Teaching and Learning Introductory Programming - A Model-Based Approach

Jens Bennedsen

Teaching and Learning Introductory Programming – A Model-Based Approach

Jens Bennedsen

Executive Summary

The dissertation identifies and discusses impact of a model-based approach to teaching and learning introductory object-oriented programming both for practitioners and for computer science education research.

Learning to program is notoriously difficult. This dissertation investigates ways to teach introductory object-oriented programming at the university level. It focuses on a model-based approach, describes and argues for this approach and investigates several of its aspects. It gives an overview of the research in teaching introductory programming in an objects-first way. The dissertation also investigates ways for university teachers to share and document best practices in teaching introductory object-oriented programming through pedagogical patterns.

The dissertation addresses both traditional young full-time students and experienced programmers (although not in object-orientation) participating in part-time education. It examines whether the same success factors for learning programming apply to a model-based approach as to introductory programming courses in general for full-time students and gives a general overview of research in success factors for introductory programming. Some factors are the same, because students' math competence is positively correlated with their success. The dissertation examines how experienced programmers link a model-based programming course to their professional practices. The general answer is that the part-time students do not need to have a direct link to their specific work-practice, they expect to create the link themselves; but the teacher must be aware of the conditions facing the part-time students in industry. Furthermore, the dissertation addresses interaction patterns for part-time students learning model-based introductory programming in a net-based environment. A previously prepared solution to an exercise is found to mediate the interaction in three different ways.

Design patterns have had a major impact on the quality of object-oriented software. Inspired by this, researchers have suggested pedagogical patterns for sharing best practices in teaching introductory object-oriented programming. It was expected that university teachers' knowledge of pedagogical patterns was limited, but this research proved that to be wrong; about half of the teachers know pedagogical patterns. One of the problems this dissertation identifies is the lack of a structuring principle for pedagogical patterns; potential users have problems identifying the correct patterns to apply. An alternative structuring principle based on a constructivist learning theory is suggested and analysed.

Preface

This dissertation is submitted as partial fulfilment of the requirements for the Dr. Philos degree in the Faculty of Mathematics and Natural Sciences, University of Oslo, Norway.

This dissertation was part of a research project, COOL (Comprehensive Object-Oriented Learning – see Berge, Fjuk, Groven, Hegna, and Kaasbøll (2003) and COOL (n.d.), initiated by Kristen Nygaard shortly before his death in 2002. The funding for my work has come from IT University West, Denmark.

The dissertation is composed of an introductory essay and eight research articles. The essay presents the problem area, the research goals and methods, related research, a discussion of the findings, and the PhD research's contributions to practice, as well as computer science education research and suggestions for further research. The eight articles listed below (five journal articles and three conference articles) are included in the dissertation as appendices.

Bennedsen, J. and Caspersen, M. E. (2004). Programming in Context: a Model-First Approach to CS1. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, United States, 477 – 481.

Bennedsen, J., & Caspersen, M. E. (2005b). Revealing the Programming Process. *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, United States, 186 – 190.

Bennedsen, J. and Fjuk, A. (2006). Learning Object-Orientation by Professional Adults. *International Journal of Continuing Engineering Education and Life-Long Learning*, 16(6), 453 – 465.

Bennedsen, J., Berge, O. and Fjuk, A. (2005). Examining social interaction patterns for online apprenticeship learning – Object-oriented programming as the knowledge domain. *European Journal of Open, Distance and E-learning*. 2005 / II.

Bennedsen, J. and Caspersen, M. E. (2006a). Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming? *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education),* 38(2), 39 – 43.

Bennedsen, J. and Caspersen, M. E. (2005a). 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, Washington, United States, 155 – 163.

Bennedsen, J. and Eriksen, O. (2006). Categorizing Pedagogical Patterns by Teaching Activities and Pedagogical Values. *Journal of Computer Science Education,* 16(2), 157 – 172.

Bennedsen, J. (2006b). The Dissemination of Pedagogical Patterns. *Journal of Computer Science Education*, 16(2), 119 – 136.

Acknowledgement

When I started to teach Advanced Computer Studies (in Danish: Datamatikeruddannelsen) at Aarhus Business College back in 1988, I shared an office with Michael Caspersen. Since then we have become very good professional and private friends. Michael and I have always had very inspiring and constructive discussions on how to teach introductory programming, and many of the views stated in the model-based approach are views formulated and shared by Michael and me – thanks, Michael!

A PhD study hopefully qualifies a student to become a researcher. This includes knowledge of research methods, which I did not have before I started this work. Fortunately, the COOL project started with Annita Fjuk as the project manager the same time that I started my research. I joined the project on a loosely coupled basis, but had the privilege to work with Annita, who – in her very gentle and productive way – introduced me to methodical considerations in my research. Thank you, Annita, for your great support!

When I started teaching back in 1988, Ole Eriksen was head of the department of the Advanced Computer Studies. Ole and I have therefore known each other since then, and have shared a mutual interest in teaching. I have had the pleasure to work with Ole on my research in pedagogical patterns, resulting in many inspiring and thought provoking discussions.

In Seattle, Washington, United States, at the first ICER workshop, I met Carsten Schulte. We quickly found a common interest in teaching introductory object-oriented programming, and since then I have had the pleasure of working with him. We have spent numerous hours Skype'ing and have developed a very productive way of collaborating across the Danish-German boarder. It has been a pleasure to get to know you Carsten, and to work with you.

Jens Kaasbøll has spent several hours reading preliminary versions of this dissertation in order to give precise and constructive comments on my work. Jens has furthermore helped me with navigating the formalities of the doctorial program at the Faculty of Mathematics and Natural Sciences. Thank you, Jens!

Gitte Møldrup-Nielsen inspired me to start this PhD project. She has furthermore made it possible for me to use some of my time at IT University West for my PhD study, and has supported me financially; without her support, my PhD work would have been much harder.

In the final stage of this dissertation, Ole Lehrmann Madsen used his very sparse spare time to read and comment on a version of my dissertation. His deep insight into object-orientation and his interest in teaching gave me many inspiring comments for the final version of the dissertation. Thank you, Ole!

Arnold Pears, who I had the pleasure to work with in a working group at ITiCSE in Dundee, took the time to comment on the final version of this dissertation. He gave me many suggestions for improvements based on his great insights in both introductory programming and educational theory. Thank you, mate!

Last, but definitely not least, I would like to express my deepest gratitude to my wife, Helle, and my daughters, Sofie and Laura, for their encouragement, support, understanding and tolerance. I thank you all from the bottom of my heart, for enduring the uncountable hours I have worked on this dissertation.

Table of Contents

1 Introduction

Two roads diverged in a wood, and I –
I took the one less travelled by,
And that has made all the difference.

Robert Frost (1874–1963), "The Road Not Taken"

This chapter shortly introduces the problem area. It describes and argues for the research questions guiding this dissertation.

Learning to program is notoriously difficult. For example, Bergin and Reilly (2005) note that "it is well known in the Computer Science Education (CSE) community that students have difficulty with programming courses and this can result in high drop-out and failure rates." (p. 293). For almost 40 years teaching programming to novices has been considered a big challenge – and it still is (Dijkstra, 1969; Gries, 1974; McCracken et al., 2001; Robins, Rountree, & Rountree, 2003; Soloway & Spohrer, 1989; Tucker, 1996); in fact, it is considered one of seven grand challenges in computing education (McGettrick et al., 2005). Teaching introductory programming at the university level has been the basis for many lively discussions among computer science teachers (Astrachan, Bruce, Koffman, Kölling, & Reges, 2005; Bailie, Courtney, Murray, Schiaffino, & Tuohy, 2003; Bruce, 2005; SIGCSE-members, 2005) and the basis for a substantial number of articles (e.g., searching "CS1" at ACM's digital library gives 1331 hits and searching "introductory programming" gives 1309 (search results compiled August, 31 2007)). Universal approaches to the design of computing curricula also have been taken – the most well known and influential being the series of curricula recommendations made by the ACM (American Association for the Computing Machinery – (*Association for computing machinery*, n.d.)) and IEEE (Institute of Electrical and Electronics Engineers, Inc. – (IEEE, n.d.)). They first described a standardized curriculum for introductory programming at the university level in 1968 (Atchison et al., 1968). Currently, a revision and enlargement of the curriculum recommendations are underway, broadening their scope from traditional computer science to the broader field of IT (Shackelford, 2005), from Information Systems (Gorgone et al., 2002) to Computer Engineering (Soldan, 2004). Within the bachelors' program, where introductory programming is described, the latest description is from 2001, known as the "Computing Curriculum 2001," or CC2001, for short (Engel & Roberts, 2001). For an overview of the development of the ACM curricula, see Hemmendinger (2007).

This dissertation is within the field of computer science education research, more specifically, research on teaching and learning introductory object-oriented programming.

The next two sections present the research questions guiding the PhD project along with the rationale for these questions. The questions are organized and argued in relation to the didactic triangle: Content, student and teacher. The last section presents the organization of the remainder of the dissertation.

1.1 Research Focus and Research Questions

One of the first courses students encounter when learning computer science or related fields (e.g., multimedia, information science) is an introductory programming course. As CC2001 concluded:

> … the programming-first model is likely to remain dominant for the foreseeable future (p. 24).

This gives us an extra incentive to focus on problems related to introductory programming courses (commonly known as CS1 courses). However, as Doran and Langan (1995) note:

> Unfortunately, initial Computer Science courses are often characterized by: (i) a great deal of learning frustration; (ii) a fairly significant student drop rate; (iii) poorly defined exit behaviours (i.e., what students should be able to do by the time the course is completed) and (iv) inadequate tools to determine if the students learned "enough". (p. 218)

The overall focus of this dissertation is teaching and learning object-oriented introductory programming, and the overall research theme is:

> What is a good approach to teaching and learning introductory programming using the objects-first approach?

A good approach avoids the problems noted by Doran and Langan (see above). A *good approach* to learning is different from student to student (e.g., the Dunn and

Dunn learning style focuses on “each individual’s multidimensional characteristics in order to determine what will most likely trigger each student’s concentration, *and* cause long-term memory” (Dunn, 1990, p. 224)). Many different learning styles exist with associated teaching methods (Cassidy, 2004)[1]. It is therefore important to notice the indefinite article **a** good approach; we do not have the hypothesis that one approach suitable to all students and/or teachers exists.

There are several basic assumptions and hypotheses in the theme. Learning programming takes place in an institutional setting (more precisely at a university) where the traditional way to learn something is to follow a course. In this course, the course designer (typically the teacher) defines the goals and content of the course and the way to evaluate these goals. Consequently, it is the course designer who defines what *programming* means, in what *programming language* the programming is expressed, what *tools and resources* is to be used and what *level of programming expertise* is required.

Given these prerequisites, this dissertation aims at giving one answer within the research theme. It does so by describing one approach to teaching introductory object-oriented programming (the word “approach” is used in a broad sense including the goals of the course, pedagogical/didactical decisions, programming language, and tools). The dissertation further researches ways to document and exchange teaching experience among teachers teaching introductory programming and success factors for students participating in learning introductory object-oriented programming.

In “Computer Science Education Research” Fincher and Petre (2004) write as the first sentence that “computer science education research is an emergent area and is still giving rise to a literature” (p. 1). The area draws on many different fields of scholarship and research – computer science, psychology, education, and technology – with the aim of building theories for teaching and learning computer science education specific topics. Since computer science in itself is a relatively young field (ACM – Association for Computing Machinery, the first society in comput-

[1] Learning styles have also been used in research within introductory programming (L. Thomas, Ratcliffe, Woodbury, & Jarman, 2002). However, this is not the focus of this dissertation.

ing – celebrates its 60th anniversary in 2007), there are many unexploited research questions in this field.

1.1.1 The Research Questions

Didactics (Greek: didáskein (διδασκειν) = to teach; lore of teaching) is defined by Merriam-Webster's On-Line Dictionary as

> systematic instruction (and a synonym for pedagogy, which is defined as "the art, science, or profession of teaching") (Merriam-Webster, 2007)

A better and operational definition is given by Wikipedia as

> the theory of teaching and, in a wider sense, the theory and practical application of teaching and learning. Slight variations on the term are widely used in many cultures, but rarely used in English-speaking cultures. (Wikipedia, 2007)

In the Anglo-American part of the world, didactics has a negative connotation. However, as Hamilton (1999) note

> The European discourse of didactics is, I suggest, very close to the Anglo-American discourse of pedagogies. Only their language divides them. (p. 135)

According to Diederich (1988) the relevant elements in a teaching situation can often be described by the didactic triangle (see figure 1). According to Peterßen (2001), this triangle originates from Johann Friedrich Herbart; a German philosopher, psychologist, and by some defined as the founder of pedagogy as an academic discipline.

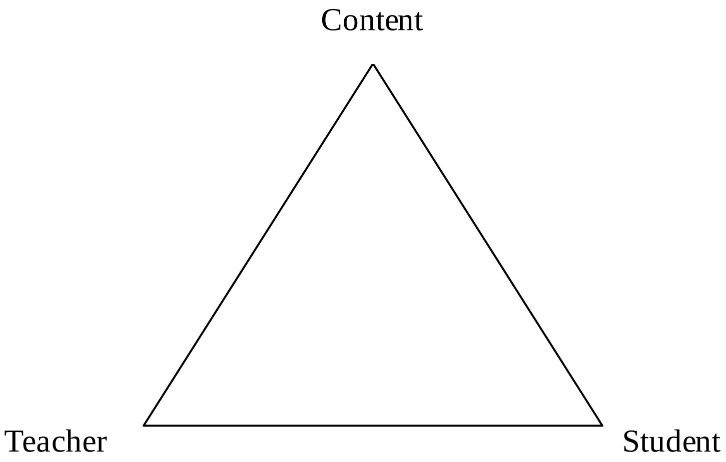


Figure 1: The didactic triangle

The didactic triangle describes the relevant elements in a teaching situation; the model also stresses the relations between them. In this dissertation the didactic triangle is not used as an analytic framework but as a means to identify the relevant elements to study. More specific didactic models exist; one is discussed later in the dissertation (section 4.1).

This dissertation addresses all three elements of the didactic triangle within the area of object-oriented introductory programming. It describes, motivates and argues for a particular didactical design of an introductory programming course. It researches aspects of students being taught using this didactical design and teachers teaching within the area of introductory object-oriented programming.

The particular design of an introductory programming course under investigation is termed “a model-based approach”. The approach will be described in detail in section 6.1, but it is founded on four principles:

1. Objects from day one – in the beginning students use predefined classes to create objects, then they imitate the implementation of a class and finally they create classes,
2. A balanced view of the three perspectives on the role of a programming language (Knudsen & Madsen, 1988),
3. Enforce the use of a systematic way to implement the description of a solution, and
4. An explicit focus on the programming process, not just the programming language.

One of the main outcomes of this dissertation is such a design and research of it. The design is based on pedagogical theories helping to fulfil the principles. The explicit focus on the programming process broadens the content category to include process not just traditional static content.

The second main outcome of this dissertation is a way to structure and generalize teaching experiences in the form of pedagogical patterns based explicitly on learning theories. Graphically the outcome is illustrated in figure 2.

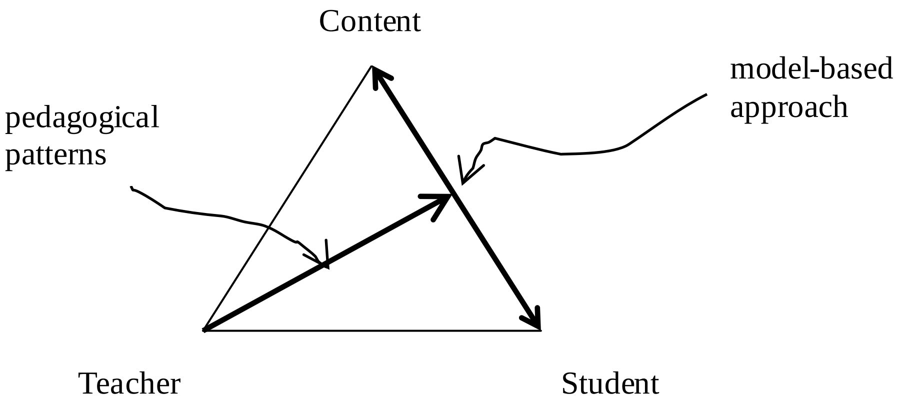


Figure 2: Main outcomes of dissertation

1.1.1.1 Content

CC2001 characterize the current state of affairs of programming courses in the following way:

> 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. (p. 23)

Consequently, we need to find ways to teach programming so that there is a broader scope than just the syntax and particular characteristics of the programming language and with a goal of learning a systematic programming process other than trial and error.

Knudsen and Madsen (1988) describe three perspectives on the role of a programming language:

> **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.

> **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 and separate compilation.
>
> **Conceptual modelling**: The programming language is used for expressing concepts and structures. The focus is on constructs for describing concepts and phenomena (p. 25).

These represent a widespread three-level perspective on object-oriented programming as represented by the three abstraction levels for the interpretation of UML class models (OMG, 2007): conceptual level, specification level and code/implementation level (Fowler & Scott, 2000).

Designing a programming course requires a decision about how much time, effort, and focus are given to each of these three perspectives. It is possible to focus only on the first, instructing the computer, and to ignore the two others. This result in a course in which the programming language's details are in focus, but whose students do not learn the underlying programming paradigm as the one described in the first paragraph in the quote from CC2001. If, on the other hand, the focus is only on conceptual modelling (using a case-tool to generate code), the result is a course whose students cannot produce code by themselves.

Most of the descriptions and discussions of the objects-first strategy tend to focus on *instructing the computer* and *managing the program's description* ("Programming in Context: a Model-First Approach to CS1"). Robins, Rountree and Rountree (2003) conclude, "Typical introductory programming textbooks devote most of their content to presenting knowledge about a particular programming language" (p. 141); that is to say, the majority of them are based on an "instructing the computer" or "managing the computer" perspective. We have been able to find only four articles discussing the adoption of conceptual modelling in introductory programming courses (Alphonce & Ventura, 2002; Knudsen & Madsen, 1996; Madsen, Torgersen, Røn, & Thorup, 1998; Sicilia, 2006).

In "Grand Challenges in Computing: Education – A Summary" McGettrick et al. (2005) identify seven challenges for computer science education. One is programming issues (challenge #4) where they describe the challenge as

> Understand the programming process and programming practice to deliver effective educational transfer of knowledge and skills (p. 46)

The problems identified above and the objective of a course not just focusing on one of the perspectives of the object-oriented programming language has led to the following research question:

> R1: Is it possible to make a course design where the students learn a systematic programming process, conceptual models[2] as a structuring mechanism, and coding?

This research question is a kind of existence proof seeking a course design that meets the design criteria *systematic programming process, conceptual models as a map of the program* and where the students still learn *coding*[3]. A requirement for the design is that the students learn the above mentioned criteria; this is interpreted as the number of students passing the exam is at least as high as traditional courses. This dissertation describes such a course design called model-based.

1.1.1.2 Student

For the students, the main interest is in factors related to their success in learning object-orientation.

Most of the studies of learning introductory object-oriented programming focus on traditional, full-time university students. A growing target group in computer science education is adults with previous knowledge of programming but not object-oriented programming. Denmark's IT industry employs more than 100,000 people (Ministeriet for Videnskab - Teknologi og Udvikling, 2003). Only 15,000 of them have a university degree (at least two years of study at the college level). In other words, more than 85,000 people are professionally working in the IT industry without a college education. Many of these work as programmers, mostly trained in an imperative way of thinking[4]. Given the current trend of changing to

[2] The term "conceptual model" is to be understood in the way it is used in the three perspectives on the programming language described by Knudsen and Madsen (1988). It is not to be understood as a synonym for models in the referent system (see page 33)

[3] The term "coding" refers to the implementation of a specification in a programming language.

[4] See e.g. Raadt, Watson and Toleman (2002) who found in 2002 that "Over half of all students are initially taught using a Procedural approach" (p. 333). Levy (1995) reported that in 1995 Pascal "is still taught by most schools in the United States" (p. 24)

object-orientation (Benander, Benander, & Sang, 2004), it is relevant to study how these people learn object-oriented programming. This group knows what programming is, so one interesting question is how they link the different aspects of a course on introductory object-oriented programming to their professional practices. This has led to the following research question:

> R2a: How do the different aspects of a model-based programming course influence professionals' practice?

Lifelong learning is seen as one of the key answers to challenges that the European Union faces (European Union, 2006b). One of the elements in implementing lifelong learning is e-learning, thereby making it possible for adult students to combine their working-, family- and study-life. The addition of e-learning increases the complexity and influences the learning. But how do the interaction patterns[5] between the learners when they are adults and "not-traditional" students evolve, and how does a technology rich environment, influence the interaction?

> R2b: What are the interaction patterns for professionals learning object-oriented programming with a focus on a systematic programming process and conceptual models as structuring mechanisms in a technology rich distance education setting?

Factors for predicting success in learning object-oriented programming can be divided into two groups – those that teachers cannot do anything about and those that teachers can address. Several studies on this subject have focused on predictors that the teachers have no control over (see section 3.2.3 and appendix 11.9). These studies report mixed findings (some report that over 60% of a student's exam grade can be predicted by a few factors such as math grade or abstraction ability – e.g., Kurtz (1980); others found that none of the studied variables have any effect – e.g., Ventura and Ramamurthy (2004)). Only a few of these studies were done in an objects-first course.

Are the factors that promote the students' success when taught in a model-based way the same as in other approaches? The factors investigated for this dissertation

[5] Please do not confuse the use of the word "interaction pattern" here with the use of the word "pattern" in pedagogical pattern or design pattern. In this context it means "observable characteristics of the interaction of students"

include among other variables previous math grades, gender, and abstract reasoning ability.

> R3: Do model-based introductory programming courses have the same success factors as more traditional courses?

The rationale for conducting this kind of study is to see if the oft-cited predictors apply equally to a model-based course. This rationale is two-fold: first, if the same predictors apply, it could be taken as an argument for the generalizability of previous studies' findings. Second, if the traditional predictors accurately predict the success of students participating in a model-based course, studies focusing on more detailed elements need not be done; it would be more relevant to find ways to improve these success factors.

Focusing on the two subgroups of students – professionals with previous imperative programming experience and traditional young students who have just finished high-school – calls for a comparison of the success factors between these two groups. Unfortunately, the number of professionals with previous imperative programming experience we can address is too limited to do statistical analysis for differences between the groups. Consequently – and in order for the results to be comparable with findings in related research – the group researched for success factors will be traditional young students who have just finished high-school.

1.1.1.3 Teacher

In "Grand Challenges in Computing: Education – A Summary" McGettrick et al. (2005) describe the grand challenges for computing education: Perception of computing, Innovation, Competencies, Programming issues, Formalism, about e-learning and Pre-university issues. The basis for the selection of challenges was a conference held in Newcastle on 30 and 31 March 2004. However, as Lister (2005a) notes:

> The Grand Challenges paper ignores one important area, and that is ourselves. All of us who teach computer science were students once, and unusually successful students. This makes us blind to some of the problems faced by typical students, and makes us reluctant to change from the content and teaching style that so appealed us. Computer education research should study teachers as much as students. (p. 15)

Kansanen and Meri (1999) discusses the relation between the teacher and the content:

> In the relation between the teacher and the content the teacher's competence in content is in focus. From the point of view of subject didactics the question is of the balance between subject knowledge and pedagogy…. Of importance is also that the teacher's relation to the content is sufficiently many-sided and there is pedagogical competence enough. (p. 113)

Typically, teachers who teach computer science – like most other university teachers – have no formal education in teaching. Many are more interested in research than in teaching (Lister, 2006). This research therefore focuses on ways that teachers not necessarily interested in pedagogy can improve their teaching of introductory object-oriented programming.

Others have focused on this problem. One approach we find promising is the use of patterns to capture knowledge of teaching and learning object technology (Bennedsen & Eriksen, 2003; Bergin, n.d.; Eckstein, 2001; Fincher & Utting, 2002; *The pedagogical patterns project*, n.d.) in the form of pedagogical patterns. Pedagogical patterns try "to capture expert knowledge of the practice of teaching and learning" (*The pedagogical patterns project*, n.d.) in a compact way which is easy to communicate.

Design patterns (Gamma, Helm, Johnson, & Vlissides, 1995) have had an enormous impact on the dissemination of good design practice within object-oriented software developers. Several other areas have tried to use this approach to assimilate and distribute knowledge of solutions to common, recurring problems (Alexander, Ishikawa, & Silverstein, 1977; Fowler, 1997; Hall & Hord, 2000).

Concrete teaching advice in CS1 is highly rated by teachers. There are numerous conferences and workshops focusing on teaching programming: general ones like SIGCSE: *The Annual Technical Symposium on Computer Science Education* (1970)[6], ITiCSE: *The Annual Conference on Innovation and Technology in Computer Science Education* (1996), FIE: *Frontiers in Education* (1971) and ACE: *Australasian Computing Education Conference* (1996) as well as specialized ones like *Program Visualization Workshop* (2000, biannual, held in even years in con-

[6] The number in parenthesis marks the year of origin

junction with the ITiCSE conference), JICC: *Java & the Internet in the Computing Curriculum conference* (1997) and CCSCSE: *Annual Consortium for Computing Sciences in Colleges CCSC: Southeastern Conference* (1987). The examples and advice offered at these conferences are enormous, but does not seem to be described in a way that it is transferable and useful for practitioners. Haberman (2006) discusses the same problems for the sub-group of High-school computer science teachers (we believe the same argument is true for university computer science teachers):

> The computer science high school teaching community of practice possesses a rich, distributed practical knowledge base consisting of individual pieces of knowledge regarding pedagogical expertise. However, without a proper means of communication and common rules of discourse, the individual pieces of knowledge might remain as personal property only, and will be not transferred within the community, and eventually might even get lost. (p. 87)

Haberman continues to discuss and illustrate how pedagogical patterns can be used as a communication and reflection means.

Fincher (1999) shares the same hope in her article "Analysis of Design: An Exploration of Patterns and Pattern Languages for Pedagogy":

> For the purposes of education, it would be of great utility if we could discover and develop ways to exchange knowledge about the design of effective learning environments. (p. 331)

Although pedagogical patterns have been around since 1995 (Eckstein, 2001), several authors claim that their impact is limited. Fincher and Utting (2002) state: "Pedagogical patterns still lack widespread acceptance" (p. 201). This statement is compliant with the remarks made by Bennedsen and Eriksen (2003), who said: "To our knowledge there are no reports on the awareness of pedagogical patterns among teachers, no systematic evaluation of the effect of pedagogical patterns or other ways of evaluating pedagogical patterns" (p. T4A-3).

Two obvious answers to the question why pedagogical patterns have a limited impact are: Teachers do not know of pedagogical patterns and/or teachers do not find pedagogical patterns useful. This has led to the first research question aimed at teachers:

> R4a: What are the awareness of, use of, and attitude toward pedagogical patterns among computer science university teachers around the world?

Pattern and pattern languages originate from Christopher Alexander's works "The Timeless Way of Building" (Alexander, 1979) and "A Pattern Language. Towns, Buildings, Construction" (Alexander et al., 1977) on the architectural design of buildings and towns. Alexander defines a pattern as a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem. He describes 253 patterns in a consistent typographical style (a pattern template) in "A Pattern Language". A "language" has two senses for Alexander according to Fincher (1999): "The first sense is that we all share a design language that his work merely articulates […] The other sense of "language" is of an organising principle which facilitates the use of patterns. Just as a dictionary is (normally) organised on an alphabetic principle, thus allowing the user to find the required word easily, so the "language" of the patterns organises them to facilitate access." (p. 333). For an introduction to the work of Christopher Alexander see also Lea (1994).

A pattern language needs an organising principle in order to be useful. The organizing principle in "A Pattern Language" is scale: Town, street, house. Patterns are organized according to their usefulness from "city-relevant" to "house-relevant". The organizing principle in Gamma, Helm, Johnson, and Vlissides (1995) (commonly known as the GoF book) is: *Creational* (object creation), *Structural* (structures in a class model) and *Behavioural* (ways to organize recurring algorithmic problems). The collection of pedagogical patterns has "neither order, nor organising principle: instead, four separate "indexes" are provided, each on a different axis: A *Learning Objectives Index*, a *Teaching/Learning Element Index*, an *Alphabetical Index* and an *Author Index*." (Fincher, 1999, p. 340)

The lack of order or structure to the pedagogical patterns could be one of the answers to the limited impact of them. This has led to the following research question:

> R4b: How can pedagogical patterns be structured to make them more useful for university teachers?

1.2 A Summary of the Research Questions

This dissertation investigates six more specific questions of the general research theme (learning introductory programming using an objects-first approach). The six questions are grouped in four (model-based programming course (R1), professional adults learning introductory programming (R2a + R2b), success factors for a model-based course (R3) and pedagogical patterns (R4a+R4b)):

> R1: Is it possible to make a course design where the students learn a systematic programming process, conceptual models as a structuring mechanism, and coding?
>
> R2a: How do the different aspects of a model-based programming course influence professionals' practice?
>
> R2b: What are the interaction patterns for professionals learning object-oriented programming with a focus on a systematic programming process and conceptual models as structuring mechanisms in a technology rich distance education setting?
>
> R3: Do model-based introductory programming courses have the same success factors as more traditional courses?
>
> R4a: What are the awareness of, use of, and attitude toward pedagogical patterns among computer science university teachers around the world?
>
> R4b: How can pedagogical patterns be structured to make them more useful for university teachers?

1.3 Organization of the Rest of the Dissertation

This dissertation is composed of an introductory essay (chapters 1 through 9) and eight research articles, provided as appendices. Chapter 2 gives background and motivation for this dissertation. Chapter 3 presents the research field "computer science education" with a specific focus on the areas relevant to this PhD work. Chapter 4 discusses learning theories. It gives a very brief general overview of learning theories and more detailed description of the pedagogical approached used in this research. Chapter 5 gives an account of the research methods. It argues for the methods used, describes the data sources and discusses trustworthiness of both the quantitative as well as the qualitative studies. Chapter 6 answers the research questions. Chapter 7 points out some implications for practice of this research both in the area of computer science education and pedagogical patterns. Many issues are debates in the community of computer science educators, chapter 8 discusses this dissertation's relation to some of these issues and the last chapter (chapter 9) points to some further research.

The titles of the eight research articles included as appendices are:

- Article I: Programming in Context: a Model-first Approach to CS1.
- Article II: Revealing the Programming Process.
- Article III: Learning Object-Orientation by Professional Adults.
- Article IV: Examining Social Interaction Patterns for Online Apprenticeship Learning – Object-oriented Programming as the Knowledge Domain.
- Article V: Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming?
- Article VI: An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course.
- Article VII: Categorizing Pedagogical Patterns by Teaching Activities and Pedagogical Values.
- Article VIII: The Dissemination of Pedagogical Patterns.

The eight articles address the four groups of research questions in the following way:

	R1	**R2**	**R3**	**R4**
Article	I, II	III, IV	V, VI	VII, VIII

2 Background and Motivation

> *Ability is what you're capable of doing.*
> *Motivation determines what you do.*
> *Attitude determines how well you do it.*
> Lou Holtz (1937-1980), American Football Coach

This chapter discusses the background of the research presented. It starts with a section on computer science education in a world-wide perspective to indicate some of the implications a better teaching of computer science could have. The next section discusses teaching and learning programming and gives a background for the different ways to teach introductory programming defined by ACM. The following section gives a background for teaching object-oriented programming; it defines object-orientation, discusses what programming is and, as a consequence of the discussion, what needs to be taught in a programming course. The next section gives a brief account of my personal background in the area of teaching introductory programming – a background that naturally has had a major impact of the scope of this dissertation. The last section introduces the project in which part of this dissertation was done.

IT is embedded in everything. IT has a dominant position in the western world's economy; in Denmark the IT industry's part of the total wealth creation in the private sector was 8.7% in 2001 (Danmarks statistik & Ministeriet for Videnskab, teknologi og udvikling, 2005, p. 10). The declining number of students enrolling in computer science programs (Denning, 2004; Madsen, 2006) has therefore become an issue in the political debate (Ministeriet for Videnskab - Teknologi og Udvikling, 2006). Added to that problem is the low retention rate within computer science (Beaubouef & Mason, 2005; Denning, 2004; Hundhausen, Farley, & Brown, 2006; Kaasbøll et al., 2004, p. 1; Kumar, 2003, p. 40). These two problems stress the importance of building knowledge of the challenges and pitfalls in computer science education.

2.1 A World-Wide Perspective on Computer Science Education

UNESCO (United Nations Educational, Scientific and Cultural Organization, (UNESCO, 2007)) collects data on the number of students who enrol and graduate in different fields worldwide. In Table 1 the number of students enrolled in tertiary computing education and the number of students graduating in regions covered by UNESCO is given (UNESCO, n.d.). Tertiary studies last from two to eight years. The enrolment does not include the United States, India and China, so we estimate that more than two million students per year enrol in computing studies worldwide.

	Number of students enrolled	Number of students graduated	Number of countries who have supplied data[7]
1999	665,058	154,164	43
2000	818,442	205,928	53
2001	1,053,813	239,795	58
2002	1,628,815	319,198	59
2003	1,696,519	374,972	59
2004	1,505,437	368,765	62
Average	*1,228,014*	*277,137*	*56*

Table 1: Enrollment and graduation in tertiary computer science education.

To give some indication of how difficult students find computer science, we have contrasted the numbers from the years 1999 (enrolment) and 2004 (graduation); this is a rough estimation of a world-wide pass (or fail) rate[8]. For some countries, 1999 enrolment numbers or 2004 graduation numbers were not accessible; in those cases we have used numbers from the neighbouring years. The result is shown in figure 3. From this figure, the number of students graduating in 2004

[7] Some countries have only given numbers for the enrollment or graduation. The number is the total number of countries, including the ones who only have supplied one of the numbers. For example, the United States supply graduation numbers but not enrollment.

[8] World-wide the lengths of study programs differ, so the number is a rough approximation to the percentage of students passing a computer science program.

was only 27% of the number of students enrolled in 1999[9]. In other words, it seems that there are a huge number of students enrolling in tertiary education who do not graduate; with the reservations mentioned above, 1.45 million students drop out of their computer science program. In this light, just a small improvement of the pass rate of CS1 would cause a gigantic increase in the number of students passing (and perhaps eventually graduating) — a one percent increase in the pass rate means 20,000 students extra passing CS1. This underlines the importance of building up knowledge on how to teach programming!

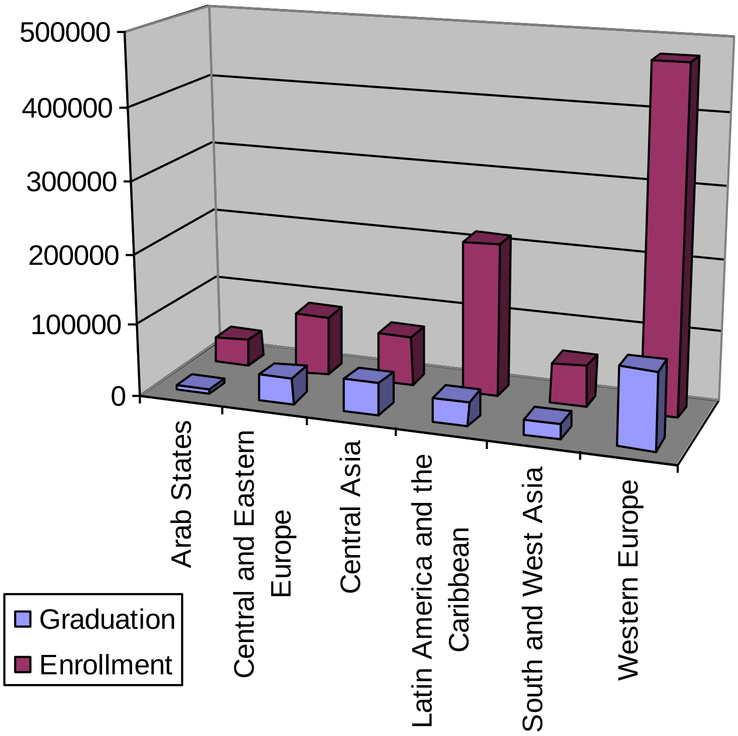


Figure 3: Enrollment and graduation in computing by region

2.2 Teaching and Learning Programming

Over the years, ongoing discussions have debated the content of an introductory programming course. In order to define a common curriculum, including an introductory course, ACM and IEEE established the Joint Task Force on Computing Curricula 2001 (CC2001). The charter was, “To review the Joint ACM and IEEE CS Computing Curricula 1991 and to develop a revised and enhanced version for the year 2001 that will match the latest developments of

[9] Note that a student graduating in 2004 need not to be enrolled in 1999, so the number take into account that students may use more or less than the scheduled study time.

computing technologies in the past decade and endure through the next decade." The resulting report gives a de facto description of an introductory programming course.

CC2001 describes the CS body of knowledge. This body of knowledge is organized hierarchically into three levels. The highest level of the hierarchy is the *area***,** which represents a particular disciplinary subfield. There are fourteen areas such as *programming fundamentals* (PF) or *software engineering* (SE). The areas are broken down into smaller divisions called *units***,** which represent individual thematic modules within an area; as an example, fundamental programming constructs is a unit on *programming fundamentals*. Each unit is further subdivided into a set of *topics***,** which are the lowest level of the hierarchy. It lists the topics that <u>must</u> be included and some topics that <u>can</u> be included. (Engel & Roberts, 2001, p. 14)

It furthermore describes three approaches to teaching introductory programming with a focus on programming: imperative-first, objects-first, and functions-first:

> **Imperative-first:** The imperative-first approach is the most traditional of the models we have included in this report…[the first course] offers an introduction to programming in an imperative style, using a structure similar to that in CS1 as defined in Curriculum '78 (Austing, Barnes, Bonnette, Engel, & Stokes, 1979; Koffman, Miller, & Wardle, 1984). [The second course] then extends this base by presenting much of the material from the traditional CS2 course (Koffman, Stemple, & Wardle, 1985), but with an explicit focus on programming in the object-oriented paradigm … the first course focuses on the imperative aspects of that language: expressions, control structures, procedures and functions, and other central elements of the traditional procedural model. The techniques of object-oriented design are deferred to the follow-on course (p. 29).
>
> **Objects-first:** The objects-first model also focuses on programming, but emphasizes the principles of object-oriented programming and design from the very beginning … The first course in either sequence begins immediately with the notions of objects and inheritance, giving students early exposure to these ideas. After experimenting with these ideas in the context of simple interactive programs, the course then goes on to introduce more traditional control structures, but always in the context of an overarching focus on object-oriented de-

> sign. The follow-on courses then go on to cover algorithms, fundamental data structures and software engineering issues in more detail (p. 30).
>
> **Functional-first:** The functional-first style was pioneered at MIT in the 1980s (Abelson & Sussman, 1985) and is characterized by using a simple functional language, such as Scheme, in the first course … To cover the material that is essential in the first year, an introductory course that follows the functional-first strategy must be followed by an intensive course that covers object-oriented programming and design (p. 30).

2.3 Teaching and Learning Object-Oriented Programming

Many define Simula as the origin of object-orientation – and consequently Kresten Nygaard and Ole Johan-Dahl as the inventor of object-orientation (ACM, 2002; Rentsch, 1982).

Madsen, Nygaard and Møller-Petersen (1993) defined object-oriented programming 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. (p. 286).

One important word here is "model," which can represent either an abstract description of the problem domain, a specification model describing the overall structure of the solution or a concrete implementation model. However, these models are connected and can be seen as a hierarchy of abstractions. This underlines the importance of the students' understanding of models and how, in a systematic way, specification models can be transformed into implementation models (code).

Object-orientation has become a main paradigm of programming in industry, as well as in computer science and engineering education (Benander et al., 2004; Dale, 2005; Douglas & Hardgrave, 2000; Lucas, 2003; Raadt et al., 2002; Schulte & Bennedsen, 2006). As reported by Schulte and Bennedsen (2006), more than 85% of all introductory programming courses include object-oriented concepts, and of these more than 60% are objects-first courses (p. 5).

Learning introductory programming is considered difficult by teachers. Schulte and Bennedsen (2006) report that:

> In general it seems that learning programming is seen as difficult. […] "25% of the students have difficulty with the subject" (p. 5).

Many different research topics exist within the research on teaching introductory object-oriented programming. Ragonis and Ben-Ari (2005a) did

> an extensive survey of the literature on teaching OOP, which showed that the following subjects have interested researchers: OOP vs. procedural programming, languages, environments and other tools such as visualizations, teaching approaches and relevant teaching theories such as constructivism and cognitive apprenticeship (p. 206).

They found that the underlying OOP model must be taught early, and that both classes and objects must be treated, but detailed treatment of control statements and advanced object-oriented topics such as inheritance should be postponed. They furthermore found that educators agree that there is a need for a different pedagogical approach to teaching OOP, and that there is lack of provable pedagogical approaches, appropriate books and suitable environments for novices. They conclude: "We did not find any publications that reported a long and formal research project on teaching OOP" (p. 207).

2.3.1 What is Programming?

There is a long and ongoing discussion on the goals for an introductory programming course. Not even the curriculum recommendations in CC2001 give precise competence goals for a CS1 course. The programming fundamentals (PF) area of the CS body of knowledge consists of five units: Fundamental programming constructs (PF1, six topics), Algorithms and problem-solving (PF2, five topics), Fundamental data structures (PF3, 12 topics), Recursion (PF4, six topics), and Event-driven programming (PF5, three topics). Within these units, programming skills are only mentioned as three topicss:

> **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.

A discussion of the learning objectives is outside the scope of this dissertation; it suffices to note that there is very little focus on programming skills: as Caspersen (2007) notes:

> 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 (p. 70)

Madsen, Nygaard and Møller-Pedersen (1993) describe a general framework for system development (see figure 4Figure). Their framework builds upon their perspective on objected-oriented programming (see quote on page 30).

Traditionally, abstraction in the referent system is known as *analysis*, modelling from the referent system to the model system is known as *design* and abstraction in the model system is known as *coding*.

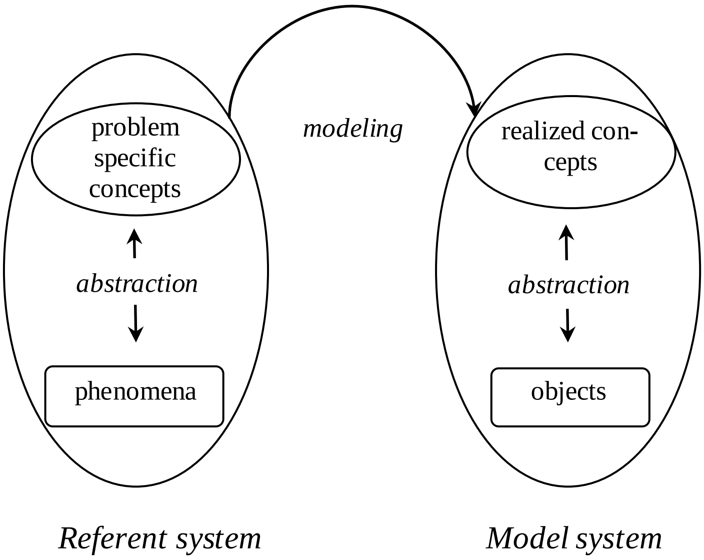


Figure 4: The system development process (Madsen et al; 1993, p. 286)

Buck and Stucki (2000) argue that the traditional way to teach programming is “to recapitulate the systems development process, because that is the order in which we have learned to apply our craft.” (p. 76). However, as they argue based on an analysis of cognitive levels of competences founded on Bloom’s taxonomy of cognitive learning (Bloom, Krathwohl, & Masia, 1956), the typical order in (waterfall) system development (analysis → design → coding) suggest that the highest cognitive level is taught first. They conclude that “based on these pedagogical influences, the process of learning software engineering should turn the Development Life Cycle on its head, incrementally building towards design and then analysis.” (p. 76)

Doran and Langan (1995) describe[10] the development of their CS1 course based on “strategic sequencing and associated levels of mastery of key topics based on Bloom levels” (p. 218). In general they find their approach satisfactory, but as they notice, they lack a control group for assessment purposes. Their results indicate that the students perceived a benefit in the explicit expectations and goals.

Adams (1996) discusses what software design methodology should be taught in CS1 when a change is made from an imperative programming language (Pascal at the time of the article) to an object-oriented programming language (C++ at the time of the article). He conclude

[10] Their description includes both their CS1 and CS2 course

> Two common answers are (i) continue teaching structured design in CS 1 and switch to object-oriented design in CS2; or (ii) teach object-oriented design from the outset in CS 1. We believe that both of these approaches have significant drawbacks. (p. 78)

He argues that teaching structured design in CS1 requires the students to "unlearn" this when they learn object-oriented design in CS2 – a major burden on the students. Learning object-oriented design from the outset requires the students to know the entire object-oriented conceptual framework[11] – a very demanding task.

2.3.2 What Should Students Learn when Learning to Program?

A substantial amount of research has been conducted regarding teaching introductory programming (e.g., Christensen (2004); Gries (1974) and Pattis (1993)). Before the mid-nineties, all studies were on students learning imperative style of programming; see e.g., Winslow (1996) for a summary of these studies. Later studies have also focused on learning an object-oriented style of programming. One aspect of this discussion is – naturally – what novices find problematic when they learn programming (see e.g., Détienne (1997); Goold and Rimmer (2000); Lahtinen, Ala-Mutka, and Järvinen (2005); Rountree, Rountree, Robins, and Hannah (2004) plus Wiedenbeck, Fix, and Scholtz (1993). A comprehensive review of this is Robins et al (2003)).

Several authors have asked teachers about their view on topics for CS1. Milne and Rowe (2002) asked students and lecturers from Ireland and the United Kingdom to rank the difficulty of several topics from the C++ programming language. The three most difficult topics were pointers, virtual functions and dynamic allocation of memory (with malloc). Dale (2006; 2005) asked teachers, in a free-text form, the following question: "In your experience, what is the most difficult topic to teach in CS1?" Based on these answers, Dale performed a content analysis and abstracted a list of topics teachers find relevant. Dale furthermore asked about the relevance of several other topics by asking the respondents to give the number of

[11] The object-oriented conceptual framework include notions of concepts and phenomena, identification of objects, identification of classes, classification, generalization and specialization, multiple classification, reference- and part-of composition (Madsen, Nygaard, & Møller-Pedersen, 1993).

hours they teach the particular topic. Schulte and Bennedsen (2006) performed a study on the importance, difficulty and relevance of 28 topics compiled from the study of Dale, Milne and Rowe as well as CC2001. They found that the teachers found the "classic" topics (e.g. selection and iteration, simple data structures, parameters) to be the most relevant and those which should be taught at the highest competence level (on average at the analysis level according to Bloom's taxonomy (Bloom et al., 1956)). They found that the teachers found recursion, algorithm efficiency, polymorphism and inheritance, generics and advanced data structures to be the most difficult topics for the students to learn (all topics close to "Very Difficult (50% have problems)").

In the 1980s, a series of experiments were undertaken in order to understand the problems a novice faces when learning programming. du Boulay (1989, p. 283) summarized these and described five overlapping areas that a student must be able to master:

> **General orientation**: What is the general idea of programs, what are they for, and what can be done using them?
>
> **The notional machine**: An abstract model of the machine when it executes programs (i.e., the running program's meaning). The notional machine's properties are language dependent (du Boulay, O'Shea, & Monk, 1999, p. 265).
>
> **Notation**: The syntax and semantics of the programming language used.
>
> **Structures**: (Abstract) solutions to standard problems, a structured set of related knowledge.
>
> **Pragmatics**: The skills of planning, developing, testing, debugging and so on.

In other words it is not enough just to present the syntax and semantics of a programming language as is the dominant way to teach introductory programming (Kölling, 2003; Robins et al., 2003), we also need to teach the other areas.

Schulte and Bennedsen (2006) included a study on teachers' view of the role of the five areas in an introductory programming course. They found the following percentage of areas covered (table from p. 24 in the article):

Taught	Overall	University	College	High school	OO covered	
					Yes	No
General orientation	53%	52%	63%	45%	56%	53%
Notional machine	29%	29%	32%	27%	30%	33%
notation +	57%	58%	65%	43%	60%	58%
structures	58%	57%	67%	51%	61%	60%
pragmatics	48%	45%	56%	45%	52%	44%

Table 2: Percentages of areas taught (no sign. differences between OO covered yes/no)

Apart from the low coverage of the notional machine, all areas are covered equally.

The general orientation should give the students an understanding of computers and the problems that are relevant to solve using computers. We should give the students an understanding of the difference in the referent system and the model system; in the referent system it is possible to use a prototypical view (or definition) of a given concept whereas in the model system we must give an Aristotelian definition of a given concept. Madsen, Nygaard and Møller-Pedersen (1993) define prototypical view and Aristotelian view as follows (the intension of a concept is a collection of properties that in some way characterize the phenomena in the collection of phenomena that the concept somehow covers):

> **Aristotelian view**: The intension of a concept is a collection of properties that may be divided into two groups: the defining properties that all phenomena in the extension must have and the characteristic properties that the phenomena may or may not have. The nature of the properties is such that it is objectively determinable whether or not a phenomenon has a certain property. (p. 292)
>
> **Prototypical view**: The intension of a concept consists of examples of properties that phenomena may have, together with a collection of typical phenomena covered by the concept, called prototypes. (p. 293)

The understanding of the different views on concepts in the referent and model system does not dependent on the paradigm used, i.e. we need to make sure that the students learn this regardless of the actual programming paradigm taught.

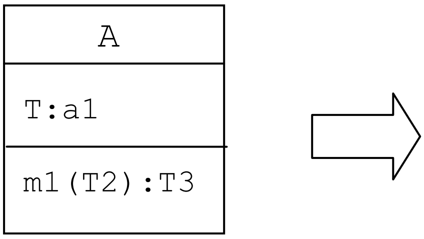
The notional machine depends on the programming paradigm taught. If it is an imperative paradigm, the execution can be described by the call stack (given that we do not dynamically allocate memory). In the object-oriented paradigm, we furthermore need a description of the objects and there links. Several articles have found that students often construct wrong models of an object-oriented program's execution (Lahtinen et al. (2005); Milne and Rowe (2002) plus Ragonis and Ben-Ari (2005b)). Milne and Rowe (2002) conclude that most reported learning problems are due to the "inability [of students] to comprehend what is happening to their program in memory, as they are incapable of creating a clear mental model of its execution." (p. 55). Lahtinen, Ala-Mutka, and Järvinen (2005) studied what domains students found problematic. They concluded: "For example, there are often misconceptions related to variable initialization, loops, conditions, pointers and recursion. Students also have problems with understanding that each instruction is executed in the state that has been created by the previous instructions" (p. 15). These findings are consistent with the findings of Guzdial (1995) "A specific problem that students encounter is creating collaborative objects – students have difficulty creating and understanding connections between objects" (p. 182). Ragonis and Ben-Ari (2005a) have conducted a longitudinal study among high school students learning object-oriented programming; they conclude, "students find it hard to create a general picture of the execution of a program that solves a problem" (p. 214).

The structures (see p. 35) are related to the programming paradigm, e.g. in the imperative paradigm we could teach the general sweep algorithm (see Figure 5) in order to teach a student how to find one or many elements in a collection fulfilling a given criteria:

```
result ← <empty>
do <collection is not empty>
    <select element>
    if <element fulfils criteria>
        <add element to result>
    fi
od
return result
```

Figure 5: General find-all sweep algorithm

In the object-oriented paradigm, the class model describes the static structure of the solution (the conceptual modelling perspective described by Knudsen and Madsen (1988)). Consequently, we need to teach systematic ways to implement class models (see figure 6 for an example) as well as standard solutions for implementation of methods (e.g. iteration through a collection of objects in order to find all that fulfils a given criteria – the find-all sweep algorithm or the sweep algorithm from figure 6).



```
class A {
        private T a1;
        public A(T a1) {
          this.a1 = a1;
        }
        public T3 m1(T2 param) {
          …
        }
}
```

Figure 6 Coding pattern for a single class

As described above, the programming process is not a linear one with a fixed set of rules. Robillard (2005) describes two creative mental activities: systematic and opportunistic. A set of cognitive actions is *systematic* when all the knowledge required to complete a task is available making it possible to follow a well-structured plan. A set of cognitive actions is *opportunistic* when further exploration is necessary. He argues that “any engineering activity that involves some creativity will result in a mixture of opportunistic and systematic approaches.” (p. 63). It is therefore necessary that we teach the students how to handle these different approaches and that a “golden road” from problem to final program does not exists. Soloway (1986) noted back in 1986 that

> Expert programmers know a great deal more than just the syntax and semantics of language constructs […] They have built up large libraries of stereotypical solutions to problems as well as strategies for coordinating and composing them. Students should be taught explicitly about these libraries and strategies for using them. (p. 850)

Experts are not necessarily conscious of the knowledge and strategies they employ to solve a problem, write a program, etc. (Soloway, 1986). Often, experts refer to “intuition, gut feel, etc.” as the sources of their inspiration. This is the same in related fields (e.g. mathematics (Michener, 1978; Sweller, Mawer, & Ward, 1983) and physics (Larkin, McDermott, Simon, & Simon, 1980; Larkin, McDermott, Simon, & Simon, 1980)) and has lead to ways to handle this in teaching: cognitive apprenticeship (Collins, Brown, & Newman, 1989; Collins, Brown, & Holum, 1991). For a more elaborated discussion on cognitive apprenticeship, see section 4.6.

Consequently, we need to adapt a way of teaching that can disclose these strategies to the students – in other words we need to teach programming as both a verb and a noun rather than just as a noun.

2.4 My Personal Background for this Research

I have taught introductory programming for many years (since 1988) and have always found it to be challenging. This interest has naturally led me to the focus of this dissertation.

When I graduated in Computer Science from the University of Aarhus, I started to teach at the Advanced Computer Studies at Aarhus Business College (Advanced Computer Studies, n.d.). It is a practice-oriented, full-time educational study lasting two years and three months. It is divided into five semesters consisting of a combination of IT-disciplines and a number of commercial subjects. The first four semesters are subject-oriented and in the fifth semester the main dissertation is elaborated in close co-operation with a company in Denmark or abroad.

When I started, it was at a time where studying computer science was popular and many students enrolled; many of the students fund the relatively short study attractive compared to a more traditional university masters degree (in Denmark it was – and still is – rather uncommon to stop studying after a bachelors degree). However, many of the students dropped out. One of the major factors to this was that they (also) found programming to be very difficult. As a consequence, Michael Caspersen and I analysed the problems we could observe and the foundation for the approach described and researched in this dissertation was laid. It is described in Bennedsen and Caspersen (1995). We found the following problems (back then we used Pascal (Jensen, Wirth, Mickel, & Miner, 1991) as the programming language):

> **Algorithmic patterns**. We would like to focus on algorithmic patterns and techniques, but used the first 75% of the course teaching about the programming language.
>
> **Details**. We bombarded the students with language details (all the types in the language, pointers, all the possible ways to do iteration etc.). The students lost sight of the wood for all the trees.
>
> **Abstraction ability**. Almost all of our exercises started from scratch; the students should program everything. The students did not get used to use pre-programmed abstractions (modules).
>
> **Closed/open exercises**. Many exercises required too much work before the students got to the point where they could test their solutions. Furthermore, the exercises did not motivate the students do continue working on them.
>
> **Role model**. We expected that the students could write a program after they had seen one or two. The students need to read a lot of well-designed programs before they write their own.
>
> **Motivation**. Many exercises were uninteresting. When exercises start from scratch, the ambition level is limited and consequently the problems that students can solve do not show the power of computer applications.

The model-based approach described and analysed in this dissertation addresses all of the above problems and proposes solutions for them. Naturally, the concrete instructional design depends on many factors, e.g. the factors described by Hiim and Hippe (2006) – see section 4.1. Young liberal arts students who have to learn about programming to have an understanding of the problems that one can solve using programming or experienced imperative programmers who should learn to program object-oriented have different learning premises and different learning objectives. Ten students in a class is different than 300 in a lecture hall and consequently requires a different didactical design.

2.5 Project COOL

The Comprehensive Object-Oriented Learning project (Berge et al., 2003; COOL, n.d.; Nygaard, 2002) started in October 2002 and concluded at the end of 2005. Its main partners were InterMedia and the Department of Informatics, both at the University of Oslo; the Norwegian Computing Center; and Simula Research Laboratory. I was associated with the project and have conducted research with several of its participants ("Programming in Context: a Model-First Approach to CS1"; "Examining social interaction patterns for online apprenticeship learning – Object-oriented programming as the knowledge domain"; Bennedsen, 2006a; Fjuk, Holmboe, Jahreie, & Bennedsen, 2006; Fjuk, Berge, Bennedsen, & Caspersen 2004). The project was initiated by Kristen Nygaard[12]. With its foundation in the Scandinavian approach to object-orientation (Knudsen & Madsen, 1988), the project has been concerned with exploring challenges met in learning this paradigm and its corresponding concepts. Some of the project's findings were published in the anthology "Comprehensive Object-Oriented Learning: The Learner's Perspective" (Fjuk, Karahasanovic, & Kaasbøll, 2006).

Project COOL has explored the challenges of learning object-orientation by studying relationships between tools and programming environments, types of learners, pedagogical approaches and learning strategies, learning resources and ICTs. The research has been carried out by conducting a number of case studies, design experiments and controlled experiments (Fjuk, Karahasanovic, & Kaasbøll, 2006). My contributions to the project are concerned with teaching and learning introductory object-oriented programming, with a particular interest in the model-based approach.

[12] Kristen Nygaard passed away just a few months before the start of the project. A memorial site is located at http://www.ifi.uio.no/in_memoriam_kristen/

3 Computer Science Education Research

There is nothing like looking, if you want to find something.
You certainly usually find something, if you look,
but it is not always quite the something you were after
J.R.R. Tolkien (1892-1973), “The Lord of the Rings”

This chapter describes the field of computer science education research, an emerging field since computer science itself is a rather novel field. The chapter gives a general, historical description of the field, a general introduction to research in introductory programming, followed by a more thorough description of the specific subfields: research in teaching introductory object-oriented programming, research in novices’ programming process and research in success factors for learning introductory programming. Then follows a description of research in the area of pedagogical patterns. A more elaborate description of the field of computer science education research can be found in Détienne (2002) plus Fincher and Petre (2004).

3.1 Historic Overview of Computer Science Education Research

Computer science is a novel field, and so is computer science education research. As Fincher and Petre (2004) note:

> Computer science (CS) education research is an emergent area and is still giving rise to literature.
>
> While scholarly and scientific publishing goes back to Philosophical Transaction (first published by the Royal Society in England in 1665), one of our oldest dedicated journals, Computer Science Education was established less than two decades ago, in 1988 (p. 1).

The researchers in the field naturally come from the field of computer science, but also from established fields such as education, psychology, engineering and technology. According to Holmboe (2005), most of the field’s work emerged from one of three fields: cognitive psychology, computer science teaching and human computer interaction.

The main forums for computer science education research are the annual American-based conference on computer science education (*Technical Symposium on Computer Science Education*[13]**,** SIGCSE) and its European counterpart, Innovation and Technology in Computer Science Education (ITiCSE). SIGCSE will be held for the 39th time in 2008, and ITiCSE will be held for the 13th time in 2008. Most of the publications within the field of computer science education research are "practitioner reports" (Carbone & Kaasbøll, 1998; Fincher & Petre, 2004; Holmboe, 2005, p. 15), but a current change is marked by more research-based publications. ACM is hosting a conference aimed specifically at Computer Science Education Research (International Computing Education Research Workshop[14], ICER), to be held for the fourth time in 2008.

In the late sixties and early seventies, a special interest in programming as a domain to study cognition emerged. Many consider the book, "The Psychology of Computer Programming," by Gerald M. Weinberg in 1971, to be the first book within the field (it was republished in 1998 in a silver anniversary edition (Weinberg, 1998)). In the book's preface he writes, "This book has only one major purpose – to trigger the beginning of a new field of study: computer programming as a human activity" (p. vii). In the seventies, programming cognition became an active field with a focus on how expert programmers differ from novices. One of the major contributions was Brooks (1977), who described a theory for studying programming. The theory "postulates understanding, method-finding, and coding processes in writing programs, and presents an explicit model for the coding process" (p. 737). The article was reprinted in the 30th anniversary issue in 1999; an issue that reprinted the most cited articles from International Journal of Human-Computer Studies (formerly International Journal of Man-Machine Studies). Several overviews of this early research have been published (e.g., Détienne (2002); Robins et al. (2003) and Winslow (1996)). However, Sheil (1981) noted: "As a body of psychological studies, the research on programming is very week methodologically" (p. 112). This is also noted by Weinberg, (1971): "it has not

[13] ACM names its conferences based on the expected number of attendants. A technical symposium is a conference with 100-300 participants expected – even though more than 1200 attended in 2006. It is a conference with peer-reviewed papers.

[14] ACM names conferences with less than 100 expected participants 'workshops' – ICER is a conference with peer reviewed papers.

been possible to support certain ideas with 'scientific' evidence.'" (p. vii). Brooks (1977) has a cognitive psychology perspective in his research, where he uses theories of long- and short-term memory to analyse the behaviour of expert programmers when programming. He is one of the exceptions to Sheil's comments.

In the following decade (the eighties,) much knowledge in the field of expert programmers was amassed. Some of those findings are presented in (Hoc, 1990; Soloway & Spohrer, 1989). von Mayrhauser and Vans (1994) sum up much research and describe the characteristics of an expert programmer as follows:

> Experts organize knowledge structures by functional characteristics of the domain in which they are experts. Knowledge possessed by novices is typically organized by surface features of the problem. For instance, novices may have knowledge about a particular program organized according to the program syntax. An example of a functional category is algorithms. Experts may organize knowledge about programs in terms of the algorithms applied rather than the syntax used to implement the program.
>
> Experts have efficiently organized specialized schemas developed through experience. Experts not only used general problem solving strategies such as divide-and-conquer, but also more specialized design schemas. These schemas differed in granularity and seemed to be abstracted from previously designed software systems.
>
> Experts are flexible in approaches to problem comprehension. In addition, experts are able to let go of questionable hypotheses and assumptions more easily than novices. Experts tend to generate a breadth-first view of the program and then refine hypotheses as more information becomes available (p. 5-6).

Winslow (1996) sums up studies on novices. They

- lack an adequate mental model of the area (Keesler & Anderson, 1989),
- are limited to a surface knowledge of the subject,
- have fragile knowledge (something the student knows but fails to use when necessary) (Perkins & Martin, 1986)
- use general problem solving strategies (i.e., copy a similar solution or work backward from the goal to determine the solution) rather than strategies dependent on the particular problem,

- tend to approach programming through control structures, and
- use a line-by-line, bottom up approach to problem solving (Anderson, 1985)

(p. 18)

In the nineties, the field of computer science education research broadened to include more areas such as visualization, tools, and ethics.

In the new millennium the field has matured even more. Conferences have been established that focus on computer science education research (*ICER*, 2006) in a broad sense (as opposed to the more specialized field of programming, where there has been a long series of conferences in the psychology of programming interest group (*Psychology of programming interest group*, n.d.)), and methodological books are beginning to appear (e.g., Fincher and Petre (2004)).

Several methodological reviews have been made in order to characterize the field of computer science education research (Randolph et al., 2005; Randolph, Bednarik, & Myller, 2005; Simon, 2007b; Simon, 2007a; Valentine, 2004).

Valentine (2004) analysed 444 entities (articles, workshop reports and panels) dealing with traditional CS1/2 topics from the SIGCSE conferences from 1984 to 2003. He used a six-fold taxonomy to classify the type of articles: Experimental (the author made an attempt to assess the "treatment" with some scientific analysis), Marco Polo ("I went there and I saw this"), Philosophy ("the author made an attempt to generate debate on an issue, on philosophical grounds, among the broader community"), Tools ("things helping education – not all software tools, but also rubrics"), Nifty ("small elements used in teaching such as nifty assignments") and John Henry[15] ("every now and then a colleague describes a course that seems so outrageously difficult (in my opinion) that it is suspected of telling us more about the author than about its pedagogy"). The general conclusions from his research were:

[15] John Henry was an American worker who worked as a steel-driver for the Chesapeake & Ohio Railroad. The folktale says that he was the strongest and fastest worker who had ever drilled holes using a hammer and a steel spike. One day a salesman came with a steam-powered drill and claimed that it could do better than any man. John Henry competed, won, but died due to strain (American Folklore, 2006).

- 21% of the articles presented (94 out of 444) fell into the experimental category (i.e., a kind of research article).
- The portion of Experimental articles has been on the rise since the mid-nineties.
- The Marco Polo articles dropped from approximately 35% in 1984 to 19% in 2003 (a statistically significant drop).
- The total number of entities presented increased during the period.
- The percentage of first year entities remained almost the same throughout the period.

However, as noticed by Randolph, Bednarik, and Myller (2005),

> Valentine's findings, however, should be accepted with a fair amount of scepticism because the methodology used to conduct Valentine's review was lacking (e.g., there were no estimates of reliability about his categorizations) (p. 104).

Randolph et al. (2005) analysed the 59 articles from the first four Koli Calling Conferences (Korhonen & Malmi, 2004; Kuittinen, 2001; Kurhila, 2003; Sutinen, 2002).

> It was found that the majority of articles did not report research involving human participants. Experimental/quasi-experimental and exploratory descriptive were the two most commonly used methodology categories; the post-test-only with controls design was the most commonly used experimental design. (p. 108)

Again, the majority of the articles were not what traditionally is seen as research articles. "By far, the most frequently published type of paper in the Koli proceedings are pure program (project) descriptions" (Randolph et al., 2005, p. 107). This observation is supported by Carbone and Kaasbøll (1998). Randolph et al. reports on inter-rater reliability for the classification of articles. They also analysed the content of the articles: structure and amount of specific parts. The amount was measured by inches of text, a debateable measurement (e.g., a literature review is not necessarily better the longer it is, it could be that it just is wordier.)

Simon (2007b) analysed research articles presented at the Koli proceedings from 2001 to 2006. He categorised the articles according to four article categories: experiment ("reports on a clear and deliberate research experiment"), analysis ("authors have set out to answer a particular question, gathered existing data as appropriate, and analysed it"), report ("describing something that has been tried or developed in an educational context") and position ("elucidates the authors' thoughts on a matter, or perhaps sets out plans for future work"). He concludes that there has been a growth in what he classifies as research articles (experiment and analysis) from less than 10% to more than 40%. Simon developed the categorization when he categorized articles from four Australian and three New Zeeland based conferences (Simon, 2007a).

In general, studies in computer science are small-scale studies within the context of one course. Pears, Seidman, Eney, Kinnunen, and Malmi. (2005) have made a taxonomy with four areas in order to classify computer science education research articles. In an ITiCSE working-group report (Pears et al., 2007) applying the taxonomy in order to create a body of research literature to inform instructors when designing a new introductory programming course, the working group found that

> The majority of studies reviewed were located in area A of the Lisbon taxonomy table [(Pears et al., 2005)]. The concentration of small-scale studies is understandable, since it is easy for practitioners to study their own instructional settings. However, it is difficult to aggregate results from such papers to produce well-supported arguments for particular approaches to teaching. A more consistent approach to reporting the results of individual studies would facilitate generalizability. (p. 2)

The ITiCSE working group (Pears et al., 2007) produced a list of 45 articles after having considered 179 articles. They found the citation index for each article using http://scholar.google.com. The average citation index of the articles was 18.7, the mean 9. The average age of the articles (i.e. time since the article was published) was 6.8 years.

Overall, we conclude that computer science education research is an emerging field without a well-established research method. The field draws on other fields such as education and computer science teaching, and it is therefore relevant to look to these fields for inspiration on research methods. Chapter 5 will discuss the research method used in this PhD work.

3.2 Research on Teaching Introductory Programming

Fincher and Petre (2004) map the area of computer science education research in the following ten areas:

> student understanding, animation/visualization/simulation systems, teaching methods, assessment, educational technology, transfer of professional practice into the classroom, incorporation of new development and new technologies into the classroom, transferring to remote teaching ("e-learning"), recruitment and retention of students, and, finally, the construction of the discipline itself. (p. 3)

Even if we restrict ourselves to research of teaching introductory programming, we will find much research in all ten areas. Consequently, it seems almost impossible to give an overview of the entire area. In the following section (3.2.1) we will focus on research in the objects-first approach to teaching introductory programming.

Many approaches to teaching introductory programming have been proposed including a procedures early approach (Pattis, 1993), a top-down approach (Hilburn, 1993; Reek, 1995), a graphics approach (Matzko & Davis, 2006). Even within object-oriented introductory programming, many different approaches exist: objects early (Alphonce & Ventura, 2002), inheritance early (Schmolitzky, 2004), GUIs early (Wolz & Koffman, 2000), concurrency early (Reges, 2000), events early (Stein, 1998), components early (Howe, Thornton, & Weide, 2004b), etc.

All of these articles about teaching introductory programming describe different peoples' approaches. However, many are in the "Marco Polo" style of reporting research in introductory programming, or to be more precise they are arguing that a certain approach is better than other approaches based on the assumption that certain learning outcomes should be promoted.

Berglund, Daniels and Pears (2006) argues that "conducting studies which are theoretically anchored in pedagogy, as well as in computing, can help us to draw more solid and significant conclusions about how students learn computing." (p. 25). The section on learning theories will discuss and argue for the pedagogy applied in the courses used as empirical basis for this dissertation. However, using real-world observations (as opposed to controlled experiments) for studying programming makes this somewhat difficult since it is very difficult to characterize the pedagogy in a course solely by one pedagogical theory (Gilmore, 1990). As Biggs (2003) note in his book "Teaching for Quality Learning at University: what the student does":

> I assume that most teachers, including readers of this book, are not particularly interested in theories of learning so much as in improving their teaching. For that we need a framework to aid reflection: a theory of learning that is broad-based and empirically sound, and that easily transfers into practice. To my mind that means constructivism[16], with its emphasis on what students have to do, rather than on how they represent knowledge. Both emphasize that the student creates knowledge […] so that knowledge is not imposed or transmitted by direct instruction. (pp. 12-13)

Section 4 will describe pedagogical theories in general and the theories applied in the courses under study in particular.

3.2.1 Research on Teaching and Learning Introductory Object-Oriented Programming

The objects-first approach to teaching object-orientation has gained much attention in recent years.

> Indeed one need only look to modern object-oriented languages such as C++, Java, and Microsoft's C# as proof of OOP's success. Since the mid-1990s, there has been considerable attention given to teaching object-oriented programming early. (Ventura, 2003, p. 2)

Programming languages have always elicited debate in the teaching community. The current dominant programming language is Java (Dale, 2005; Raadt et al., 2002; Schulte & Bennedsen, 2006). Because it is impossible in Java to create methods that are not associated with a class, it is implied that teachers using Java have been forced to rethink their curriculum (Bergin, Koffman, Proulx, Rasala, &

[16] Biggs have a broad definition of constructivism: it is theories where the learner has to *do* to create knowledge. (J. B. Biggs, 2003, p. 12)

Wolz, 1999; Culwin, 1999; Kölling & Rosenberg, 2001; Mitchell, 2000; Weber-Wulff, 2000).

Teaching introductory object-oriented programming is considered difficult. Kölling and Rosenberg (2002) believe

> that this is not due to any intrinsic complexity associated with OO, but is caused by a set of other factors which includes:
>
> - A lack of experience and knowledge about how to teach these courses well, which results in mistakes being made.
> - Many teachers are not as familiar with object orientation as they are with the procedural language they taught earlier and have technical difficulties.
> - The teaching materials are not as mature as they were for earlier languages. Many books were produced very quickly with clear compromises in quality. In addition, the authors often do not have the necessary teaching experience in this new paradigm themselves.
> - The software tools are often not suited to the new paradigm. Many development environments are variations of procedural environments that fail to capture the full potential of object orientation.
> - In conjunction with the introduction of object orientation into first year courses there has been a shift in focus from an algorithms centred view to a software engineering centred view. This introduces new topics and new material into the course.
> - Technological advances have forced additional material into the course. (pp. 1-2)

Kölling (2003) stressed his point by analysing 39 major selling textbooks on introductory programming. The overall conclusion of his survey was that all books are structured according to the programming language's language constructs, not by the programming techniques that should be taught to the students.

In 1994 Decker and Hirshfield (1994) described what they found to be the top 10 reasons for not teaching object-orientation in an introductory programming course. Their conclusion supports the conclusions from Kölling and Rosenberg.

Since the computing curriculum's publication in 2001, a lively discussion has arisen on the idea of "objects-first." Back in March 2004 a dynamic discussion took place on the SIGCSE mailing list (SIGCSE-members, 2005). For a summary of this discussion, see Bruce (2005). The discussion was analyzed in order to explore "how the CS community debates the issue and whether contributors' positions are supported by the research literature on novice programmers" (Lister et al., 2006, p. 146). At SIGCSE 2005, a discussion entitled "Resolved: Objects Early Has Failed" (Astrachan et al., 2005), and at the Annual Consortium for Computing Sciences in Colleges sponsored CCSC: Eastern Conference in 2003, a panel discussion entitled "Objects first - does it work?" (Bailie et al., 2003) took place. In the fall of 2006, Michael Kölling reopened the debate with his "I Object" contribution to the SIGCSE mailing-list.

Objects-first is defined only vaguely in CC2001: "The objects-first model also focuses on programming, but emphasizes the principles of object-oriented programming and design from the very beginning […] The first course in either sequence begins immediately with the notions of objects and inheritance, giving students early exposure to these ideas. After experimenting with these ideas in the context of simple interactive programs, the course then goes on to introduce more traditional control structures, but always in the context of an overarching focus on object-oriented design" (p. 30). This is in concordance with the notes made by Lewis (2000) who note that the term objects-first is

> often used to convey the general idea that objects are discussed early in the course and established as a fundamental concept. Beyond that, however, these phrases seem to take on a variety of meanings, with important implications (p. 246).

In order to get a better understanding of the term objects-first, Bennedsen and Schulte (2007) did a content analysis (Stemler, 2001) of over 200 responses from teachers around the world. The data was the respondents from the study in Schulte and Bennedsen (2006). Three different categories of the concept were found:

> **Using objects**: At the beginning of the course, the student uses objects implemented beforehand. When students have understood the object concept, they move on to defining classes by themselves. Focus is on usage before implementation.

> **Creating classes**: Student both defines and implements classes and creates instances of the defined classes. Focus is on the concrete-creative part of programming.
>
> **Concepts**: Students learn about general principles and ideas of the object-oriented paradigm, not just focusing on programming but on creating object-oriented models in general. Focus is on the conceptual aspects of object-orientation.

The respondents were categorized into these three categories. Thirty-two percent (75 in total) of the descriptions were too short to categorize. Moreover, many of the descriptions defined objects-first as “something else” or in opposition to procedural-first–indicating a focus on object-orientation but not necessarily an objects-first approach: “Starting with object-oriented concepts rather than procedural concepts.” Fifty fell in the category ‘other’. They found that most (respondents) use a *creating classes*’ approach (51), followed by 34 who use *concepts* and 28 *using objects*.

From the categorization of the responses it was possible to check if there were any differences in the three groups’ evaluation of the relevance, difficulty, or teaching level of topics in an introductory programming course. The important implications (indicated by Lewis (2000) in the above quote) of these three categories are not prominently apparent in the list of topics for an introductory programming course. Further research is needed to establish those implications. They could be in the use of different teaching methods or the sequence of topics taught.

Many articles have described different authors’ experiences with an objects-first approach. There have been workshops at computer science conferences, e.g., Bruce, Danyluk, and Murtagh (2001b) and even panels discussing how to start an objects-first course ("Day One of the Objects-First First Course: What to Do” (Bergin, Clancy, Slater, Goldweber, & Levine, 2007)). In Table 3 there is a short description of different authors’ approaches to objects-first. The selection criteria were that the article describes (parts of) a way to teach an objects-first CS1 course. Articles with a focus on conceptual modelling is excluded from the overview, but discussed later in contrast to research question R1. The table is by no means complete but serves the purpose of showing different authors views on what is relevant in CS1.

<table>
<tr><th>Author(s)</th><th>Article</th><th>Language</th></tr>
<tr><td>Adams (1996)
Adams and Frens (2003)</td><td>Object-centered design: a five-phase introduction to object-oriented programming in CS1–2
Object centered design for Java: teaching OOD in CS-1</td><td>Java (originally C++)</td></tr>
<tr><td colspan="3">They claim that design is an important skill and consequently it must be taught in CS1. They describe three phases of a design process: 1) Use only objects. Find objects and methods by noun/verb inspection. 2) Classes and methods. Abstract over the nouns to create concepts (=classes) if they can not be represented by a primitive type. 3) Inheritance. If two or more classes have common attributes, consolidate those attributes into a super class.</td></tr>
<tr><td>Woodworth and Dann (1999)</td><td>Integrating console and event-driven models in CS1</td><td>C++</td></tr>
<tr><td colspan="3">They find that it is important for the students to know of event-driven programs. They focus on three elements: 1) fundamentals of problem-solving and control structures, 2) students should design their own abstractions, and 3) emphasizing two types of functions: class methods and library functions</td></tr>
<tr><td>Culwin (1999)</td><td>Object imperatives!</td><td></td></tr>
<tr><td colspan="3">Culwin state that the initial educational experience in computer science will have a major impact on the students' subsequent progression. Based on this, he argues that courses needs to be rooted in current concerns and practice in order to prepare the students for a professional life. He then discusses how this influences the curriculum including a discussion on objects-first vs. objects-late.</td></tr>
<tr><td>Buck and Stucki (2000)</td><td>Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development</td><td>Java</td></tr>
</table>

<table>
<tr><td colspan="3">Buck and Stucki propose a structuring of a CS1 course based on Bloom’s taxonomy. They argue that the traditional system development life cycle (analysis – design – coding) starts with competences at the highest level and moves downwards. As a consequence, they argue that students should start by modifying code and later design (parts of) a system. The argue for an approach where the language constructs are introduces as needed</td></tr>
<tr><td>Proulx (2000)</td><td>Programming patterns and design patterns in the introductory computer science course</td><td></td></tr>
<tr><td colspan="3">Proulx have developed a pattern catalogue aimed at helping novices learning to program. It includes patterns like “Name Use Pattern: Every programming language uses 'identifiers' to represent different kinds of entities in a program. Yet the use of all identifiers follows a set pattern: declare - define/build – use - destroy.” (p. 81)
She introduces programming via key concepts and the relevant patterns. Students read the patterns, see several completely solved problems related to the pattern and gets a number of unsolved problems (including a suggestion for a problem solving strategy)</td></tr>
<tr><td>Cooper, Dann, and Pausch (2000; 2003a; 2003b)
Moskal, Lurie, and Cooper (2004)</td><td>Teaching objects-first in introductory computer science.
Using Animated 3D Graphics to Prepare Novices for CS1.
Alice: a 3-D tool for introductory programming concepts
Evaluating the effectiveness of a new instructional approach.</td><td>Alice-lingo</td></tr>
<tr><td colspan="3">They describe their use of a 3D micro-world (Alice) to teach programming. They emphasis design and a low number of details the student must master.</td></tr>
</table>

<table>
<tr><td colspan="3">Their argumentation for Alice is: 1) Working with an easy-to-use 3D graphics environment is attractive and highly motivating to today’s generation of media-conscious students. 2) The visual nature and immediate feedback of program visualizations makes it easy for students to see the impact of a statement or group of statements. Further, it makes debugging easier. 3) The drag-and-drop editor prevents students from making syntax errors that are prevalent for beginners. 4) The 3D modelled classes and instantiated objects in Alice provide a very concrete notion of the concept of an object and support an objects-first approach. (Moskal et al., 2004, p. 76)</td></tr>
<tr><td>Wick (2001)</td><td>Kaleidoscope: using design patterns in CS1</td><td></td></tr>
<tr><td colspan="3">Wick finds that design patterns are an essential competence of a professional programmer and therefore it must be included in CS1. Given this, he presents a program - kaleidoscope – including several design patterns and a way to use this in CS1 (including a proposed sequence of topics).</td></tr>
<tr><td>Nguyen and Wong (2001)</td><td>OOP in introductory CS: Better students through abstraction</td><td></td></tr>
<tr><td colspan="3">Nguyen and Wong argue for an “objects-first-with-design-patterns” introductory sequence. Thereby, according to them, the students train abstract thinking. They do this by teaching the students to abstractly decompose problems and express their ideas in terms of abstract object models</td></tr>
<tr><td>Bruce, Danyluk, and Murtagh (2001a; 2001b; 2001c; 2004)</td><td>Event-driven programming facilitates learning standard programming concepts.
Event-driven programming is simple enough for CS1.
Events and objects first: an innovative approach to teaching JAVA in CS 1
A library to support a graphics-based objects-first approach to CS 1</td><td>Java</td></tr>
<tr><td colspan="3">They make a case for an event driven approach to CS1 using their own library where the students inspects code before they write it on their own.
The implemented graphics library wraps Java’s AWT framework, thereby making</td></tr>
</table>

<table>
<tr><td colspan="3">it possible for beginning students to use it.
They have found that the event-driven approach familiarize students with object-oriented concepts.</td></tr>
<tr><td>Kölling and Rosenberg (2001)</td><td>Guidelines for teaching object orientation with Java</td><td>Java</td></tr>
<tr><td colspan="3">Kölling and Rosenberg describes a set of guidelines for teaching object-oriented programming: Start with objects (introduce objects from the beginning and not a small imperative style program), don’t start with a blank screen (the students shall start by making small changes to existing code, not by creating code themselves), read code (students should read well written code and mimic the style and idioms), use “large” projects (the benefits of object-orientation is best seen in large projects where the syntactic overhead is minor), don’t start with main (the main method has nothing to do with object-orientation), don’t use “hello world” (hello world is not an object-oriented program), show program structure (the classes and their structure are a central issue in object-oriented programming, learn the students who this can be expressed), be careful with the user interface (making a nice interface is time consuming). They discuss how to address the guidelines using BlueJ.</td></tr>
<tr><td>Becker (2001)</td><td>Teaching CS1 with Karel the robot in Java</td><td>Java</td></tr>
<tr><td colspan="3">Becker describes the progression of the first five weeks of their course 1. Instantiating and using objects, 2. Extending existing classes, 3. Selection and iteration, 4. Methods and parameters, and 5. Instance variables. He furthermore describes how a concrete micro-world (Karel the robot) is used in such a sequence.</td></tr>
<tr><td>Lawhead et al. (2002)</td><td>A road map for teaching introductory programming using LEGO© mindstorms robots</td><td>Java</td></tr>
<tr><td colspan="3">The authors make a case for the use of robots in a CS1 course. The robot exhibits physical behaviour, the objects representing the model in a natural way has state, and the state change can directly be observed. Inheritance can also be illustrated by extending models of robots.
They present several examples of exercises and give tips to get started.</td></tr>
</table>

<table>
<tr><td>Barnes (2002)</td><td>Teaching introductory Java through LEGO MINDSTORMS models</td><td>Java</td></tr>
<tr><td colspan="3">Barnes argues that the use of LEGO MINDSTORMS can motivate students. He discusses how MINDSTORMS can be used in an introductory course and addresses some problems: polling a sensor vs. adding an event listener, the correspondence between the physical model and the program and the physical limitations of the MINDSTORMS brick.</td></tr>
<tr><td>Proulx, Raab, and Rasala (2002)</td><td>Objects from the beginning - with GUIs</td><td>Java</td></tr>
<tr><td colspan="3">Proulx, Raab and Rasala describe the use of GUI's as an introduction to object-oriented programming. In the first lab, the students use a picture manipulation application and change the state of the pictures. Later the students inspect the code for the picture class. Their second lab is a turtle micro world where the turtle can be manipulated using traditional Java syntax. Their third lab, the students adds functionality to a ticket selling machine. In the fourth lab, the students implement a subclass of the picture class presented in lab one.</td></tr>
<tr><td>Christensen and Caspersen (2002)</td><td>Frameworks in CS1: a different way of introducing event-driven programming</td><td>Java</td></tr>
<tr><td colspan="3">Christensen and Caspersen argue that today's programmers reuse code as much as they produce code. One goal of a CS1 course must consequently be that students get used to reuse and extend code. They present a concrete framework used in their teaching.</td></tr>
<tr><td>Roumani (2002)</td><td>Design guidelines for the lab component of objects-first CS1.</td><td>Java</td></tr>
<tr><td colspan="3">This article describes not an entire CS1 course but the lab component. Roumani put forward that labs must be seen as an integrated component in the students learning.</td></tr>
</table>

<table>
<tr><td>Nevison and Wells (2003; 2004)</td><td>Teaching objects early and design patterns in Java using case studies
Using a maze case study to teach: object-oriented programming and design patterns</td><td>Java</td></tr>
<tr><td colspan="3">They suggest an approach to teaching based on a case study. In the beginning, simple versions of the case study can be used for the first concepts and increase the complexity of the case study during the course. An important characteristic of the case study is that is must use a graphical and interactive presentation. They use design patterns as an organizing principle. They describe two concrete cases: an elevator system and a maze.</td></tr>
<tr><td>Towell and Towell (2003)</td><td>Reality abstraction and OO pedagogy: results from 5 weeks in virtual reality</td><td>MOO</td></tr>
<tr><td colspan="3">Towell and Towell describe a five week introductory sequence where the students are exposed to an OO virtual environment. The students explored the environment as avatars sending messages to other objects (other avatars, rooms etc). They find that the approach helps the students understand abstraction, inheritance and method override but not state.</td></tr>
<tr><td>Bruhn and Burton (2003)</td><td>An approach to teaching Java using computers</td><td>Java</td></tr>
<tr><td colspan="3">Bruhn and Burton describes an approach to CS1 using studio teaching: the students have their own laptop and are able to test and run their code in class. The instructor write on his computer (which is projected to a large screen) and the students can use their computers as well.</td></tr>
<tr><td>Guzdial (2003)
Guzdial and Forte (2005)</td><td>A media computation course for non-majors
Design process for a non-majors computing course</td><td>Python</td></tr>
<tr><td colspan="3">Guzdial discusses the challenges related to the broader student intake (students majoring in other areas than computer science). They describe and discuss the design of a course that uses computation for communication as a guiding princi-</td></tr>
</table>

<table>
<tr><td colspan="3">ple, where the students manipulate (writing programs for this manipulation) digital media.</td></tr>
<tr><td>Edwards (2003a; 2003b)</td><td>Rethinking computer science education from a test-first perspective
Improving student performance by evaluating how well students test their own programs</td><td>Java</td></tr>
<tr><td colspan="3">He argues that students think a program is finished when it compiles. He suggests a test-driven strategy to all assignments. The students write then on their own tests. The test and coding style is a part of the grading.</td></tr>
<tr><td>Schmolitzky (2004; 2006)</td><td>"Objects first, interfaces next" or interfaces before inheritance
Teaching inheritance concepts with Java</td><td>Java</td></tr>
<tr><td colspan="3">Schmolitzky distinguish between hierarchies of types (interfaces in Java) and hierarchies of implementation (classes in Java). He argues that that it is important to use this as a way to differentiate “what” from “how”
The students see this difference from day one. In the beginning it is supported by the use of javadoc vs. implementation; later the students are taught the interface concept in Java explicitly.</td></tr>
<tr><td>Howe, Thornton, and Weide (2004a)</td><td>Components-first approaches to CS1/CS2: principles and practice</td><td>C++/Java</td></tr>
<tr><td colspan="3">The authors argue that the availability of software components significantly affects how professionals think about building their software. Students might benefit from learning this early in the CS curriculum.
They argue that it is vital for the students to be more able to distinguish between the interface and the implementation of a concept, and argues that traditional objects-first courses ignore this difference. Loop invariants are another difference – they are introduced early in the component-based approach (as opposed to traditional objects-first courses)</td></tr>
</table>

<table>
<tr><td>Bierre, Ventura, Phelps, and Egert (2006)
Bierre and. Phelps (2004)
Egert, Bierre, Phelps, and Ventura (2006)</td><td>Motivating OOP by blowing things up: an exercise in cooperation and competition in an introductory java programming course
The use of MUPPETS in an introductory java programming course
Hello, M.U.P.P.E.T.S.: using a 3D collaborative virtual environment to motivate fundamental object-oriented learning</td><td>Java</td></tr>
<tr><td colspan="3">Bierre et. al. claim that students find traditional programming assignments as toy problems not matching the graphics-rich, interactive notion of programming students have on beforehand. They propose the use of a micro-world (TankBrains) implemented in their MUPPETS system (Multi-User Programming Pedagogy for Enhancing Traditional Study).</td></tr>
<tr><td>Ventura, Egert, and Decker (2004)</td><td>Ancestor worship in CS1: on the primacy of arrays</td><td>Java</td></tr>
<tr><td colspan="3">They make a case that the introduction of object-oriented programming implies that the students should use some Collection class rather than arrays</td></tr>
<tr><td>Leska and Rabung (2005)</td><td>Refactoring the CS1 course</td><td>Java</td></tr>
<tr><td colspan="3">They use games as a way of motivating the students. They claim that it is important to start with design without coding and gradually include coding. The programming language features are included by “the need of” game projects.</td></tr>
<tr><td>Xinogalos, Satratzemi, and Dagdilelis (2006)</td><td>An introduction to object-oriented programming with a didactic microworld: <i>objectKarel</i></td><td>Karel++</td></tr>
<tr><td colspan="3">The authors argue that the use of objectKarel overcomes some of the difficulty traditionally found when teaching using a traditional programming language: 1) The extended instruction set of programming languages. Karel++ is a small language. 2) The notional machine. objectKarel animates the program execution via robots. 3) Students focus on the syntactic details. ObjectKarel uses a structure</td></tr>
</table>

<table>
<tr><td colspan="3">editor.
ObjectKarel includes small lessons on given subjects and problems the student can solve</td></tr>
<tr><td>Roumani (2006)</td><td>Practice what you preach: full separation of concerns in CS1/CS2</td><td></td></tr>
<tr><td colspan="3">Roumani believes that the main problem in CS1 stems from abandoning a long held principle in computer science: separation of concerns. CS1 should be taught from a client perspective. This means we only write main programs that use existing components. In the article, he presents his order of topics.</td></tr>
<tr><td>Pecinovský, Pavlíčková, and Pavlíček (2006)</td><td>Let's modify the objects-first approach into design-patterns-first</td><td></td></tr>
<tr><td colspan="3">The authors argue that design patterns have gained great importance and consequently must be taught in CS1. They describe the content of their first five weeks of their course: 1. lesson: Show the students an example of an object based program (with the property that the objects are not merely representation of real life objects), Create test classes, and introduce design patterns. 2. lesson: Create a handwritten program, introduce the first design pattern (Servant) and introduce the concept of interface. 3. lesson: Introduce more design patterns (Crate and observer). 4. lesson: one class – one task (split classes into two) and more design patterns (Bridge, strategy). 5. lesson: Inheritance of interfaces, design patterns (State and proxy)</td></tr>
</table>

Table 3: Approaches to objects-first

The review of articles describing the objects-first approach shows that a unified approach does not exist. However, it seems that there are some important areas of contention:

> **Events**: Several authors argue that one of the learning objectives should be that the students can create event-based programs.
>
> **Components**: Several authors argue that it is vital for students to be better to (re)use existing components. It is important that the students know the

difference between the specification and the implementation, and that it is normal to use code others have created.

Micro-worlds: Several different micro-worlds have been created and used in objects-first courses. The authors argues either by the motivational factor of a micro-world (students play many games) or by the reduction in complexity of the programming language or development tool (or both).

Design: Several authors argue that design (modelling in figure 4Figure , p. 33) is a key competence of computer science students and consequently must be (one of) the primary learning goal(s). Others argue that design is too difficult for novices and consequently it shall be postponed to later courses.

Concrete exercisers: Several authors describe concrete examples they have used in their teaching. Most of the examples have "motivation" as a key quality parameter.

Trends in industry: Several authors argue that new trends in industry require new learning goals in their CS1 course.

The overall conclusion is that there are many different approaches to teaching an objects-first introductory programming course. Many good ideas are described, but the contexts of the concrete experiences are lacking thus making it difficult to transfer to other practitioners. This could call for a more uniform way of describing teaching experiences – a description based on pedagogical theories and didactics to make it more transferable.

Others have discussed how to teach object-orientation with a focus on the conceptual model aspect. The conceptual model aspect is one of the defining characteristics (Knudsen, Lofgren, Madsen., & Magnusson, 1993 chapter 4) of what is known as the "Scandinavian approach" to object-orientation (Madsen, 1995). In the following, these other articles are discussed and their differences to a model-based approach highlighted. For a short description of the model-based approach see page 15; for a more elaborated description see section 6.1.

Knudsen and Madsen (1996) describe how they have used object-orientation as a common basis for the system development part of the curriculum at the University of Aarhus in 1996. In their description, object-orientation was introduced in the second programming course; the first programming course used an imperative programming language and focused on common, imperative algorithms. The main course design criteria used by Knudsen and Madsen was that the "programming courses must introduce object-orientation in such a way that the concepts, languages and [development] tools are applicable in other courses" (p. 55). The focus of a model-based approach is not that the development tools shall be applicable in other courses, but the concepts and language should be. However, Knudsen and Madsen do not focus on a systematic construction process as done in the model-based approach (p. 15). Knudsen and Madsen's "main emphasis in the programming course is on modelling and design" (p. 59). In a model-based approach, the specification models are given to the students in the beginning, but the connection between the domain model, the specification model and the program code is made clear. In this way, students see that it is possible to understand the code at an abstract level. Later the students are required to create their own specification models and implement those models on their own.

The focus of the programming course Knudsen and Madsen describe is "programming-in-the-large;" this is in contrast to the studies done in dIntProg (5.2.4) and IOOP (5.2.1). This implies a focus on many software engineering elements such as "exception handling," "software testing," "persistence and object-oriented databases" and "concurrent programming." Another difference, although not explicitly stated by Knudsen and Madsen, is the study program the students follow. Knudsen and Madsen's students are all going to major in computer science, whereas the students in dIntProg follow many different study programs.

Madsen, Torgersen, Røn, and Thorup (1998) describe a course for proficient C programmers learning object-orientation. That course objective is to teach both object-oriented programming and object-oriented modelling (p. 3). They focus on the underlying object-oriented conceptual framework, and start by teaching the students the conceptual framework. They find that:

> Using the conceptual framework as the foundation for the object-oriented programming course gives students a generally applicable knowledge, which is independent of any concrete programming language. This is very powerful and useful in many other settings because it gives the students a tool to criti-

> cally understand and evaluate new object-oriented programming languages and methodologies (p. 5).

Madsen et al. uses both a graphical and textual notation as in a model-based approach.

> The main motivation for using a graphical, as well as a textual, syntax was to emphasize that the main difference between OOA/OOD languages and OOP languages is a matter of syntax. We often experience that people have difficulties in realizing that OOA/OOD and OOP languages are based on (almost) the same set of abstract language constructs. (p. 6)

This is consistent with the model-based approach, which presents the students with a specification model expressed as a UML class model and the corresponding code. In order to focus on a systematic approach to programming, the students abstract general coding patterns from many concrete implementations of specification models.

Alphonce and Ventura (2002) describe an introductory programming course with "a strong design and thematic object orientated philosophy" (p. 72). Their approach begins, like the approach of Madsen, Røn, Krab, and Torgersen, with an introduction to (elements of) the object-oriented conceptual framework. In a later article (Alphonce & Ventura, 2003), they describe how they use a small, graphics library called NGP ("Nice Graphics Package") in order to teach their course.

Alphonce and Ventura use a code generation tool so that the students develop designs in UML and generate the corresponding code. The model-based approach requires the students to implement the class structure by themselves. The rationale is twofold. First, the students experience hands-on the connection between the specification model and the code (they see how a concrete model is implemented in a given programming language); second, the students get hands-on experience with abstraction, because they realise that the implementation of an element of the specification model can be described in an abstract way (they se that there is a coding skeleton for most UML class-model constructs).

Hadar and Hadar (2007) discuss the balance between the concrete language and the conceptual framework when teaching objects-first. They – like Alphonce and Ventura (2002) – argue for the use of a code generating tool. They use the tool to generate skeleton programs from UML diagrams, thereby showing the students the equivalence of the UML model and the programming code.

Zhu and Zhou (2003) discuss the difference between teaching a programming language (in their article this is specifically C++) and the general object-oriented conceptual framework. They conclude that the methodology is more important than the programming language (in concordance with Luker (1989; 1994)), and suggest six steps to introduce object-orientation:

> 1) Discuss fundamental principles of object-orientation with respect to conventional thinking;
>
> 2) Introduce an object concept by observing the real world;
>
> 3) Acquire the class concept by abstraction of many common objects;
>
> 4) Introduce instantiation after the class concept is learned;
>
> 5) Illustrate subclasses by adding more details to an existing class and super classes by finding common things among several classes;
>
> 6) (Optional) Discuss meta-classes to master completely the class and object concepts. (p. 141)

They use formal description and not a concrete programming language (e.g., their description of an object is a “quadruple: Object ::= <*N*, *s*, *M*, *X* >, where *N* is an identification or name of an object; *s* is a state or a body represented by a set of attributes; *M* is a set of methods (also called services or operations) the object can perform; and *X* is an interface that is a subset of the specifications of all the methods of the object.” (p. 142)). Furthermore, they seem only to pay attention to the concepts “object”, “class” and “inheritance” and consequently leaving out major parts of the object-oriented conceptual framework such as association and aggregation.

Sicilia (2006) raises “a number of problematic issues regarding the teaching of object-oriented concepts with Java in CS1” (p. 16). He categorises the problems in three areas:

1. From object models to Java programs,
2. Teaching extensibility, and
3. Design by contract and exceptions.

Overall, the ideas Sicilia presents comply with the ideas in a model-based approach. He finds that starting with object-models is very fruitful, because it makes the students aware of the phenomena in a given problem domain. The objects can then be grouped into sets, thereby introducing the idea of a class. He describes his approach as an "instances first" approach (p. 4).

Sicilia introduces coding patterns for the different conceptual constructs. He finds the container classes in Java to be problematic to introduce, hence he uses `arrays` to implement to-many associations.

The overall conclusion is that there exist many different approaches to designing and teaching an objects-first introductory programming course. A few of these approaches focus on the conceptual framework as a guiding principle in the course design.

3.2.2 Research in Novices' Programming Process

One of the problems students struggle with in programming is not the product (i.e., the program), but the construction process (Soloway, 1986; Spohrer & Soloway, 1986; Spohrer & Soloway, 1986).

In the eighties, a series of studies on misconceptions or bugs represents one line of research in this area. The following are a few examples of this research. For an overview of some of the early studies on program comprehension and program generation see Robins, Rountree and Rountree (2003). For a list of different bugs/errors/problems, see table 1 (pp 46-48) in Ko and Myers (2005).

Bayman and Mayer (1983) found that students who learned BASIC through a self-study course were capable of generating programs but "possessed a wide range of misconceptions concerning the statements they have learned" (p. 677). They found that most of the problems were related to misconceptions of what the execution of a given language construct stands for.

Bonar and Soloway (1985) argues that "many novice programming bugs can be explained as inappropriate use of knowledge used in writing step-by-step specifications in natural language" (p. 134), i.e. the students wrongly transfers the semantics of natural language words to programming language. As an example, they discuss the difference between the meaning of the word “then” in a programming language and in natural language.

Goldenson and Wang (1991), based on analyses of log-files, characterized the processes by which students interact with the Pascal GENIE novice programming environment. However, their focus was not to draw general conclusions of novice programming processes but to evaluate the usefulness of syntax directed structure editing.

Many other studies were performed at that time in order to describe bugs and give explanations to what causes the bugs. Several researchers tried to describe a catalogue of bugs containing over 100 bugs (Johnson, Soloway, Cutler, & Draper, 1983; Spohrer et al., 1985).

All of the above studies were done in the context of an imperative programming style. Ramalingam and Wiedenbeck (1997) and Wiedenbeck, Ramalingan, Sarasamma and Corritore (1999) focused on the differences between the mental representation of object-oriented programs and imperative programs. They found that the comprehension of imperative programs was better, but there were differences in the mental representations. The imperative programs focused on program-level knowledge whereas the mental representations of the object-oriented programs focused more on domain-level knowledge. Their findings supports the findings of Gilmore and Green (1984) who, from a comparison of answers to questions to programs in the procedural and declarative notation, found that the notation highlights some information at the expense of others. The procedural notation, naturally, supported questions about what happens in a program after some specific action is performed, whereas the declarative notation was better when answering questions about what combination of circumstances causes a given action. The transferability of studies from the imperative paradigm to the object-oriented paradigm is therefore not straightforward.

The above mentioned research is mainly focused on program comprehension and not the programming process. Comprehension plays a role in a novice’s programming process, but reading a program in order to understand it is a different activity than creating a program to solve some task.

Research in the problems a student encounters regarding their programming process seems limited. Hundhausen, Brown, Farley and Skarpas (2006) note

> 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. (p. 59)

Their goal was not to generalize problems students encounter in their programming process but to evaluate three different forms of semantic feedback for their ALVIS! Live programming environment (for a description of this environment see Hundhausen and Brown (2007)). They found four distinct behavioural patterns that could describe 29 of 35 participants: *No Feedback* (completes solution with few missteps), *Automatic* (succeeds through persistence), *On Request* (cannot get on track, despite honest effort) and *No Feedback* (gives up quickly).

Eckerdal, Thuné, and Berglund (2005) have studied what it takes to learn program thinking from a student perspective. Using a phenomenological research method, they have analysed the interviews of 14 students and found five categories of program thinking:

> Learning to program is experienced as to understand some programming language, and to use it for writing program texts.
>
> As above, and in addition learning to program is experienced as learning a way of thinking, which is experienced to be difficult to capture, and which is understood to be aligned with the programming language.
>
> As above, and in addition learning to program is experienced as to gain understanding of computer programs as they appear in everyday life.
>
> As above, with the difference that learning to program is experienced as learning a way of thinking which enables problem solving, and which is experienced as a ”method” of thinking.
>
> As above, and in addition learning to program is experienced as learning a skill that can be used outside the programming course. (p. 137)

Many teachers teaching introductory programming courses claim that the students learn ‘problem solving’ and consequently expect that their students should be at Eckerdal, Thuné, and Berglund’s highest level when they have finished their programming course. The natural question here is what is meant by ‘problem solving’? Eckerdal, Thuné, and Berglund do not define what they mean by ‘problem solving’, so students may have many different views on the concept. Palumbo (1990) made an extensive review of literature regarding the connections between learning programming languages and problem solving skills. He summarises many studies (done in the imperative style of programming and most of them in a primary or secondary school setting) on the transferability of learning a programming language and problem solving skills. Palumbo distinguish between two types of transfer: the distance of transfer and generalizability of transfer. Near transfer is “skills and expertise to a new problem solving domain that is similar in its stimulus features to the domain in which proficiency and skill have already been obtained and established “(p. 70), whereas distant transfer is “the transfer of skills and expertise to a new problem-solving domain that is distinctly different in its stimulus features to the problem-solving domain where expertise had already been established” (p. 71). Specific transfer “can be defined as transfer of one or more specific skills to a new problem-solving domain. Generalized transfer can be defined as the transfer of general problem-solving strategies and procedures from one problem solving context to another.” (p. 70). His conclusion is that one can not expect neither distant nor general transfer to happen in an introductory course, since the time possible to practice is simply too limited.

Eckerdal, Thuné and Berglund restrain themselves from giving advice on how to enhance the students “program thinking” level. However, they find that “in object-oriented programming as in mathematics there are standard solutions to certain type of problems, ’canonical procedures’ to learn and discover. They are used by experienced programmers and necessary for the simplification and speed up of the work. It is thus desirable to help the students to discern such procedures” (p. 140). Canonical procedure is a term defined by Hazzan (2003):

> A *canonical* procedure is a procedure that is more or less automatically triggered by a given problem. This can happen either because the procedure is naturally suggested by the nature of the problem, or because prior training has firmly linked this kind of problem with this procedure. (p. 108)

Consequently, Eckerdal, Thuné and Berglund ask an important question:

> How can we help beginning programming students to discern ’canonical procedure’ when learning object-oriented programming? (p. 141)

In this light, the answer to research question R1 (*Is it possible to make a course design where the students learn a systematic programming process, conceptual models as a structuring mechanism, and coding?*) can be seen as one response to Eckerdal, Thuné, and Berglund’s question: Coding patterns describe ‘canonical procedures’ that programmers can use when they encounter the problem of implementing a class model, the focus on standard algorithmic solutions to standard problems like finding one object among many objects and process recordings describe ways to integrate these ‘canonical patterns’. However, if the problem students shall solve is distant and/or more general than the problems they have learnt one can not (and should not) expect them to be able to solve it (Palumbo, 1990). deRaadt (2007) did an experiment in order to see if explicit teaching of programming plans had an impact. They taught the same curriculum to two groups of students during a weekend, but in one of the groups he explicitly taught plans. In the final programming assignment, he found that 44% of the solutions in the group where the plans were taught included plans whereas only 28% of the solutions included plans in the other group[17].

One of Hazzan’s compatriots Orna Muller’s answer to ‘canonical procedures’ is pattern-oriented instruction (Muller, 2005; Muller, Ginat, & Haberman, 2007). She has described a number of algorithmic patterns that are:

> solutions to basic recurring algorithmic problems and form the building blocks for developing algorithms. Algorithmic patterns refer to a classification of algorithmic problems according to their goal (Muller, 2005, p. 60)

From a study based on two groups of students – one group that has received pattern-oriented instruction and one that has not – she found that the group who was exposed to pattern-based instruction exhibited a better problem-solving competence than the other group (Muller et al., 2007).

As for the teaching of the programming process, Caspersen (2007) concludes

[17] Please note that the groups only contained four people, so he was not able to check for statistically significant differences.

> 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. (p. 73)

The claim made by Caspersen is further illustrated by the lack of focus on the programming process in computer science textbooks. Lin, Lin and Wu (1999) analysed the examples in 16 textbooks for teaching computer science in high schools and found that all of the textbooks neglects the programming process when presenting examples. Only two of the textbooks include examples where debugging and correction of logic or semantic errors were included.

Using the pedagogical theory of cognitive apprenticeship, this dissertation describes a way to unfold the programming process to novices.

3.2.3 Research on Success Factors for Learning Introductory Object-Oriented Programming

A substantial amount of research has been conducted to identify general variables that predict the success of students aiming for a university degree. The variables investigated encompass gender (Rountree et al., 2004; Sanders 1998; Ventura, 2005), parents' educational level (Ting & Robinson, 1998), ACT/SAT[18] scores (Brooks & DuBois, 1995; Butcher & Muth, 1985; Sanders 1998), performance in prior courses (Chamillard, 2006), emotional factors (Bennedsen & Caspersen, 2008c; Pritchard & Wilson, 2003; Szulecka, Springett, & de Pauw, 1987), and the application of a consistent memory model (Caspersen, Bennedsen, & Larsen, 2007; Dehnadi, 2006; Dehnadi & Bornat, 2006). Research has been conducted within the general context of education, within computer science and in the more topic specific area of introductory programming (Bennedsen, 2003; Bergin & Reilly, 2005; Byrne & Lyons, 2001; Hagan & Markham, 2000b; Leeper & Silver, 1982; Pillay & Jugoo, 2005). Even in the area of introductory object-oriented pro-

[18] ACT: is formerly known as the American College Test. An American, nation-wide college entrance exam. It assesses high school students' general educational development and their ability to complete college-level work. It is a multiple-choice test that covers four skill areas: English, mathematics, reading, and science. The Writing Test, which is optional, measures skill in planning and writing a short essay (ACT). SAT (formerly known as the Scholastic Aptitude Test and Scholastic Assessment Test) is a standardized reasoning test taken by United States high school students applying for college. It covers two areas – verbal and mathematics. (SAT)

gramming research has tried to establish general factors to predict particular students' success or failure. In particular, Phil Ventura's work (2003) focuses on a systematic evaluation of hypotheses related to the success factors of an introductory programming course using an objects-first approach. The results are documented in Ventura (2003); Ventura and Ramamurthy (2004) and Ventura (2005).

In the sixties a lot of work on creating and validating psychological test to select programmers were performed. Much of the work of the Special Interest Group in Computer Personnel Research (SIGCPR) was about psychological tests for the selection of computing staff. Back then there were not many people educated in the field but industry had a huge demand for manpower. In 1966 the number of programmers and system analyst was 170.000 – 200.000; and the number was expected to rise to 400.000 in 1970 (Dickmann & Lockwood, 1966). Simpson (1973) published a bibliography in 1973 containing 152 publications describing test for programming ability.

Evans and Simkin (1989) sum up the arguments given in many studies for performing these kinds of studies:

1. Discriminating among enrolment applicants
2. Advising students' 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

(p. 1322)

Evans and Simkin's study was carried out during a time when many students enrolled in computer science classes. This is not the case today, and so the first argument is not currently relevant because most computer science departments now accept all students who apply.

The following table is an extract of appendix 11.9 presenting the findings of selected studies. The presentation in the appendix describes the researcher, the number of students involved in the research, the programming language involved, the variables investigated, the findings, and the research method. Studies focused on formal reasoning – or abstraction ability – are excluded, these are addressed later (apart from Werth (1986) who mainly focuses on other variables but includes cognitive development level).

Study and year	**Predictors**
Gotterer and Stalnaker (1964)	The two significant factors are the E-51 (the tested must solve a problem presented in program form) and the relationship between part three of the IBM programmer aptitude test (PAT) and the math achievement.
Biamonte (1964)	Both PAT and BLT (a paper and pencil analogue of the Logical Analysis Device) correlated with the exam-score (PAT r=0.56, BLP r=0.44).
Bauer, Mehrens and Vinsonhaler (1968)	All variables were significantly correlated with course grade at p<0.05. GPA predicted most (46%). Three variables (GPA, figures series of aptitude test for programmer personnel (ATPP) and computer programmer scale of strong vocational interest blank) predicted 65.6%
Alspaugh (1972)	Math background, personality (thusstone temperament schedule) and critical thinking (Watson-Glaser critical thinking appraisal) accounts for 61.4 % of the variation. Math alone accounts for the most variation.
DeNelsky and McKee (1974)	The aptitude assessment battery: programming predicts 16% of the students' grade.
Newsted (1975)	Programming ability, GPA and time spend predicts 41 %. Time spend is a negative factor (i.e. more time spend means lower result).
Capstick, Gordon and Salvadori (1975)	A non-significant relationship between ATPP scores and COBOL. The correlations between ATPP scores and FORTRAN are significant at p< .05. The larger and most significant correlation is between FORTRAN, and the arithmetical reasoning test component.
Mazlack (1980)	None of the variables showed a correlation with the final grade.
Leeper and Silver (1982)	All but math and science correlate with the final grade. The regressions formula predicts 26 %. 64 % of the predicted fail-students did actually fail in the following year.

Konvalina, Wileman and Stephens (1983)	Students who passed the course (228) had significantly more mathematics background than students who withdrew (154). There were statistically significant differences between the students who passed and failed wrt. all but one of the success factors (word problems).
Hostetler (1983)	GPA, diagramming and reasoning score, mathematics background, and Sober/Happy-go-lucky accounted for 42.6 % of the variance.
Nowaczyk (1983)	Performance in prior mathematics and English courses, amount of previous computer experience, expected grade in the course, and performance on selected logic and algebraic-word problems accounted for a statistically significant amount of variation (does not say how much). Neither personality factors nor computer anxiety correlated with success.
Sorge and Wark (1984)	In order to make a satisfactory progress in the program studied (computer science major), a student should have SAT math score of at least 560, SAT-verbal score of at least 500, a score of at least 5 on a trigonometry test, a grade of A or B in the first CS course.
Werth (1986)	Correlation between College grades/high school math ($R=0.252$, $p<0.05$), cognitive style ($R=0.317$, $p<0.01$) and Piagetian intellectual development ($R=0.232$, $p<0.05$).
Mayer, Dyck and Vilberg (1986)	Word problem translation, word problem solution, and following directions accounted for 50% of the variance.
Austin (1987)	High school composite achievement, quantitative end algorithmic reasoning abilities, vocabulary end general information abilities, self-assessed math ability, and cognitive style – self-assessed introvert accounted for 64% of the variance.
Evans and Simkin (1989)	Combinations of the variables accounted for at most 23% of the variance of the six different outcome variables. No pattern can be found in the combinations.
Taylor and Mounfield (1989)	Students with prior computer science courses as well as male students perform better, hours worked and major had no impact.
Gibbs (2000)	No correlation between field dependent /.field independent and design or coding was found.
Goold and Rimmer (2000)	Average score in other units, relative abstraction, dislike programming, problem solving, raw secondary score and gender accounted for 43% of the variance.
Hagan and Mark-	The more programming languages a student knew prior to taking the

ham (2000b)	course, the higher the performance.
Byrne and Lyons (2001)	Math (*r* = 0.353) and science (*r* = 0.572, but only 35 students had studied a science subject) correlates with programming performance.
Rountree, Rountree and Robins (2002)	The strongest single indicator of success was the grade the student expected to get on the course. Other factors that relate to success include whether students think their background is science, commerce, or humanities; whether they have recent university math experience; and what year of study they are in.
Stein (2002)	There is a correlation between math and success in CS1. Students who study calculus do at least as well as students who study discrete math in their CS1-CS2 courses.
Wilson (2002)	Comfort level, math background, attribution of success/failure to luck (negative correlation), work style preference and attribution of success/failure to task difficulty accounted for 40% of the variance.
Rountree, Rountree, Robins and Hannah (2004)	Four rules to identify students who will potentially fail. The rules identify 53% ± 11% instead of the expected 27%. Two rules for successful students.
Holden and Weeden (2003)	Prior experience (sum of knowledge of encapsulation and inheritance (IE), number of semesters in high school involving programming and programming as part of work) as well as IE in itself is an advantage in the first course in a programming sequence, but not in later courses. Comfort level does not correlate with success.
Ramalingam, LaBelle and Wiedenbeck (2004)	Mental model, self-efficacy, and previous programming and computer experience accounted for 30% of the variance.
Ventura and Ramamurthy (2004)	Prior knowledge of C++ or Java was not a predictor of success for their objects-first CS1. There were no correlation between programming self-efficacy and success.
Bergin and Reilly (2005)	Student’s perceived understanding of the module, gender, Irish learning certificate mathematics grade, and comfort level accounted for 79% of the variance.
Ventura (2005)	Percent lab usage, comfort level, and SAT math score accounted for 52.9% of the variance.

Wiedenbeck (2005)	Prior computer and programming experience, self-efficacy, and knowledge organization accounted for 30% of the variance.
Thomas, Karahasanovic and Kennedy (2005)	There is a fairly strong, negative correlation between typing speed and coding performance.
Bergin and Reilly (2006)	Irish learning certificate mathematics score, number of hours playing computer games (negative effect), and programming self-esteem accounted correctly classified 80% of the students in the categories weak and strong.
Rauchas, Rosman, Konidaris and Sanders (2006)	English predicts better than math.
Simon et al. (2006)	A positive correlation between deep learning and the exam score, surface learning was negatively correlated. Spatial visualization skills are positively correlated. A rich articulation of search strategies is positively correlated.

Table 4: Predictors of success for programming

For another overview of prior research in the area of predicting success in programming, see Caspersen (2007) table 5-1.

Many computer science educators argue that abstraction is a core competency for computer professionals, and that it therefore must be a learning goal of a given computer science curriculum – see, e.g., Alphonce and Ventura (2002); Kurtz (1980); Nguyen and Wong (2001); Or-Bach and Lavy (2004) plus Sprague and Schahczenski (2002).

Nguyen and Wong (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 introductory programming with a strong focus on abstract thinking and developing the students' abstract skills.

Or-Bach and Lavy (2004) argue that abstraction is a fundamental concept in programming in general, and in object-oriented programming, in particular. The authors describe a four-level ordering of cognitive abstraction activities that students employ when solving a given problem: 1) defining a concrete class, 2) defining an abstract class with attributes only, 3) defining an abstract class including methods, and 4) defining an abstract class including abstract methods. They do not use a general definition of cognitive development, and therefore no standard test instruments that others have proven valid and reliable. Their definitions (and test instrument) are very specific to object-orientation.

Sprague and Schahczenski (2002) also argue that abstraction is the key concept for computer science students. They furthermore argue that object-orientation "facilitates, and even forces, a higher level of abstraction than does procedural programming" (p. 211).

Hudak and Anderson (1990) found that a measurement of learning style (concrete experiencing, measured via Kolb's learning style inventory (Kolb, 1976)) and formal operation (the level of formal operations, measured by formal operational reasoning test (Roberge & Flexer, 1982)) correctly classified 72% of computer science students using a cut-off of 80% or better as a criterion of success in the course.

Kurtz (1980) used what seems to be a sub-scale of Inhelder and Piaget's stage theory (Inhelder & Piaget, 1958) for his study, and found that the measured abstraction level strongly predicted programming success (it predicted 66% of the final score). However, the study's sample was very small (n=23). He found that the first and last developmental levels were "strong predictors of poor and outstanding performance, respectively; and the [developmental] level predicts performance on tests better than performance on programs" (p. 114). Barker and Unger (1983) did a follow-up study (almost a replication but with a shortened version of Kurtz's instrument) with a much larger sample (n=353). They found that the intellectual development level (Inhelder & Piaget, 1958) accounted for 11.6% of the students' final grade.

As described in section 5.4.3, it can be difficult to generalize from one cultural context to another. Most of the above-cited studies were done in universities in the United States. Care is required when directly comparing the results found in these studies to results that we find when studying Danish university students. For example, Phil Ventura's finding about math's missing impact on his students' success might not be directly transferable to a Danish cultural context, and therefore would not necessarily contradict others' findings.

From the above we conclude that it is difficult to make a general conclusion on the success factors for programming. Most of the factors are predictive in some studies but non-predictive in others. One of the few factors that seem to recur in the prediction is students' mathematical ability. Success is in almost all studies defined as the result of the examination.

3.3 Research on Pedagogical Patterns

As described in section 1.1.1.3, the goal of pedagogical patterns is to capture expert knowledge of teaching and make the transfer of such knowledge possible to practitioners. Like design patterns, pedagogical patterns do not express new pedagogical ideas but tried and "proven" solutions to pedagogical problems.

Most of the research within the field of pedagogical patterns is done in a positivistic style, merely describing the expectations of the pedagogical patterns' developers (e.g., Eckstein (2001); Seffah and Grogono (2002) plus Sharp, Manns, and Eckstein (2000; 2003)).

A few descriptions in the "Marco Polo style" were also noted from teachers who have used pedagogical patterns to develop their teaching or who have reflected on their teaching – see, e.g., Erickson and Leidig (1997); Muller, Haberman, and Averbuch (2004) plus Seffah and Grogono (2002).

The Journal of Computer Science Education has published a thematic issue on pedagogic patterns (this is the issue where both of the articles on pedagogical patterns included in this dissertation appear): Volume 16 issue 2, 2006. In this issue, the following articles are presented:

> Denning, P. J., & Hiles, J. E. (2006). Transformational Events, 77-85
>
> Haberman, B (2006). Pedagogical Patterns: A means for Communication within the CS Teaching Community of Practice, 87-103.
>
> Retalis, S., Georgiakakis, P. & Dimitriadis, Y.(2006). Eliciting Design Patterns for e-learning Systems, 105-118.

Bennedsen, J. (2006b). The Dissemination of Pedagogical Patterns, 119-136.

Derntl, M., & Botturi, L. (2006). Essential Use Cases for Pedagogical Patterns, 137-156.

Bennedsen, J., & Eriksen, O. (2006). Categorizing Pedagogical Patterns by Teaching Activities and Pedagogical Values, 157-172.

Denning and Hiles (2006) describes a pattern for teaching and explaining how innovations occur. According to their pattern, it occurs in three stages *Period of increasing ferment*, *Transformational moment* and *Period of adoption and settling*. They describe how they have used this pattern in their Cornerstones of Computer Science course where the students identify modern problem areas ready for transformation.

Haberman (2006) describes how she has used pedagogical patterns as a tool to discuss and retain pedagogical development and expertise among high-school computer science teachers. She proposes a five stage process to foster the use of pedagogical patterns as a communication tool among teachers:

1. A beginners workshop so that newcomers can be introduced to the concept of patterns,
2. Identify new patterns. Each teacher identify new patterns based on their experience,
3. Writers' workshop. Presentation, discussion and identification of recurrence of the proposed patterns,
4. Feedback and assessment. The teachers should give feedback and assess the patterns, pattern languages and value systems, and
5. Shepherding. The teachers at this stage should help newcomers in order to expand the community.

Retalis, Georgiakakis, and Dimitriadis (2006) discuss the process of creating patterns for e-learning systems. They argue that these patterns must contain both pedagogical and technical advice. As they say: "Creating patterns for e-learning systems thus signifies an explicit commitment to a set of values that characterize well-being, well-learning, and a good social and physical convivial learning environment.” (p. 109). With this boundary condition, they propose a four step development process in order to create a design pattern language for e-learning systems: Identify and sketch design patterns, draft a design pattern, critique patterns, and identify related patterns.

Derntl and Botturi (2006) define three key terms: Design pattern, pattern language and pattern system. By seeing the development of patterns as a system development process, they describe functional and non-functional requirements for pattern systems, add structure to these requirements, and derive essential use cases. Finally, “implications concerning the pedagogical use of pattern-based design are drawn, concluding that a stronger focus on the underlying (pedagogical) value system is required in order to make a pattern system a meaningful tool for effective educational design” (p. 137).

Apart from the special issue, research in pedagogical patterns is hard to find. Jalloul (2000) describe and evaluate their development of three pedagogical patterns (Link, Reuse and Evaluation) used in the teaching of object-oriented software engineering. Their biggest challenge was the

> gap between the theory introduced in classrooms and the real habitat of the theory, namely work environments and challenges. In particular it was hard to teach object-oriented software engineering concepts […] effectively and convincingly when the practical application of these concepts was confined to class work or lab projects. (p. 76).

They find that the application of these three patterns have enhanced the learning outcome of the students.

From this we conclude that the research area of pedagogical patterns is emerging and no clear research tradition exists. It is necessary to focus on the foundation of pedagogical patterns like structuring of patterns or a template that emphasizes the pedagogical aspects of the patterns.

4 Research on Teaching and Learning

Experience is the best teacher, but a fool will learn from no other.
Benjamin Franklin (1706-1790)

As opposed to computer science, the field of learning has a long tradition. This chapter starts by presenting a more elaborated didactical model than the didactical triangle (see section 1.1.1). Following that an overview of different learning theories are given and a short description of published applications of learning theory in computer science education. The next section gives a more detailed description and argumentation of the learning theories applied in the concrete courses used as empirical basis for this dissertation – constructivism, situated learning and apprenticeship learning (and more specific cognitive apprenticeship). The last section discusses the differences between young students learning programming as their formal education and professionals learning a new programming paradigm, i.e. the difference between freshmen and professionals.

4.1 Didactical Models

Many factors influence teaching and learning and consequently have to be taken into account when designing a course. For example, Hiim and Hippe (2006) have developed a didactic model that identifies six aspects that influence students' learning outcomes[19] [20]:

> **Student's learning premises** – knowledge, experiences, attitudes and skills that the students already possess when they come for the course's first lesson.

[19] Translation from Danish done by the author.

[20] Other didactical models exists (e.g. Weitl, Süß, Kammerl and Freitag (2002) describes a didactical model for e-learning or Hansen and Tams (2006) who describes what they call the diamond-model (seven elements influence teaching and learning: goals and purpose, the teacher, the institution, the student, the content, the form , and the evaluation)) but the purpose of this section is not to discuss different models but to show that many elements influence a teaching and learning situation. For that purpose Hiim and Hippe's model is found very appropriate.

External conditions – conditions that limit or make learning possible such as equipment, artefacts, time, place, classroom settings, teacher's resources, learning resources, etc.

Objectives for the learning activity – what the students should learn from the course/activity in terms of knowledge, skills, attitudes and competencies.

Contents – what the course is about, and how its content is selected, adjusted and presented.

Learning process – the process of change within the learning subjects (i.e., the students), and reflections on how the intended changes are facilitated.

Evaluation – assessment or evaluation in relation to the teaching process, in relation to the course's objectives and in relation to the students' learning.

All of the above aspects influence each other (e.g., the content can not be selected without knowing the student's learning premises, the objectives …); consequently their model is normally drawn as in figure 7:

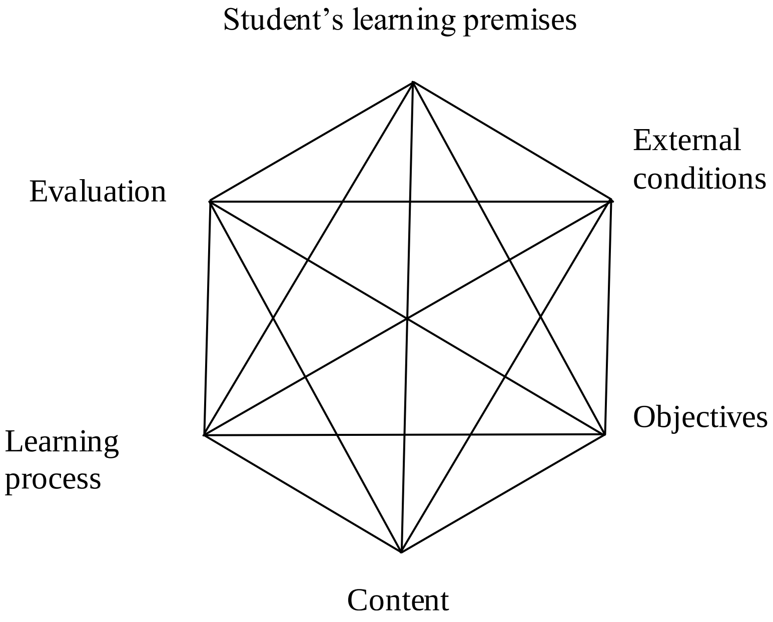


Figure 7: Hiim and Hippe's didactical model

Hiim and Hippe's model can be seen as a detailing of the didactical triangle as shown in figure 8.

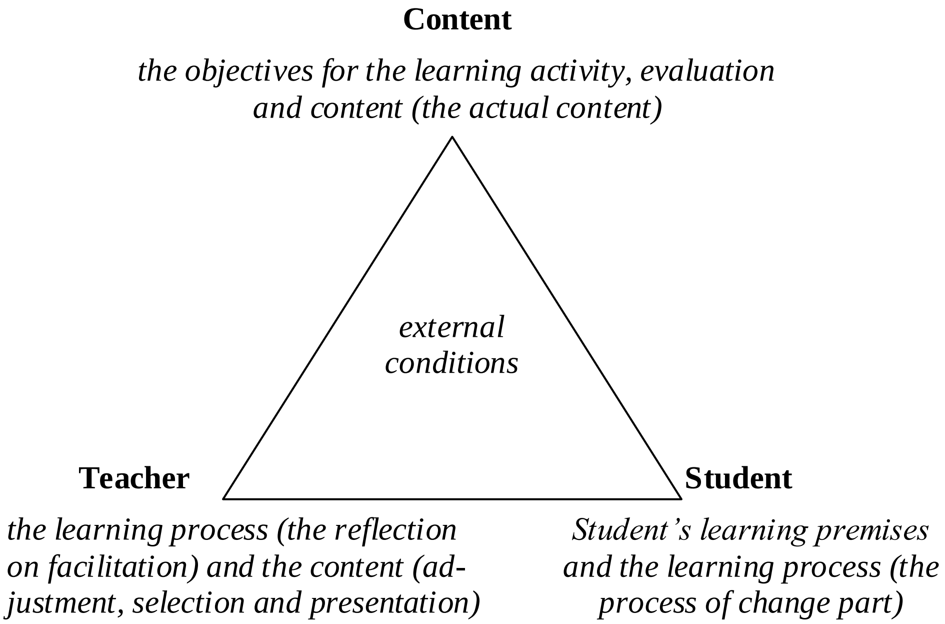

Content
the objectives for the learning activity, evaluation and content (the actual content)
external conditions
Teacher
the learning process (the reflection on facilitation) and the content (adjustment, selection and presentation)
Student
Student's learning premises and the learning process (the process of change part)

Figure 8 Relation between the three elements of the didactic triangle and the *six elements of Hiim and Hippe's didactic model (in italics)*

Kaasbøll (1998) describes three didactic models for programming: *semiotic ladder*, *cognitive objectives taxonomy* and *problem solving*. Semiotic ladder is described as: "the teaching and learning sequence starts out from syntax, and proceed to semantics and pragmatics of the language-like tools" (p. 196), cognitive objectives taxonomy: "The sequence of instruction comprised using an application program, reading the program, an changing the program. Creating a program may also be added." (p. 196), and problem solving: "Through solving problems, the students should extend their experience and repertoire of practice, and the basis for the process is the knowledge structure of the field of programming. The problem solving process is guided by methods and environments." (p. 196). Based on five interviews with teachers at computer science departments in Australia he concludes:

> … interviews with teachers of introductory programming in universities indicated that suggested models for teaching had not been adapted, and that most courses had no clear teaching model.

From the above, we conclude two things. Firstly, many factors influence a learning situation and consequently it is difficult to generalize from one educational context to another. Secondly, it is a prerequisite for others to evaluate the results to know the context in which the research is done. For a start, it seems problematic to compare results from CS1 courses simply due to the fact that the learning objectives are not compatible (or at least stated explicitly so that one knows what to compare). The observations by Kaasbøll on the different didactic models can be seen as a difference between the learning objectives, e.g., is it the mastering the programming language or mastering problem solving and the ability to express a solution to this problem in some formal language that is the goal. Consequently, in this research the learning objectives, students, etc. are described in order to help others to evaluate the transferability of the results.

4.2 Learning Theories

Cognition and learning are central concepts in education. There is a long tradition for research in this area, and many different "schools" of education exists. Consequently, many different descriptions and classification can be given. The following overview is based on the categorization from Greeno, Collins, and Resnick (1996) and consequently uses their terminology. They have described three general views of knowing and learning they find dominant in European and North America: *empiricist*, *rationalist*, and *pragmatist-sociohistoric*. The empiricist view is typified by authors like Thorndike (1931), Skinner (1965), and Watson (1925). Empiricism stresses the importance of consistency of knowledge and experience and sees learning as building, strengthening and weakening associations between stimuli and responses. An alternative name for the empiricist view is behaviourist view, a term introduced by Watson (1913), the "father" of behaviourism (Marton & Booth, 1997, p. 4). The rationalist view is typified by authors like Piaget (Inhelder & Piaget, 1958), Bruner (1990) and Glasersfeld (2002). This view seeks "conceptual coherence and formal criteria of truth" (Greeno et al., 1996, p. 16). An alternative name for the rationalist view is constructivist view. The pragmatist-sociohistoric view is typified by authors like Dewey (Sleeper, 2001) and Mead (1974). Alternative names for the pragmatist-sociohistoric view are social constructivism or situated learning.

Many theorists have contributed to the three views; naturally, the following is a rough characterization of the views.

Greeno, Collins, and Resnick (1996) divide their description of the different groups of learning theories under three headlines:

- the views on *knowing*,
- the views on *learning* and *transfer*, and
- the views of *motivation* and *engagement*

In the following we will characterize the three general views of knowing and learning under these three headlines.

4.2.1 Knowing

The empiricist view: What a person knows is often a reflection on what that person has experienced. This is captured in the famous quote of Watson (1925):

> Give me a dozen healthy infants, well-formed, and my own specified world to bring them up in and I'll guarantee to take any one at random and train him to become any type of specialist I might select–doctor, lawyer, artist, merchant chief, and yes, even beggar-man and thief, regardless of his talents, penchants, tendencies, abilities, vocations and race of his ancestors. (p. 82)

Much of the theoretical foundation of the empiricist view was based in stimulus-response association theory, where the assumption is, that behaviour is to be understood as the responses to given stimuli in a given situation. For example, Watson has stated two "laws" that describe the habit[21] of stimuli-response (Ormrod, 2004):

> **Law of frequency:** The more frequently a stimulus and response occur in association with each other, the stronger that stimuli-response habit will become.
>
> **Law of recency**: The response that has most recently occurred after a particular stimulus is the response most likely to be associated with that stimulus.

Later, Hull (1943; 1952) introduced organismic characteristics into the behaviourist learning theory. He proposed that a number of other factors than just the pres-

[21] The term *habit* was adopted by Watson and other behaviorists from William James as a description of behaviour that requires relatively little conscious control (James, 1950).

ence of a particular stimuli and a persons past experiences but unique to each organism and each occasion must be considered in order to predict the likelihood and strength of a response's occurrence.

The rationalist view sees knowledge as structures of information and processes that form these structures (usually known as schemas) and retrieve relevant structures according to the problem at hand. It builds upon a memory model of two memory systems: short term memory and long term memory. The processing takes place in short-term memory and the storage of schemas in long-term memory. This view on knowing is the one taken in many articles on program comprehension (e.g., Soloway and Ehrlic (1984) and Soloway (1986)). Another line of research based on this view on knowledge it the work on "misunderstandings" (e.g., Pea (1986))

The "cognitive load theory" described by John Sweller (Clark, Nguyen, & Sweller, 2006; Sweller, van Merrienboer, & Paas, 1998) has this understanding of knowledge as its basis. The short-term memory is limited whereas the long-term memory holds an unlimited capacity. Consequently, cognitive load theory has described a number of so called effects (e.g., worked examples effect (Paas & Van Merrienboer, 1994; Ward & Sweller, 1990)) that optimizes[22] the load on short-term memory.

The pragmatist-sociohistoric view sees knowledge as distributed in the world among individuals as opposed to the individual views described above. This view includes tools, artefacts and communities of practices where individuals participate in the view on knowledge; enhancing the view from the brain to the society. Communities or groups are composed of individuals but has rules and roles that are part of the (collective) knowledge.

The work of the Russian psychologist Vygotsky (1978) is an example of this view on knowledge. As he writes:

> Every function in the child's cultural development appears twice: first, on the social level, and later, on the individual level; first, between people (interpsychological), and then inside the child (intrapsychological). This applies

[22] optimize in the sense that most possible short-term memory capacity is used on creation/modification/extension of schemas

> equally to voluntary attention, to logical memory, and to the formation of concepts. All the higher functions originate as actual relations between human individuals. (p. 57)

4.2.2 Learning and Transfer

Transfer "is the process of applying knowledge in new situations. Educators want the knowledge that is acquired in school to apply generally in students' lives, rather than being limited to the situations of classrooms where it is acquired." (Greeno et al., 1996, p. 21)

The empiricist view: The view on knowledge is a stimuli-response view. Consequently, learning is the formation, strengthening and adjustment of stimuli-response associations. To apply these associations in new contexts, the learner needs to broaden the "things" that can cause a given response (and also broaden the response). Thorndike expressed this as the effect of "belongingness" (Bevan & Dukes, 1967) – the stimuli-response connection is strengthened if the elements of association somehow belong together.

The rationalist view: Knowledge is represented as schemas and transfer from one context to another depends on the ability to acquire a schema that composes a structure that is invariant across situations. Learning is thus the creation and modification of such schemas. In general, transfer is difficult to verify (Gick & Holyoak, 1983), and research have focused on various aspects of problems and to study the relative effects of these aspects on spontaneous access to and mapping of the previous learned situation (Bassok, 1990). One thing noticed is that learning the general schema has to be acquired along with practice in applying the schema to examples.

The pragmatist-sociohistoric view: Knowledge is seen as practices of communities and the ability of persons to participate in these practices. Learning is therefore seen as enhancing the practices and the participants' abilities to participate. Many of the reforms in education is based on students becoming more active in the learning community, the students must formulate and evaluate hypotheses and questions and do group work. The students must learn such skills. Since learning is tightly linked to a practice, transfer becomes problematic since it is another practice. Typically, there is a distinction between transfer within a community or to a practice outside the community (e.g., from school to job). Many of the tools and artefacts used in one practice is not available in the new practice and “the problem of transfer becomes one of marshalling the resources needed to be successful in the new situation” (Greeno et al., 1996, p. 24)

4.2.3 Motivation and Engagement

The empiricist view: Engagement is assumed to occur mainly due to extrinsic motivations: rewards, punishments or other (positive or negative) incentives. The motivations are extrinsic to the person but require the person’s desire to receive or avoid them. Behaviourists believe that there are basic needs of a person (food, sleep etc) and undesires (pain) and these are the basic motives. If other motives can be associated with these, they influence in the same way. Thorndike conducted a famous puzzle-box experiment, where a hungry cat was placed in a box where a lever could open the door and the cat could get food (see figure 9). The cat performed random tries to open the door; at some point in time hit the lever and the door opened. The cat was placed in the box again (after becoming hungry again), and the same scene happened. But at some point in time, the cat had “learned” that hitting the lever resulted in the door opening. This was due to the basic desire for food.

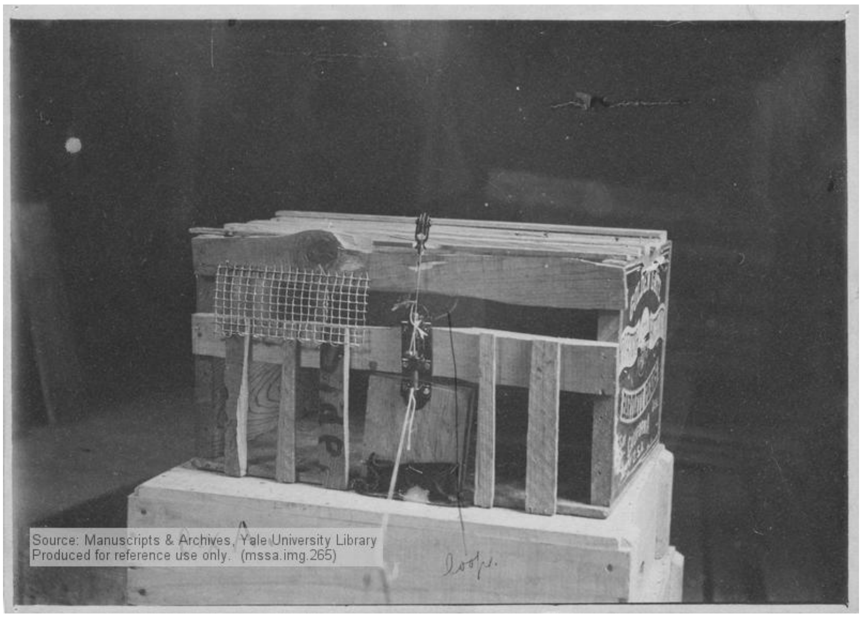

Source: Manuscripts & Archives, Yale University Library
Produced for reference use only. (mssa.img.265)

Figure 9: Thorndike's puzzle-box[23]

The rationalist view: The motivation is seen as intrinsic to a person. Students are expected to be interested in the domain. Children are seen as naturally motivated to learn when their experience is inconsistent with their current knowledge (adjustment of schemas) or they encounter regularities in information (generalization of schemas). Consequently, rationalists have been trying to figure out ways to foster students' natural tendency to learn.

The pragmatist-sociohistoric view: Since learning is seen as a social activity, students become engaged and motivated by participating in communities where learning is valued.

4.3 Learning Introductory Object-Oriented Programming

In general, many things influence a given learning situation (see section 4.1); consequently thinking that there is one right pedagogy to be used in all introductory programming courses is mistaken. This implies the need for documenting the applied pedagogy when doing computer science education research. Unfortunately, it is difficult to find research in the area of computer science education where description of the applied pedagogy is explicit. In general, descriptions of the con-

[23] The picture is from Yale University Library Digital Collections (http://mssa.library.yale.edu/madid/showzoom.php?id=mss&msrg=569&msrgext=0&pg=1&imgNum=265).

text of the research are short description of the university and the content of the course.

A few researchers have published articles describing how a given pedagogy can be applied within the area of computer science. Mordechai (or Moti) Ben-Ari has published a few articles discussing application of constructivism and situated learning in computer science education (1998; 2001; 2004; 2005). Machanick (2007) discuss how social constructivism can be applied in computer science whereas Kölling and Barnes (2004), Astrachan and Reed (1995), Astrachan, Selby and Unger (1996), Chee (1995) as well as Schulte, Magenheim, Niere, and Schäfer (2003) discuss the application of an apprentice based pedagogy in computer science education. In general, it seems that the influence of pedagogy on the design and development of introductory programming courses is limited.

Rowland (1999) discuss the question "what theory underlies this course?" His conclusion – based on a discussion of a course in teaching and learning for higher education teachers – is, that in practice it is not possible to point out a single theory, but

> One possibility is that the question is not looking for 'an underlying theory' but rather 'a set of theoretical resources or insights' …There is not one overarching theory which draws it together, but rather a more pluralistic or eclectic range of insights, methods, approaches, and so on (pp. 304-305)

In the following sections, the pedagogy or theories influencing the design of the model-based courses studied in this dissertation is described and discussed. The above mentioned articles describing the application of a given pedagogy will be addressed in the following sections.

4.4 Constructivism

An old Chinese proverb says that *Tell me about it, and I'll forget it. Show me it, and I'll remember it. Let me do it, and I'll learn it*. This is in short the essence of constructivism. Biggs (2003) quote Tyler (1949) from his book "Basic Principles of Curriculum and Instruction":

> Learning takes place through the active behaviour of the student: it is what he does that he learns, not what the teacher does (p. 25)

Biggs argues that most modern teachers find that this form of constructivism is the best way to enhance learning among the students (Biggs, 2003, p. 13). However, as Phillips (1995) notes, constructivism is not just one theory but it "has many sects" (p. 5), so one needs to be precise when talking about constructivism. Based on Phillips different "constructivist schools", Perkins (1999) calls for what he call pragmatic constructivism – the view of constructivism as a toolbox that the teacher can use for problems of learning; teachers should use whatever works.

Ben-Ari (1998; 2001) discusses constructivism in computer science education. As described above (p. 86), the assumption in constructivism is that cognitive schemas are build on top of other cognitive schemas, i.e. the prior knowledge and understanding of programming and the computer influences the way students build their (new) knowledge of the field. As a consequence (if you accept constructivism as the pedagogical theory) Ben-Ari lists ten issues one must consider, two of which are that "the model (or as du Boulay (1989) calls it "the notional machine") must explicitly be taught" and "don't start with abstractions". Following this, he discusses the problems with an objects-first approach and concludes that "for an objects-first approach to work, teachers will have to develop ways of explaining the underlying models without destroying the abstractions." (p. 63).

Hadjerrouit (1998; 1999; 2005) describes "a pedagogical framework rooted in constructivist epistemology for teaching object-oriented design and programming" (1999, p. 171). In his framework, Hadjerrouit describes seven principles:

1. Object-oriented knowledge must be *actively constructed* by learners, not passively transmitted by teachers.
2. Program development must be guided by *object-oriented concepts*, not by language technicalities.
3. Students' *prior knowledge* in programming needs to be evaluated whether it conflicts with the object-oriented approach.
4. In order to be useful for problem-solving, object-oriented concepts must be *strongly linked* to problem situations and to the object-oriented language being used, which must themselves be strongly connected to each other.
5. The process of constructing strongly linked object-oriented knowledge requires *particular types of problem-solving skills*.

6. Traditional modes of teaching such as lectures must be replaced by a *set of activities* where the students are actively engaged in the knowledge being constructed.
7. To get students actively involved in problem-solving, the activities must focus around a set of *realistic, intrinsically motivating problems*. (1999, pp. 172-173)

The seven principles fit nicely into the goals of a model-based introductory programming course; constructivism is seen as one pedagogical pillar of a model-based course. However, it seems like Hadjerrouit focuses mostly on the conceptual framework of object-orientation not much on the programming process.

4.5 Situated Learning

One of the goals of the model-based approach is that students learn a systematic approach to the construction of programs. This requires a focus on the programming process. But what constitutes the programming process? And how do we make the process explicit? Other professions face the same problem; they need to teach students how to construct things, need to show the students that the solution is not build in one, big, monolithic step but in small steps. Carpenters solve several small tasks when building a house, work out problems they encounter during the construction, use tacit knowledge about the best solution etc. Modern software development is the same (Beck, 1999; Beck et al.; Larman, 2005; Schwaber, 2004); it focuses on iteration, test, refactoring and development in small steps towards a solution. We need to make this explicit to the students and find pedagogical theories that make this transfer of (tacit) knowledge possible.

The social interactions between the experienced master and his apprentices or learners are the driving force in the learners' progressively constructing a deeper understanding of, and experience in, the profession (Aboulafia & Nielsen, 1997). One of the most well-known descriptions is by Lave and Wenger (1991). They demonstrate how midwives, African tailors, United States Navy quartermasters, meat-cutters and non-drinking alcoholics in Alcoholics Anonymous build their knowledge of those fields. Probably, the most cited description is the way Vai and Gola tailors learn by first observing master(s), then sewing and finally cutting the cloth. The learners participate in what Lave and Wenger call the community of practice (CoP). They start in the peripheral of the community, and then move more and more toward the middle. Lave and Wenger call this learning process legitimate peripheral participation (LPP). The essence of LPP is that learning takes place within the community where the knowledge is used, as opposed to learning in conventional schools which teach knowledge that is de-contextualized (Lave & Wenger, 1991, p. 40)

It should be noted that Lave and Wenger explicitly state that LPP is not a pedagogical strategy:

> We should emphasize, therefore, that legitimate peripheral participation is not in itself an educational form, much less a pedagogical strategy or a teaching technique. It is an analytical view on learning, a way of understanding learning (p. 40)

However, as Ben-Ari (2005) argues, it is difficult not to identify LPP with apprenticeship. Consequently, he has "chosen to identify the term with apprentice-like learning that occurs within a community that uses the knowledge." The understanding of situated learning expressed in this dissertation resembles this.

Ben-Ari (2004; 2005) discusses several problems with this analytic way of learning computer science at high schools or universities:

- LPP as it is presented in Lave and Wenger (1991) describes the entire educational process that a newcomer undergoes on her way to full-fledged membership in the community. It is hard to see how to accomplish this with students who aspire to membership in high-tech communities engaged in science, engineering, medicine and finance,
- apprenticeship requires that the future occupation of a student be determined at a very early age,
- apprenticeship is susceptible to nepotism,
- real problems should be presented to the students, and
- it would be unreasonable, as well as terribly inefficient, to burden masters with the entire education of young students, (Ben-Ari, 2004, pp. 87-88)

Based on this problematization of situated learning, Ben-Ari asks the question “what can be learned from situated learning?” His answer includes pedagogy, where he finds

> CSE experts should devote time to analyzing what actually happens in real CoPs, and then to create learning activities that simulate such tasks as well as possible within the constraints of a school. (I emphasize the word simulate, because I want to distance myself from naive LPP that insists on ‘‘real’’ situations and contexts.) This has certainly been attempted in courses of software engineering and other project-based educational activities (Fincher, Petre, & Clark, 2001), but I would extend it to earlier levels of learning. (Ben-Ari, 2004, pp. 93-94)

Ben-Ari’s view on LPP is echoed in the design of the courses studied in this dissertation.

4.6 Apprenticeship Learning

Given the modelling aspect of the studied way of teaching programming and the importance of connecting the learning process to a programming practice, teaching inspired by apprenticeship learning in the craft or craft-like forms of production are found promising (Bennedsen & Caspersen, 2003; Bennedsen, 2006a; Fjuk et al., 2004). This view is shared by others, both in the field of general education (e.g., Levin and Waugh (1998)) and in the more specific field of introductory programming (e.g., Kölling & Barnes (2004)). One example of an apprenticeship-inspired approach to teaching introductory programming is given by Astrachan and Read (1995). Their work describes what they term an application-based apprenticeship model for learning; starting by reading programs written by experts, then modifying them and at the end creating programs by themselves (their didactical model is a good example of the cognitive objectives taxonomy described by Kaasbøll (1998)). Astrachan, Selby and Unger (1996) describe how they have used this approach for a data structure course.

Nielsen and Kvale (1997) suggest two facets of apprenticeship learning: the person-centred facet and the de-centred facet. Concerning the person-centred facet, the master reflects on and thinks aloud about the particular action, making it visible and a source of identification. As such, the apprentice learns from observing the master performing the profession’s embedded actions. Concerning the de-centred facet, knowledge construction is considered as LPP; i.e., the attention focuses on the learner's participation in a community of practitioners where the master, or a more experienced peer, legitimizes the individual learner's skills and knowledge.

Jordan (1989) identified, from a study of village midwifes in Mexico, several characteristics of traditional apprenticeship learning:

- Work is the driving force. The apprentices' progressive mastering of tasks is appreciated for "getting the problem solved," not as a step toward a distant, symbolic goal (such as a certificate).
- Apprentices start with skills that are expected by the master to be easiest and where mistakes are least costly. (The tailors start out sewing, not cutting, the cloth.)
- Learning is focused on bodily performance. It involves the ability to do, rather than the ability to talk about something. It often focuses on skills rather than academic competencies.
- Standards of performance are embedded in the work environment.
- Teachers and teaching are largely invisible.

Wenger (1998) finds schools unsuitable for learning:

> if we believe that information stored in explicit ways is only a small part of knowing, and that knowing involves primarily active participation in social communities, then the traditional format does not look so productive (p. 10)

Traditional university courses make it difficult to apply an apprenticeship-inspired pedagogy or using LPP as an inspiration to design learning situations. There is no direct "work" to be done, so the work cannot be the driving force. This implies that it can be problematic to have students participate or even identify themselves in future participation in a community of practice whose practice is system development. This is especially true in the case of introductory programming courses, because many of the students participating in these courses do not want to major in computer science, but in related fields (e.g., one of the courses studied has participants from computer science, mathematics, geology, nano-science, mathematical economics and multimedia). However, e.g., Guzdial and Tew (2006) describe how they have created a community of practice for their students to be involved in; students who are not majoring in computer science. Others have taken LPP as their starting point and discussed how schooling can be seen as more or less LPP. One example is Joseph and Nacu (2003), who talk about instruction as being "*aligned*" if students recognise that their school activity leads toward a community of practice they value. This implies that students participating in a course whose goals and purposes cannot be understood as relating to a long-term goal (e.g., becoming a member of a community of practice) will not be motivated to learn. Likewise, Shaffer and Resnick (1999) also discuss authenticity. They describe different kinds of authenticity that are necessarily directly related to a workplace.

The goal of university courses is not the acquisition of skill, but academic competencies; the learning takes place in an educational institutional setting, not a workplace. Collins, Brown and Newman (1989) have described a form of apprenticeship they term "cognitive apprenticeship":

> We call this rethinking of teaching and learning in school "cognitive apprenticeship" to emphasize two issues. First the method is aimed primarily at teaching the processes that experts use to handle complex tasks. Where conceptual and factual knowledge are addressed, cognitive apprenticeship emphasizes their uses in solving problems and carrying out tasks …
>
> Second, our term, cognitive apprenticeship, refers to the learning-through-guided-experience on cognitive and metacognitive, rather than physical, skills and processes […] The externalization of relevant processes and methods makes possible such characteristics of apprenticeship as its reliance on observation as a primary means of building a conceptual model of a complex target skill. (p. 457)

Ben-Ari (2005) recognizes the usefulness of cognitive apprenticeship:

> An example of the benign use of insights from situated learning is cognitive apprenticeship (Collins et al. 1989), in which teachers attempt to present students with activities that introduce students to the nature of actual practice; for example, the nature of writing is that the written text must be analyzed and criticized, and the nature of mathematics is that results come from a struggle with unfruitful attacks on a problem. Having students assume the role of

> mathematicians may be excellent pedagogy, but only in carefully crafted activities that bear little resemblance to actual practice. (p. 374)

To program is to create a program with certain features. It is a process that requires certain skills. These skills are not necessarily well-known and explicitly described, but they are skills that experienced programmers possess. Traditional pedagogical practices do not show students key aspects of expertise– "too little attention is paid to the reasoning and strategies that experts employ when they acquire knowledge or put it to work to solve complex or real-lift tasks" (Collins et al., 1991, p. 6). One example of this type of problem was noted by Schoenfeld (1985), who found that students relied on surface knowledge and standard textbooks for "one direct way from problem to goal." When the problems the students face do not match the standard pattern, they are lost about what to do.

According to Collins et al. (1991), traditional apprenticeship has four important aspects: modelling, scaffolding, fading and coaching.

Modelling is

> supposed to give models of expert performance. This does not refer only to an expert's internal cognitive processes, like heuristics and control processes, but also to model the expert's performance, tacit knowledge as well as motivational and emotional impulses in problem solving (Järvelä, 1995, p. 241).

Scaffolding is support given by the master to the 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).

Fading is – as the word suggests – the master pulling back more and more, leaving the responsibility for performing the task more and more to the apprentice.

Coaching is the entire process of apprenticeship – overseeing the process of learning. The master coaches the apprentice in many ways: choosing tasks, giving feedback, challenge the apprentice, encouraging the apprentice, etc.

Collins et al. (1991) have described how these aspects can be used in a more traditional, school-based educational system. In short, "In order to translate the model of traditional apprenticeship to cognitive apprenticeship, teachers need to:

- identify the process of the task and make it visible to students;

- situate abstract tasks in authentic contexts so that students understand the relevance of the work; and
- vary the diversity of situations and articulate the common aspects so that students can transfer what they learn.” (p. 8)

The teachers must give the right level of support while the learners are solving problems. The support depends on the learners’ previous knowledge, working habits etc. It is important that the teacher builds up knowledge about the individual learner in order to give support. An important issue in cognitive apprenticeship is the learners' reflection on the differences between their knowledge and the reflections made by the master and fellow learners.

Several examples of cognitive apprenticeship in elementary school are described – e.g., Palinscar and Brown (1984) describe how reading can be taught using cognitive apprenticeship; and Scardamalia, Bereiter, and Steinbach (1984) have developed an approach to teaching writing based on elements of cognitive apprenticeship. In college, Schoenfeld’s (1985) way of teaching mathematics is an example.

Schulte, Magenheim, Niere, and Schäfer (2003) describe a learning environment for introducing object-oriented technology in upper secondary schools based on a cognitive apprenticeship pedagogy. They have a strong focus on the modelling aspect of object-orientation and introduce object-oriented technology by two tools: Fujaba (a CASE tool that generates executable Java code from a class and behaviour UML-diagram, (Fujaba, 2007)) and dobs (a plug-in for BlueJ (Kölling, Quig, Patterson, & Rosenberg, 2003) that shows the dynamic object structure of a running program, (Dobs, 2005)). Schulte et al. found a very positive influence of their learning environment and their pedagogical model using fading guidance.

In 2002 David Gries wrote an invited editorial to Inroads. In his editorial, he reflected on teaching programming:

> 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. The ongoing switch from C/C++ to Java in introductory courses is a great step forward […]

> 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. (Gries, 2002, p. 5)

From the above we conclude that cognitive apprenticeship is a relevant pedagogical theory to apply when designing an introductory course. However, as described by Hiim and Hippe (see section 4.1), many other factors influence the concrete design of a course. One of the typical general conditions in university teaching is the use of large lecture halls or – in the case of further education – the use of e-learning. Consequently, we need to explore and investigate ways to divulge the programming process under different conditions and explore ways to apply a cognitive apprenticeship inspired pedagogy under such conditions.

4.7 Freshmen vs. Professionals

Teaching experienced students enjoys a long tradition. In 1926 the American Association for Adult Education was formed. According to Merriam (2001), the two most important theory-building efforts within the field of adult education have been andragogy (Knowles, 1968; Knowles, 1980) (a learner-centred method as contrasted to a traditional teacher-oriented method) and self-directed learning (Knowles, 1975; Tough, 1967; Tough, 1972). Even though these two theories have had a major impact on adult education research, much debate has arisen about whether anything actually differentiates adult learners from young learners. As Merriam (2001) noticed:

> … ongoing to this day, is the extent to which the assumptions are characteristic of adult learners only. Some adults are highly dependent on a teacher for structure, while some children are independent, self-directed learners. The same is true for motivation (p. 5).

Imel (1989) summarized research on educators who say they believe in using an andragogical approach and concluded that they did not necessarily use a different style when teaching adults. Beder and Darkenwald (1982) found from a study of teachers teaching both adults and pre-adults that the teachers found

> adults were perceived as more intellectually curious, more concerned with the practical applications/implications of learning, more motivated to learn, less confident in their ability as learners, more willing to take responsibility for their own learning, clearer about what they want to learn, more willing to work hard at learning, and less emotionally dependent on the teacher. (p. 152)

Gorham (1985) actually observed teachers that taught both adults and pre-adults. She found no difference in how they instructed adults and pre-adults, even though they said they taught the two groups differently. One way to approach the question of whether teaching adults are different is by examining the types of learning in which adults engage. Cranton (1994) classified adult learning into three categories:

> **Subject-oriented adult learning.** The primary goal is to acquire content. The educator "speaks of covering the material, and the learners see themselves as gaining knowledge or skills" (p. 10).
>
> **Consumer-oriented adult learning**. The goal is to fulfil the expressed needs of learners. Learners set their learning goals, identify objectives, select relevant resources, and so forth. The educator acts as a coach "and does not engage in challenging or questioning what learners say about their needs" (p. 12).
>
> **Emancipatory adult learning**. The goal is to free learners from the forces that hinder their opportunities and control their lives; forces that they have taken for granted or seen as beyond their control. Emancipatory learning results in transformations of learner perspectives through critical reflection (Mezirow, 1991). The educator plays an active role in fostering critical reflection by challenging learners to consider why they hold certain assumptions, values, and beliefs.

Of the three types of adult learning, only emancipatory has been described as unique to adulthood since "it is only in late adolescence and in adulthood that a person can recognize being caught in his/her own history and reliving it" (Mezirow, 1981, p. 11)). According to Imel (1995), subject-oriented learning is the most common form of learning engaged in by youth. Collaborative and cooperative learning and other types of experiential learning that are more consumer-oriented are also found in youth classrooms.

The goal of the introductory programming course is not to free learners from forces that limit their opportunities; it is to learn object-oriented programming. From this perspective – the type of learning adults engage in – there is no difference between young or adult learners.

Examining what adult learners expect from teaching provides another perspective on whether teaching adults is different. Donaldson, Flannery, and Ross-Gordon (1993) have combined and reanalyzed research they previously have done individually that examined adult college students' expectations of effective teaching and compared them with those of traditional students. They found that the six most frequently mentioned attributes adult learners expected of effective instructors were:

- to be knowledgeable
- to show concern for student learning
- to present material clearly
- to motivate
- to emphasize relevance of class material
- to be enthusiastic (p. 150)

The adults had preferences for both student-centered (e.g., concern for student learning) and teacher-centered (e.g., knowledgeable) instruction. When they compared adult expectations for good teaching with those of traditional students, many similarities existed in how the two groups characterized good teaching. However, four teacher characteristics mentioned by adults were not among the top items for undergraduates:

- creates a comfortable learning atmosphere
- uses a variety of techniques

- adapts to meet diverse needs
- dedicated to teaching

Based on this description it seems natural to ask the question:

> Is teaching adults different? Based upon the literature discussed here, the answer is both *yes* and *no*. Perhaps a better way to frame the question would be '*Should* teaching adults be different?' The answer to that would, of course, depend upon the purpose of the teaching-learning situation, including which approach and methods seem to be appropriate, as well as the needs of the learners. (Imel, 1995, p. 4).

In this dissertation we are not interested in the general differences between adult learners and young learners but in the way that professionals with prior programming experience learn object-oriented programming. In particular, we are interested in how they connect different aspects of a course (content, tools used and pedagogical model) to their working life.

5 Method

It is common sense to take a method and try it.
If it fails, admit it frankly and try another.
But above all, try something

Franklin D. Roosevelt (182-1945)

This chapter gives an account of the research method employed in this PhD project in order to answer the research questions. As summed up in section 1.2, they were:

> R1: Is it possible to make a course design where the students learn a systematic programming process, conceptual models as a structuring mechanism, and coding?
>
> R2a: How do the different aspects of a model-based programming course influence professionals' practice?
>
> R2b: What are the interaction patterns for professionals learning object-oriented programming with a focus on a systematic programming process and conceptual models as structuring mechanisms in a technology rich distance education setting?
>
> R3: Do model-based introductory programming courses have the same success factors as more traditional courses?
>
> R4a: What are the awareness of, use of, and attitude toward pedagogical patterns among computer science university teachers around the world?
>
> R4b: How can pedagogical patterns be structured to make them more useful for university teachers?

Section 5.1 describes the research design and its rationale based on the research questions. Section 5.2 describes the empirical data used. Some research questions are best answered using quantitative methods, others using qualitative methods or combination of both. The last two sections discuss the trustworthiness of the quantitative and qualitative research.

5.1 Research Design

Some of the research questions can be answered using an empirical approach while others are more analytic or constructive in nature. The more analytic or constructive research questions are R1 and R4b, whereas the rest requires empirical methods in order to answer the questions.

5.1.1 Analytic/Constructive Questions

This subsection discusses the two research questions requiring the creation of something.

5.1.1.1 Model-Based programming course

The first research questions (R1):

> R1: Is it possible to make a course design where the students learn a systematic programming process, conceptual models as a structuring mechanism, and coding?

requires several elements in order to give an answer. The first is a course design with a focus *on systematic programming process*, *conceptual models* as a structuring mechanism and *coding*. If it is possible to create such a design, the prerequisite for answering "yes" to the question is fulfilled. However, we still need to make sure that the students actually learn the goals. This is traditionally done at the exam, so if the exam measures the goals of the course (i.e. systematic programming process, conceptual models as a structuring mechanism and coding) and the results of the exam is at least as god as traditional exam results, then we will be able to answer "yes" to R1.

5.1.1.2 Structuring of pedagogical patterns

Research question

> R4b: How can pedagogical patterns be structured to make them more useful for university teachers?

can be approached in several ways. One possibility is the use of design methods inspired by participatory design (Schuler & Namioka, 1993), i.e. design processes that include the teachers as active participants in the process. However, as argumented elsewhere, the group of university teachers is large and hard to access;

consequently we find it hard to find representative teachers. Therefore we will use an approach where pedagogical theory is the guide for the development and the result subsequently can be evaluated by the users.

5.1.2 Empirical Questions

Information Systems (IS) research has its roots in natural sciences, and is dominated by positivist approaches to research relying on quantitative methods (Chen & Hirschheim, 2004; Goles & Hirschheim, 2000; Orlikowski & Baroudi, 1991). This rooting of IS research has had a major impact on the research methods applied in computer science education research (Fincher & Petre, 2004; Lister, 2005b). However, the clear-cut distinction between research being either qualitative or quantitative is fading (Alvesson & Skoldberg, 2000; Creswell, 2002; Lister, 2005b) and mixed methods are becoming more frequent in research in general (Johnson & Onwuegbuzie, 2004) and in computer science education research (Lister, 2005b) in particular.

A research design entails several decisions. Creswell (2002, p. 5) conceptualized these decisions in the following three questions:

1. What knowledge claims is the researcher making?
2. What strategies of inquiry will inform the procedures?
3. What methods of data collection and analysis will be used?

He (Creswell, 2002) describes four alternative knowledge positions: *Postpositivism* (also called quantitative research), *Advocacy/Participatory*, *Constructivism*, and *Pragmatism*.

Positivist science is grounded in a correspondence theory of knowledge and a realist ontology (Nielsen, 1990). This view presumes an objective reality apart from a subjective person who can use language and symbols to accurately describe and explain the truth of this objective reality. The origin of this realist orientation is usually credited to Descartes, but it can, in fact, be traced as far as back Galileo, who stated that “whatever cannot be measured and quantified is not scientific” (Capra, 1989, p. 133). Knowledge must be attained through an objective distance from the world, and if this distance is not maintained, the risk of tainting reality with our own subjective beliefs and biases arises (Guba, 1990; Heshusius, 1994).

Creswell (2002) uses the word “postpositivism” to indicate a foundation in a positivist view, but with the additional stipulation that there cannot be an absolute truth of knowledge.

The advocacy/participatory knowledge claim arose during the 1980s and 1990s. Researchers who found that the traditional view by postpositivist knowledge claims did not fit marginalized individuals or groups advocated intertwining the research with a political agenda (Fay, 1987; Heron & Reason, 1997; Whyte, 1989). “Thus, the research should contain an action agenda for reform that may change the lives of the participants, the institutions in which the individuals work or live, and the researcher’s life” (Creswell, 2002, pp. 9-10).

The constructivist knowledge claim holds the position that meanings are constructed by humans engaging with the world. It arises from works such as Berger and Luckmann (1969) plus Lincoln and Guba (1985). The generation of meaning is always done in a social context. The researchers acknowledge that their own personal, cultural and historical background influences the interpretations, and they position themselves within the research field.

The pragmatic knowledge claim derives from the work of, among others, C.S. Peirse, William James, George Herbert Mead, John Dewey and Cornel West (Cherryholmes, 1992, p. 13). The focus is on “what works” and solutions to problems (Patton, 1990). As Creswell (2002) writes, “Truth is what works at the time; it is not based on a strict dualism between the mind and a reality completely independent of the mind” (p. 12). A mixture of methods – both quantitative and qualitative – is therefore used.

In general, there have been two inquiry strategies within educational research: quantitative (Ayer, 1959; Maxwell & Delaney, 1999) and qualitative (Lincoln & Guba, 1989; Smith, 1984).

The quantitative approach uses primarily postpositivist claims to build knowledge (e.g., tests of theories and hypotheses, cause and effect and measurement of specific variables). It employs strategies of inquiry such as experiments and surveys using predetermined instruments giving statistical data (Gall, Gall, & Borg, 2003). The most common quantitative research techniques include (Creswell, 2002):

- Observation

- Experimentation
- Surveys

Since the late sixties, qualitative research in education has flowered (Bogdan & Biklen, 1982, p. 3), but it has a long history from, among other things, anthropology and sociology (Bogdan & Biklen, 1982). In the qualitative approach, the researcher often uses a constructivist knowledge claim to build knowledge (e.g., knowledge is socially constructed with multiple meanings of experiences). The intent is often to build a theory from the data. The researcher collects open-ended, emerging data. The most common qualitative research techniques include:

- In-depth interviews
- Focus groups
- Projective methods
- Case studies
- Pilot studies

In the last two or three decades an increasing number of researchers have argued that it is better to combine these two philosophies than to use only one of them, in order to better answer the complex research questions in, e.g., educational research (see e.g., Angen (2000); Howe (1988); Johnson and Onwuegbuzie (2004) plus Onwuegbuzie and Leech (2005)). As Johnson and Onwuegbuzie (2004) notice:

> We hope the field will move beyond quantitative versus qualitative research arguments because, as recognized by mixed methods research, both quantitative and qualitative research are important and useful (p. 14).

This combination is known as a mixed method approach or a compatibility dissertation:

> The compatibility thesis supports the view, beginning to dominate practice, that combining quantitative and qualitative methods is a good thing and denies that such a wedding of methods is epistemologically incoherent. On the contrary, the compatibility thesis holds that there are important senses in which quantitative and qualitative methods are inseparable. (Howe, 1988, p. 10)

This definition is compliant with the definition of Johnson and Onwuegbuzie (2004) (italics in the original):

> Mixed methods research is formally defined here as the class of research where the researcher mixes or combines quantitative and qualitative research techniques, methods, approaches, concepts or language into a single study. (p. 17)

In this work, the knowledge will be claimed using a pragmatic view with a mixed method approach. The argument for using this approach is the complexity of the research questions. It will not be physically one single study, however, but several combined studies aimed at enlightening different aspects of the research questions.

The research questions R2a, R2b and R3 address different groups of students. They involves both testing of the hypothesis that traditional success factors for learning introductory programming also apply to model-based courses, and more elaborative elements such as interaction patterns and ways to link the educational setting to the learners' own work practice. In the other part of the research question, the goal is to generate new knowledge and a qualitative approach will be used (articles "Learning Object-Orientation by Professional Adults "and "Examining Social Interaction Patterns for Online Apprenticeship Learning – Object-oriented Programming as the Knowledge Domain"). This will also give several views on the same research question, thereby enhancing the findings' validity. This is in line with the position of Johnson and Onwuegbuzie (2004):

> What is most fundamental is the research question – research methods should *follow* research questions in a way that offers the best chance to obtain useful answers. Many research questions and combinations of questions are best and most fully answered through mixed research solutions (pp. 17-18).

As they note:

> For example, the major characteristics of traditional *quantitative* research are a focus on deduction, confirmation, theory/hypothesis testing, explanation, prediction, standardized data collection and statistical analysis …The major characteristics of traditional *qualitative* research are induction, discovery, exploration, theory/ hypothesis generation, the researcher as the primary "instrument" of data collection and qualitative analysis (p. 18).

Research in an educational setting is a challenge. As Daniels, Petre and Berglund (1998) conclude:

> Overcoming the constraints of the educational setting is often a matter of combination: accumulating results from a series of studies, compiling results from studies at different sites; mixing qualitative and quantitative techniques […]. Combining methods allows a sort of 'triangulation' among multiple perspectives, which can achieve a more complete account. Consolidating results from different sources can improve representativeness and strengthen conclusions, as long as the data arise from cognate studies. (p. 205).

The following is a discussion and argumentation for the research design for each question.

5.1.2.1 Influence of a model-based programming course on professional practice

The research question

> R2a: How do the different aspects of a model-based programming course influence professionals' practice?

is targeting professionals with prior knowledge in programming, but not necessarily object-oriented programming. The research question is explorative in nature aiming at examining the learner's own experiences of the effects of transforming their achieved and new knowledge to their daily work situations. Change of working habits takes time, so the gathering of data must be with students who have participated in a model-based course and been back in their daily work situations for some time. Several possibilities exists: observing people using ethnographical methods, analysing old and new code in order to find if there are traces of object-oriented concepts in the code after the course, or interviewing students who have participated. Due to time constraints and the unwillingness of the students and their employers to let us observe them in their working situations, ethnographical studies were excluded. Inspection of source code reveals only the impact of object-orientation on some of the artefacts one hope to address; the more soft elements (discussions, the way of thinking about programming) are not addressed. Furthermore the students involved worked in teams on projects that had been ongoing for some time and consequently it would be very hard to identify the impact since not all team members had taken a model-based course. Consequently, semi-structured interviews were chosen.

In order to address research question R2a, we have interviewed professionals participating in a model-based introductory course right after the course. The interviews addressed among other things the connection between practice and participation in a model-based introductory course. However, the data were collected right after the course ended and therefore the impact of learning was not known. Therefore we decided to re-interview the students approximately one year after the course was taken.

The next element to address was which “aspects” to look for? One of the foci of the interviews done right after the exam was the connection of the course and the students’ professional situation. Analysing these interviews, noting topics mentioned by the interviewed professionals in relation to their working life, extracts the students’ own focus points in the transfer from learning to practice. We categorised the topics mentioned: *content*, *tools* and *social communities*. *Content* addresses the content of the course: programming language, assignments, examples etc. *Tools* addresses the tools used in the course: BlueJ, instant messaging, videos showing the programming process etc. *Social communities* included the pedagogy in the course (cognitive apprenticeship), connections to other professionals etc. Based on these three categories, we designed an interview protocol according to the guidelines given in (Selltiz, Wrightsman, & Cook, 1976) and to be used with a positivist approach (Silverman, 2001, p. 90) in order to acquire information about the transfer of knowledge from the course setting to the students’ daily working practices.

Since I was the teacher of the course, we decided to have another person to perform the interviews in order not to influence the students. The interviews were audiotaped. The analysis consisted of categorising the answers according to the three categories (content, tools, and social communities) and then analysing these categorized answers for the impact that the students said they had on their professional practices.

5.1.2.2 Interaction patterns

The research question

> R2b: What are the interaction patterns for professionals learning object-oriented programming with a focus on a systematic programming process and conceptual models as structuring mechanisms in a technology rich distance education setting?

is to a large extend dependent on the 'e-learning setting' studied. One extreme is a course that is merely an electronic version of traditional correspondence school; the other extreme is a course that uses the new possibilities technology give. The decentered aspect of the apprenticeship-based pedagogy applied in IOOP03 was found to be almost non-existing; that form of distance learning environment did not foster much collaboration ("Examining social interaction patterns for online apprenticeship learning – Object-oriented programming as the knowledge domain"; Berge & Fjuk, 2006a). Consequently, another form of technology must be used and evaluated, a form that focus much more on collaboration and the decentered aspect of apprenticeship. Web-based collaboration tools seem to be able to support this; in such tools it is possible to (in something close to real-time) collaborate by talking, sharing applications (e.g., BlueJ), exploring web-pages, make drawings, etc. As a consequence, a design experiment (Brown, 1992) was established with the focus of using a particular web-based collaboration tool to enhance the decentered aspect of the apprenticeship inspired pedagogy and study the interaction that took place between the learners in such a setting. Design experiments differ in interesting ways from both laboratory experiments and naturalistic investigations. Ludvigsen and Mørch (2003) put it very clearly:

> Design experiments can be seen as intervention in the educational practice because the researchers, in collaboration with teachers, try to change the way the students work. These shifts often presuppose a change in participation structures and how agency and division of labour are distributed between the teacher and the students. (p. 67)

The activities were recorded and analysed. In addition to observations, the participating students were interviewed in groups after the on-line session.

In this research question the focus in on distance education. In a distance education setting the possibility to demonstrate the programming process is different, e.g., it is not possible to do live coding in the lecture theatre. Consequently, an alternative way needs to be defined. In this research, streaming video is used, where the students, sitting at home, sees the teachers screen and hear his voice as he thinks aloud demonstrating his approach to the problem at hand. Doing so makes it possible to store the recordings and make them available for later retrieval. Is this way of exposing the programming process experienced by the students as being relevant and useful? Do they actually watch it? Does it reveal the programming process? In order to answer these questions – and consequently extend the possible “yes” to R1 to include distance learning and thereby be able to answer R2b – an evaluation of the use and attitude towards these process recordings must be done. Since we do not have a comparable control group (the students taking IOOP are adults with previous programming knowledge whereas the students taking dIntProg are young freshmen) it is not possible to compare the two ways of revealing the programming process. Consequently, we measure the usage of and attitude towards the process recordings using a questionnaire where the students indicate their use of the process recordings and evaluate the resulting learning outcome (using a 5 item likert scale from ‘none’ to ‘very high’). In order to ensure the validity of this, a representative selection of the students will be interviewed using a semi-structured interview.

5.1.2.3 Success factors

The research question

> R3: Do model-based introductory programming courses have the same success factors as more traditional courses?

compares success factors from more traditional courses with a model-based introductory programming course. Previous studies have been conducted by statistically analyzing different variables and investigating their correlation with the final exam score. This is normally done using multiple linear regression analysis (Schroeder, Sjoquist, & Stephan, 1986) creating a formula for predicting the exam result. The linear regression formula has the form: $\lambda = \alpha + \beta_1\chi_1 + ... + \beta_n\chi_n$, where λ is the predicted variable (exam score), α is a constant displacement, and each χ_i is a predictor variable with corresponding coefficient β_i. In a multiple linear regression analysis, the coefficient of multiple correlation (R) “indicates the degree to which variation in the dependent variable is associated with variations in the several variables taken simultaneously" (Schroeder et al., 1986, p. 33). R^2 defines the percentage of impact of the regression formula as a whole or of each variable.

In order to be comparable to previous studies, a similar approach was used to identify predictors for success in a model-based introductory programming course. Based on the analysis of previous work on success factors (see section 3.2.3), we have selected the following variables to be included in the analysis in "An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course": *mathematical ability*, the students' *work* during the course, *gender*, *intended major*, number of *years at the university*, and the *team*[24]. Due to technical problems, it was not possible to include previous programming experience.

Some of the data (gender, number of years at the university, intended major and exam score) can be found in the university's administrative system. In order to evaluate the work the students do during the course, the teaching assistants were instructed to keep track of each student using a three point scale describing the students performance with respect to their weekly assignment: 'Perfect', 'OK but small errors' and 'not acceptable'. Since the university does not record the students high-school math score, the students were asked for this. Finally, the website of the course describes the team each student is assigned to.

Most authors of articles in computer science education research argue that abstraction is a core competency, but defer to describe an evaluation tool for abstraction ability. The measurement of the students' abstraction ability used in this research is based on Adey and Shayer's theory of cognitive development (Adey & Shayer, 1994; Shayer & Adey, 1981); this theory is a refinement of Inhelder and Piaget's (1958) stage theory. Inhelder and Piaget's theory included four stages: *sensorimotor*, *pre-operational*, *concrete operational* and *formal operational*. The first two stages cover infancy and early childhood (until the age of approximately seven); the next from seven to eleven is characterized by the appropriate use of

[24] All students are grouped into teams of approximately 22 students. Each team has an associated TA and spends their lab hours together.

logic. The last stage, formal operational, commences around 11 years of age (puberty) and continues into adulthood. It is characterized by acquisition of the ability to think abstractly and draw conclusions from the available information.

Adey and Shayer specialize the last three stages of Inhelder and Piaget by defining eight stages of pupils' cognitive development (Adey & Shayer, 1994, p. 30) – see Table 5.

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

Table 5: Cognitive development stages

Adey and Shayer based their stages of cognitive development on a very large research project, CASE, aimed at finding the cognitive development stages of pupils in primary and secondary schools (Adey & Shayer, 1994, pp. 78ff). Their research showed a different result from 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 path expected by Piaget, i.e. the path where it is the age of the person that defines the cognitive development. Epstein (n.d.) have, from Adey and Sheyer's work as well as experimental results in the literature, estimated the percentage of persons at each cognitive level at each age (from 1-18 and adult). Epstein as well as Kühn et al. (1977) found that only a third of adults can reason formally.

Shayer and Adey (1994) have developed several tests to determine students' cognitive development level. These tests 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 used the so called "pendulum test" which has been used for a long time to test young peoples' understanding of the laws of the physical world and thus the interaction between several variables (Bond, 2004).Adey and Shayer argue that the pendulum test is particularly focused on testing the cognitive development stages from 2B to 3B (Adey & Shayer, 1994, 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 informed that the test's outcome would not be communicated to the lecturer before the exam.

Before performing this type of empirical quantitative research, it is important to consider the required sample size so that the risk of rejecting a hypothesis when it actually is true (type II error), or the other way around (type I error) is acceptable (for a description of type I and II errors see section 5.3.4). In order to estimate the sample size, we consider three concepts:

- effect size (f^2)
- alpha level (α, or confidence level)
- power (1-ß)

Having α=0.05, 1-ß=0.9 and f^2 = 0.15 and one predictor variable, we can calculate the required sample size to be 71 (Soper, 2007). Given that there are over 235 students participating in the course, it seems reasonable to perform the statistical analysis.

As in most other studies, the result of the final exam is used as an indicator for success – higher grade, more success. As described in appendix 11.6, the score in dIntProg is binary (pass/fail) and based on a practical programming test. To provide numeric grades, we postmarked the answers. The grades of the Danish tenary scale are: 00, 03, 5, 6, 7, 8, 9, 10, 11 and 13 (Exam-scale) was used. A student needs at least a "6" to pass an exam. Table 6 shows the official conversion from the Danish scale of marks to the ECTS scale of marks (European Union, 2006a). Unfortunately, an official conversion to the North American scale does not exist; we have included our conversion.

Danish scale of marks	ECTS scale of marks	North American scale of marks
11 and 13	A	A
10	B	B
8 and 9	C	C
7	D	C
6	E	D
03 and 5	Fx	F
00	F	F

Table 6: Conversion from Danish scale of marks to A-F scale of marks

In general, the result of the grade that a student gets is solely determined by the final exam; mandatory assignments are a prerequisite for the final exam, but these assignments do not contribute to the final exam score.

5.1.2.4 Awareness, use and attitude towards pedagogical patterns

The research question

> R4a: What are the awareness of, use of, and attitude toward pedagogical patterns among computer science university teachers around the world?

deals with the hypothesis that many teachers do not know pedagogical patterns. It is a measurement of knowledge and consequently a quantitative study design seems appropriate using a questionnaire to measure the attributes. Since the respondents are from all over the world, an electronic distribution of the questionnaire is suggested. The response rate among computer science teachers are expected to be higher using this form than using a paper and pencil version.

The biggest problem is which teachers to address. The research question mentions "computer science teachers at the university level." This is a large group of people, for whom there exists no database from which to draw a random selection of people. An accessible group of teachers are the group of teachers attending specific computer science conferences. However, the attendance-list for conferences is traditionally not accessible, but the authors of articles for conferences are accessible in the proceedings. The following six groups of teachers where chosen:

- The authors of articles for Koli Calling 2004, the 4th Annual Finnish / Baltic Sea Conference on Computer Science Education (Korhonen & Malmi, 2004);
- Article authors and panellists at the 36th Technical Symposium on Computer Science Education (SIGCSE'05, 2005);
- Article authors and panellists at the 10th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'05, 2005);
- The authors of research articles at the 4th International Conference on Advanced Learning Technologies(ICALT'04, 2004);
- The authors of articles at the Australian Computers in Education Conference (ACEC'04, 2004); and
- The ACM (Association for Computing Machinery) Special Interest Group in Computer Science Education’s e-mail list (SIGCSE mail-list, n.d.).

The reason for choosing these groups is that they consist of teachers seemingly interested in computer science education (SIGCSE, ITiCSE, Koli, ACEC and the SIGCSE e-mail list), and of teachers interested in teaching technology but not necessarily interested in computer science education (ICALT). The four computer science education conferences (SIGCSE, ITiCSE, Koli and ACEC) attract teachers from all over the world. SIGCSE is held in the United States, ITiCSE in Europe and ACEC in Australia, whereas Koli is held in Finland and attracts teachers mostly from the Baltic Sea area. The participants in ICALT were from all over the world. The SIGCSE mailing list was used to address teachers who were not necessarily authors, but who still had an interest in computer science education. Section 5.3.6 will further discuss the representativity of the selected groups.

Another way to address teachers from universities around the world would have been to select representative universities and write to the faculty members of the computer science departments. The problem with this approach would have been the difficulty in finding universities spanning the differences in computer science education. Another problem would be whether the teachers who actually responded to the questionnaire would be representative.

Due to time constraints in my PhD work, we do not supplement the research with qualitative studies. This could have given more insight in what the teacher actually understand by the term “pedagogical pattern”, a more detailed understanding of their usefulness etc.

5.2 Data Sources

This section describes the different data sources used in this PhD work. As described above, a mixed method approach is used and, consequently, several data sources. The section describes the teaching context or community where the data were gathered, and the method used to gather them. Several of the qualitative studies have been undertaken in collaboration with fellow researchers in the COOL project.

5.2.1 The IOOP course

This section describes the course upon which articles “Learning Object-Orientation by Professional Adults” and “Examining Social Interaction Patterns for Online Apprenticeship Learning – Object-oriented Programming as the Knowledge Domain” were based.

The course *Introduction to Object-Oriented Programming* (IOOP) has been taught as an on-campus course at University of Aarhus in Denmark during the last decade. In 2003 it was changed to a net-based form (for a description of this transformation’s goals, see Bennedsen and Caspersen (2003) plus Fennefoss and Bennedsen (2003)).

It was a course in master programs in further education at the university level. It was part of three study programs: Master of multimedia, Master of software construction and Master of ICT and organization. Its participants were both adults with previous knowledge of programming (but not object-oriented programming) and adults with no previous knowledge of programming. The participants should have at least two years of professional experience prior to taking the course.

5.2.1.1 Goals

The official description of the course goals is[25] as follows:

> The course covers principles and techniques for systematic construction of programs, especially object-oriented programs. The course focuses on two different abstraction levels and the transition between them – on one hand a model level, with topics such as conceptual modelling, ways of structuring, and design patterns; on the other hand a programming level, whose primary focus is on how programs are realized with the help of different techniques.
>
> The students will learn fundamental programming concepts and techniques and to use object-orientation to solve advanced problems such as programming for the World Wide Web and smaller administrative systems (IOOP, 2004).

5.2.1.2 Form

The course lasted for 15 weeks. Each week included a video mediated virtual meeting, in which the teacher (or master) displayed his competence, based on problems suggested by the learners or by means of his own examples. Theoretical aspects (e.g., learning the underlying object-oriented conceptual framework) were addressed mostly in face-to-face weekend seminars, of which there were three during the course. For a more thorough description of the course, and especially of the net-based form, see Bennedsen and Caspersen (2003); "Programming in Context: a Model-First Approach to CS1" plus Berge and Fjuk (2006b).

5.2.1.3 Content

The learning objectives of the course were divided into two parts: an abstract, conceptual level and a concrete, practical level.

On the abstract level, the learners should construct an understanding of the underlying conceptual framework of object-orientation and how this framework could be expressed in an object-oriented programming language (in this case, Java™). This conceptual framework was used at three levels: (i) the domain level (or analysis level), to describe domain models, (ii) the specification level (or design level) to describe specifications models and (iii) the implementation level to code

[25] Translated from Danish by the author.

actual programs. The students are not expected to be able to create models on levels i) or ii), but only to read them.

On the concrete level the learners should be familiar not only with the elements of the chosen programming language but also with the programming process. They should know techniques and systematic ways to create programs; not just the programming language and tools, but how an experienced programmer creates, in a systematic way, a programming solution to a given problem.

It should be noted that it is a programming course, not a system development course; hence, it focuses on programming, not on creating models of problem domains. However, the interplay between the problem domain's model, the specification model and the code is studied.

5.2.1.4 Exam

Based on a program that solves one of the mandatory assignments, the student sits down by the computer and is asked to modify or extend the program. During the modification, the student and the examiner talks about why the change is made and what concepts is involved in the program and the change. The evaluation is pass/fail.

5.2.2 Case Study: IOOP 03

I was the lecturer in this instance of the course taught in fall 2003. I was also one of the two designers of the net-based version of the course. The case study was conducted by Ola Berge and Annita Fjuk, University of Oslo. For a more through description of the course and the data from this case study, see Berge (2006). Ola Berge and Annita Fjuk conducted semi-structured, audio-recorded interviews (Kvale, 2005) with nine students and the teaching assistant, each lasting approximately 30 minutes. These were carried out just after the final examination. The interviews focused on student collaboration, learning resources and the pedagogical design (especially the mix of the course's online and distance elements).

In order to evaluate the course's impact on the adult target group's professional practice, a second set of interviews were conducted with the same nine students one year after the first interviews. These interviews were also audio-recorded, and notes were taken during the interviews to support the subsequent analysis. To be as neutral as possible, a research assistant conducted the interviews by telephone using an interview protocol designed according to the guidelines given in Selltiz (1976) and using a positivist approach. The purpose was to acquire information about the transfer of knowledge from the course setting to the learners' daily working practices.

The case study's results are reported in "Examining Social Interaction Patterns for Online Apprenticeship Learning – Object-oriented Programming as the Knowledge Domain". The results of the interviews are reported in "Learning Object-Orientation by Professional Adults".

5.2.3 Design Experiment: IOOP 04

A design experiment was conducted as a follow-up activity to the case study, and took place during week six of the 15-week course in the spring semester of 2004. The aim was to explore critical issues identified during the case study with a particular focus on learners' collaboration.

The delivery situation in the spring semester of 2004 was nearly identical to the delivery situation in the fall semester of 2003 (presented in Section 5.2.2). The design, content and objectives were the same, and the learners were part-time learners committed to a daily work situation. However, the course was delivered as part of the MS program in *ICT and organization* or *multimedia* (and not in software construction as the previously). Another major difference was that a new teacher replaced me.

The purpose of this study was to explore some issues identified in the IOOP 03 case study, namely, the missing interactions in the ICT mediated learning environment. The findings from the IOOP 04 design experiment are reported in "Examining Social Interaction Patterns for Online Apprenticeship Learning – Object-oriented Programming as the Knowledge Domain".

5.2.4 The dIntProg Course

This section describes the course upon which the articles "Revealing the Programming Process", "Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming?" and "An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course" are based.

The course spanned the first half of CS1 at the University of Aarhus. The course ran for seven weeks, and two weeks after the course there was a lab examination with a binary pass/fail grading.

The instances of the courses included approximately 250 students each from a variety of study programmes, e.g., computer science, mathematics, geology, nano-science, economics and multimedia. Forty percent of the students are majoring in computer science, and they are the only group of students to continue with the second half of CS1. The rest of the students proceed to other programming courses related to their field (e.g., multimedia programming, scientific computing, etc.). As opposed to the IOOP students, the dIntProg students were traditional young, full-time students with no professional experience.

5.2.4.1 Goals

The purpose of the course was for students to learn the foundation for systematic construction of simple programs, and through this to obtain knowledge of conceptual modelling's role in object-oriented programming. Another goal was that students became familiar with a modern programming language, fundamental programming language concepts and selected class libraries.

After the course, students would have insight into principles and techniques for systematic construction of simple programs and practical experience with implementation of specification models using a standard programming language and selected standard classes.

Goals: Upon completion, the students must be able to:

- *apply* fundamental constructs of a common programming language.
- *identify* and *explain* the architecture of simple programs.
- *identify* and *explain* the semantics of simple specification models.
- *implement* simple specification models in a common programming language
- *apply* standard classes for implementation tasks.

5.2.4.2 Form

The course ran for seven weeks; every week had four lecture hours and four hours with a TA (in the instance "Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming?" was based on, there were two lab hours and two class hours. In the instance where "An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course" is based, the students had one lab hour plus three class hours). In addition to the scheduled hours, students worked approximately seven hours per week in study groups or on their own.

The four lecture hours per week was used to present and discuss general concepts and the programming process. The programming process was revealed through live programming in front of the students in the lecture theatre using a computer and projector, and through process recordings (narrated, screen-captured video recordings of program development sessions); see "Revealing the Programming Process".

Every week (except for the first) a mandatory assignment had to be handed in to the TA. The TA examined the assignments and gave personal, as well as collective, feedback to the students. Approval of five out of the six weekly assignments was a prerequisite for the final exam, but did not count as part of the grade.

5.2.4.3 Content

The course content included fundamental programming language concepts, object-orientation and techniques for the systematic construction of simple programs:

> **Fundamental programming language concepts**: variable, value, type, expression, object, class, encapsulation, control structure, method/procedure, recursion and type hierarchies.
>
> **Object-orientation**: modelling; class structures (specialization, aggregation and association); and the use of selected class libraries (in particular, collection libraries), interfaces and abstract classes.
>
> **Systematic development of small programs**: modularisation, stepwise refinement/incremental development and testing.

This is a logical listing of the content; it is *not* the order in which the content was covered. The content was covered using a spiral approach (Bergin, n.d.); for further details of the course's structure and content, see Caspersen and Christensen, (2000).

5.2.4.4 Exam

Bennedsen and Caspersen (2006b) discuss how an exam can be designed for a model-based introductory course. The described approach, which was the form used in dIntProg, takes into account both the product (i.e. the program) and the process used to create it. The lab examination has as its characteristics that it:

- provides a valid and accurate evaluation of the student's programming capabilities,
- evaluates the process as well as the product,
- encourages the students to practice programming throughout the course, and
- can be used to assess approximately 150 students per day.

Bennedsen and Caspersen claim that traditional examination forms evaluate only the product, and not the process the students use to create it.

5.3 Verification of Quantitative Studies

This section discusses the validity of the quantitative studies. The section starts with a general discussion of the validity, reliability and generalizability of the quantitative studies:

> **Validity**: the accuracy of research findings (do the variables measure what we claim)
>
> **Reliability**: consistency of research findings (will replicated measuring give the same result)
>
> **Generalizability**: are the results transferable to new contexts

Following that is a detailed discussion of the trustworthiness of the study of traditional predictors, abstraction ability and knowledge of pedagogical patterns.

5.3.1 Validity

Within quantitative research, the question is whether a variable actually measures what it is supposed to. Maxim (1999) describes four categories of validity: content validity, criterion validity, construct validity, and convergent and discriminant validity (Maxim, 1999, p. 208).

Whether an indicator refers to what it is supposed to refer to is called content validity. In all the studies, this was addressed by discussing the questionnaires with fellow researchers before using them in order to validate that the questions cover the indicators and that the indicators actually refer to what they are intended to refer to. Criterion validity "is the correspondence between an operational indicator and its latent factor or normal definition" (Maxim, 1999, p. 209). In most of the data we have used records from the university (e.g., the courses' exam results), but some data were provided by the students (e.g., their math grade). Construct validity is the correlation between two separate notations of the same construct. The issue where in doubt is "abstraction". It was measured using stages of cognitive development (Inhelder & Piaget, 1958). As discussed in the article, it is debatable whether the measurement of the stage of cognitive development is a good construct for abstraction. Convergent and discriminant validity requires at least two different measures of the same concept. Due to time constraints, we have not made different measurements of the same constructs.

5.3.2 Reliability

Within quantitative research, reliability is traditionally concerned with designing a reliable instrument. For questionnaires this boils down to the construction of questions. Fowler (1993) describes three properties that a good question has:

- The researcher's side of the question-and-answer process is entirely scripted, so that the questions as written fully prepare a respondent to answer them.
- The question means the same thing to every respondent.
- The kinds of answers that constitute an appropriate response to the question are communicated consistently to all respondents (p. 71).

In the articles that use questionnaires as a data source, the questionnaires have been discussed with fellow researchers, and pilot studies have been undertaken.

5.3.3 Generalizability

Generalization of research concerns the question of the findings' transferability from one context to another. It is an issue in quantitative research, where the answer is to have large populations in order to minimize a single respondent's effect. It is furthermore included in the statistical test, which always questions a result's statistical significance. Statistical significant means that something is probably true; it is expressed as a percentage, denoted by "p" and the rule of thumb is that p must be below 5% in order for it to be valid, i.e., there is less than a 5% risk that something occurred out of luck.

Other studies have tried to generalize to the population of either introductory programming teachers or students. "The Dissemination of Pedagogical Patterns" addresses teachers and characterizes them with respect to their knowledge of pedagogical patterns. The articles "Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming?" and "An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course" address students and try to describe general characteristics of successful students.

Our findings do not confirm previous findings (chapter 3.2.3); but that fact alone does not allow us to conclude that they contradict earlier findings. That conclusion would be valid only if previous findings were generalizable to our context (or vice versa), which we cannot be sure of. A number of circumstances are sufficiently different to prevent an immediate generalisation.

Even though ACM and IEEE have made curriculum recommendations, it seems like there is no universally agreed-upon list of topics that need to be taught in an introductory programming course. This implies that studies in one CS1 course may not be generalizable, since the content of the course differs from the content of other CS1 courses. This diversity has, as described in section 3.2.1, lead to many articles at conferences which describe various approaches to a CS1 course. Even in 2007, there are workshops with the focus of defining the goals of a CS1 course (e.g., "Every year CS departments are faced with addressing the following issues when they plan their CS1 courses, some issues being unique to liberal arts schools and their goals: 1: What are the goals we want to achieve with our students? …" (Krone & Bressoud, 2007)). Schulte and Bennedsen (2006) have tried to create a general, worldwide picture of teachers' opinions about what should be taught in introductory programming courses. This was done by focusing on three aspects: what topics teachers believe is relevant to teach (*relevance*), what they actually teach (*level*) and what they think students find most difficult (*difficulty*). In addition, they explored how these specific topics fit into a larger conceptual classification. Earlier studies of topics taught in introductory programming (Dale, 2006; Dale, 2005; Milne & Rowe, 2002) focus only on one dimension of a given topic – either its relevance or difficulty. From the study, Schulte and Bennedsen conclude that teachers covering object-oriented topics in general find the material easier for their students, and tend to include more topics, than teachers not covering object-oriented topics do. Furthermore, they conclude that there is a big variation on what topics a CS1 course contains. This underpins the restraining of generalizing from one study to another.

Very many of the studies done in computer science research are done in the United States. Freshmen at Danish universities are generally a few years older than freshmen at United States colleges. The difference in age is likely to imply a difference in maturity and degree of dependency on parents which may be reflected in the way students responded to the questionnaire.

In Denmark, university education is free; furthermore, students receive financial support from the state. In the United States the average cost of a private four-year college was 23,940 USD per year (U.S. Department of Education, 2003) and 9,828 USD per year in a public four-year college in 2003. Furthermore, parents often contribute some or all of this cost, and support their children financially as well. The resulting stronger dependency between United States students and their parents may cause United States students to feel stronger pressure from their parents whether the pressure is real or not.

The culture in Denmark is different from the culture in the United States. Denmark is a land of homogeneity and equality much more than the United States. In Denmark it is considered important that everyone has equal rights and opportunities regardless of social background. Also, it is considered inappropriate if someone stands too much out from the crowd (there is a prevailing who-do-you-think-you-are attitude often caricatured in Danish literature); in the United States, this character of personality seems to be encouraged and highly appreciated. Cultural differences make it questionable whether it is possible to generalize United States findings to a Danish context (and vice versa); consequently, we cannot conclude a contradiction with earlier findings.

5.3.4 Traditional Predictors

In order to use the multiple regression model, five prerequisites need to be fulfilled: *linearity*; *normal distribution*; *homoscedasticity* (the conditional distribution has constant standard deviation throughout the range of values of the explanatory variables); *no collinearity* (two or more variables have a strong linear relationship, i.e. explains the same); and *no problematic outliers* (an observation falls far from the rest of the data and the mean is highly influenced). Scatter-plot of the data used for “An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course” indicates that the requirements are met. Levene’s tests was performed for the variables (mathematical ability, the students’ work during the course, gender, intended major, number of years at the university, and the team) and showed homogeneity of variances. Pearson correlation tests show that none of the variables have strong linear relationships apart from team and intended major as described in appendix 11.6.

Using regression analysis raises the question of the statistical power of the results. In other words, given the results, what are the odds of confirming the hypothesis correctly? Traditionally, this is discussed according to these four concepts (Cohen, 2003):

sample size (n). The number of respondents (students) accessible to the study

effect size (f^2), or the salience of the treatment relative to the noise in measurement. 0.02, 0.15, and 0.35 are considered small, medium, and large, respectively (Cohen, 2003).

alpha level (α, or confidence level), or the odds that the observed result is due to chance. Traditionally 0.05 (Mazen, Graf, Kellogg, & Hemmasi, 1987)

power (1-ß), or the odds that you will observe a treatment effect when it occurs. Traditionally 0.8 (Cohen, 1988)

Given the values of three of the concepts, it is possible to calculate the fourth.

There can be two types of errors: We can reject a hypothesis when it actually is true (called a type II error, denoted ß), or we can accept a hypothesis when it is actually wrong (called a type I error, denoted α). In Table 7, a description of the two types or errors can be found (the hypothesis H_1 is that there is a relation, the null-hypothesis (H_0) is that there is no relation).

In reality ⇨ What we conclude ⇩	H_0 is true (i.e. there is no relationship – our hypothesis is wrong)	H_1 is true (i.e. there is a relationship – our hypothesis is correct)
We accept H_0 (i.e. there is no relationship – our hypothesis is wrong)	1- α **The confidence level** The odds of saying there is no relationship when there is none. The odds of *correctly rejecting* the hypothesis. Traditionally at least 0.95	β **Type II error** The odds of saying there is no relationship when there is one. The odds of *incorrectly rejecting* the hypothesis. Traditionally at most 0.2
We accept H_1 (i.e. there is a relationship – our hypothesis is correct)	α **Type I error** The odds of saying there is a relationship when there is none. The odds of *incorrectly confirming* the hypothesis. Traditionally at most 0.05	1 – β **Power** The odds of saying there is a relationship when there is one. The odds of *correctly confirming* the hypothesis. Traditionally at least 0.8

Table 7: Type I and type II errors

A good description of the relation between the four concepts can be found at The Decision Matrix on Trial (Becker, 1995) web-page where the OJ Simpson trial is used to explain the impact of manipulating the different concepts and the effect it have on the validity of the verdict.

We can calculate the power given the observed correlation (0.241), number of students (220) and significance (0.001) using the type two error rate calculator (Soper, 2007). This gives a β almost equal to zero; consequently the type I and type II errors in my study are almost non-existing and the results can be considered to be trustworthy.

5.3.5 Abstraction Ability

One question is whether Adey and Shayer’s scale can evaluate the group that was this study’s focus (young people approximately 20-24 years old). Adey and Shayer’s study was done in primary and secondary schools in England, with pupils in the age range from six to 16. The concern is that the variation of results will be very limited and therefore not useful in the statistical analysis. Epstein (n.d.) concludes that, in general, only 34% of the eighteen year old persons have reached the formal operational level, so for the general population this scale will give results that can be statistically analysed.

Others have used Adey and Shayer’s scale or the more course grained scale by Inhelder and Piaget on students in college, e.g., Nielsen and Thomsen (1983) plus McKinnon and Renner (1971). Nielsen and Thomsen found that many students in Danish colleges have problems with theoretical exercises and found that their cognitive development stage could reliable be measured by Adey and Shayer’s test. McKinnon and Renner as well as Hudak and Anderson (1990) tested American college students for the Piagetan cognitive development level. We therefore conclude that the scale is useful for the selected group of first year university students.

The distribution of the observed cognitive development level as measured by the pendulum test is a number between 5 and 10. The actual distribution can be seen in figure 10.

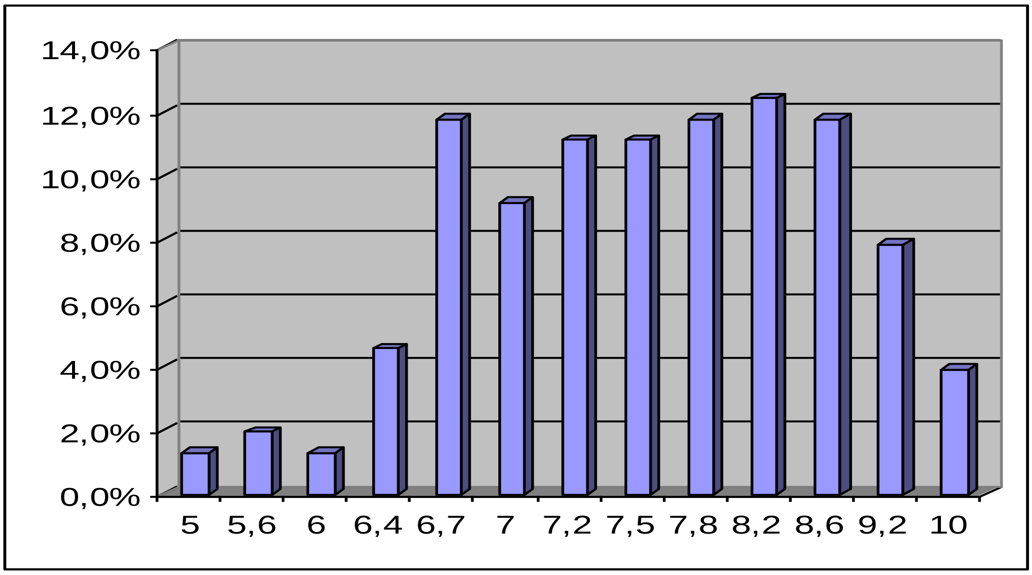


Figure 10: Distribution of cognitive development

A Kolmogov-Smirnov test verifies that the distribution can be described as normal. The test instrument used did give scores for cognitive development useful for statistical analysis.

5.3.6 Knowledge of Pedagogical Patterns

According to the International Association of Universities (International handbook of universities, 2005), there are approximately 9.000 universities all over the world. Added to that is a large number of colleges (in United States alone, there are almost 7.000 universities and colleges (National centre for educational statistics, 2007)) Unfortunately, we do not have world wide numbers of the percentage of universities with a computer science program, but in the United States, 649 institutions offer a bachelors or master degree in computer science or mathematics and computer science (National centre for education statistics, 2007), i.e. 10% of the universities and colleges offers a computer science degree. If each university and college with a degree in computer science has 20 faculty members, the total population of computer science university teachers in the United States is approximately 18,000. How large should the sample size be? This depends on the level of precision (if one finds that 30% of all teachers know of pedagogical patterns and the precision is 5%, then the conclusion is that between 25% and 35% know of pedagogical patterns), the confidence level (if a 95% confidence level is selected, 95 out of 100 samples will have the true population value within the range of precision) and variability (the more heterogeneous a population, the larger the sample size required to obtain a given level of precision. Note that 50% indicates a greater level of variability than either 20% or 80%. This is because 20% and 80% indicate that a large majority do not or do, respectively, have the attribute of interest.). Yamene (1973) provides a formula to calculate the sample size (given variability of 0.5 and a confidence level of 0.95):

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

Formula 1: sample size

where *n* is the sample size, N is the size of the population and *e* is the level of precision. In our case this gives

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

It is debatable whether the selected groups are representative of computer science university teachers. It might be argued that teachers who participate in conferences focused on teaching have a stronger interest in the field than those who do not. However, we use the results as indicators of trends, and not as definite measurements of, e.g., knowledge of pedagogical patterns. We do not claim that this study is representative of all computer science teachers, but it is as representative as is practically possible, and is therefore useful as an indicator of awareness.

358 answered the questionnaire. The required sample size was 391, so the precision of the research is not as high as requested. Using Formula 1, the precision is 0.053. If the group of participants were selected randomly, then the conclusion would be that between 41% and 51% of the teachers know pedagogical patterns. However, as argued above, the participants are not selected randomly.

The respondents came from most of the world. In figure 11 the geographical distribution of respondents is given. As can be seen from the figure, a major part was from the United States, but in general the distribution matches the expectation for the number of teachers around the western world; we would expect that countries like India should be represented in order to be representative for the entire globe. However, as described in section 5.1.2, there is no knowledge of the numbers of computer science university teachers in the different countries.

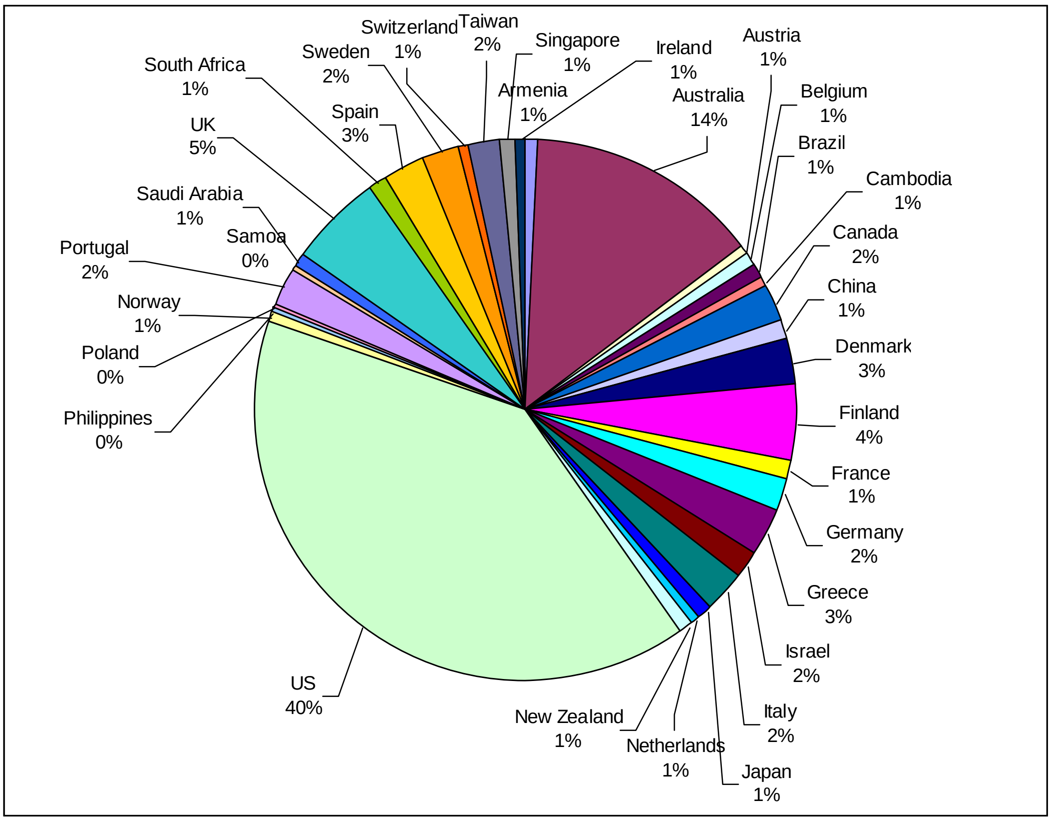


Figure11: The geographical distribution of respondents for knowledge of pedagogical patterns

5.4 Verification of Qualitative Studies

The discussion of verification of knowledge for qualitative studies is – like quantitative studies – traditionally discussed in terms of *validity*, *reliability*, and *generalizability*. We reflect on the qualitative research studies in terms of these three issues.

5.4.1 Validity

In quantitative research, validity is about whether a variable measures what it is purported to measure (Maxim, 1999). In qualitative studies, validity refers to the research findings' accuracy (Creswell, 2002), involving issues of truth and the nature of knowledge. Hammersley (1990) defines it as: "truth: interpreted as the extent to which an account accurately represents the social phenomena to which it refers" (p. 57).

Case studies can address validity by using multiple sources of evidence (Yin, 2003, p. 90) and thereby applying data triangulation. This technique was used in the IOOP03 study. Another technique to increase validity is respondent validation; to take the findings back to the subjects being studied for review (Silverman, 2001). The IOOP03 study's findings were done mainly by Ola Berge and Annita Fjuk, but we have discussed the findings extensively. Furthermore the provisory findings were reported back to the participants during the course.

The extent of the data material is relevant to the validity of the research – is the data material sufficient for exploring the topics of interest? Kvale's advice is to "interview as many subjects as necessary to find out what you need to know" (Kvale, 2005, p. 108). Silverman (2001) advocates "analytic induction," which "boils down to two simple techniques:

- the use of the constant comparative method
- the search for deviant cases" (p. 238)

The IOOP 03 study did inform the IOOP 04 study in the sense that what we observed in the IOOP 03 (namely, a missing utilization of the de-centred apprenticeship facet) was used as a starting point for the design experiment. In the interview after the design experiment, we focused, among other things, on the students' views of de-centred apprenticeship.

The validity of the research is also bolstered by providing information on the research process leading to the findings, by providing rich descriptions of the research setting when conveying the findings, and by providing extracts of the interactions that are subject to analysis.

5.4.2 Reliability

Some authors argue that the question of reliability arises only in the quantitative research tradition (Marshall & Rossman, 1989, p. 147; Silverman, 2001, p. 226). However, Silverman (2001) advocates striving for an approach with high reliability in qualitative research as well.

Reliability of research concerns the consistency of the research findings. Merriam-Webster's medical desk dictionary (Merriam-Webster's medical desk dictionary, 2002) defines reliability as "the extent to which an experiment, test or measuring procedure yields the same results in repeated trials." Hammersley (1992) says that reliability "refers to the degree of consistency with which instances are assigned to the same category by different observers or by the same observer on different occasions" (p. 67).

LeCompte and Goetz (1982) distinguish between internal and external reliability. The first is what traditionally comes under the heading of "inter-rater reliability." External reliability is the replicability of an entire study – "Would other researchers studying the same or similar settings generate the same findings?" (Seale, 1999, p. 140).

LeCompte and Goetz (1982) say that external reliability can be improved by addressing five issues:

- identification of the status position of the researcher in the field,
- as much information as possible about who offered the information,
- information about the social context in which the research was done,
- a full account of the theories and ideas that informed the study, and
- a detailed account of all aspects of methods used.

In the articles using qualitative research ("Examining Social Interaction Patterns for Online Apprenticeship Learning – Object-oriented Programming as the Knowledge Domain", "Examining Social Interaction Patterns for Online Apprenticeship Learning – Object-oriented Programming as the Knowledge Domain" and "Categorizing Pedagogical Patterns by Teaching Activities and Pedagogical Values"), we have accounted for the above information.

Internal reliability is enhanced via the following five features (LeCompte & Goetz, 1982):

- use low-inference descriptors
- use multiple researchers
- use participant researchers
- use peer examination
- record data mechanically

"Low-inference descriptors" (Silverman, 2001, p. 226) involve

> recording observations in terms that are as concrete as possible, including verbatim accounts of what people say, for example, rather than researchers' reconstructions of the general sense of what a person said, which would allow researchers' personal perspectives to influence the reporting (Seale, 1999, p. 148).

Not only the observations, but also the transcription and analysis of the data must have low inference (Kvale, 2005).

The major part of the data material from the IOOP case study as well as the IOOP design experiment is low inference. The video-streamed meetings, textual interactions (instant messaging and discussion forum), captured collaborations, and interviews all provide persistent verbatim accounts of what took place, available to other researchers for inspection. The same is true of the comments made by teachers on pedagogical patterns. The notes from the interviews are based more on interpretations of what took place than on verbatim transcriptions, but these are used as pointers into the data material, not as raw data. Similarly, the audiotaped interviews were analysed for the purpose of noting topics the interviewed learners mentioned in relation to their daily working lives. We categorized these topics and formed the following categories: *content*, *tools* and *social communities*. Based on this categorization we designed an interview protocol. The categorization was used only to construct the interview protocol. However, only small parts of the data were selected for transcription and presentation in this dissertation. This selection itself can be regarded as part of the analysis. The analyses has been carried out in collaboration with other COOL researchers and discussed with colleagues. Participant researchers were not used, but most of the findings have been taken back to the participants for discussion – see Section 5.4.1.

5.4.3 Generalizability

Yin (2003) uses the term "external validity" for generalizability within qualitative research. Many have argued that case studies cannot be generalized because they are based only on a single case study. Yin notice that

> survey research relies on *statistical* generalization, whereas case studies rely on *analytical* generalization. In analytical generalization, the investigator is striving to generalize a particular set of results to some broader theory (p. 36) (italics in original).

Silverman (2001) discusses three ways to achieve generalizability:

- combining qualitative research with quantitative measures of populations
- purposive sampling guided by time and resource
- theoretical sampling (p. 249)

In this research, we have compared the case to other findings in the field. The IOOP 04 study finds communicative patterns similar to those found by other researchers. This is one of the three ways that Hammersley (1992) suggests to improve generalizability.

The selection of a case also influences the generalizability of the findings. A case is not just selected at random, but "because it illustrates some feature or process in which we are interested" (Silverman, 2001, p. 250). For example, the IOOP case was selected because it represents the specific pedagogy and the target group in focus. Theoretical sampling means that qualitative research should strive for generalizing theories, instead of generalizing from populations or universes (Bryman, 1988).

One finding from the studies includes proposals for interaction patterns in ICT-mediated dialogue ("Examining Social Interaction Patterns for Online Apprenticeship Learning – Object-oriented Programming as the Knowledge Domain"). Taken together with related research on distributed computer-supported collaborative learning, these findings contribute to the generation of theory regarding communication and learning in ICT-mediated learning environments. Another finding is the connections between professional working life and learning object-oriented programming ("Learning Object-Orientation by Professional Adults"). Taken together with general research on teaching introductory programming and adult learning, this contributes to a theory of professional adults learning object-oriented programming.

In this chapter, we have accounted for the method used in this dissertation. We have argumented for the research design for each research question. The research method is a mixed method based on both quantitative and qualitative studies. We have furthermore accounted for the data sources used for both types of studies.

Lastely, we have accounted for the trustworthiness of the studies. This is done in terms of the validity, reliability and generalizability of the findings.

6 Research Findings

Man has made some machines that can answer questions provided the facts are profusely stored in them, but we will never be able to make a machine that will ask questions. The ability to ask the right question is more than half the battle of finding the answer.

Thomas J. Watson (1874-1956), President of IBM

This chapter discusses how the articles included in appendices 1-8 relate to the research questions formulated in chapter 1. It also discusses the research findings with respect to other relevant research in the field.

The overall research theme of this dissertation was:

> What is a good approach to teaching introductory programming using the objects-first approach?

The overall research theme is broken down into six more specific questions in four different topic areas: model-based programming (R1), professional adults learning introductory programming (R2a + R2b), success factors for a model-based introductory programming course (R3), and pedagogical patterns (R4a + R4b):

> R1: Is it possible to make a course design where the students learn a systematic programming process, conceptual models as a structuring mechanism, and coding?
>
> R2a: How do the different aspects of a model-based programming course influence professionals' practice?
>
> R2b: What are the interaction patterns for professionals learning object-oriented programming with a focus on a systematic programming process and conceptual models as structuring mechanisms in a technology rich distance education setting?
>
> R3: Do model-based introductory programming courses have the same success factors as more traditional courses?

> R4a: What are the awareness of, use of, and attitude toward pedagogical patterns among computer science university teachers around the world?
>
> R4b: How can pedagogical patterns be structured to make them more useful for university teachers?

This chapter will discuss the research with respect to these four groups of questions.

6.1 A Model-Based Programming Course

This section describes the approach to teaching introductory object-oriented programming investigated in this dissertation. It is described in the article "Programming in Context: a Model-First Approach to CS1". This article is a "Marco Polo" style of article (Valentine, 2004). The article is enhanced in "Model-Driven Programming" (Bennedsen & Caspersen, 2008b) to be published by Springer-Verlag in 2008. The focus on the programming process is discussed in "Revealing the Programming Process" (enhanced in the article "Exposing the Programming Process" (Bennedsen & Caspersen, 2008a)).

This section gives an answer to the first research question:

> R1: Is it possible to make a course design where the students learn a systematic programming process, conceptual models as a structuring mechanism, and coding?

Many suggestions have been put forward on how to teach introductory object-oriented programming. Most of these suggestions are structured according to the programming language and are focused on its syntax. The model-based (model-first is used as a synonym in the article) approach is structured according to the complexity of the class model to be implemented. As described in chapter 1, the four principles are:

1. Objects from day one – in the beginning students use predefined classes to create objects, then they imitate the implementation of a class and finally they create classes,
2. A balanced view of the three perspectives on the role of a programming language (Knudsen & Madsen, 1988),

3. Enforce the use of a systematic way to implement the description of a solution, and
4. An explicit focus on the programming process, not just the programming language.

The model-based approach to teaching object-orientation has been developed for the last 15 years together with Michael Caspersen.

6.1.1 Objects from Day One

One of the typical misconceptions students have when learning introductory object-oriented programming is the confusion of the concepts of "object" and "class" (Fleury, 2000; Holland, Griffiths, & Woodman, 1997). Eckerdal and Thuné (2005) use a phenomenographic research metod to get a better understanding of the way students understand these two concepts (a more detailed description of the study can be found in (Eckerdal, 2004)). Eckerdal and Thuné conclude that in their data, they can find three different understandings of the concept of "object:"

> 1) Object is experienced as a piece of code.
>
> 2) As above, and in addition, object is experienced as something that is active in the program.
>
> 3) As above, and in addition, object is experienced as a model of some real world phenomenon (p. 91).

and three different understandings of the concept of "class:"

> 1) Class is experienced as an entity in the program, contributing to the structure of the code.
>
> 2) As above, and in addition class is experienced as a description of properties and behaviour of the object.
>
> 3) As above, and in addition class is experienced as a description of properties and behaviour of the object, as a model of some real world phenomenon. (p. 91)

It should be noted that one of the aims of phenomenographic analysis is to develop a hierarchy of understandings. The goal of a model-based introductory course should be for students to understand the connection between the real world concept and a class in the program's text, as well as the connection between objects created by the program's execution and real world phenomena. As described below, this is done by showing the students models of the real world (domain models), specification models and how specification models can be implemented in a given programming language. This naturally implies a balanced view the role of the programming language instead of a course that focuses only on syntax and semantics of a given programming language.

The model-based approach uses a read-modify-write approach. This implies that the student first sees and studies one or more examples of a given concept (loop, method, class, association ...) – the reading phase – followed by changes to existing programs – the modification phase – and lastly the creation from scratch (typically an addition to an already existing program without the concept) – the write phase. The progression in terms of read-modify-write and the object-oriented conceptual framework is illustrated in figure 12. The overall idea is the following: The student starts by reading methods (and classes since methods are in classes). When the student has read several methods, (s)he starts to modify methods (and thereby also modify classes, but that is not the focus). After that, the focus starts on classes (read classes and afterwards modify classes – concurrently with write methods as modifying classes normally requires to write one or more methods) followed by a focus on associations etc.

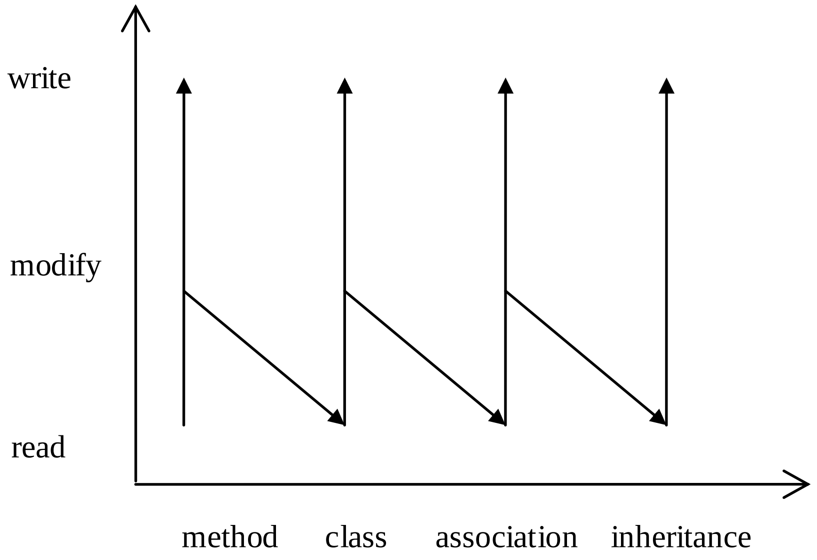


Figure 12: Progression in a model-based approach

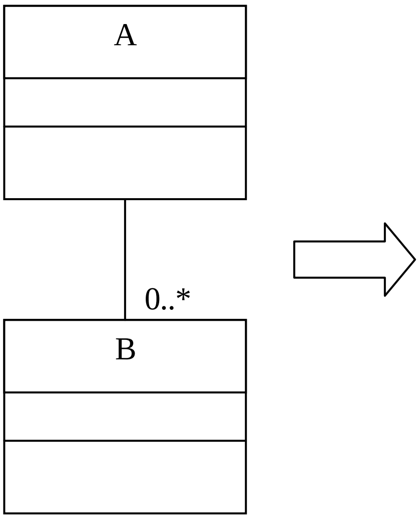
Students initially use pre-created classes to instantiate objects. This has been done in several ways; one is using the “Shapes Demo” from ahapter 1 in (Barnes & Kölling, 2005), another is by the use of “Turtle Graphics” (Caspersen & Christensen, 2000).

In many courses BlueJ is used as the development environment. One of the goals of BlueJ is to give students a clear visual understanding of objects and classes (Kölling & Rosenberg, 2001). Even though we have not done research on the topics, it is our firm belief that the students in general do not confuse the concepts of “class” and “object.” It is supported by the high number of students passing the exam where they shall create objects and implement classes (Bennedsen & Caspersen, 2006b). This is consistent with the findings of Hagan and Markham (2000a), who did not find that students misunderstood “object” and “class” when they switched to BlueJ.

6.1.2 A Balanced View of the Programming Language’s Role

In the model-based approach, coding and conceptual modeling are done hand-in-hand, with the latter leading the way. Introducing the different language constructs is subordinate to the need to implement a given concept in the conceptual framework. After introducing a concept from the object-oriented conceptual framework, its representation in UML and a corresponding coding pattern is introduced; a coding pattern is a guideline for the translation from UML to code that implements the concept from the conceptual framework thereby making an explicit link between the implementation view and the specification view. In figure 13 an example of a “to-many” association at the conceptual level and the corresponding coding pattern is given.[26]

[26] The description could lead to the impression that the actual teaching is deductive. This is not the case. The students abstract coding patterns from a number of concrete examples.



```
import java.util.*;
public class A {
  private List<B> bs;
  public A(){
    bs = new ArrayList<B>();
  }
  public void addB(B b) {
    bs.add(b);
  }
  public void removeB(B b) {
     bs.remove(b);
  }
}
```

Figure 13: A coding pattern for a "to-many" association

The conceptual framework for object-orientation is comprehensive; for introductory programming the coverage is restricted to association, composition and specialization, which seems to be the most used concepts in object-oriented modelling and programming. This restriction is a choice based on many years of teaching experience; other choices and sequences of concepts are, of course, possible.

As described above, the first concept the students encounter is object. The next is a class and its properties. One of the properties of a class can be an association to another class; consequently, the next topic is association. This correlates nicely with our impression from talking to many professional object-oriented software developers that association (reference) is the most common structure between classes (objects). Composition is a special case of association, taught in the next round of the spiral. The last structure to be thoroughly covered is specialization. Specialization is the least common structure in class models, but is the foundation for, e.g., design patterns, which then could be the focus of the following course.

6.1.3 A Systematic Way to Implement a Specification

In general novices do not know how to approach a programming problem without guidance (Barrett, 1996; Beck, 1999; Caspersen & Kölling, 2006; Gantenbein, 1989; Gries, 1974). As Beck remembers from his first, larger programming assignment when he learned to program:

> 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! (p. 43)

Caspersen and Kölling put it this way

> 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. (p. 892)

In the model-based approach, the programming process is made explicit. This is done in several ways:

1. Using an explicit programming process focusing on the fact that a program is not developed in one, big shot (as seems to be the impression students get from the programming textbooks) but through stepwise refinement (Wirth, 1971). For a more thorough description, see (Caspersen & Kölling, 2006; Caspersen, 2007)

2. Focusing on programming as a skill and showing the way an experienced programmer works; explicitate the internal cognitive processes (assumptions, considerations, tool use etc.) that the experienced programmer applies. This is done as cognitive apprenticeship. Revelation of the process can be done as live programming (Alford, 2003) or as recorded sessions (process recordings as they are termed in "Revealing the Programming Process") where the experience programmer "thinks aloud".

3. Using exercises where the process is very explicit in the beginning (i.e. the exercise is described in many small steps mimicking the development process).In later exercises, the steps are gradually made bigger and bigger.

4. An explicit focus on different levels of abstraction and use of programming techniques for each level:

    a. *Problem domain → model*: A UML class model of the problem domain. The UML class model is normally provided to the students.

b. *Functionality → model*: Specify properties and distribute responsibility among classes. Normally included in the specification model given to the students.

c. *Model → Java code*: Create a skeleton for the program using the coding patterns.

d. *Implementation of classes*: Use class invariants (Meyer, 1997) describing the internal constraints that have to be fulfilled before and after each method call.

e. *Implementation of methods*: Use algorithmic patterns (Muller, 2005) for traditional algorithmic problems such as searching and sweeping. Use loop-invariants (Cormen, Leiserson, Rivest, & Stein, 2003) to systematically construct loops. Use design by contract (Bolstad, 2004; Meyer, 1992; Pedroni & Meyer, 2006) to describe the responsibilities between a method's users and the method itself.

Caspersen and Kölling (2006) have described a programming process that can be used in a model-based introductory programming course.

A typical goal of a programming course is to teach the students to appreciate and achieve good 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 knowing how to achieve them is even harder. We need to teach students guidelines for achieving quality programs and a vocabulary that enables them to talk abut their programs in order to help them build quality programs.

In an introductory programming course applying a model-based approach, the starting point (the specification model) is normally given to the students in the beginning. The reason for this is that the students normally find it difficult to design (Barrett, 1996; Lidtke & Zhou, 1999; Moritz, Wei, Parvez, & Blank, 2005), and the focus is on implementation, and not description of the problem domain or the creations of specification models (i.e., it is a programming course, not a design and analysis course). Later on in the course (or in following courses) the students can start to develop specification models themselves. In this way, a “read – modify – write” approach can be taken to design as well.

Class invariants should, as with the rest of the concepts covered in a model-based approach, be taught to help rather than burden students. This implies that the goal is not to exercise formal, logical predicates, but to learn informally, typically including drawings or other informal ways of describing invariants. The purpose is to make the students aware of the state of an object represented by the values of its attributes and the dependencies among them.

As described by Sicilia (2006, pp. 14), a simplified version of design by contract should be taught. As he proposed, we have found that the introduction of pseudo comment tags such as `@pre` and `@post` helps the students to focus on the method’s responsibilities and the caller’s requirements.

6.1.3.1 An example of “model → Java code”

The development of a coding pattern for a zero to one association is given as an example of this approach. Through a number of progressive examples, we illustrate that an association is a relation between classes (or between the same class as in the following example), a class can have more than one association, and an association is a dynamic relation.

Students extend a previous example of a single class Person with a recursive association. One example is that a person can be married_to another person or can be another person’s lover. This results in the model in figure 14.

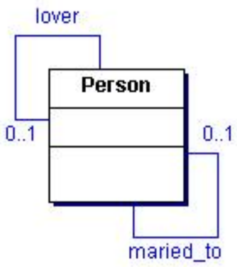


Figure 14: One class with two associations

In order to implement a zero-to-one association, the student needs to know about programming language elements (e.g., reference and the null value). This also gives the students an understanding of interactions between objects (calling methods on other objects) and reference semantics.

Another example of a recursive association is a simple adventure game whose rooms are connected to other rooms in different directions. This can be modelled as follows (figure 15):

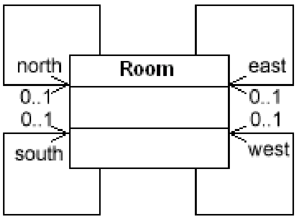


Figure 15: One class with four recursive associations

Again, the idea is that the students see many implementations of the same general concept from the conceptual framework, and realize the general coding pattern (figure 16Figure):

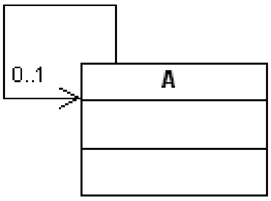


Figure 16: General, recursive association

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;
   }
}
```

6.1.4 The Programming Process

One of the problems students struggle with in programming is not the single elements of the programming language, but the construction process (Soloway, 1986; Spohrer & Soloway, 1986; Spohrer & Soloway, 1986). Gries (1974) focused on the process in the early seventies when he 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! (p. 82)

In “Revealing the Programming Process” we have described and evaluated a way to make the programming process more explicit to students in the form of “process recordings” (p. 188). We found that students found process recordings to support their learning of introductory programming. This is one way of practicing cognitive apprenticeship by the use of technology.

In general, Michael Caspersen has in his PhD dissertation (2007) described in more detail ways to address the problem of teaching novices a programming process.

6.1.5 Results of Teaching Model-Based

Many teachers report high numbers of students not passing or dropping out of CS1 courses (Andersson & Roxå, 2000; Börstler, Johansson, & Nordström, 2002; Forte & Guzdial, 2005; Herrmann et al., 2003; Nagappan et al., 2003). In section 2.1, based on numbers from UNESCO, we indicated that only 27% of the students globally complete their computer science study. Bennedsen and Caspersen (2007b) have done an initial study in order to collect data on the average pass and failure rates of CS1 courses; they found the average pass rate to be 67% (but the number of respondents was low: 64).

From the above studies it is difficult to define a precise global pass and fail rate. It is even difficult to find precise definitions of these terms. However, based on the findings of Bennedsen and Caspersen and the numbers from UNESCO, it seems reasonable to call a course a success with respect to pass- and fail rates if the pass rate is above 75% (and consequently the fail + drop-out rate is below 25%).

In almost all of the studies of success factors for learning programming, the exam result is used as success factor. We will use the same to indicate the feasibility of a model-based approach. It should be noted that the claim is not that a model-based approach is better than other approaches, but that it is possible to learn the students programming using this approach.

Bennedsen and Caspersen (2007a) evaluated the exam of a model-based introductory programming course (actually the dIntProg course, taught by Michael Caspersen). They found the following numbers:

	2003	**2004**	**2005**	**2006**
Students	276	220	295	326
Abort	63	26	28	44
Exam	213	194	267	282
Exam rate	77.2 %	88.2 %	90.5 %	86.5 %
Skip	13	5	3	1
Fail	15	19	29	17
Pass	185	170	235	264
Pass rate	86.9 %	87.6 %	88.0 %	93.6 %
Retention rate	67.0 %	77.3 %	79.7 %	81.0 %

Table 8: Results from dIntProg 2003-2006 (table 2 in (Bennedsen & Caspersen, 2007a)**)**

Their terms are defined as follows:

Variable	**Description**
Students	students enrolled in 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 9: Definition of the variables in Table 8

The same numbers form the IOOP courses in 2003 and 2004 can be found in Table 10

	2003	**2004**
Students	21	20
Abort	4	3
Exam	17	17
Exam rate	81%	85%
Skip	0	0
Fail	0	0
Pass	17	17
Pass rate	100%	100%
Retention rate	81%	85%

Table 10: Numbers for IOOP

Based on the numbers in Table 8 and Table 10, we conclude that it is possible to teach a model-based course with the same success in terms of pass and fail rates as traditional CS1 courses.

In "Revealing the Programming Process" the results of the evaluation of the learning outcome of process recordings was as follows: None 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. Consequently, we conclude that process recordings can be seen as a good way to make the programming process explicit in a distance education setting where the students are adults with prior programming knowledge.

6.1.6 Conclusion

The first research question was

> R1: Is it possible to make a course design where the students learn a systematic programming process, conceptual models as a structuring mechanism, and coding?

In this chapter we have described a course design that has an explicit focus on the programming process in two ways: Showing how a master performs when he or she actually solves programming problems (both in a lecture hall setting and in an e-learning setting) and teaches the students a systematic way to address programming problems. The course further uses the object-oriented conceptual framework in a consistent way to introduce conceptual modelling and the relationship between the models and the code. Consequently, we conclude that it is possible to design such a course. Based on the evaluations presented in article "Programming in Context: a Model-first Approach to CS1" and "Revealing the Programming Process", we also conclude that it is possible to teach such a course.

The model-based approach teaches the constructs of the programming language and the elements of the conceptual framework hand-in-hand. We find that a model-based approach both gives the students a good understanding of the conceptual framework, and gives the students competencies in the five areas du Boulay describes as necessary in order to do programming (see page 35).

6.2 Professional Adults Learning Introductory Programming

This section will discuss the two research questions related to professional adults:

> R2a: How do the different aspects of a model-based programming course influence professionals' practice?
>
> R2b: What are the interaction patterns for professionals learning object-oriented programming with a focus on a systematic programming process and conceptual models as structuring mechanisms in a technology rich distance education setting?

6.2.1 Linking the Learning of Model-Based Introductory Programming to Professional Practice

Research in adult learning has shown that adults are more focused than children when learning in the sense that they typically are more selective about what they learn. They "want their learning to help them solve problems, build new skills, advance in their jobs, make more friends – in general, to do, produce or decide something that is of real value to them" (Wlodkowski, 1999, p. 27), see also section 4.7.

Benander, Benander, and Sang (2004) found that the most important factor for learning Java was previous knowledge of object-orientation. They further found that those "who did have some graduate education reported finding all of the tested features less difficult to learn than did those who had no graduate education" (p. 106). However, these two factors are precisely what do not characterize the above-mentioned target group. Turk (1997) advocates a smooth transition from the procedural paradigm to the object-oriented paradigm by gradually changing a procedural solution into a more object-oriented solution. His reason is to avoid the difficulty of creating an object-oriented design from scratch. This is consistent with the idea of the model-based approach, which gives the learners the design to implement. However, we find it problematic to transform a program made in one paradigm into another.

Adults want their study program to be flexible (Lorentsen, 2004). They want to study what they find relevant, when and where they want to. One popular way to create flexibility for learners is to use distance education (King, 2002).

The research focuses on how a link to the learners' prior programming experience should be made when the learners are adults with previous programming experience, when a model-based approach is used, and when the course is net-based. The research is positioned in research on learning object-orientation in ICT-mediated learning environments. Previous studies have provided insights into how adults in general connect their working practice to their learning situation. Nevertheless, few studies have addressed adults learning object-orientation in a net-based setting. The article "Learning Object-Orientation by Professional Adults" discusses three categories of linking: the course's content, the programming tools used, and the pedagogical model of the IOOP course. The research consists of two parts: first, an analysis of the interviews from IOOP 03 in order to establish the three categories; second, interviewing and analyzing the same learners one year later to investigate the link to their professional practice.

The learners found that the course did impact their working life. They indicated that the teacher's role of performing the practice of programming by showing which software tools to handle (e.g., BlueJ, Java, Java libraries, etc.) and what intellectual tools were necessary (e.g., analytical skills, reasoning, trial-and-error, combining natural and scientific language, etc,) was important for their understanding of object-orientation.

du Boulay (1989, p. 283) describe five overlapping areas that a learner must be able to master: *general orientation*, *notional machine*, *notation*, *structure*, and *pragmatics* (see page 35). The learners in question know a lot about the general orientation and some also the notation. They also know about the notional machine, structures and pragmatics in general, but not in the context of object-orientation. One interpretation of the findings could be that the learners found the teacher showing areas 2-5 in action was the most important aspect for their learning. As noted by, e.g., Gries (1974) and Gantenbein (1989), it is important to focus explicitly on the programming process; this was also explicitly mentioned by the learners. In this respect, a systematic way of developing a program is included in the pragmatic and structural areas above.

The results of the study imply that a direct link between daily work and course content is not needed. The learners expected to easily be able to transfer the examples shown in the course to their daily working lives, and accepted that it was their own responsibility to do so. This is in line with the arguments made by Joseph and Nacu (2003) that the learners need the instruction to be aligned with their work.

The course under study used BlueJ. It is a tool designed especially for learning object-orientation. It is much different from the feature-rich professional development tools the learners use in their daily life. The learners did not find this to be problematic. Rather than requiring professional programming tools to learn, they focussed on the best tool for learning object-orientation. This could be seen in contrast to the "functional link" described by Engeström (1994). However, as Wlodkowski (1999) notices, they "want to learn new skills" and apparently have found that BlueJ helps them achieve their goal more easily than other tools. In this respect, the goals of the BlueJ developers (Kölling & Rosenberg, 2001) are met.

The learners participated in at least two communities of practice: the community at their workplace and the community of learners. According to the learners' own words, transfer from the community of learners to the community of the daily work practice occurred both during the course and following its completion.

The research question, "How do professionals link the different aspects of a model-based programming course to their professional practices?" cannot completely be answered. The first reason is that the research focuses on only three aspects – content, tools, and social interaction, and other aspects need to be considered. Furthermore, the research is done in the special setting of net-based education. This naturally shapes the design of the course and therefore the participants' experience. However, our research has given some indications that a direct link between the participants' daily work practice and the educational setting is not needed in all cases.

6.2.2 Social Interaction Patterns for Learning Model-Based Introductory Programming

This research question is addressed in "Examining Social Interaction Patterns for Online Apprenticeship Learning – Object-oriented Programming as the Knowledge Domain". This article is positioned in research on learning object-orientation in ICT-mediated learning environments. Many developers of net-based learning have a tendency to use these new technologies in traditional ways (Dehoney & Reeves, 1999; Kearsley, 1998). Previous studies have provided insights into how various pedagogical approaches that are grounded in co-located situations can be applied in distributed learning settings. However, few studies have addressed pedagogical designs based on ideas from apprenticeship learning in such contexts. This article discusses social interaction patterns that developed during shared events in the IOOP course. The research consists of two parts: first is a case study of IOOP 03, with focus on the online meetings. This study identifies some problematic issues. Second, a design experiment addressing the problematic issues was carried out in IOOP 04.

Based on a cluster analysis of 569 e-learning courses' activities, Buelens, Roosels, Wils, and van Rentergem (2002) found that the majority (85%) of courses were "course document oriented" (i.e., the overwhelming majority of activities learners do are up- and downloading documents, not sending messages or other communicative actions) (p. 172). Therefore, the low activity identified in the online meetings could be expected.

The case study supports the preliminary findings from Fjuk et al. (2004). The low learner activity in the online meetings indicates limited success with regard to the de-centred facet of the apprenticeship approach. The online meetings were almost monological and did not support the de-centred facet of apprenticeship learning. In addition, the modest interaction is found to be problematic with respect to the teacher's opportunity to obtain insights into the learners' understanding and progress. Such insights are important for successful scaffolding from the teacher.

The design experiment was set up to explore how the de-centred facet of apprenticeship could be facilitated. Analysis of these small groups and problem-oriented sessions indicates that they have the potential to resolve the issues from the case study. The master's meta-communicative actions are found to be essential in this respect. The study also suggests that proposals for solutions, prepared in advance individually by the learners, shaped the groups' interaction patterns. These shared artefacts (and their presentation by one member of the group) constitute important communicative instruments for establishing productive collective activities online. It extends the discussion of Fjuk et al. (2004) by analyzing how some design intentions described in that article were carried out in practice. The analyses are concerned with patterns of social interaction, in which the artefacts mediating these actions are integral parts of the analyses. One finding indicates that the online meetings adequately mediated apprenticeship's person-centred facet, but not its de-centred facet. The article's second major finding is the role of the learners' initial proposals for solutions in shaping the social interactions.

Other aspects of the design experiment are analysed in Bennedsen (2006a).

6.3 Success Factors for an Model-Based Introductory Programming Course

This section address research question R3:

> R3: Do model-based introductory programming courses have the same success factors as more traditional courses?

Many studies have examined success factors in programming – see section 3.2.3 for a listing.

A model-based introductory programming course is a special type of objects-first programming course. Within the area of success in objects-first courses, Phil Ventura's PhD study (2003) is one of the most exhaustive. He asked: "What variables are most important for success among credit hours, hours worked at job, high school average, high school rank, number of years of high school math, SAT math, SAT verbal, number of office hour visits, recitation attendance, comfort level, critical thinking ability, programming self-efficacy, attribution for success/failure, prior programming experience, number of labs submitted, and number of exams taken?" (Ventura, 2003, p. 57). Furthermore, he was concerned not only with success, but also with retention rates, i.e. not only how many students passed the introductory course but also how many students continued their study.

To measure his 39 research variables, Ventura used a battery of tests such as the Cornell Critical Thinking Test (Level Z) developed by Ennis, Millman and Tomko (1985), to assess the students' critical thinking ability, and the Computer Programming Self-Efficacy Scale developed by Ramalingam & Wiedenbeck (1998) to measure the students' beliefs about their own programming ability.

In neither Phil Ventura's nor the studies reported in this dissertation did gender or intended major predict the final grade. Concern has been increasing about females' low enrolment in computer science (Cuny & Aspray, 2002; Margolis & Fisher, 2003; Roberts, Kassianidou, & Irani, 2002). We find it encouraging that the model-based approach does not discriminate against female students; it is important that an introductory programming course offers equal opportunities for both sexes to learn successfully.

A growing number of students are learning programming but not majoring in computer science (Rosson, Ballin, & Rode, 2005; Scaffidi, Shaw, & Myers, 2005). At the University of Aarhus, for example, only 40% of the students participating in the introductory programming course are computer science majors. This growing student group makes it crucial that CS1 designs do not alienate non-CS majors early on, but instead shows these students that programming is useful and interesting. It is encouraging that the model-based approach is equally successful for teaching students who are majoring in CS and those who are not.

One of the findings is in contrast to Ventura's findings; he found that previous math scores did not predict students' success, but we found that high school math score was the best predictor of students' success. This difference could be the result of the data's origin. Ventura's and our data come from two different implementations of an objects-first introductory programming course in two different cultural settings; we can draw conclusions only for the particular implementation studied. Conclusions about success factors for objects-first programming courses in general require research to be generalizable. The generalizability of the research in this dissertation is discussed in section 5.3. One example of this cultural difference is the way that the exam result is calculated. Ventura's study calculated the exam result using three elements: homework and quizzes (10%), lab average (40%) and exam component (50%); our study based the exam result solely on a practical examination, in which the student solved nine small, progressive programming tasks. The exam components in Ventura's study "were two in-class exams and a final exam. All exams were short answer format; that is, they required students to write small amounts of code, produce or modify class diagrams, perform matching and answer True/False (with justification) questions" (Ventura & Ramamurthy, 2004, p. 241). Care must consequently be taken when likening one result to other results.

In partial conclusion to the research question "Do model-based introductory programming courses have the same success factors as more traditional courses?" is – "Yes". From the literature review of studies in success factors we concluded that math seemed to be a general predictor. In our study we have found the same to be true.

6.3.1 Abstraction as Success Factor

The research did not find that abstraction ability measured as cognitive development predicted success. This is in contrast to Cafolla (1987-88) who found that the level of cognitive development and verbal ability accounted for 49.5% of the variation in exam score when students were learning BASIC. Verbal ability was the strongest predictor (R=0.62), followed by the level of cognitive development (R=0.59). Cafolla noted, "Verbal ability may have played a large role in this study because the final exam required interpreting written instructions that explained the parameters of the programming problem. It may be that the final examination measures how well students interpret written instructions, in addition to computer programming ability" (p. 52).

The finding of no correlation between the students' high school math grade and their cognitive development level was surprising, but in agreement with Cafolla, who found no correlation between the students' cognitive development and their mathematical ability (Cafolla, 1987-88, p. 53). Combined with the finding of a correlation between high school math grade and exam grade, this led to the conclusion 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. Nothing in the course requires a mathematical background; the lecturer explicitly stated that his examples and exercises should be in an interesting domain that the students could identify with, and not mathematical examples such as prime numbers. One reason for the correlation could be that math, like most programming languages, is a formal language, so the ability to use and understand formal languages correlates with math and programming skills.

We could speculate about why abstraction ability does not correlate with the exam score. One explanation could be the nature of the model-based approach, with its explicit focus on coding patterns founded in the specification model. The cognitive load on the students is presumably less than for traditional problem-solving oriented courses because they do not have to worry about both the specification models and solution domains; the former is given by the teacher. This is supported by Détienne (2002) who emphasises that designers "use knowledge from at least two different domains, the application (or problem) domain and the computing domain, between which they establish a mapping" (p. 22), and consequently design is a difficult process for students.

Is an exam result an indicator of success? The obvious answer is, "Yes – of course!" Nevertheless, this requires that the examination measure the factors that we believe are crucial for success. We believe that most teachers have experienced a student asking "Is this included in the exam?" and if the answer is, "No," more than half of the students stop listening or leave the lecture hall. Rowntree (1988) put it like this, "The spirit and style of student assessment define de facto the curriculum" (p. 1). Ramsden (1992) makes a similar observation: "The type of grading influences the students' learning approach."

The investigated variables' low predictive power could lead to the conclusion that there is no way to predict a student's success. This, however, is not the case. Chamillard (2006) advocates the use of scores in prior courses to predict success in succeeding courses. Using those, he can predict up to 75% of a student's score. However, this does not help to identify factors that indicate success in an introductory programming course. Dehnadi and Bornat (2006) conducted an initial study which suggests that success in the first stage of an introductory programming course is predictable by noting consistency in use of the mental models that students apply to a basic programming problem even before they have had any contact with programming notation. In their initial study, they claim to be able to distinguish between passing and failing students: “if it [their test] were 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%) failed” (p. 16). Caspersen, Bennedsen and Larsen (2007) replicated the study in a model-based course (dIntProg). They were not able to verify the findings of Dehnadi and Bornat.

In conclusion the findings of Barker and Unger (1983) (summed up at page 77), Ventura (2003; 2005) and Kurtz (1980) (summed up at page 77) are not supported by this study. We must therefore partly answer the question, “Do model-based introductory programming courses have the same success factors as more traditional courses?” with a “no!” – abstraction ability is not an indicator of success in this type of course, as it is in an imperative-first course.

In order to answer the more general question of success factors for an objects-first introductory programming course, we need data from various implementations of this teaching strategy. Furthermore, we need to be much more aware than we currently are of the different circumstances that influence the success indicator (normally the exam score). However, it seems that traditional factors are not useful as predictors in our case.

6.4 Pedagogical Patterns

This section addresses the research questions R4a and R4b:

> R4a: What are the awareness of, use of, and attitude toward pedagogical patterns among computer science university teachers around the world?
>
> R4b: How can pedagogical patterns be structured to make them more useful for university teachers?

The pedagogical patterns project tried “to capture expert knowledge of the practice of teaching and learning” in a compact and easily communicable way (The pedagogical patterns project, n.d.). This is a worthy goal, but is it achievable? Several authors (e.g., Fincher and Utting (2002)) claim that pedagogical patterns’ impact are very limited.

Our research found a high awareness of the concept of pedagogical patterns (a little less than half of the respondents were aware of them). In order to describe the teachers who were aware of pedagogical patterns, correlations between a number of factors and the knowledge of pedagogical patterns was investigated. None was found except that teachers in the United States seem to have a higher awareness than teachers from the rest of the world.

Teachers have a positive attitude toward the idea of pedagogical patterns and they believe that they can use them as a tool to develop their own teaching standards. These findings seem to contradict the claims made by Fincher and Utting (2002) and by Bennedsen and Eriksen (2003).

It is questionable whether these results can be directly generalized to the entire population of university teachers because of the selection of the response group: authors at conferences focusing on teaching computer science. We would expect this group to have a higher interest in teaching than the average university teacher and because of this probably also a higher awareness of concepts related to the pedagogical field. In other words, we do not claim that almost 50% of all computer science university teachers have knowledge of pedagogical patterns.

It seems that pedagogical patterns are not used in teaching or in research on teaching. Since knowledge of pedagogical patterns seems to be quite high, this naturally gives rise to the question, “Why not?”

In design patterns, the underlying values that all patterns strive for are low coupling and high cohesion. In the software development community it is commonly accepted that these two values are good and worth striving for. Within teaching it is very different, because many different theories of teaching exist, each with different underlying values of what is considered important in teaching and learning. So, just describing pedagogical patterns without explicitly stating what teaching values are considered important makes it very difficult for a teacher to evaluate whether a given pedagogical pattern is acceptable to him/her and solves the teaching problem (s)he faces.

The main contribution of the research on pedagogical patterns is a proposal for a universal pedagogical pattern categorization based on teaching values and activities. The article argues that, with pedagogical patterns' current structure, it is very difficult for their supposed users (teachers) to find a pedagogical pattern that is suitable for a given problem. The article suggest a categorization of pedagogical patterns based on teaching values and teaching activities in the same way that design patterns (Gamma et al., 1995) are organized by two criteria: purpose and scope. As Fincher (2006) notes, the aim is

> a ‘‘structuring principle,’’ a meaningful organisation that will allow other educators easier access to the knowledge the patterns contain. (p. 75)

Values of pedagogical patterns are understood in the same sense as the Agile Manifesto's four values of software development (Cockburn, 2002). The Agile Manifesto's authors use the term "value" because they believe that the presence of the four agile values will improve software development, with respect to both the constructed systems' quality and the process used for each project. Furthermore, the values are not debatable; they are a matter of faith, like religion. Bennedsen and Eriksen (2003) describe three pedagogical values derived from the pedagogical theories of Steen Larsen (1998):

> "When teaching, we prefer:
>
> 1. **Working students** over listening or reading students
>
> 2. **Emotional involvement** over discipline and external motivation
>
> 3. **Students working with tasks they almost are able to solve** over students working with routine or impossible tasks" (p. T4A-5).

Larsen's values reflect the constructivist way of teaching (the rationalist view – see pp. 84). Focusing on social constructivism or situated learning (the pragmatist-sociohistoric view see pp. 84), an additional value to the three above could be:

When teaching we prefer:

- **Collaborating students** over individually working students

As for teaching activities, three categories were chosen: *planning*, *performing* and *evaluating*:

> **Planning**: working out the details of something in advance, including establishing learning goals, a schedule, evaluation criteria, and a sequence of topics.
>
> **Performing**: carrying the plan through, including lecturing, mentoring, supervising, etc.
>
> **Evaluating**: judging students' performance in relation to the evaluation criteria defined in the planning.

These categories are chosen mainly because they are useful for practical teaching purposes.

Using these values and categories suggests a uniform classification of pedagogical patterns.

Based on the findings, we conclude that the knowledge of pedagogical patterns among computer science teachers around the world are larger than expected; we had expected that almost no teachers knew about the concept.

We furthermore conclude that it is possible to structure pedagogical patterns in a more uniform way via structuring mechanism based on learning theories. This way of structuring pedagogical patterns makes it possible for a teacher – based on a pedagogical analysis of the problem – to identify relevant patterns to use.

7 Implications for Practice

Creativity is not the finding of a thing,
but the making something out of it after it is found.
James Russell Lowell (1819-1891), United States Poet

This section will discuss this PhD work's impact on the practice of teaching computer science.

This PhD project's central research focus is teaching and learning introductory object-oriented programming. During my more than 20 years of teaching experience, I have taught object-oriented programming to many different people and in many different settings; from traditional freshmen in semester-long university courses held in lecture halls to three-day courses for professional programmers in the IT industry. The model-based approach is a synthesis of all my teaching experience. It has been interesting and enlightening to evaluate this approach's impact more methodically than we have in the past.

Other approaches to teaching object-oriented programming have been proposed. However, most, if not all, are based on the teachers' own belief and not on methodical evaluations. This dissertation is an attempt to change this and use a scientific approach to evaluating a model-based approach's impact in many settings – from university courses in lecture halls teaching freshmen to distance education teaching professional programmers.

Concrete teaching advice is always at high rate. One problem, however, is how to present advice. In software design, patterns have been the answer for some time, so it is obvious to try to do the same with teaching advice. This is what the pedagogical pattern project has done. Its success, however, has been limited; few articles, workshops, etc. discuss, enhance and evaluate pedagogical patterns even though the awareness of pedagogical patterns was higher than we initially expected. In this PhD project we have proposed another way of categorizing and describing pedagogical patterns, forming a universal pedagogical pattern categorization based on teaching values and activities. This will force the teacher to state and address the underlying teaching values in much the same way as software values are addressed in design patterns.

7.1 Computer Science Education Practice

Many different ways are used to teach introductory programming; this PhD work presents one ("Programming in Context: a Model-first Approach to CS1"). This is done in a way that other teachers can follow the ideas behind the design and, hopefully, be inspired in their course design to focus more on object-orientation's conceptual modelling aspects than on the language details of the particular programming language they use. Madsen, Nygaard and Møller-Pedersen (1993) emphasise the role of conceptual modelling; the model-based approach can be seen as one way to highlight conceptual modelling.

There is a tendency to focus on technology in net-based learning (or e-learning as it is popularly called); pedagogical principles seem not to be included in the learning management system[27] (Bixler & Spotts, 1998; Firdyiwek, 1999). We find it vital to balance the technology, pedagogy and learning materials when designing a net-based course. Georgsen (2004) asserts that teachers must ask themselves the following fundamental questions:

1. How do I wish to teach and organize the learning processes?
2. How can technology be used (if at all) to support learning, communication and collaboration?
3. How can the learning materials be developed, made available, and distributed?[28] (p. 11).

This is in line with the argument of Govindasamy (2001):

> Most of the pedagogical principles that apply to the traditional classroom delivery method also apply to e-Learning. However, these principles need to be extended to accommodate and provide for the rapid changes in technology. Pedagogical principles must form the very basis for inclusion of features in

[27] According to Wikipedia (http://en.wikipedia.org/wiki/Learning_management_system), "A *Learning Management System* (or *LMS*) is a software package, usually on a large scale (that scale is decreasing rapidly), that enables the management and delivery of learning content and resources to student."

[28] Translated from Danish by the author.

> LMS. Better still, these principles should be integrated into the LMS where every feature included is accompanied by explicit guidelines on the best method of their use to effect pedagogically sound instruction (p. 288).

The research on cognitive apprenticeship in a net-based setting ("Examining Social Interaction Patterns for Online Apprenticeship Learning – Object-oriented Programming as the Knowledge Domain") can be seen as a description of one way to implement a pedagogical principle in an e-learning course. The research showed that the implementation was seen as successful from the students' point of view, but problematic from the teacher's point of view. The design experiment describes one possible way of giving the teacher more feedback. The research shows that the solution to a given exercise strongly influences the progress of collaboration. This could give some hints to teachers on their advice to students when they are preparing for a collaborative online session. The role of the master (teacher) did have an important impact on the session's structure. The research can be used by teachers to reflect on their role in such an online meeting.

Teachers of introductory programming have strong opinions on the characteristics of well-performing students. These, among other things, are reflected in the selection of variables that have been studied in order to predict students' success in introductory programming. Our studies ("Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming?" and "An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course") have found only the students' prior math grade and their work during the course to be influential. My colleague, Michael Caspersen, has already used the finding of the correlation between coursework and success to motivate students to be active, when he introduced his course in introductory programming this year. According to him, the fact that it was a scientific result was seen by the students as making the argument more persuasive. We hope that other teachers can use this result in a similar way.

Many teachers see abstraction as positively influencing students learning to program. The lack of correlation between abstraction and success in learning programming is indeed surprising. This result could influence the way teachers think of students and abstraction's role in introductory programming curricula. If the result had been the other way around, teachers could have used some of, e.g., Adey and Shayer’s (1994) teaching materials to enhancing their students’ cognitive level. Now it seems that this will not positively influence their success in a first model-based programming course.

Adult, experienced procedural programmers participating in further education are a growing student population. In “Learning Object-Orientation by Professional Adults” we have presented the results of our investigation of how representatives from this group connect a course in introductory object-oriented programming to their professional life. In general, the students found it to be mainly their own responsibility to connect their learning from the course to their professional life. One implication of this is that teachers designing courses for this particular target group do not need to use professional programming tools, but can use pedagogical tools such as BlueJ. The need to “speak the same language” could imply that teachers undertaking this kind of teaching should have worked in industry at some point in their career.

7.2 Pedagogical Pattern Practice

Design patterns have had a major impact on software design. They give designers a common language to discuss common design problems and their solutions. This idea has been applied to many other areas, including pedagogy. One crucial difference between design patterns and pedagogical patterns is the underlying values that a given pattern strives for. In software the two values that most design patterns aim for are coupling and cohesion. These two values are commonly accepted in the software community, and therefore are not a centre for heated discussion. The opposite is true in teaching, which includes many different pedagogical theories, each with its own set of values. Ignoring the values makes the discussion difficult because a given pattern can support one participant’s values while counteracting another participant’s values. Making the values explicit in the categorization facilitates the discussion. It will also help the teacher to select an appropriate pattern given the knowledge of his problem (i.e., what value is “problematic”- work, involvement, upper limit or collaboration) and the situation he is in (planning, teaching or evaluating):

	Planning	**Performing**	**Evaluating**
Work	Peer Feedback, Embrace Corrections, Grade It Again Sam, Active Students, Prefer Writing, Invisible Teacher, Explore For Yourself, Study Groups	Try It Yourself, Own Words, Peer Feedback, Students Online Portfolio, Mock Exam, Challenge Understanding, Honor Questions, Shotgun Seminar, Test Tube, Teacher Select Teams, Explore For Yourself, Adopt-an-Artefact, Critique, Role Play, Students Design Sprint, Larger than Life	Peer Grading, One Grade For All
Involvement	Feedback, Different Experience Level, Students Decide, Real World Experience, War Game	Feedback, feedback sandwich, gold star, Mock exam, acquire participants' feedback, anonymous feedback, honor questions, teacher select teams adopt an artefact role play larger than life	Fair grading
Upper Limit	Feedback, Embrace Corrections, Early Warning, Grade it Again Sam, Different Experience Level, Explore Yourself, Study Groups, Expand the Known World	Feedback Differentiated Feedback, Try it yourself, Self test, Own Words, Early Warning, Challenge Understanding, Honor Questions, Test Tube, Explore Yourself,	

		Critique, Expand the known World	
Collaboration	Peer Feedback, Grade It Again Sam, Invisible Teacher, Students Decide, Groups Work, Study Groups, War Game	Peer Feedback, Acquire Participants' Feedback, Teacher Select Teams, Adopt-an-Artefact, Groups Work, Role Play Students Design Sprint	Peer Grading, One Grade For All

Table 11: Pedagogical patterns categorized according to pedagogical values (work, involvement, upper limit and collaboration) and teaching activities (planning, performing and evaluating). The table is from "Categorizing Pedagogical Patterns by Teaching Activities and Pedagogical Values"

If a teacher, for example, is preparing his course and is afraid that the students will work solo (i.e., not collaboratively), he could look at the pedagogical pattern "Invisible Teacher" (Active Learning, n.d.) to get inspiration on how to solve this problem.

As described above, several authors claim that pedagogical patterns do not have widespread use. We have been able to find only a few articles (e.g., Jalloul (2000); Haberman (2006) plus Derntl and Motschnig-Pitrik (2004)) describing or discussing the use of pedagogical patterns for either development or documentation of pedagogical practice. Fincher (1999) claims

> a Pattern Language of Pedagogy is possible and achievable and that for historic and disciplinary reasons, CS is singularly well positioned to create such a tool and a particularly fertile ground for its use (p. 331).

However, we have not found this to be the case – there is no commonly accepted catalogue of pedagogical patterns. Fincher (1999) observes that

> Most obviously in this respect they lack a cohesion of scale, encompassing material from the use of learning theory concepts to improve the effectiveness of lecture sessions […] to a precise method for teaching the use of accessors and mutators for accessing an object's private data (p. 336).

The proposed categorisation in "Categorizing Pedagogical Patterns by Teaching Activities and Pedagogical Values" and the accompanying template in Bennedsen and Eriksen (2003) should give a better and more abstract description of pedagogical patterns.

Derntl and Botturi (2006) describe key terms related to a pattern system and describe use cases for them. They stress that “a pedagogical pattern collection needs to be based on some clearly expressed pedagogical baseline” (p. 139-140). The template’s and classification’s grounding in a specific pedagogical theory (constructivism or social constructivism) could be seen as one solution to this, thereby enhancing pedagogical patterns’ popularity.

One of the hypotheses in “The Dissemination of Pedagogical Patterns is that pedagogical patterns can be used as an artefact for creating and communicating about pedagogical practice at an institution. This is also the idea of Haberman (2006). She describes a series of activities that use pedagogical patterns as a means of communication about pedagogy. Her series of activities is promising, but does not seem to address the issue of a pedagogical baseline. The respondents in “The Dissemination of Pedagogical Patterns” also find pedagogical patterns useful for creating and communicating about pedagogical practice at an institution. The explicit focus on the underlying pedagogical value system is important and – using a template with an explicit focus on pedagogical values – it should give rise to more qualified pedagogical discussions.

8 Discussion

He that is not open to conviction, is not qualified for discussion.
Richard Whately (1787-1863), English logician

In this chapter we will discuss the research presented here in relation to some of the current "hot topics" in the computer science education research community today. The chapter is not intended to cover all of the hot topics, but only topics we find relevant to comment on.

8.1 Objects-First Debate

As described in section 3.2.1, there is no common opinion of what an "objects-first" approach is, nor is there an agreement among teachers whether an objects-first course is an easier or harder way for students to learn programming (Lister et al., 2006). As an example of this debate, Hu (2004) discusses in detail the usefulness of an objects-first approach. His conclusion is "Teaching objects-first may well be (and in fact, is, in many schools) counter-productive and harmful." (p. 216). He finds four problems that still need to be answered in order to use an objects-first approach:

> 1. The students in CS1 are not prepared for the complexity and the level of abstraction of OO.
>
> 2. There's too much material for a single CS1 course to handle.
>
> 3. The weakened coverage on traditional algorithms due to the shifted emphasis and time constraints.
>
> 4. The concerns of less or no coverage on structured programming. (p. 210)

Indeed, these are relevant problems but – as noticed above – there is a big diversity in the goals, content and teaching methods of courses using an objects-first approach. His problems seem to be unsubstantiated claims, which may be true for some concrete courses and students. In the implementations of the model-based approach researched in this dissertation, Hu's problems were not experiences as such neither by the teacher of the course nor by the teachers of the following courses. As stated in section 6.1, the model-based approach addresses explicitly the different abstraction levels in a program, uses traditional structured programming principles like loop invariants, covers almost the entire book (the book used was "Objects first with Java: a practical introduction using Blue J" (Barnes & Kölling, 2005)) and covers traditional sweep algorithms (find one object that fulfills a given criteria, find all objects that fulfill a given criteria). However, there is no coverage of traditional algorithms like binary search and sorting. The students use standard implementations of these algorithms thereby exposing the students to a more relevant topic: reuse of code and addressing the "not invented here" syndrome (Oliver & Dalbey, 1994). In short, the model-based approach fits nicely to modern software engineering methods and the competences the students get are useful in industry as well as in their further studies.

As the answer to research question R1 indicates, we find the model-based approach useful and the results are encouraging. It shows that it is possible to teach introductory programming to students using an objects-first approach. However, we do not claim that the very fine results can be ascribed only to the model-based approach; many things influence the outcome of a student.

8.2 The Role of Conceptual Modeling in Introductory Programming Courses

As descried in the introduction, Knudsen and Madsen (1988) describe three perspectives on the role of the programming language: *Instructing the computer, Managing the program description* and *Conceptual modelling*. Including the conceptual modelling perspective has a positive impact on the students' skills and their understanding of the programming process ("Revealing the Programming Process"; "Learning Object-Orientation by Professional Adults"; Gantenbein, 1989). It is our experience that the general omission of conceptual modelling is the major reason for the problems identified in CC2001:

> 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. (p. 23)

CC2001 generally ignores conceptual modelling in the objects-first recommendations for CS1. Aspects of conceptual modelling are mentioned only briefly and the recommended time to be spent on the subject is four core hours[29]!

In her PhD dissertation, Pauline H. Mosley (2002) makes a textbook analysis of ten major selling object technology textbooks. She counted the number of pages devoted to different topics. One of her observations is that

> Approximately, 95% of the texts do not discuss the Unified Modeling Language. 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. (p. 76)

Textbooks seem to be a good indicator of the teaching taking place; Weiss, Banilower, McMahon and Smith (2001) found that 96% of secondary science teachers (and 94% of math teachers) used one or multiple textbooks, and Goldstein (1978) estimated that 75% of the classroom time and 90% of homework time involved textbook use. The focus on "instructors' kits" in the advertisement of introductory programming books for university courses lead us to believe that the same is the case for introductory programming courses at university level. We see the absence of UML models as an indication of the focus of programming courses. The courses focus on the semantic and syntactic details of a given programming language and not on the given programming language as a concrete way to express solutions in a given programming paradigm.

8.3 What are the Goals of an Introductory Programming Course?

In general it seems like many discussions among computer science teachers boils down to differences in the learning goals for an introductory programming course. If, as it seems to be in many cases, the goals are not stated explicitly, the discussions very quickly become unproductive and may even end in mudslinging.

[29] A core hour is defined in CC2001 as "the in-class time required to present the material in a traditional lecture-oriented format" (p. 15). … "students are expected to devote three hours of time outside class for each in-class hour," (p. 16). One core-hour therefore equals four hours of student work.

Biggs (1996) argues for "constructive alignment". Constructive alignment is

> a marriage of the two thrusts [constructivist learning theory and instructional design literature], constructivism being used as a framework to guide decision-making at all stages in instructional design: in deriving curriculum objectives in terms of performances that represent a suitably high cognitive level, in deciding teaching/learning activities judged to elicit those performances, and to assess and summatively report student performance. The "performances of understanding" nominated in the objectives are thus used to systematically align the teaching methods and the assessment. (p. 347)

The design of teaching and learning activities are guided by the learning outcomes, see e.g., figure 2.2 in Biggs (2003).

We hope that the interim community review of the curriculum recommendations currently undertaken by ACM will entail that a selection of more explicitly stated goals will be available for the designers of introductory programming courses to choose from. It is furthermore our hope that future research in the area of introductory programming addresses the lack of explicitly stating the goals of the course/courses under investigation, and – if the research is empirically based – deepens on the description of the goals of the course so that the results of the research better can be weighted against the readers own teaching practice and course goals.

8.4 Computing Education Challenges

In the "grand challenges" for computing education (McGettrick et al., 2005), the authors present a sub-challenge:

> The concept of IQ, though controversial, provides a measure, relative to the whole population average, of innate intellectual capability, independent of age. Similarly, we suggest a sub-challenge to develop a 'programmers quotient' to give a measure of programming ability, relative to the whole population, that would remain the same independent of programming experience (p. 46)

We find this an unachievable goal. As Lister (2005a) notes "One thing is certain: there is no "silver bullet". That is, there is no simple and accurate test for PQ. If there was a silver bullet, then we would already have found it." (p. 15). The studies in this dissertation support Lister's comments. Although we have found a correlation between high-school math and programming, the correlation only accounts for 16% of the exam score.

8.5 Computer Science Education Research

Generalizations in the area of educational research are difficult since many different variables influence the students' learning (see e.g., section 4.1 on didactical models). The thorough description of the goals, pedagogy and content of the courses under study in the dissertation hopefully helps other teachers to evaluate if the findings in this research can have implications for their own teaching.

Another way to make the results more generalizable is to increase the number of different participating entities (teachers, students, institutions, cultural contexts etc). This is the case in the large-scale research that Adey and Shayer used to prove that abstraction was teachable in primary school and that such teaching indeed enhanced the learning outcome of students.

Fincher et al. (2005) describe design considerations when designing computer science education research projects that are multi-institutional and multi-national. The multi-national studies aims at making the results more generalizable. However, it might be the case that one compares "apples and oranges" since the context of participating entities differ too much. In e.g., Lister et al (2004) they compare students ability to trace code. However, the only discussion on differences relates to the programming language used and the familiarity with multiple choice questions; there is no discussion on the difference between goals of the courses. It might be the case that some courses have code tracing as an explicit goal while others pay no attention to this at all. Again, we would hope for a more elaborate description of the research context as seems to be the case in other areas of social science research.

8.6 Transfer of Teaching Experiences

Transfer of experience within a community is normally highly valued. This also seems to be the case in the community of computer science teachers where e.g., the nifty assignments session at SIGCSE (Parlante, 2007) is one of the most popular and well-sessions at the entire conference. However, this is only about exercises, not teaching experiences in general. The pedagogical pattern project aims at transferring experiences; the natural question is whether this is the right way to transfer experience (written and generalized in a particular template) or there are too many factors involved in a teaching situation so that the “bandwidth” of the media is simply not big enough. Are other forms like e.g. pair-teaching superior to transferring experience by written communication? This question seems not possible to give a yes/no answer to; it seems to depend on the available resources and the participating teachers. From our own experience pair-teaching is very effective given that the teachers respect and feel secure with each other. However, the cost of human resources in that form of experience transfer is relatively high

Haberman (2006) argues that traditional forms of transfer of experience are unsuitable in the case of computer science teachers:

> Unlike other communities of practice, the transfer of practical expertise to novice teachers cannot be achieved through classical ongoing ‘‘face-to-face’’ apprenticeship because teaching is in the end an individual activity. Usually, teachers carry out expert – novice support as well as expert – expert cooperation through sharing self-made learning materials (i.e., assignments and exams), a means of communication that does not guarantee the transfer of teaching expertise. (p. 88)

We do not sanction the categorical claim made by Haberman. There are many ways of transferring experience, each with is strength and weaknesses. The strength of pedagogical patterns is, in our view, especially in three areas:

1. The teacher(s) creating the pedagogical pattern can use it as a reflection tool for his/her own teaching,

2. Experiences are made public and given a name (we have e.g. often used the word spiral approach to communicate that a given topic was covered several times in the course, each time at a deeper level),

3. Teachers are not reliant on other teacher(s) to be present (and the use of human resources is consequently lower).

Pedagogical patterns are not the only way to transfer experience, but we see them as a valuable supplement to all the other ways to do this.

9 Further Work

Prediction is very difficult, especially about the future.
Niels Bohr (188-1962), Danish physicist

This dissertation contributes to building knowledge of how to teach introductory object-oriented programming. However, many more interesting research questions could be answered for model-based teaching; naturally much more research needs to be done in order to give better explanations for, and guidance on, teaching model-based introductory programming.

It is generally accepted that it is very difficult to learn to programming. However, the belief that there generally are high drop-out and failure rates does not seem to originate from any official statistic or sound investigation of the subject – it is more in the realm of folk-wisdom, and claims that have been said so often that they are accepted as truths. The only source of information that we know of is authors who give pass-rates for their particular introductory course in articles describing other issues (e.g., Guzdial and Forte (2005)). We find it problematic not to have more solid evidence for this claim. False views on failure and pass rates can have serious implications for the quality of introductory programming courses. A lecturer with a high failure rate might accept that “this is just the way programming courses are since all programming courses have high failure rates” and consequently fail to take action to improve the course in order to reduce the failure rate. It would be relevant to design a study aiming at finding the average failure and pass rate for CS1 courses around the world so that the discussion can be based at facts rather than folks wisdom. One suggestion could be that the ACM Education Council and others engage in this work in order to provide reliable and representative data from as many institutions as possible.

Focusing on the programming process is important. It could be interesting to “turn the camera around” and have the students record their own problem solving activities in the same way as the process recordings. In this way, the exercises could focus on both the product (the program) and the process. By having the students to deliver both a program and a recording of their programming process, the teacher has a way to evaluate and give feedback to the process used by the students. It would be interesting to investigate this and see if it positively impacts learning and how a practical set up can be made. Simply having the students to hand in an entire recording of their programming process seems unproductive.

The described categorization of pedagogical patterns would indeed be interesting to implement. It is – as described in the article – a way to restructure the pedagogical patterns web-site in order to give structure to the pattern catalogue. It will be interesting to implement the suggested categorization and evaluate its usefulness of this. It is rare that one structure fits all, so it would be interesting to expand the implementation with the possibility for users to create their own categorizations in the same way as it is possible to do "social book-marking" on del.icio.us (http://del.icio.us/)

It seems like textbooks shape much of the teaching activities. Textbooks highly influence the content and structure of a course (Good, 1993; Lin et al., 1999). As a consequence, it seems necessary to produce a good textbook using the model-based approach. Many good textbooks exist, but none is organized according to a model-based approach and with a strong focus on the programming process.

Success (or the lack of it) in learning to program is a major concern for many researchers. It seems like looking for other variables that predict success is not very productive. The studies done as a part of this dissertation and the ones analyzed in section 3.2.3 point in very different directions and no general conclusion is obtainable. This indicates that the goal of finding general success factors for learning to program is unachievable; we need to aim at a less general level. In this research the level has not been learning to program in general but learning to program using a model-based approach. Another limitation is that the research is only done in one context (University of Aarhus); it would be interesting to repeat the research in other contexts.

As a supplement to the quantitative studies of success factors it would be relevant to do more qualitative studies. This will give us the possibility to use a more "open-minded" research approach indeed in the sense that it is not about answering hypotheses but creating new hypotheses and hopefully new hypotheses that explain better why students struggle with programming and what we can do to help students learn programming better. We have started to do this in a small scale by interviewing students about their own experience of why they think they passed the introductory programming course; the results are partly documented in (Caspersen et al., 2007), but extending this research to a larger group of students might reveal new success factors; most likely factors that are not as easily measurable as "previous math score"

When learning to program the student needs a tool to practice programming. Currently we use BlueJ, a tool that illustrate the connection between aspects of the class model that can be derived from the code and the code itself. Yet, it is only possible to derive a "usage" link between classes, not an association or an aggregation. It would be interesting to try to extend BlueJ with a facility to show associations/aggregations (with cardinality) and evaluate the influence of this in a model-based programming course. Furthermore, BlueJ does not focus much on communicating objects. Matilda Östling (2004) has created a sequence diagram editor for BlueJ; this could be used as a starting point for exploring the learning outcome of usage of such a tool.

One of the topics students seem to have difficulties with is the mental model of program execution. We have evaluated a proposed competency hierarchy regarding object interaction and its role in teaching and learning (Bennedsen & Schulte, 2006), and we have evaluated the hierarchy among teachers in (Schulte & Bennedsen, 2006). This hierarchy is found to be both relevant and useful. We would like to explore the use of this hierarchy further; among other things, to try to create validated and standardized instruments that can be used to assess a given student's level of competence. We also would like to evaluate the hierarchy by assessing the students' object-interaction competence in the middle of the course and at the end of the course, to see if growth in object-interaction competency correlates strongly with students' overall learning and to see if students participating in a model-based course differ from other students.

Learning to program is notoriously difficult – and so is teaching it. Many students participate in introductory programming courses so answering research questions that can move both the theoretical and practical side of computer science education forward is much needed. Future research must therefore be guided by a search for new theories and answers to research questions that acknowledges both the theoretical and practical side of this field; research that has a solid methodological foundation and produce findings that are relevant for teachers in this field.

10 References

Abelson, H., & Sussman, G. J. (1985). *Structure and interpretation of computer programs*. Cambridge, MA, United States: MIT Press.

Aboulafia, A., & Nielsen, J. L. (1997). Situated learning – nogle videnskabsteoretiske synspunkter (situated learning- some theory of science based views. in Danish). In O. Danielsen, L. Dirckinck-Holmfeld, B. Holm Sørensen, J. Nielsen & B. Fibiger (Eds.), *Læring og multimedier* (). Aalborg, Denmark: Aalborg Universitetsforlag.

ACEC'04. (2004). Papers presented at the *Proceedings of the Australian Computers in Education 2004 Conference*, Adelaide, Australia. Retrieved June 14, 2006 from http://www.acec2006.info/confpapers/default2.asp?pid=7211

ACM. (2002). *Computing's highest honor awarded to inventors of dominant programming style (A.M. turing award)*. Retrieved September 4, 2007, from http://info.acm.org/announcements/turing_2001.html

ACT. *ACT assessment*. Retrieved August, 31, 2006, from http://www.act.org/aap/

Active Learning. (n.d.). *Patterns for active learning*. Retrieved October 25, 2006, from http://www.pedagogicalpatterns.org/current/activelearning.pdf

Adams, J. C. (1996). Object-centered design: A five-phase introduction to object-oriented programming in CS1–2. *SIGCSE '96: Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, Pennsylvania, United States. 78-82.

Adams, J., & Frens, J. (2003). Object-centered design for java: Teaching OOD in CS-1. *SIGCSE '03: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, Reno, Nevada, United States. 273-277.

Adey, P., & Shayer, M. (1994). *Really raising standards. cognitive intervention and academic achievement*. London, United Kingdom: Routledge.

Advanced Computer Studies. (n.d.). *Aarhus Business College - computer science*. Retrieved May, 10, 2007, from http://www.aabc.dk/sw7327.asp

Alexander, C. (1979). *The timeless way of building*. New York, Ney York, United Stated: Oxford University Press.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A pattern language. towns, buildings, construction*. New York, New York, United States: Oxford University Press.

Alford, K. L. (2003). Video faqs - instruction-on-demand. *Proceedings of the 33rd Frontiers in Education Conference.* Boulder, Colorado, United States. S2e-20.

Alphonce, C., & Ventura, P. (2002). 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. 70-74.

Alphonce, C., & Ventura, P. (2003). Using graphics to support the teaching of fundamental object-oriented principles in CS1. *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications,* Anaheim, CA, United States. 156-161.

Alspaugh, C. A. (1972). Identification of some components of computer programming aptitude. *Journal for Research in Mathematics Education, 3*(2), 89-98.

Alvesson, M., & Skoldberg, K. (2000). *Reflexive methodology: New vistas for qualitative research*. London, United Kingdom: Sage Publications.

American Folklore. (2006). *John henry, the steel-driving man: A west virginia folktale from american folklore.* Retrieved November, 20, 2007, from http://www.americanfolklore.net/folktales/wv2.html

Anderson, J. R. (1985). *Cognitive psychology and its implications*. New York, New York, United States: W.H. Freeman.

Andersson, R., & Roxå , T. (2000). Encouraging students in large classes. *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, Austin, Texas, United States. 176-179.

Angen, M. J. (2000). Evaluating interpretive inquiry: Reviewing the validity debate and opening the dialogue. *Qualitative health research, 10*(3), 378-395.

Association for computing machinery. (n.d.). Retrieved April, 21, 2006, from http://www.acm.org/

Astrachan, O., & Reed, D. (1995). AAA and CS1: 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.

Astrachan, O., Selby, T., & Unger, J. (1996). An object-oriented, apprenticeship approach to data structures using simulation. *Proceedings of the 26th Frontiers in Education Conference.* Salt Lake City, Utah, United States. , 130-134.

Astrachan, O., Bruce, K., Koffman, E., Kölling, M., & Reges, S. (2005). Resolved: Objects early has failed. *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, United States. 451-452.

Atchison, W. F., Conte, S. D., Hamblen, J. W., Hull, T. E., Keenan, T. A., Kehl, W. B., et al. (1968). Curriculum 68: Recommendations for academic programs in computer science: A report of the ACM curriculum committee on computer science. *Communications of the ACM, 11*(3), 151-197.

Austin, H. S. (1987). Predictors of Pascal programming achievement for community college students. *SIGCSE '87: Proceedings of the Eighteenth SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, United States. 161-164.

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

Ayer, A. J. (1959). *Logical positivism*. New York, New York, United States: The Free Press.

Bailie, F., Courtney, M., Murray, K., Schiaffino, R., & Tuohy, S. (2003). Objects first - does it work? *Journal of Computing in Small Colleges, 19*(2), 303-305.

Barker, R. J., & Unger, E. A. (1983). 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. 154-158.

Barnes, D. J. (2002). Teaching introductory java through LEGO MINDSTORMS models. *SIGCSE '02: Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education,* Cincinnati, Kentucky, United States. 147-151.

Barnes, D. J., & Kölling, M. (2005). *Objects first with Java: A practical introduction using BlueJ* (2nd ed.). New York, New York, United States: Prentice Hall.

Barrett, M. L. (1996). Emphasizing design in CS1. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 28*(1), 315-318.

Bassok, M. (1990). Transfer of domain-specific problem-solving procedures. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 16*(3), 522-533.

Bauer, R., Mehrens, W. A., & Vinsonhaler, J. F. (1968). Predicting performance in a computer programming course. *Educational and Psychological Measurement, 28*, 1159-1164.

Bayman, P., & Mayer, R. E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM, 26*(9), 677-679.

Beaubouef, T., & Mason, J. (2005). Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 37*(2), 103-106.

Beck, K. (1999). *Extreme programming explained: Embrace change*. Reading, Massachusetts, United States: Addison-Wesley Professional.

Beck, K., et al. *Manifesto for agile software development*. Retrieved November, 6, 2006, from http://www.agilemanifesto.org/

Becker, R. M. (1995). *The decision matrix on Trial/OJ trial analogy*. Retrieved May, 13, 2007, from http://www.socialresearchmethods.net/OJtrial/ojhome.htm

Becker, B. W. (2001). Teaching CS1 with Karel the robot in Java. *SIGCSE '01: Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education*, Charlotte, North Carolina, United States. 50-54.

Beder, H. W., & Darkenwald, G. G. (1982). Differences between teaching adults and pre-adults: Some propositions and findings. *Adult Education Quarterly, 32*(3), 142-155.

Benander, A., Benander, B., & Sang, J. (2004). Factors related to the difficulty of learning to program in java - an empirical study of non-novice programmers. *Information and Software Technology, 46*, 99-107.

Ben-Ari, M. (1998). Constructivism in computer science education. *SIGCSE '98: Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education,* Atlanta, Georgia, United States. 257-261.

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

Ben-Ari, M. (2004). Situated learning in computer science education. *Computer Science Education, 14*(2), 85-100.

Ben-Ari, M. (2005). Situated learning in ‘This high-technology world’. *Science & Education, 14*(3), 367-376.

Bennedsen, J. (2003). Teaching java programming to media students with a liberal arts background. *Proceedings for the 7th Java & the Internet in the Computing Curriculum Conference (JICC 7),* London, United Kingdom.

Bennedsen, J. (2006a). Collaborating in learning object-orientation in a synchronous, net-based environment. In A. Fjuk, A. Karahasanovic & J. Kaasbøll (Eds.), *Comprehensive object-oriented learning: The learner's perspective* (pp. 157-181). Santa Rose, California, United States: Informing Science Press.

Bennedsen, J. (2006b). The dissemination of pedagogical patterns. *Journal of Computer Science Education, 16*(2), 119-136.

Bennedsen, J., Berge, O., & Fjuk, A. (2005). Examining social interaction patterns for online apprenticeship learning object-oriented programming as the knowledge domain. *European Journal of Open and Distance Learning, 2005*(II)

Bennedsen, J., & Caspersen, M. (2003). Rationale for the design of a web-based programming course for adults. *Proceedings for the International Conference on Open and Online Learning (ICOOL 2003),* University of Mauritius, Mauritius.

Bennedsen, J., & Caspersen, M. (2006a). Abstraction ability as an indicator of success for learning object-oriented programming? *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 38*(2), 39-43.

Bennedsen, J., & Caspersen, M. (2006b). 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, United States. M4E-16-M4E-21.

Bennedsen, J., & Caspersen, M. E. (1995). Programmering med klodser - et moderne programmeringskursus (programming by using modules - a modern programming course. in Danish). *DMLF bladet, 2* Retrieved May 11, 2007 from http://www.daimi.au.dk/~jbb/publications/PROGRAMMING_USING_MODULES.PDF

Bennedsen, J., & Caspersen, M. E. (2004). Programming in context: A model-first approach to CS1. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* Norfolk, Virginia, United States. 477-481.

Bennedsen, J., & Caspersen, M. E. (2005a). 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, Washington, United States. 155-163.

Bennedsen, J., & Caspersen, M. E. (2005b). Revealing the programming process. *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education,* St. Louis, Missouri, United States. 186-190.

Bennedsen, J., & Caspersen, M. E. (2007a). Assessing process and product – A practical lab exam for an introductory programming course. *ITALICS, Innovation in Teaching and Learning in Information and Computer Sciences, 6*(4), 183-202. Retrieved October 30, 2007 from http://www.ics.heacademy.ac.uk/italics/vol6iss4/Bennedsen.pdf

Bennedsen, J., & Caspersen, M. E. (2007b). Failure rates in introductory programming. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 39*(2)

Bennedsen, J., & Caspersen, M. E. (2008a). Exposing the programming process. In J. Bennedsen, M. E. Caspersen & M. Kölling (Eds.), *Reflections on the teaching of programming* Berlin/Heidelberg, Germany: Springer-Verlag.

Bennedsen, J., & Caspersen, M. E. (2008b). Model-driven programming. In J. Bennedsen, M. E. Caspersen & M. Kölling (Eds.), *Reflections on the teaching of programming* Berlin/Heidelberg, Germany: Springer-Verlag.

Bennedsen, J., & Caspersen, M. E. (2008c). Optimists have more fun, but do they learn better? – on the influence of emotional and social factors on learning introductory computer science. *Journal of Computer Science Education, 18*(1), Accepted for publication.

Bennedsen, J., & Eriksen, O. (2003). Applying and developing patterns in teaching. *Proceedings of the 33rd ASEE/IEEE Frontiers in Education Conference,* Westminster, Colorado, United States. T4A-2-T4A-7.

Bennedsen, J., & Eriksen, O. (2006). Categorizing pedagogical patterns by teaching activities and pedagogical values. *Journal of Computer Science Education, 16*(2), 157-172.

Bennedsen, J., & Fjuk, A. (2006). Learning object-orientation by professional adults. *International Journal of Continuing Engineering Education and Life-Long Learning, 16*(6), 453-465.

Bennedsen, J., & Schulte, C. (2006). A competence model for object-interaction in introductory programming. *Proceedings of the 18th Workshop of the Psychology of Programming Interest Group,* Brighton, United Kingdom.

Bennedsen, J., & Schulte, C. (2007). What does "objects-first" mean? an international study of teachers' perception of objects-first. *Proceedings of the 7th Baltic Sea Conference on Computing Education Research - Koli Calling,* Koli, Finland. not yet published.

Berge, O. (2006). *Reuse of digital learning resources in collaborative learning environments*. Unpublished Philosophiae Doctor, Faculty of Mathematics and Natural Sciences, University of Oslo, Norway.

Berge, O., & Fjuk, A. (2006a). Understanding the roles of online meetings in a net-based course. *Journal of Computer Assisted Learning, 22*(1), 13-23.

Berge, O., & Fjuk, A. (2006b). Reuse of learning resources in object-oriented learning. In A. Fjuk, A. Karahasanovic & J. Kaasbøll (Eds.), *Comprehensive object-oriented learning: The learner's perspective* (pp. 131-155). Santa Rose, California, United States: Informing Science Press.

Berge, O., Fjuk, A., Groven, A., Hegna, H., & Kaasbøll, J. (2003). Comprehensive object-oriented learning - an introduction. *Journal of Computer Science Education, 13*(4), 331-335.

Berger, P. L., & Luckmann, T. (1969). *The social construction of reality. A treatise in the sociology of knowledge*. London, United Kingdom: Penguin Books.

Bergin, J., Koffman, E., Proulx, V. K., Rasala, R., & Wolz, U. (1999). Objects: When, why, and how. *Journal of Computing in Small Colleges, 14*(4), 216–219.

Bergin, J. (n.d.). *Fourteen pedagogical patterns*. Retrieved August 31, 2007, from http://csis.pace.edu/~bergin/PedPat1.3.html

Bergin, S., & Reilly, R. (2005). The influence of motivation and comfort-level on learning to program. *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group,* University of Sussex, Brighton United Kingdom. 293-304.

Bergin, J., Clancy, M., Slater, D., Goldweber, M., & Levine, D. B. (2007). Day one of the objects-first first course: What to do. *SIGCSE '07: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education,* Covington, Kentucky, United States. 264-265.

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, United States. 411-415.

Bergin, S., & Reilly, R. (2006). Predicting introductory programming performance: A multi-institutional multivariate study. *Computer Science Education, 16*(4), 303-323.

Berglund, A., Daniels, M., & Pears, A. (2006). Qualitative research projects in computing education research: An overview. *ACE '06: Proceedings of the 8th Australian Conference on Computing Education,* Hobart, Australia. 25-33.

Bevan, W., & Dukes, W. F. (1967). Stimulus-variation and recall: The role of belongingness. *The American Journal of Psychology, 80*(2), 309-312.

Biamonte, A. J. (1964). Predicting success in programmer training. *SIGCPR '64: Proceedings of the Second SIGCPR Conference on Computer Personnel Research,* New York, United States. 9-12.

Bierre, K. J., & Phelps, A. M. (2004). The use of MUPPETS in an introductory java programming course. *CITC5 '04: Proceedings of the 5th Conference on Information Technology Education,* Salt Lake City, Utah, United States. 122-127.

Bierre, K., Ventura, P., Phelps, A., & Egert, C. (2006). Motivating OOP by blowing things up: An exercise in cooperation and competition in an introductory java programming course. *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education,* Houston, Texas, United States. 354-358.

Biggs, J. (1996). Enhancing teaching through constructive alignment. *Higher Education, 32*(3), 347-364.

Biggs, J. B. (2003). *Teaching for quality learning at university. what the student does*. Maidenhead, United Kingdom: Open University Press.

Bixler, B., & Spotts, J. (1998). Screen design and levels of interactivity in web-based training. *Proceedings of the 1998 IVLA Annual Conference: Visual Literacy in an Information Age,* Athens, Georgia, United States. 43-50.

Bloom, B. S., Krathwohl, D. R., & Masia, B. B. (1956). *Taxonomy of educational objectives. the classification of educational goals. handbook I: Cognitive domain*. New York, United States: Longmans, Green.

Bogdan, R. C., & Biklen, S. K. (1982). *Qualitative research for education. an introduction to theory and methods*. Boston, Massachusetts, United States: Allyn and Bacon.

Bolstad, M. (2004). Design by contract: A simple technique for improving the quality of software. *Proceedings of the 2004 DoD HPCMP Users Group Conference,* Williamsburg, Virginia, United States. 303-307.

Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction, 1*(2), 133-161.

Bond, T. G. E. (2004). Piaget and the pendulum. *Science & Education, 13*(4), 389-399.

Börstler, J., Johansson , T., & Nordström, M. (2002). Teaching OO concepts - a case study using CRC-cards and BlueJ. *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference,* Boston, Massachusetts, United States. T2G-1-T2G-6.

Brooks, J. H., & DuBois, D. L. (1995). Individual and environmental predictors of adjustment during the first year of college. *Journal of College Students Development, 36*, 347-360.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. [reprinted in International Journal of Human-Computer Studies 51(2) 197-211] *International Journal of Man-Machine Studies, 9*(6), 737-751.

Brown, A. L. (1992). Design experiments: Theoretical and methodological challenges in creating complex interventions in classroom settings. *Journal of the Learning Sciences, 2*(2), 141-178.

Bruce, K. B. (2005). Controversy on how to teach CS 1: A discussion on the SIGCSE-members mailing list. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 37*(2), 111-117.

Bruce, K. B., Danyluk, A. P., & Murtagh, T. P. (2001a). 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. 1-4.

Bruce, K. B., Danyluk, A., & Murtagh, T. (2001b). Events and objects first: An innovative approach to teaching JAVA in CS 1. *CCSC '01: Proceedings of the Sixth Annual Consortium for Computing Sciences in Colleges CCSC: Northeastern Conference,* Middlebury, Vermont, United States. 1.

Bruce, K. B., Danyluk, A., & Murtagh, T. (2001c). A library to support a graphics-based object-first approach to CS 1. *SIGCSE '01: Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education,* Charlotte, North Carolina, United States. 6-10.

Bruce, K. B., Danyluk, A., & Murtagh, T. (2004). Event-driven programming facilitates learning standard programming concepts. *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications,* Vancouver, British Columbia, Canada. 96-100.

Bruhn, R. E., & Burton, P. J. (2003). An approach to teaching java using computers. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 35*(4), 94-99.

Bruner, J. (1990). *Acts of meaning*. Cambridge, Massachusetts, United States: Harvard University Press.

Bryman, A. (1988). *Quantity and quality in social research*. London, United Kingdom: Unwin Hyman.

Buck, D., & Stucki, D. J. (2000). Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 32*(1), 75-79.

Buelens, H., Roosels, W., Wils, A., & Rentergem, L. (2002). One year E-learning at the KU leuven: An examination of log-files. *Proceedings of the New Educational Benefits of ICT in Higher Education,* Rotterdam, the Netherlands. 170-174.

Butcher, D. F., & Muth, W. A. (1985). Predicting performance in an introductory computer science course. *Communications of the ACM, 28*(3), 263-268.

Byrne, P., & Lyons, G. (2001). 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. 49-52.

Cafolla, R. (1987-88). Piagetian formal operations and other cognitive correlates of achievement in computer programming. *Journal of Educational Technology Systems, 16*(1), 45-55.

Campbell, D. P. S. (1966). *Vocational interest blanks: Manual*. Stanford, California, United States: Stanford University Press.

Capra, F. (1989). *Uncommon wisdom: Conversations with remarkable people*. Toronto, Canada: Bantam Books.

Capstick, C. K., Gordon, J. D., & Salvadori, A. (1975). Predicting performance by university students in introductory computing courses. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 7*(3), 21-29.

Carbone, A., & Kaasbøll, J. J. (1998). A survey of methods used to evaluate computer science teaching. *ITiCSE '98: Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education,* Dublin City University, Ireland. 41-45.

Caspersen, M. E. (2007). *Educating novices in the skills of programming*. Unpublished PhD, University of Aarhus, Denmark.

Caspersen, M. E., Bennedsen, J., & Larsen, K. D. (2007). Mental models and programming aptitude. *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education,* Dundee, Scotland.

Caspersen, M. E., & Christensen, H. B. (2000). 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. 34-40.

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, United States. 892-900.

Cassidy, S. (2004). Learning styles: An overview of theories, models, and measures. *Educational Psychology, 24*(4), 419-444.

Chamillard, A. T. (2006). Using student performance predictions in a computer science curriculum. *ITICSE '06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* Bologna, Italy. 260-264.

Chee, Y. S. (1995). Cognitive apprenticeship and its application to the teaching of smalltalk in a multimedia interactive learning environment. *Instructional Science, 23*(1), 133-161.

Chen, W. S., & Hirschheim, R. (2004). A paradigmatic and methodological examination of information systems research from 1991 to 2001. *Information Systems Journal, 14*(3), 197-235.

Cherryholmes, C. H. (1992). Notes on pragmatism and scientific realism. *Educational Researcher, 21*(6), 13-17.

Christensen, H. B. (2004). 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. 142-145.

Christensen, H. B., & Caspersen, M. E. (2002). 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. 75-79.

Clark, R., Nguyen, F., & Sweller, J. (2006). *Efficiency in learning: Evidence-based guidelines to manage cognitive load*. San Francisco, United States: Jossey-Bass.

Cockburn, A. (2002). *Agile software development*. Boston, Massachusetts, United States: Addison-Wesley.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2. revised edition ed.). Hillsdale New Jersey, United States: Lawrence Erlbaum Associates.

Cohen, J. (2003). *Applied multiple regression/correlation analysis for the behavioral sciences*. Mahwah, N.J.: Lawrence Erlbaum Associates, Inc.

Collins, A., Brown, J. S., & Newman, S. E. (1989). Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics. In L. B. Resnick (Ed.), *Knowing, learning and instruction: Essays in honour of Robert Glaser* (). Hillsdale New Jersey, United States: Erlbaum.

Collins, A. M., Brown, J. S., & Holum, A. (1991). Cognitive apprenticeship: Making thinking visible. *American Educator, 15*(3), 6-11, 38-46.

COOL. (n.d.). *Comprehensive object-oriented learning*. Retrieved April, 21, 2006, from http://www.intermedia.uio.no/projects/research/cool.html

Cooper, S., Dann, W., & Pausch, R. (2000). Alice: A 3-D tool for introductory programming concepts. *CCSC '00: Proceedings of the Fifth Annual Consortium for Computing Sciences in Colleges CCSC: Northeastern Conference,* Ramapo College of New Jersey, Mahwah, New Jersey, United States. 107-116.

Cooper, S., Dann, W., & Pausch, R. (2003a). Using animated 3D graphics to prepare novices for CS1. *Computer Science Education, 13*(1), 3-30.

Cooper, S., Dann, W., & Pausch, R. (2003b). Teaching objects-first in introductory computer science. *SIGCSE '03: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education,* Reno, Nevada, United States. 191-195.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2003). *Introduction to algorithms*. Cambridge, Massachusetts, United States: MIT Press.

Cranton, P. (1994). *Understanding and promoting transformative learning. a guide for educators of adults*. San Francisco, California, United States: Jossey-Bass.

Creswell, J. W. (2002). *Research design: Qualitative, quantitative, and mixed methods approaches*. Thousand Oaks, California, United States: Sage Publications.

Culwin, F. (1999). Object imperatives! *SIGCSE '99: The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, New Orleans, Louisiana, United States. 31-36.

Cuny, J., & Aspray, W. (2002). Recruitment and retention of women graduate students in computer science and engineering: Results of a workshop organized by the computing research association. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 34*(2), 168-174.

Dale, N. (2005). Content and emphasis in CS1. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 37*(4), 69-73.

Dale, N. B. (2006). Most difficult topics in CS1: Results of an online survey of educators. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 38*(2), 49-53.

Daniels, M., Petre, M., & Berglund, A. (1998). Building a rigorous research agenda into changes to teaching. *ACSE '98: Proceedings of the 3rd Australasian Conference on Computer Science Education*, The University of Queensland, Australia. 203-209.

Danmarks statistik & Ministeriet for Videnskab, teknologi og udvikling. (2005). *Nøgletal om informationssamfundet Danmark 2005 (key numbers about the information society Denmark 2005. in Danish)*. Copenhagen, Denmark: Danmarks statistik & Ministeriet for Videnskab, teknologi og udvikling.

Decker, R., & Hirshfield, S. (1994). The top 10 reasons why object-oriented programming can't be taught in CS 1. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 26*(1), 51-55.

Dehnadi, S. (2006). Testing programming aptitude. *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group*, Brighton, United Kingdom. 22-37.

Dehnadi, S., & Bornat, R. (2006). *The camel has two humps*. Unpublished manuscript.from http://www.cs.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf

Dehoney, J., & Reeves, T. C. (1999). Instructional and social dimensions of class web pages. *Journal of Computing in Higher Education, 10*(2), 19-41.

DeNelsky, G. Y., & McKee, M. G. (1974). Prediction of computer programmer training and job performance using the aabp test. *Personnel Psychology, 27*(1), 129-137.

Denning, P. J. (2004). The field of programmers myth. *Communications of the ACM, 47*(7), 15-20.

Denning, P. J., & Hiles, J. E. (2006). Transformational events. *Computer Science Education, 16*(2), 77-85.

deRaadt, M. (2007). Incorporating programming strategies explicitly into curricula. *Proceedings of the 7th Koli Calling Conference,* Koli, Finland.

Derntl, M., & Botturi, L. (2006). Essential use cases for pedagogical patterns. *Computer Science Education, 16*(2), 137-156.

Derntl, M., & Motschnig-Pitrik, R. (2004). Patterns for blended, person-centered learning: Strategy, concepts, experiences, and evaluation. *Proceedings of the 2004 ACM Symposium on Applied Computing,* Nicosia, Cyprus. 916-923.

Détienne, F. (1997). Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams. *Interacting with Computers, 9*(1), 47-72.

Détienne, F. (2002). *Software design. cognitive aspects*(F. Bott Trans.). London, United Kingdom: Springer.

Dickmann, R. A., & Lockwood, J. (1966). Computer personnel research group, 1966 survey of test use in computer personnel selection. *SIGCPR '66: Proceedings of the Fourth SIGCPR Conference on Computer Personnel Research,* Los Angeles, California, United States. 15-25.

Diederich, J. (1988). *Didaktisches denken. eine einführung in anspruch und aufgabe, möglichkeiten und grenzen der allgemeinen didaktik*. Weinheim, Germany: Juventa.

Dijkstra, E. W. (1969). *Notes on structured programming* No. EWD 249). Eindhoven, the Nederlands: Technological University Eindhoven. Retrieved December 4, 2007 from http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF

Dobs. (2005). *BlueJ: Dobs extension*. Retrieved August 6, 2007, from http://life.uni-paderborn.de/index.php?level1_open=3&level2_open=&level3_open=&storyid=67

Donaldson, J. F., Flannery, D., & Ross-Gordon, J. (1993). A triangulated study comparing adult college students' perceptions of effective teaching with those of traditional students. *Continuing Higher Education Review, 57*(3), 147-165.

Doran, M. V., & Langan, D. D. (1995). A cognitive-based approach to introductory computer science courses: Lesson learned. *SIGCSE '95: Proceedings of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education,* Nashville, Tennessee, United States. 218-222.

Douglas, D. E., & Hardgrave, B. C. (2000). Object-oriented curricula in academic programs. *Communications of the ACM, 43*(11), 249-256.

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, New Jersey, United States: Lawrence Erlbaum.

du Boulay, B., O'Shea, T., & Monk, J. (1999). The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Human-Computer Studies, 51*(2), 265-277.

Dunn, R. (1990). Understanding the Dunn and Dunn learning styles model and the need for individual diagnosis and prescription. *Reading, Writing and Learning Disabilities, 6*, 223-247.

Eckerdal, A. (2004). *On the understanding of object and class*. Unpublished manuscript. Retrieved November 12, 2006, from http://www.it.uu.se/research/publications/reports/2004-058/2004-058-nc.pdf

Eckerdal, A., & Thuné , M. (2005). 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. 89-93.

Eckerdal, A., Thuné, M., & Berglund, A. (2005). What does it take to learn 'programming thinking'? *ICER '05: Proceedings of the 2005 International Workshop on Computing Education Research*, Seattle, Washington, United States. 135-142.

Eckstein, J. (2001). Pedagogical patterns: Capturing best practice in teaching object technology. *Software Focus, 2*(1), 9-12.

Edwards, S. H. (2003a). Improving student performance by evaluating how well students test their own programs. *ACM Journal of Educational Resources in Computing, 3*(3), 1.

Edwards, S. H. (2003b). 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, California, United States. 148-155.

Egert, C., Bierre, K., Phelps, A., & Ventura, P. (2006). Hello, M.U.P.P.E.T.S.: Using a 3D collaborative virtual environment to motivate fundamental object-oriented learning. *OOPSLA '06: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications,* Portland, Oregon, United States. 881-886.

Engel, G., & Roberts, E. (2001). *Computing curricula 2001 computer science, final report*. Retrieved August 31, 2007, from http://www.acm.org/education/curric_vols/cc2001.pdf

Engeström, Y. (1994). *Training for change: New approach to instruction and learning in working life*. Geneva, Switzerland: International Labour Office.

Ennis, R. H., Millman, J., & Tomko, T. N. (1985). *Cornell critical thinking tests level X & level Z manual* (3rd ed.). Pacific Grove, California, United States: Midwest Publications.

Epstein, H. T. (n.d.). *The four R or why Johnny can't reason*. Retrieved May 11, 2007, from http://www.brainstages.net/4thr.html

Erickson, C., & Leidig, P. (1997). A pedagogical pattern for bringing service into the curriculum via the web. *ITiCSE '97: Proceedings of the 2nd Conference on Integrating Technology into Computer Science Education,* Uppsala, Sweden. 54-56.

European Union. (2006a). *ECTS - European credit transfer and accumulation system*. Retrieved August 16, 2007, from http://ec.europa.eu/education/programmes/socrates/ects/index_en.html

European Union. (2006b). *EUROPA - education and training - eLearning - designing tomorrow's education*.http://ec.europa.eu/education/programmes/elearning/index_en.html

Evans, G. E., & Simkin, M. G. (1989). What best predicts computer proficiency? *Communication of the ACM, 32*(11), 1322-1327.

Exam-scale. Retrieved August, 4, 2006, from http://www.retsinfo.dk/_GETDOCM_/ACCN/B19950051305-REGL

Fay, B. (1987). *Critical social science*. Ithaca, New York, United States: Cornell University Press.

Fennefoss, T., & Bennedsen, J. (2003). Fleksibel webundervisning vokser med opgaverne (flexible webteaching grows by its tasks. in Danish). *Tidsskrift for universiteternes efter- og videreuddannelse, 1*, June 8, 2006 from http://www.unev.dk/files/turid_fennefoss_jens_bennedsen.pdf

Fennema, E., & Sherman, J. A. (1976). Fennema-sherman mathematics attitudes scales: Instruments designed to measure attitudes toward the learning of mathematics by females and males. *Journal for Research in Mathematics Education, 7*(5), 324-326.

Fincher, S. (1999). Analysis of design: An exploration of patterns and pattern languages for pedagogy. *Journal of Computers in Mathematics and Science Teaching: Special Issue CS-ED Research, 18*(3), 331-348.

Fincher, S. (2006). Editorial comments: Special issue on CSE pedagogic patterns. *Computer Science Education, 16*(2), 75-75.

Fincher, S., Lister, R., Clear, T., Robins, A., Tenenberg, J., & Petre, M. (2005). Multi-institutional, multi-national studies in CSEd research: Some design considerations and trade-offs. *ICER '05: Proceedings of the 2005 International Workshop on Computing Education Research,* Seattle, Washington, United States. 111-121.

Fincher, S., & Petre, M. (2004). *Computer science education research*. London, United Kingdom: Routledge Falmer.

Fincher, S., Petre, M., & Clark, M. (2001). *Computer science project work. principles and pragmatics*. London, United Kingdom: Springer.

Fincher, S., & Utting, I. (2002). Pedagogical patterns: Their place in the genre. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 34*(3), 199-202.

Firdyiwek, Y. (1999). Web-based courseware tools: Where is the pedagogy? *Educational Technology, 39*(1), 29-34.

Fjuk, A., Karahasanovic, A., & Kaasbøll, J. (Eds.). (2006). *Comprehensive object-oriented learning: The learner's perspective*. Santa Rose, California, United States: Informing Science Press.

Fjuk, A., Holmboe, C., Jahreie, C. F., & Bennedsen, J. (2006). Contextualizing object-oriented learning. In A. Fjuk, A. Karahasanovic & J. Kaasbøll (Eds.), *Comprehensive object-oriented learning: The learner's perspective* (pp. 11-26). Santa Rose, California, United States: Informing Science Press.

Fjuk, A., Karahasanovic, A., & Kaasbøll, J. (2006). Preface. In A. Fjuk, A. Karahasanovic & J. Kaasbøll (Eds.), *Comprehensive object-oriented learning: The learner's perspective* (pp. vii-xi). Santa Rose, California, United States: Informing Science Press.

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, Finland. 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.

Forte, A., & Guzdial, M. (2005). Motivation and nonmajors in computer science: Identifying discrete audiences for introductory courses. *Education, IEEE Transactions on, 48*(2), 248-253.

Fowler, F. J. (1993). *Survey research methods*. Newbury Park, California, United States: Sage Publications.

Fowler, M. (1997). *Analysis patterns. reusable object models*. Menlo Park, California, United States: Addison Wesley.

Fowler, M., & Scott, K. (2000). *UML distilled (2nd ed.): A brief guide to the standard object modeling language*. Boston, Massachusetts, United States: Addison-Wesley Longman Publishing Co.

Fujaba. (2007). *Fujaba tool suite*. Retrieved August 6, 2007, from http://www.fujaba.de/

Gall, M. D., Gall, J. P., & Borg, W. R. (2003). *Educational research. an introduction*. Boston, Massachusetts, United States: Allyn and Bacon.

Gamma, E., Helm, R., Johnson, R., & Vlissides , J. (1995). *Design patterns. elements of reusable object-oriented software*. Reading, Massachusetts, United States: Addison-Wesley.

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.

Georgsen, M. (2004). Introduktion (introduction . in Danish). In M. Georgsen, & J. Bennedsen (Eds.), *Fleksibel læring og undervisning. erfaringer, konsekvenser og muligheder med ikt (flexible learning and teaching. experiences, consequences and possibilities with ICT. in Danish)* (pp. 9-23). Aalborg, Denmark: Aalborg Universitetsforlag (Aalborg University Press).

Gibbs, D. C. (2000). The effect of a constructivist learning environment for field-dependent/independent students on achievement in introductory computer programming. *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education,* Austin, Texas, United States. 207-211.

Gick, M. L., & Holyoak, K. J. (1983). Schema induction and analogical transfer. *Cognitive psychology, 15*(1), 1-38.

Gilmore, D. J. (1990). Methodological issues in the study of programming. In J. Hoc, T. R. G. Green, R. Samurcay & D. J. Gilmore (Eds.), *Psychology of programming* (pp. 83-98). London, United Kingdom: Academic Press.

Gilmore, D. J., & Green, T. R. G. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies, 21*(1), 31-48.

Glasersfeld, E. v. (2002). *Radical constructivism. a way of knowing and learning*. London, United Kingdom: Routledge.

Goldenson, D. R., & Wang, B. J. (1991). Use of structure editing tools by novice programmers. In J. Koenemann-Belliveau, T. C. Moher & S. P. Robertson (Eds.), *Empirical studies of programmers: Fourth workshop* (pp. 99-120). Norwood, New Jersey, United States: Ablex Pub.

Goldstein, P. (1978). *Changing the American school book*. Lexington, Massachusetts, United States: Heath Publishing Company.

Goles, T., & Hirschheim, R. (2000). The paradigm is dead, the paradigm is dead ... long live the paradigm: The legacy of Burrell and Morgan. *Omega, 28*, 249-268.

Good, R. (1993). Editorial: Science textbook analysis. *Journal of Research in Science Teaching, 30*(7), 619.

Goold, A., & Rimmer, R. (2000). Factors affecting performance in first-year computing. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 32*(2), 39-43.

Gorgone, J. T., Davis, G. B., Valacich, J. S., Topi, H., Feinstein, D. L. & Longenecker, J.,Herbert E. (2002). *IS 2002. Model curriculum and guidelines for undergraduate degree programs in information systems*. Retrieved April, 21, 2006, from http://www.acm.org/education/is2002.pdf

Gorham, J. (1985). Differences between teaching adults and pre-adults: A closer look. *Adult Education Quarterly, 35*(4), 194-209.

Gotterer, M., & Stalnaker, A. W. (1964). Predicting programmer performance among non-preselected trainee groups. *SIGCPR '64: Proceedings of the Second SIGCPR Conference on Computer Personnel Research,* New York, United States. 29-37.

Govindasamy, T. (2001). Successful implementation of e-learning: Pedagogical considerations. *The Internet and Higher Education, 4*(3-4), 287-299.

Greeno, J. G., Collins, A. M., & Resnick, L. B. (1996). Cognition and learning. In D. C. Berliner, & R. C. Calfee (Eds.), *Handbook of educational psychology* (pp. 15-46). New York, United States: Macmillan.

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.

Gries, D. (2002). Where is programming methodology these days? *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 34*(4), 5-7.

Guba, E. G. (1990). The alternative paradigm dialog. In E. G. Guba (Ed.), *Paradigm dialog* (pp. 17-27). Newbury Park, California, United States: Sage Publications.

Guzdial, M., & Tew, A. E. (2006). Imagineering inauthentic legitimate peripheral participation: An instructional design approach for motivating computing education. *Proceedings of the 2006 International Workshop on Computing Education Research,* University of Kent, Canterbury, United Kingdom. 51-58.

Guzdial, M. (1995). Centralized mindset: A student problem with object-oriented programming. *SIGCSE '95: Proceedings of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education,* Nashville, Tennessee, United States. 182-185.

Guzdial, M. (2003). A media computation course for non-majors. *ITiCSE '03: Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education,* Thessaloniki, Greece. 104-108.

Guzdial, M., & Forte, A. (2005). Design process for a non-majors computing course. *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education,* St. Louis, Missouri, United States. 361-365.

Haberman, B. (2006). Pedagogical patterns: A means for communication within the CS teaching community of practice. *Computer Science Education, 16*(2), 87-103.

Hadar, I., & Hadar, E. (2007). An iterative methodology for teaching object oriented concepts. *Informatics in education, 6*(1), 67-80.

Hadjerrouit, S. (1998). A constructivist framework for integrating the java paradigm into the undergraduate curriculum. *ITiCSE '98: Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education,* Dublin City University, Ireland. 105-107.

Hadjerrouit, S. (1999). 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. 171-174.

Hadjerrouit, S. (2005). Constructivism as guiding philosophy for software engineering education. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 37*(4), 45-49.

Hagan, D., & Markham, S. (2000a). Teaching java with the BlueJ environment. Paper presented at the *Proceedings of Australasian Society for Computers in Learning in Tertiary Education Conference, ASCILITE 2000,* Retrieved November 13, 2006 from http://www.ascilite.org.au/conferences/coffs00/papers/dianne_hagan.pdf

Hagan, D., & Markham, S. (2000b). Does it help to have some programming experience before beginning a computing degree program? *ITiCSE '00: Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education,* Helsinki, Finland. 25-28.

Hall, G. E., & Hord, S. M. (2000). *Implementing change: Patterns, principles, and potholes*. Boston, Massachusetts, United States: Allyn & Bacon.

Hamilton, D. (1999). The pedagogic paradox (or why no didactics in England?). *Pedagogy, Culture and Society, 7*(1), 135-152.

Hammersley, M. (1990). *Reading ethnographic research. a critical guide*. London, United Kingdom: Longman.

Hammersley, M. (1992). *What's wrong with ethnography? Methodological explorations*. London, United Kingdom: Routledge.

Hansen, B. G., & Tams, A. (2006). Almen didaktiske temaer og problemstillinger i et kritisk kommunikativt perspektiv (common didactical themes and problems in a critical communicative perspective. in Danish). In B. G. Hansen, & A. Tams (Eds.), *Almen didaktik - relationer mellem undervisning og læring (general didactics - relation between teaching and learning. in Danish)* (pp. 249-271). Værløse, Denmark: Billesø & Baltzer.

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

Hemmendinger, D. (2007). The ACM and IEEE-CS guidelines for undergraduate CS education. *Communications of the ACM, 50*(5), 46-53.

Heron, J., & Reason, P. (1997). A participatory inquiry paradigm. *Qualitative Inquiry, 3*(3), 274–294.

Herrmann, N., Popyack, J. L., Char, B., Zoski, P., Cera, C. D., Lass, R. N., et al. (2003). Redesigning introductory computer programming using multi-level online modules for a mixed audience. *SIGCSE '03: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education,* Reno, Nevada, United States. 196-200.

Heshusius, L. (1994). Freeing ourselves from objectivity: Managing subjectivity or turning toward a participatory mode of consciousness? *Educational Researcher, 23*(3), 15-22.

Hiim, H., & Hippe, E. (2006). *Undervisningsplanlægning for faglærere (planning teaching for subject teachers. in Danish)* (2nd ed.). Copenhagen, Denmark: Gyldendal.

Hilburn, T. B. (1993). A top-down approach to teaching an introductory computer science course. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 25*(1), 58-62.

Hoc, J. M. (1990). *Psychology of programming*. London, United Kingdom: Academic Press.

Holden, E., & Weeden, E. (2003). The impact of prior experience in an information technology programming course sequence. *CITC4: Proceedings of the 4th Conference on Information Technology Curriculum,* Lafayette, Indiana, United States. 41-46.

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.

Holmboe, C. (2005). *Language, and the learning of data modelling*. Retrieved May 25, 2006, from http://folk.uio.no/christho/phd/thesis_Holmboe.pdf

Hostetler, T. R. (1983). Predicting student success in an introductory programming course. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 15*(3), 40-43.

Howe, K. R. (1988). Against the quantitative-qualitative incompatibility thesis or dogmas die hard. *Educational Researcher, 17*(8), 10-16.

Howe, E., Thornton, M., & Weide, B. W. (2004a). Components-first approaches to CS1/CS2: Principles and practice. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* Norfolk, Virginia, United States. 291-295.

Howe, E., Thornton, M., & Weide, B. W. (2004b). Components-first approaches to CS1/CS2: Principles and practice. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* Norfolk, Virginia, United States. 291-295.

Hu, C. (2004). Rethinking of teaching objects-first. *Education and Information Technologies, 9*(3), 209-218.

Hudak, M. A., & Anderson, D. E. (1990). Formal operations and learning style predict success in statistics and computer science courses. *Teaching of Psychology, 17*(4), 231-234.

Hull, C. L. (1943). *Principles of behavior. an introduction to behavior theory*. New York, New York, United States: Appleton-Century-Crofts.

Hull, C. L. (1952). *A behavior system. an introduction to behavior theory concerning the individual organism*. New Haven, Connecticut, United States: Yale University Press.

Hundhausen, C. D., Farley, S., & Brown, J. L. (2006). Can direct manipulation lower the barriers to programming and promote positive transfer to textual programming? an experimental study. *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, 2006. (VL/HCC 2006),* Brighton, United Kingdom. 157-164.

Hundhausen, C. D., & Brown, J. L. (2007). What you see is what you code: A “live” algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing, 18*(1), 22-47.

Hundhausen, C. D., Brown, J. L., Farley, S., & Skarpas, D. (2006). 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. 59-71.

IBM. (1964). *Manual for administrating and scoring the aptitude test for programmer personnel*. White Plains, New York, United States: IBM, Technical Publications Department.

ICALT'04. (2004). Proceedings IEEE international conference on advanced learning technologies. Joensuu, Finland.

ICER. (2006). Retrieved June 23, 2006, from http://www.cs.kent.ac.uk/events/conf/2006/icer/

IEEE. (n.d.). *Institute of electrical and electronics engineers, inc*. Retrieved April, 21, 2006, from http://www.ieee.org/

Imel, S. (1989). *Teaching adults: Is it different? ERIC digest no. 82*. Columbus, Ohio, United States: ERIC Clearinghouse on Adult, Career, and Vocational Education, Center on Education and Training for Employment, The Ohio State University.

Imel, S. (1995). *Teaching adults: Is it different? myths and realities*. Columbus, Ohio, United States: ERIC Clearinghouse on Adult, Career, and Vocational Education.

Inhelder, B., & Piaget, J. (1958). *The growth of logical thinking from childhood to adolescence. an essay on the construction of formal operational structures* [De la logique de l'enfant à la logique de l'adolescent. Essai sur la construction des structures operatoires formelles] (A. Parsons , S. Milgram Trans.). New York, New York, United States: Basic Books.

International handbook of universities(2005). . New York, New York, United States: Palgrave Macmillan.

IOOP. (2004). *Goal description for IOOP (in Danish).* Retrieved February, 14, 2006, from http://www.au.dk/da/evu/katalog/2004540.htm#_Toc63842557

ITiCSE'05. (2005). ITiCSE'05: Proceedings of the 10th annual conference on innovation and technology in computer science education. Monte de Caparica, Portugal.

Jalloul, G. (2000). Links: A framework for object-oriented software engineering. *Computer Science Education, 10*(1), 75-93.

James, W. (1950). *The principles of psychology. volume 1 + 2*. New York, New York, United States: Dover.

Järvelä, S. (1995). The cognitive apprenticeship model in a technologically rich learning environment: Interpreting the learning interaction. *Learning and Instruction, 5*(3), 237-259.

Jensen, K., Wirth, N., Mickel, A. B., & Miner, J. F. (1991). *Pascal user manual and report. ISO Pascal standard*. New York, New York, United States: Springer-Verlag.

Johnson, R. B., & Onwuegbuzie, A. J. (2004). Mixed methods research: A research paradigm whose time has come. *Educational Researcher, 33*(7), 14-26.

Johnson, W. L., Soloway, E., Cutler, B., & Draper, D. (1983). *Bug catalogue I* No. TR286). New Haven, Connecticut, United States: Computer Science, Yale University.

Jordan, B. (1989). Cosmopolitical obstetrics: Some insights from the training of traditional midwives. *Journal of Social Science Medicine, 28*(9), 925-944.

Joseph, D., & Nacu, D. C. (2003). Designing interesting learning environments when the medium isn't enough. *Convergence, 9*(2), 84-115.

Kaasbøll, J. J. (1998). Exploring didactic models for programming. *Proceedings of NIK'98 (Norwegian Computer Science Conference),* Kristiansand, Norway. 195-203.

Kaasbøll, J., Berge, O., Borge, R. E., Fjuk, A., Holmboe, C., & Samuelsen, T. (2004). Learning object-oriented programming. Paper presented at the *Proceedings of the 16th Workshop of the Psychology of Programming Interest Group,* Institute of Technology, Carlow, Ireland. 86-96. Retrieved May 7, 2007 from http://www.ppig.org/papers/16th-kaasboll.pdf

Kansanen, P., & Meri, M. (1999). The didactic relation in the teaching-studying-learning process. In B. Hudson, F. Buchberger, P. Kansanen & H. Seel (Eds.), *Didaktik/Fachdidaktik as science (-s) of the teaching profession?* (pp. 107-116). Umeå, Sweden: Thematic Network of Teacher Education in Europe. Retrieved November 22, 2007 from http://tntee.umu.se/publications/v2n1/pdf/2_1complete.pdf

Kearsley, G. (1998). Educational technology: A critique. *Educational Technology, 38*(2), 47-51.

Keesler, C., & Anderson, J. (1989). Learning flow of control: Recursive and iterative procedures. In E. Soloway, & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 229-260). Hillsdale, New Jersey, United States: L. Erlbaum Associates.

King, F. B. (2002). A virtual student not an ordinary Joe. *The Internet and Higher Education, 5*(2), 157–166.

Knowles, M. S. (1968). Andragogy, not pedagogy. *Adult Leadership, 16*(10), 350-352, 386.

Knowles, M. S. (1975). *Self-directed learning: A guide for learners and teachers*. New York, New York, United States: Association Press.

Knowles, M. S. (1980). *The modern practice of adult education: From pedagogy to androgogy.* (2nd ed.). New York, New York, United States: Cambridge Books.

Knudsen, J. L., & Madsen, O. L. (1988). Teaching object-oriented programming is more than teaching object-oriented programming languages. *ECOOP '88 European Conference on Object-Oriented Programming,* Oslo, Norway. 21-40.

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., Lofgren, M., Madsen. O.L., & Magnusson, B. (1993). *Object-oriented environments. the mjølner approach*. New York, New York, United States: Prentice Hall.

Ko, A. J., & Myers, B. A. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing, 16*(1-2), 41-84.

Koffman, E. B., Miller, P. L., & Wardle, C. E. (1984). Recommended curriculum for CS1, 1984. *Communications of the ACM, 27*(10), 998-1001.

Koffman, E. B., Stemple, D., & Wardle, C. E. (1985). Recommended curriculum for CS2, 1984: A report of the ACM curriculum task force for CS2. *Communications of the ACM, 28*(8), 815-818.

Kolb, D. A. (1976). *Learning style inventory: Technical manual*. Boston, Massachusetts, United States: McBer.

Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Journal of Computer Science Education, 13*(4), 249-268.

Kölling, M., & Rosenberg, J. (2001). Guidelines for teaching object orientation with java. *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, Canterbury, United Kingdom. 33-36.

Kölling, M. (2003). *The curse of hello world*. Oslo, Norway: Invited lecture at Workshop on Learning and Teaching Object-orientation – Scandinavian Perspectives.

Kölling, M., & Barnes, D. J. (2004). Enhancing apprentice-based learning of java. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, United States. 286-290.

Kölling, M., & Rosenberg, J. (2001). Guidelines for teaching object orientation with java. *ITiCSE '01: Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, Canterbury, United Kingdom. 33-36.

Kölling, M., & Rosenberg, J. (2002). *BlueJ - the hitch-hikers guide to object orientation* (Technical Report. Odense, Denmark: The Maersk Mc-Kinney Moller Institute for Production Technology, University of Southern Denmark.

Konvalina, J., Wileman, S. A., & Stephens, L. J. (1983). Math proficiency: A key to success for computer science students. *Communications of the ACM, 26*(5), 377-382.

Korhonen, A., & Malmi, L. (2004). *Kolin kolistelut - koli calling 2004: Proceedings of the fourth Finnish/Baltic sea conference on computer science education* (Report TKO-A42/04 ed.). Helsinki, Finland: Helsinki University of Technology, Department of Computer Science and Technology.

Krone, J., & Bressoud, T. (2007). *Enriching computer science faculty careers with a workshop on CS1*. Retrieved May, 14, 2007, from http://www.denison.edu/mathcs/cs1/about.html

Kühn, D., Langer, J., Kohlberg, L., & Haan, N. S. (1977). The development of formal operations in logical and moral judgement. *Genetic psychology monographs, 95,* 97-188.

Kuittinen, M. (2001). *Kolin kolistelut - koli calling 2001: Proceedings of the first Finnish/Baltic sea conference on computer science education. report A-2002-1*University of Joensuu, Department of Computer Science, Finland. Retrieved May 7, 2007 from http://cs.joensuu.fi/kolistelut/archive/2001/koli_proc_2001.pdf

Kumar, A. N. (2003). The effect of closed labs in computer science I: An assessment. *Journal of Computing in Small Colleges, 18*(5), 40-48.

Kurhila, J. (2003). *Kolin kolistelut - koli calling 2003: Proceedings of the third Finnish/Baltic sea conference on computer science education. report B-2003-3*. Helsinki, Finland: Helsinki University Printing House. Retrieved May 7, 2007 from http://cs.joensuu.fi/kolistelut/archive/2003/koli_proc_2003.pdf

Kurtz, B. L. (1980). 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. 110-117.

Kvale, S. (2005). *Interview. en introduktion til det kvalitative forskningsinterview (interviews - an introduction to qualitative research interviewing. in danish)*. Copenhagen, Denmark: Hans Reitzel.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H. (2005). 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. 14-18.

Larkin, J., McDermott, J., Simon, D. P., & Simon, H. A. (1980). Expert and novice performance in solving physics problems. *Science, 208,* 140-156.

Larkin, J. H., McDermott, J., Simon, D. P., & Simon, H. A. (1980). Models of competence in solving physics problems. *Cognitive Science, 4*(4), 317-345.

Larman, C. (2005). *Applying UML and patterns - an introduction to object-oriented analysis and design and iterative development* (3rd ed.). Westford, Massachusetts, United States: Pearson Education Inc.

Larsen, S. (1998). *Den ultimative formel. for effektive læreprocesser (the ultimate formula for effective learning processes. in Danish)*. Hellerup, Denmark: Steen Larsens forlag.

Lave, J., & Wenger, E. (1991). *Situated learning. legitimate peripheral participation*. Cambridge, United Kingdom: Cambridge University Press.

Lawhead, P. B., Duncan, M. E., Bland, C. G., Goldweber, M., Schep, M., Barnes, D. J., et al. (2002). 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. 191-201.

Lea, D. (1994). Christopher Alexander: An introduction for object-oriented designers. *ACM SIGSOFT Software Engineering Notes, 19*(1), 39-46.

LeCompte, M. D., & Goetz, J. P. (1982). Problems of reliability and validity in ethnographic research. *Review of Educational Research, 52*(1), 31-60.

Leeper, R. R., & Silver, J. L. (1982). Predicting success in a first programming course. *SIGCSE '82: Proceedings of the Thirteenth SIGCSE Technical Symposium on Computer Science Education*, Indianapolis, Indiana, United States. 147-150.

Leska, C., & Rabung, J. (2005). Refactoring the CS1 course. *Journal of Computing in Small Colleges, 20*(3), 6-18.

Levin, J., & Waugh, M. (1998). Teaching teleapprenticeships: Electronic network-based educational frameworks for improving teacher education. *Interactive Learning Environments, 6*(1), 39-58.

Levy, S. P. (1995). Computer language usage in CS1: Survey results. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 27*(3), 21-26.

Lewis, J. (2000). Myths about object-orientation and its pedagogy. *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, Austin, Texas, United States. 245-249.

Lidtke, D. K., & Zhou, H. H. (1999). A new approach to an introduction to computer science. *Proceedings of the 29th Annual Frontiers in Education Conference, 1999. (FIE '99),* San Juan, Puerto Rico. 12a4-23.

Lin, J. M., Lin, K., & Wu, C. (1999). A content analysis of programming examples in high school computer textbooks in taiwan. *Journal of Computers in Mathematics and Science Teaching, 18*(3), 225-244.

Lincoln, Y. S., & Guba, E. G. (1989). *Fourth generation evaluation*. Beverly Hills, California, United States: Sage Publications.

Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic inquiry*. Beverly Hills, California, United States: Sage Publications.

Lister, R. (2005a). Grand challenges. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 37*(2), 14-15.

Lister, R. (2005b). Mixed methods: Positivists are from Mars, constructivists are from Venus. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 37*(4), 18-19.

Lister, R. (2006). Call me Ishmael: Charles Dickens meets Moby Book. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 38*(2), 11-13.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., et al. (2004). A multi-national study of reading and tracing skills in novice programmers. *ITiCSE-WGR '04: Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education,* Leeds, United Kingdom. 119-150.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., et al. (2006). Research perspectives on the objects-early debate. *ITiCSE-WGR '06: Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education,* Bologna, Italy. 146-165.

Lorentsen, A. (2004). *Voksne som målgruppe for universiteternes efter- og videreuddannelse (adults as target group for the universities further- and continuing education. in Danish).* Retrieved January 26, 2005, from http://www.learning.aau.dk/download/Forskningsrapporter/Rapport_5_8790934954.pdf

Lucas, W. (2003). Making way for java in an information technology masters program. In T. McGill (Ed.), *Current issues in IT education* (pp. 35-47). Hershey, Pennsylvania, United States: IRM Press.

Ludvigsen, S., & Mørch, A. (2003). Categorization in knowledge building: Task-specific argumentation in a co-located CSCL environment. *Proceedings of the International Conference on Computer Support for Collaborative Learning,* Bergen, Norway. 67-76.

Luker, P. A. (1989). Never mind the language, what about the paradigm? *SIGCSE '89: Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education,* Louisville, Kentucky, United States. 252-256.

Luker, P. A. (1994). There's more to OOP than syntax! *SIGCSE '94: Proceedings of the Twenty-Fifth SIGCSE Symposium on Computer Science Education,* Phoenix, Arizona, United States. 56-60.

Machanick, P. (2007). A social construction approach to computer science education. *Computer Science Education, 17*(1), 1.

Madsen, K. S. (2006, July 31). Politikere: Faldende it-optag skal vendes (politicians: The falling enrolment to IT studies has to be turned. in Danish). [Electronic version]. *Computer World Denmark,* Retrieved August 31, 2006 from http://www.computerworld.dk/art/34957

Madsen, O. L., Torgersen, M., Røn, H., & Thorup, K. K. (1998). Teaching object-oriented programming to C programmers. *Proceedings Educators' Symposium, Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98),* Vancouver, Canada.

Madsen, O. L. (1995). Open issues in object-oriented programming - a Scandinavian perspective. *Software Practice end Experience, 25*(S4), 3-43.

Madsen, O. L., Nygaard, K., & Møller-Pedersen, B. (1993). *Object-oriented programming in the BETA programming language*. Wokingham, United Kingdom: Addison-Wesley.

Margolis, J., & Fisher, A. (2003). *Unlocking the clubhouse: Women in computing*. Cambridge, Massachusetts, United States: MIT Press.

Marshall, C., & Rossman, G. B. (1989). *Designing qualitative research*. London, United Kingdom: Sage Publications.

Marton, F., & Booth, S. (1997). *Learning and awareness*. Mahwah, New Jersey, United States: L. Erlbaum Associates.

Matzko, S., & Davis, T. A. (2006). Teaching CS1 with graphics and C. *ITICSE '06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, Bologna, Italy. 168-172.

Maxim, P. S. (1999). *Quantitative research methods in the social sciences*. New York, New York, United States: Oxford University Press.

Maxwell, S. E., & Delaney, H. D. (1999). *Designing experiments and analyzing data*. Mahwah, New Jersey, United States: Lawrence Erlbaum Associates.

Mayer, R. E., Dyck, J. L., & Vilberg, W. (1986). Learning to program and learning to think: What's the connection? *Communications of the ACM, 29*(7), 605-610.

Mazen, A. M., Graf, L. A., Kellogg, C. E., & Hemmasi, M. (1987). Statistical power in contemporary management research. *The Academy of Management Journal, 30*(2), 369-380.

Mazlack, L. J. (1980). Identifying potential to acquire programming skill. *Communications of the ACM, 23*(1), 14-17.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 33*(4), 125-180.

McGettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Lovegrove, G., & Mander, K. (2005). Grand challenges in computing: Education--A summary. *The Computer Journal, 48*(1), 42-48.

McKinnon, J. W., & Renner, J. W. (1971). Are colleges concerned with intellectual development? *American Journal of Physics, 39*(9), 1047-1052.

Mead, G. H. (1974). *Mind, self, and society. from the standpoint of a social behaviorist*. Chicago, Illinois, United States: The University of Chicago Press.

Merriam, S. B. (2001). Andragogy and self-directed learning: Pillars of adult learning theory. *New directions for Adult and Continuing Education, 89*, 3-13.

Merriam-Webster. (2007). *Merriam-webster's online dictionary*. Retrieved October 10, 2007, from http://mw1.merriam-webster.com/dictionary/didactics

Merriam-webster's medical desk dictionary (2002). Springfield, Massachusetts, United States: Merriam-Webster.

Meyer, B. (1992). Applying `design by contract'. *Computer, 25*(10), 40-51.

Meyer, B. (1997). *Object-oriented software construction. 2nd ed*. Upper Saddle River, New Jersey, United States: Prentice Hall.

Mezirow, J. (1981). A critical theory of adult learning and education. *Adult Education Quarterly, 32*(1), 3-24.

Mezirow, J. (1991). *Transformative dimensions of adult learning*. San Francisco, California, United States: Jossey-Bass.

Michener, E. R. (1978). Understanding understanding mathematics. *Cognitive Science, 2*(4), 283-327.

Milne, I., & Rowe, G. (2002). Difficulties in learning and teaching programming - views of students and tutors. *Education and Information Technologies, 7*(1), 55-66.

Ministeriet for Videnskab - Teknologi og Udvikling. (2003). *En værdiskabende sektor? - fokus på de danske it-erhverv (A value-creating sector? - focus on the danish IT-industry. in Danish)*. Copenhagen, Denmark: Ministeriet for Videnskab, Teknologi og Udvikling.

Ministeriet for Videnskab - Teknologi og Udvikling. (2006). *Mere attraktive uddannelser skal skabe flere IT-kandidater (more attractive study programs shall produce more IT-candidates. in Danish)*. Retrieved August 31, 2006, from http://videnskabsministeriet.dk/site/forside/nyheder/pressemeddelelser/2006/mere-attraktive-uddannelser-skal-skabe-flere-it-kandidater

Mitchell, W. (2000). A paradigm shift to OOP has occurred … implementation to follow. *CCSC '00: Proceedings of the Fourteenth Annual Consortium on Small Colleges Southeastern Conference*, Roanoke College, Salem, Virginia, United States. 94-105.

Moritz, S. H., Wei, F., Parvez, S. M., & Blank, G. D. (2005). From objects-first to design-first with multimedia and intelligent tutoring. *ITiCSE '05: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* Caparica, Portugal. 99-103.

Moskal, B., Lurie, D., & Cooper, S. (2004). Evaluating the effectiveness of a new instructional approach. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* Norfolk, Virginia, United States. 75-79.

Mosley, P. H. (2002). *The cognitive complexities confronting developers using object technology*. Unpublished PhD, School of Computer Science & Information Systems, Pace University.

Muller, O. (2005). Pattern oriented instruction and the enhancement of analogical reasoning. *ICER '05: Proceedings of the 2005 International Workshop on Computing Education Research,* Seattle, Washington, United States. 57-67.

Muller, O., Ginat, D., & Haberman, B. (2007). Pattern-oriented instruction and its influence on problem decomposition and solution construction. *ITiCSE '07: Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* Dundee, Scotland. 151-155.

Muller, O., Haberman, B., & Averbuch, H. (2004). (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. 102-106.

Myers, I. B. (1961). *Manual for the Myers-Briggs type indicator*. Palo Alto, California, United States: Consulting Psychologists Press.

Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C., et al. (2003). Improving the CS1 experience with pair programming. *SIGCSE '03: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education,* Reno, Nevada, United States. 359-362.

National centre for education statistics. (2007). *COOL: College opportunities online locator*. Retrieved May, 13, 2007, from http://nces.ed.gov/ipeds/cool/

National centre for educational statistics. (2007). Retrieved May, 13, 2007, from http://nces.ed.gov/index.asp

Nevison, C., & Wells, B. (2003). Teaching objects early and design patterns in java using case studies. *ITiCSE '03: Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education,* Thessaloniki, Greece. 94-98.

Nevison, C., & Wells, B. (2004). Using a maze case study to teach: Object-oriented programming and design patterns. *ACE '04: Proceedings of the Sixth Conference on Australasian Computing Education,* Dunedin, New Zealand. 207-215.

Newsted, P. R. (1975). Grade and ability predictions in an introductory programming course. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 7*(2), 87-91.

Nguyen, D. Z., & Wong, S. B. (2001). OOP in introductory CS: Better students through abstraction. Paper presented at the *Proceedings of the Fifth Workshop on and Tools for Assimilating Object-Oriented Concepts, OOPSLA '01*, Tampa, Florida, United States. Retrieved November 22, 2007 from http://cs.umu.se/~jubo/Meetings/OOPSLA01/Contributions/swong.pdf

Nielsen, H., & Thomsen, P. V. (1983). *Hverdagsforestillinger i fysik (everyday notations in physics. in Danish)* No. GymnasieFysikrapport nr. 1). Aarhus, Denmark: Department of physics and astronomy, University of Aarhus.

Nielsen, J. M. C. (1990). *Feminist research methods: Exemplary readings in the social sciences*. Boulder, Colorado, United States: Westview Press.

Nielsen, K., & Kvale, S. (1997). Current issues of apprenticeship. *Nordisk Pedagogik, 17*, 130-139.

Nowaczyk, R. H. (1983). Cognitive skills needed in computer programming. *Proceedings of the 29th Annual Meeting of the South-Eastern Psychological Association,* Atlanta, Georgia, United States. 1-14.

Nygaard, K. (2002). COOL (comprehensive object-oriented learning). *ITiCSE '02: Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education,* Aarhus, Denmark. 218-218.

Oliver, S. R., & Dalbey, J. (1994). A software development process laboratory for CS1 and CS2. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 26*(1), 169-173.

OMG. (2007). *Unified modeling language: Superstructure* No. version 2.1.1)Object Management Group. Retrieved December 4, 2007 from http://www.omg.org/docs/formal/07-02-05.pdf

Onwuegbuzie, A. J. N., & Leech, N. L. N. (2005). Taking the “Q” out of research: Teaching research methodology courses without the divide between quantitative and qualitative paradigms. *Quality and Quantity, 39*(3), 267-295.

Or-Bach, R., & Lavy, I. (2004). Cognitive activities of abstraction in object orientation: An empirical study. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 36*(2), 82-86.

Orlikowski, W. J., & Baroudi, J. J. (1991). Studying information technology in organizations: Research approaches and assumptions. *Information Systems Research, 2*(1), 1-28.

Ormrod, E. J. (2004). *Human learning*. Upper Saddle River, New Jersey, United States: Merrill.

Östling, M. (2004). *A sequence diagram editor for BlueJ*. Unpublished Master of Science, University of Umeå, Sweden.

Paas, F. G. W. C., & Van Merrienboer, J. J. G. (1994). Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive-load approach. *Journal of educational psychology, 86*(1), 122-133.

Palinscar, A. S., & Brown, A. L. (1984). Reciprocal teaching of comprehension-fostering and comprehension-monitoring activities. *Cognition and Instruction, 1*(2), 117-175.

Palumbo, D. B. (1990). Programming language/problem-solving research: A review of relevant issues. *Review of Educational Research, 60*(1), 65-89.

Parlante, N. (2007). *Nifty assignments*. Retrieved August 28, 2007, from http://nifty.stanford.edu/

Pattis, R. E. (1993). 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. 122-126.

Patton, M. Q. (1990). *Qualitative evaluation and research methods*. Newbury Park, California, United States: Sage Publications.

Pea, R. D. (1986). Language independent conceptual 'bugs' in novice programming. *Journal of Educational Computing Research, 2*(1), 25-36.

Pears, A., Seidman, S., Eney, C., Kinnunen, P., & Malmi, L. (2005). Constructing a core literature for computing education research. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 37*(4), 152-161.

Pears, A., Seidman, S., Malmi, L., Mannili, L., Adams, E. S., Bennedsen, J., et al. (2007). A survey of literature on the teaching of introductory programming. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), To appear*.

Pecinovský, R., Pavlíčková, J., & Pavlíček, L. (2006). Let's modify the objects-first approach into design-patterns-first. *ITICSE '06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* Bologna, Italy. 188-192.

The pedagogical patterns project. (n.d.). Retrieved May, 31, 2006, from http://www.pedagogicalpatterns.org/

Pedroni, M., & Meyer, B. (2006). The inverted curriculum in practice. *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education,* Houston, Texas, United States. 481-485.

Perkins, D. N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers,* Washington, District of Columbia, United States. 213-229.

Perkins, D. (1999). The many faces of constructivism. *Educational Leadership, 57*(3), 6-11.

Peterßen, W. H. (2001). *Lehrbuch allgemeine didaktik*. München, Germany: Oldenbourg.

Phillips, D. (1995). The good, the bad, and the ugly: The many faces of constructivism. *Educational Researcher, 24*(7), 5-12.

Pillay, N., & Jugoo, V. R. (2005). An investigation into student characteristics affecting novice programming performance. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 37*(4), 107-110.

Pintrich, P. R., & Garcia, T. (1991). Student goal orientation and self-regulation in the college classroom. *Advances in motivation and achievement, 7,* 371-403.

Poplin, M., Drew, D., & Gable, R. (1984). *CALIP: Computer aptitude, computer literacy and interest profile*. Austin, Texas, United States: Pro-Ed.

Pritchard, M. E., & Wilson, G. S. (2003). Using emotional and social factors to predict student success. *Journal of College Student Development, 44*(1), 18-28.

Proulx, V. K. (2000). Programming patterns and design patterns in the introductory computer science course. *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education,* Austin, Texas, United States. 80-84.

Proulx, V. K., Raab, J., & Rasala, R. (2002). Objects from the beginning - with GUIs. *ITiCSE '02: Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education,* Aarhus, Denmark. 65-69.

Psychology of programming interest group. (n.d.). Retrieved June 23, 2006, from http://www.ppig.org/

Raadt, M. D., Watson, R., & Toleman, M. (2002). Language trends in introductory programming courses. *Proceedings of the Informing Science + IT Education Conference,* Cork, Ireland. 329-337.

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

Ragonis, N., & Ben-Ari, M. (2005b). 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, United States. 226-230.

Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004). 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. 171-175.

Ramalingam, V., & Wiedenbeck, S. (1997). An empirical study of novice program comprehension in the imperative and object-oriented styles. *ESP '97: Papers Presented at the Seventh Workshop on Empirical Studies of Programmers,* Alexandria, Virginia, United States. 124-139.

Ramalingam, V., & Wiedenbeck, S. (1998). Development and validation of scores on a computer programming self-efficacy scale and group analyses of novice programmer self-efficacy. *Journal of Educational Computing Research, 19*(4), 367-381.

Ramsden, P. (1992). *Learning to teach in higher education*. London, United Kingdom: Routledge.

Randolph, J., Bednarik, R., Silander, P., Gonzalez, J., Myller, N., & Sutinen, E. (2005). A critical analysis of the research methodologies reported in the full papers of the proceedings of ICALT 2004. *Proceedings of the 5th IEEE International Conference on Advanced Learning Technologies, ICALT 2005,* Kaohsiung, Taiwan. 10-14.

Randolph, J. J., Bednarik, R., & Myller, N. (2005). A methodological review of the articles published in the proceedings of Koli calling 2001-2004. *Proceedings of Koli Calling 2005 Fifth Koli Calling Conference on Computer Science Education,* Koli national park, Finland. 103-108.

Rauchas, S., Rosman, B., Konidaris, G., & Sanders, I. (2006). 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, United States. 398-402.

Reek, M. M. (1995). 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. 6-9.

Reges, S. (2000). Conservatively radical java in CS1. *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education,* Austin, Texas, United States. 85-89.

Rentsch, T. (1982). Object oriented programming. *SIGPLAN Notices, 17*(9), 51-57.

Retalis, S., Georgiakakis, P., & Dimitriadis, Y. (2006). Eliciting design patterns for e-learning systems. *Computer Science Education, 16*(2), 105-118.

Roberge, J. J., & Flexer, B. K. (1982). Formal operational reasoning test. *Journal of General Psychology, 106*, 61-67.

Roberts, E. S., Kassianidou, M., & Irani, L. (2002). Encouraging women in computer science. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 34*(2), 84-88.

Robillard, P. N. (2005). Opportunistic problem solving in software engineering. *IEEE Software, 22*(6), 60-67.

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, M. (1965). *Society and the adolescent self-image*. Princeton, New Jersey, United States: Princeton University Press.

Rosson, M. B., Ballin, J., & Rode, J. (2005). Who, what, and how: A survey of informal and professional web developers. *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing,* Dallas, Texas, United States. 199-206.

Rosson, M. B., Ballin, J., & Nash, H. (2004). Everyday programming: Challenges and opportunities for informal web development. *Proceedings of the IEEE Symposia on Visual Languages and Human-Centric Computing,* Rome, Italy. 123-130.

Roumani, H. (2002). Design guidelines for the lab component of objects-first CS1. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 34*(1), 222-226.

Roumani, H. (2006). 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, United States. 491-494.

Rountree, N., Rountree, J., & Robins, A. (2002). Predictors of success and failure in a CS1 course. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 34*(4), 121-124.

Rountree, N., Rountree, J., Robins, A., & Hannah, R. (2004). Interacting factors that predict success and failure in a CS1 course. *ITiCSE-WGR '04: Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education,* Leeds, United Kingdom. 101-104.

Rowland, S. (1999). The role of theory in a pedagogical model for lectures in higher education. *Studies in Higher Education, 24*(3), 303-314.

Rowntree, D. (1988). *Assessing students. how shall we know them?*. London, United Kingdom: Kogan Page.

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. University of Illinois at Urbana-Champaign). *Dissertation Abstracts International Section B: The Sciences and Engineering, 58* (10-B), 5655.

SAT. *Encyclopædia Britannica*. Retrieved August, 31, 2006, from http://www.britannica.com/eb/article-9377802

Scaffidi, C., Shaw, M., & Myers, B. (2005). Estimating the numbers of end users and end user programmers. *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, United States. 207-214.

Scardamalia, M., Bereiter, C., & Steinbach, R. (1984). Teachability of reflective processes in written composition. *Cognitive Science, 8*(2), 173-190.

Schmolitzky, A. (2004). "Objects first, interfaces next" or interfaces before inheritance. *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications,* Vancouver, British Columbia, Canada. 64-67.

Schmolitzky, A. (2006). Teaching inheritance concepts with java. *PPPJ '06: Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java,* Mannheim, Germany. 203-207.

Schoenfeld, A. H. (1985). *Mathematical problem solving*. Orlando, Florida, United States: Academic Press.

Schroeder, L. D., Sjoquist, D. L., & Stephan, P. E. (1986). *Understanding regression analysis. an introductory guide*. Newbury Park, California, United States: Sage Publications.

Schuler, D., & Namioka, A. (1993). *Participatory design. principles and practices*. Hillsdale, New Jersey, United States: Lawrence Erlbaum Associates.

Schulte, C., & Bennedsen, J. (2006). What do teachers teach in introductory programming? *The Second International Computing Education Research Workshop,* University of Kent, Canterbury, United Kingdom.

Schulte, C., Magenheim, J., Niere, J., & Schäfer, W. (2003). Thinking in objects and their collaboration: Introducing object-oriented technology. *Computer Science Education, 13*(4), 269.

Schwaber, K. (2004). *Agile project management with scrum*. Redmond, Washington, United States: Microsoft Press.

Seale, C. (1999). *The quality of qualitative research*. London, United Kingdom: Sage Publications.

Seffah, A., & Grogono, P. (2002). Learner-centered software engineering education: From resources to skills and pedagogical patterns. *Proceedings of the*

15th Conference on Software Engineering Education and Training (CSEE&T 2002). Covington, Northern Kentucky, United States. 14-21.

Selltiz, C., Wrightsman, L. S., & Cook, S. W. (1976). *Research methods in social relations*. New York, New York, United States: Holt, Rinehart and Winston.

Shackelford, R. (2005). *Computing curricula 2005. the overview report*. Retrieved April, 21, 2006, from http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf

Shaffer, D. W., & Resnick, M. (1999). “Thick” authenticity: New media and authentic learning. *Journal of Interactive Learning Research, 10*(2), 195-215.

Sharp, H., Manns, M. L., & Eckstein, J. (2003). Evolving pedagogical patterns: The work of the pedagogical patterns project. *Computer Science Education, 13*(4), 315-330.

Sharp, H., Manns, M. L., & Eckstein, J. (2000). The pedagogical patterns project (poster session). *OOPSLA '00: Addendum to the 2000 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum),* Minneapolis, Minnesota, United States. 139-140.

Shayer, M., & Adey, P. (1981). *Towards a science of science teaching. cognitive development and curriculum demand*. Oxford, United Kingdom: Heinemann Educational.

Sheil, B. A. (1981). The psychological study of programming. *ACM Computing Surveys, 13*(1), 101-120.

Sicilia, M. (2006). Strategies for teaching object-oriented concepts with java. *Journal of Computer Science Education, 16*(1), 1-18.

SIGCSE mail-list. (n.d.). *Special interest group on computer science education mailing list*. Retrieved June 14, 2006 from http://sigcse.org/join/list.shtml

SIGCSE'05. (2005). SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on computer science education. St. Louis, Missouri, United States.

SIGCSE-members. (2005). *Archives of sigcse-members@ACM.ORG*. Retrieved March 22, 2006, from http://listserv.acm.org/archives/sigcse-members.html

Silverman, D. (2001). *Interpreting qualitative data. methods for analysing talk, text and interaction*. London, United Kingdom: Sage Publications.

Simon. (2007a). A classification of recent Australasian computing education publications. *Computer Science Education, 17*(3), 155.

Simon. (2007b). Koli comes of age. *Proceedings of the 7th Koli Calling Conference,* Koli, Finland.

Simon, Fincher, S., Robins, A., Baker, B., Box, I., Cutts, Q., et al. (2006). Predictors of success in a first programming course. *ACE '06: Proceedings of the 8th Australian Conference on Computing Education*, Hobart, Australia. 189-196.

Simpson, D. (1973). Psychological testing in computing staff selection: A bibliography. *ACM SIGCPR Computer Personnel, 4*(1-2), 2-5.

Skinner, B. F. (1965). *Science and human behavior*. New York, New York, United States: Macmillan.

Sleeper, R. W. (2001). *The necessity of pragmatism. John Dewey's conception of philosophy*. Urbana, Illinois, United States: University of Illinois.

Smith, J. K. (1984). The problem of criteria for judging interpretive inquiry. *Educational Evaluation and Policy Analysis, 6*(4), 379-391.

Soldan, D. (2004). *Computer engineering 2004. curriculum guidelines for undergraduate degree programs in computer engineering*. Retrieved April, 21, 2006, from http://www.acm.org/education/CE-Final%20Report.pdf

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *Transactions on Software Engineering, 10*(5), 595-609.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM, 29*(9), 850-858.

Soloway, E., & Spohrer, J. C. (1989). *Studying the novice programmer*. Hillsdale, New Jersey, United States: Lawrence Erlbaum.

Soper, D. S. (2007). *A-priori sample size calculator (multiple regression) free statistics calculators (online software)*. Retrieved April 20, 2007 from http://www.danielsoper.com/statcalc

Sorge, D. H., & Wark, L. K. (1984). Factors for success as a computer science major. *AEDS Journal, 17*, 36-45.

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, District of Columbia, United States: Ablex Publishing Corporation.

Spohrer, J., Pope, E., Lipman, M., Sack, W., Freiman, S., Littman, D., et al. (1985). *Bug catalogue 2,3,4* No. TR386). New Haven, Connecticut, United States: Computer Science, Yale University.

Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM, 29*(7), 624-632.

Sprague, P., & Schahczenski, C. (2002). Abstraction the key to CS1. *Journal of Computing in Small Colleges, 17*(3), 211-218.

Stein, L. A. (1998). What we swept under the rug: Radically rethinking CS1. *Computer Science Education, 8*(2), 118-129.

Stein, M. V. (2002). Mathematical preparation as a basis for success in CS-II. *Journal of Computing in Small Colleges, 17*(4), 28-38.

Stemler, S. (2001). An overview of content analysis. *Practical Assessment, Research & Evaluation, 7*(17) Retrieved August 10, 2006 from http://PAREonline.net/getvn.asp?v=7&n=17

Sutinen, E. (2002). *Kolin kolistelut - Koli calling 2002: Proceedings of the second Finnish/Baltic sea conference on computer science education. report A-2002-7* University of Joensuu, Department of Computer Science, Finland. Retrieved May 10, 2007 from http://cs.joensuu.fi/kolistelut/archive/2002/koli_proc_2002.pdf

Sweller, J., van Merrienboer, J. J. G., & Paas, F. G. W. C. (1998). Cognitive architecture and instructional design. *Educational Psychology Review, 10*(3), 251-296.

Sweller, J., Mawer, R. F., & Ward, M. R. (1983). Development of expertise in mathematical problem solving. *Journal of Experimental Psychology: General, 112*(4), 639-661.

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 & Counselling Special Issue: Counselling and health, 15*, 82-91.

Taylor, H. G., & Mounfield, L. C. (1989). The effect of high school computer science, gender, and work on success in college computer science. *SIGCSE '89: Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education,* Louisville, Kentucky, United States. 195-198.

Thomas, L., Ratcliffe, M., Woodbury, J., & Jarman, E. (2002). Learning styles and performance in the introductory programming sequence. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 34*(1), 33-37.

Thomas, R. C., Karahasanovic, A., & Kennedy, G. E. (2005). An investigation into keystroke latency metrics as an indicator of programming performance. *ACE '05: Proceedings of the 7th Australasian Conference on Computing Education,* Newcastle, New South Wales, Australia. 127-134.

Thorndike, E. L. (1931). *Human learning*. New York, New York, United States: The Century Co.

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 Students Development, 39,* 599-610.

Tough, A. M. (1967). *Learning without a teacher*. Toronto, Ontario, Canada: Ontario Institute for Studies in Education.

Tough, A. M. (1972). *The Adult's learning projects: A fresh approach to theory and practice in adult learning*. Toronto, Ontario, Canada: Ontario Institute for Studies in Education.

Towell, J., & Towell, E. (2003). Reality abstraction and OO pedagogy: Results from 5 weeks in virtual reality. *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, California, United States. 162-166.

Tucker, A. B. (1996). Strategic directions in computer science education. *ACM Computing Surveys*, *28*(4), 836-845.

Turk, M. (1997). Introducing object orientation to experienced procedural programmers. *ACSE '97: Proceedings of the 2nd Australasian Conference on Computer Science Education*, The University of Melbourne, Australia. 135-140.

Tyler, R. W. (1949). *Basic principles of curriculum and instruction*. Chicago, Illinois, United States: University of Chicago Press.

U.S. Department of Education. (2003). *Think college early: Average college costs*. Retrieved April 19, 2007, from http://www.ed.gov/students/prep/college/thinkcollege/early/students/edlite-college-costs.html

UNESCO. (2007). *United Nations educational, scientific and cultural organization*. Retrieved May, 14, 2007, from http://www.unesco.org

UNESCO. (n.d.). *UNESCO institute of statistics*. Retrieved January 20, 2007, from http://stats.uis.unesco.org/TableViewer/dimView.aspx?ReportId=251

Valentine, D. W. (2004). CS educational research: A meta-analysis of SIGCSE technical symposium proceedings. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, United States. 255-259.

Ventura, P. R. (2003). *On the origins of programmers: Identifying predictors of success for an objects first CS1*. Unpublished Ph.D. thesis, The State University of New York at Buffalo, Buffalo, New York United States.

Ventura, P. (2005). Identifying predictors of success for an objects-first CS1. *Computer Science Education*, *15*(3), 223-243.

Ventura, P., & Ramamurthy, B. (2004). Wanted: CS1 students. No experience required. *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, United States. 240-244.

Ventura, P., Egert, C., & Decker, A. (2004). Ancestor worship in CS1: On the primacy of arrays. *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications,* Vancouver, British Columbia, Canada. 68-72.

von Mayrhauser, A., & Vans, A. M. (1994). *Program understanding - A survey* No. Technical Report CS-94-120). Colorado, United States: Department of Computer Science, Colorado State University.

Vygotsky, L. S. (1978). *Mind in society: The development of higher psychological processes*. Cambridge, Massachusetts, United States: Harvard University Press.

Ward, M., & Sweller, J. (1990). Structuring effective worked examples. *Cognition & Instruction, 7,* 1-39.

Watson, J. B. (1913). Psychology as the behaviorist views it. *Psychological Review, 20*, 158-177.

Watson, J. B. (1925). *Behaviorism*. New York, New York, United States: W. W. Norton & company inc.

Weber-Wulff, D. (2000). Combating the code warrior: A different sort of programming instruction. *ITiCSE '00: Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, Helsinki, Finland. 85-88.

Weinberg, G. M. (1971). *The psychology of computer programming*. New York, New York, United States: Van Nostrand Reinhold Co.

Weinberg, G. M. (1998). *The psychology of computer programming*. New York, New York, United States: Dorset House Pub.

Weiss, I. R., Banilower, E. R., McMahon, K. C., & Smith, P. S. (2001). *Report of the 2000 national survey of science and mathematics education* from http://2000survey.horizon-research.com/reports/status/complete.pdf

Weitl, F., Süß, C., Kammerl, R., & Freitag, B. (2002). Presenting complex e-learning content on the web: A didactical reference model. *Proceedings of e-Learn 2002 World Conference on E-Learning in Corporate, Government, Healthcare, & Higher Education,*

Wenger, E. (1998). Communities of practice: Learning as A social practice. *The Systems Thinker, 9*(5)

Werth, L. H. (1986). Predicting student performance in a beginning computer science class. *SIGCSE '86: Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education*, Cincinnati, Ohio, United States. 138-143.

Whyte, W. F. (1989). Participatory action research: Through practice to science in social research. *American Behavioral Scientist, 32*(5), 513-551.

Wick, M. R. (2001). Kaleidoscope: Using design patterns in CS1. *SIGCSE '01: Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education,* Charlotte, North Carolina, United States. 258-262.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers, 11*(3), 255-282.

Wiedenbeck, S. (2005). Factors affecting the success of non-majors in learning to program. *ICER '05: Proceedings of the 2005 International Workshop on Computing Education Research,* Seattle, Washington, United States. 13-24.

Wiedenbeck, S., Fix, V., & Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: An empirical study. *International Journal of Man-Machine Studies, 39*(5), 793-812.

Wikipedia. (2007). *Didactics*. Retrieved May 29, 2007, from http://en.wikipedia.org/wiki/Didactics

Wilson, B. C. (2002). A study of factors promoting success in computer science including gender differences. *Computer Science Education, 12*(1), 141-164.

Wilson, B. C., & Shrock, S. (2001). 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. 184-188.

Winslow, L. E. (1996). Programming pedagogy — a psychological overview. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 28*(3), 17-22.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM, 14*(4), 221-227.

Witkin, H. A., Oltman, P. K., Raskin, E., & Karp, S. A. (1971). *A manual for the embedded figures test.* Palo Alto, California, United States: Consulting Psychologies Press.

Wlodkowski, R. J. (1999). *Enhancing adult motivation to learn: A comprehensive guide for teaching all adults*. San Francisco, California, United States: Jossey-Bass Publishers.

Wolz, U., & Koffman, E. (2000). Interactivity in CS1 & CS2: Bringing back the fun stuff with java. *CCSC '00: Proceedings of the Fifth Annual Consortium for Computing Sciences in Colleges CCSC: Northeastern Conference,* Ramapo College of New Jersey, Mahwah, New Jersey, United States. 1-3.

Woodworth, P., & Dann, W. (1999). Integrating console and event-driven models in CS1. *SIGCSE '99: The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education,* New Orleans, Louisiana, United States. 132-135.

Xinogalos, S., Satratzemi, M., & Dagdilelis, V. (2006). An introduction to object-oriented programming with a didactic microworld: ObjectKarel. *Computers & Education*, *47*(2), 148-171.

Yamane, T. (1973). *Statistics. An introductory analysis*. New York, New York, United States: Harper & Row.

Yin, R. K. (2003). *Case study research. design and methods*. Thousand Oaks, California, United States: SAGE Publications.

Zhu, H., & Zhou, M. (2003). Methodology first and language second: A way to teach object-oriented programming. *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, California, United States. 140-147.

11 Appendix

This chapter contains the eight research articles:

Bennedsen, J. and Caspersen, M. E. (2004). Programming in Context: a Model-First Approach to CS1. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* Norfolk, Virginia, United States, 477 – 481.

Bennedsen, J., & Caspersen, M. E. (2005b). Revealing the Programming Process. *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, United States, 186 – 190.

Bennedsen, J. and Fjuk, A. (2006). Learning Object-Orientation by Professional Adults. *International Journal of Continuing Engineering Education and Life-Long Learning,* 16(6), 453 – 465.

Bennedsen, J., Berge, O. and Fjuk, A. (2005). Examining social interaction patterns for online apprenticeship learning – Object-oriented programming as the knowledge domain. *European Journal of Open, Distance and E-learning*. 2005 / II.

Bennedsen, J. and Caspersen, M. E. (2006a). Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming? *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education),* 38(2), 39 – 43.

Bennedsen, J. and Caspersen, M. E. (2005a). 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, Washington, United States, 155 – 163.

Bennedsen, J. and Eriksen, O. (2006). Categorizing Pedagogical Patterns by Teaching Activities and Pedagogical Values. *Journal of Computer Science Education,* 16(2), 157 – 172.

Bennedsen, J. (2006b). The Dissemination of Pedagogical Patterns. *Journal of Computer Science Education*, 16(2), 119 – 136.

11.1 Programming in Context: a Model-First Approach to CS1

Bennedsen, J. and Caspersen, M. E. (2004). Programming in Context: a Model-First Approach to CS1. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, United States, 477 – 481.