Instructional Design of a Programming Course — A Learning Theoretic Approach

Michael E. Caspersen
Department of Computer Science
University of Aarhus
Aabogade 34, DK-8200, Aarhus N
Denmark
mec@daimi.au.dk

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

ABSTRACT
We present a brief overview of a model for the human cognitive architecture and three learning theories based on this model: cognitive load theory, cognitive apprenticeship, and worked examples (a key area of cognitive skill acquisition). Based on this brief overview we argue how an introductory object-oriented programming course is designed according to results of cognitive science and educational psychology in general and cognitive load theory and cognitive skill acquisition in particular; the principal techniques applied are: worked examples, scaffolding, faded guidance, cognitive apprenticeship, and emphasis of patterns to aid schema creation and improve learning. As part of the presentation of the course, we provide a characterization of model-driven programming —the approach we have adopted in the introductory programming course. The result is an introductory programming course emphasizing a pattern-based approach to programming and schema acquisition in order to improve learning.

Categories and Subject Descriptors
D2.3 [**Software Engineering**]: Coding Tools and Techniques – *object-oriented programming*

K3.2 [**Computers & Education**]: Computer and Information Science Education – *computer science education, information systems education*.

General Terms: Design.

Keywords: Cognition, learning, cognitive load theory, cognitive apprenticeship, worked examples, object-oriented programming, model-driven programming, instructional design, pattern-based approach to programming education.

ICER'07, September 15–16, 2007, Atlanta, Georgia, USA.

1. INTRODUCTION
Learning to program is notoriously considered difficult [56]. In spite of more than forty years of experience, teaching programming is still considered a major challenge; in fact it is considered one of seven grand challenges in computing education [37].

A minor but remarkable collection of programming education research from the past ten to fifteen years concerns *a pattern-based approach to instruction* which utilize a shift from emphasis on learning the syntactic details of a specific programming language to the development of general problem-solving and program-design skills [19]. The approach was motivated by a shared perception that too many students cannot write reasonable programs even after one or two semesters of programming education. The approach was also motivated by the fact that "textbooks address top-down design by admonishing students to break larger problems into smaller problems and by giving static examples that illustrate a very dynamic process." (p. 1). A static program example presented in a textbook reveals nothing about the process of developing the program. Consequently, students get no insight into how problems can be broken down and solved. The last motivating factor was an urge to take pedagogical issues into account: "There is indeed little discussion of the teaching of programming that relates to pedagogy and almost none that address how the process of learning might or should affect instruction." [19]. This paper describes how learning theories may affect instructional design of a programming course.

Researchers in cognitive science and educational psychology have developed numerous learning theories [23]. In this paper, we investigate cognitive load theory, cognitive apprenticeship, and the theory of worked examples as the learning theoretic foundation for the instructional design of an introductory programming course. According to Valentine [69], many papers dealing with CS1/2 topics fall in the so-called Marco Polo category, "I went there and I saw this". Contrary to this, our aim is to provide a reasoned, reflective description of the instructional design of an introductory programming course in terms of concepts, techniques, and effects of the aforementioned learning theories.

The learning theories we apply are based on the assumption that the human cognitive architecture consists of working memory and long-term memory and that cognition takes place through creation of schemas stored in long-term memory. The first theory we consider is cognitive load theory [11, 46, 60, 61, 62]. Working memory has a limited capacity; the fundamental axiom of cognitive load theory (based upon the model of cognitive architecture) is

that learning outcome is optimized when cognitive load fully utilizes the capacity of working memory with elements that allow for optimal schema acquisition. The second theory we consider is cognitive apprenticeship [15, 16]. The theory of cognitive apprenticeship holds that masters of a skill often fail to take into account the implicit processes involved in carrying out complex skills when they are teaching novices. To combat these tendencies, cognitive apprenticeship is designed, among other things, to bring these tacit processes into the open, where students can observe, enact, and practice them with help from the teacher. The third theory we consider is worked examples. Worked examples are "instructional devices that provide an expert's problem solution for a learner to study. Worked-examples research is a cognitive-experimental program that has relevance to classroom instruction and the broader educational research community." [2] (p. 181).

The paper is structured as follows. Section 2 provides a brief overview of related work. Section 3 gives an introduction to cognition and learning including the human cognitive architecture and a more detailed presentation of the four learning theories. Section 4 is a short characterization of goals and purpose of the course under consideration. In section 5 we reflect upon the course design according to results of cognitive science and educational psychology in general and cognitive load theory, cognitive apprenticeship, cognitive skill acquisition, and worked examples in particular. The last section is the conclusion.

Major parts of the paper are excerpts of a recent PhD dissertation; further details can be found therein [9].

2. RELATED WORK

Others in the computer science community have used cognitive load theory as a basis for computing education research [20, 38, 67]. However, we have no knowledge of the use of cognitive load theory for the instructional design of an introductory object-oriented programming course.

Tuovinen [67] discuss general principles from cognitive load theory. He is especially interested in the role of prior knowledge, the format of materials, and the variability of learning tasks. However, he does not apply the guidelines in the context of introductory programming.

Mead et al. [38] use cognitive load theory to develop anchor graphs: "An anchor graph brings together the idea of anchor concept and cognitive load to provide a structure within which course layout can be planned" (p. 182). The authors have developed an initial anchor graph for OOP focusing on the conceptual framework for object-orientation.

Muller makes use of cognitive load theory to discuss aspects of a programming course. Muller's primary focus is on algorithmic problem solving (though in the context of object-oriented programming), and the pattern focus is on algorithmic patterns only [41, 42, 43].

Upchurch and Sims-Knight [68] describe how they have used cognitive apprenticeship in a laboratory component in a software engineering course. They focus on software development processes to "support students in learning the mental habits of skilled practitioners". The authors use cognitive apprenticeship for teaching object-oriented design in a non-programming context [59].

Chee [76] describes how cognitive apprenticeship is used in technologically supported learning of Smalltalk programming. Chee discusses different aspects of cognitive apprenticeship and how it is supported by the smallTALKER learning environment.

Segal and Ahmad [58] found that worked examples when learning programming languages may be seen as being the primary source of learning material even when the examples are not fully understood, especially if the exercises bear a similarity with an assignment.

3. COGNITION AND LEARNING

An instructional design that does not take the learner into account is of limited value. The purpose of this section on cognition is to provide a basic conceptual framework for use in the rest of the paper to discuss the instructional design of an introductory programming course.

Unfortunately, there is little discussion and research of the teaching of programming that relates to pedagogy, and almost none that address how the process of learning might or should affect instruction [19]. The report on strategic directions in computer science education concurs: "We must view changes in pedagogy as opportunistically as we view changes in research specialties" [66]. There is, however, a slow but increasing awareness of the benefits of applying models and research results from cognitive science and learning theory to instructional design.

3.1 The Human Cognitive Architecture

We begin by discussing aspects of *human cognitive architecture*. All human learning and work activities rely on two of our memory systems: *working memory* and *long-term memory* and the partnership they share. As its name implies, working memory is the active partner (as you read this and think about its relevance to the paper, it is your working memory that does the processing). While in learning mode, new information from the environment is processed in working memory to form knowledge structures called *schemas*, which are stored in long-term memory. Schemas are memory structures that permit us to treat a large number of information elements as if they are a single element. New information entering working memory must be integrated into pre-existing schemas in long-term memory. For this to take place, relevant schemas in long-term memory must be activated and *decoded* into working memory, where integration takes place. The result is an encoding of extended schemas stored in long-term memory. The process is known as *schema acquisition*, and this model of the human cognitive architecture is presented in Figure 1. Our model is adopted from Newell, Rosenbloom, and Laird's Soar model described in [45].

3.1.1 Schemas

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

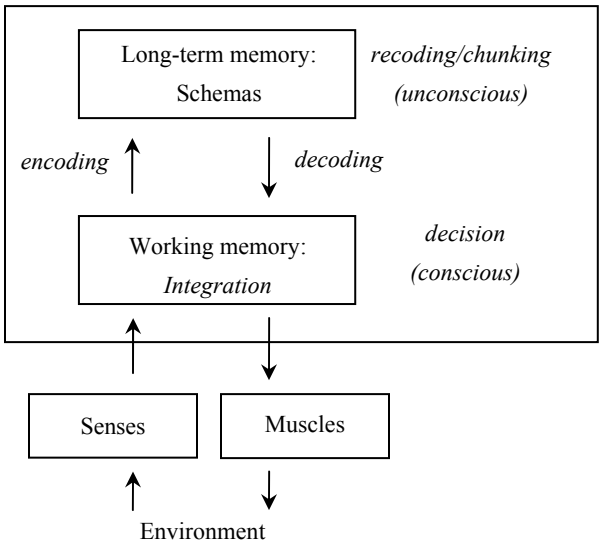

Figure 1: *A model of the human cognitive architecture*

3.1.2 Chunking

In his seminal paper [40], George A. Miller observed that the number of chunks of information is constant for working memory. More precisely, Miller found that working memory has a capacity of about "seven plus or minus two" chunks —independent of the number of bits per chunk.[1] *Recoding* or *chunking* is the process of reorganizing information from many chunks with few bits of information to fewer chunks of many bits of information. By recoding information, we can make more efficient use of working memory and consequently increase the amount of comprehensible information. Once learned, these schemas are kept in long-term memory and therefore do not affect the cognitive load of working memory.

In contrast to working memory, long-term memory has a massive capacity for information storage; however, it is the inert member of the memory partnership. All conscious processing takes place in working memory, but working memory and long-term memory work closely together. When we encode information, we create/modify schemas – also the schemas for decoding the encoded information. A schema can hold a huge amount of information, but is treated as one element of information in working memory. Thus, the more complex schemas the more advanced processing can take place in working memory. Consequently, the definition of learning is that information has successfully been encoded into long term memory.

3.2 Cognitive Load Theory

Cognitive load theory presumes that we have an unlimited amount of long-term memory [30]. Cognitive load theory is a set of learning principles that deals with the optimal usage of the working memory. To be a bit more precise:

[1] Miller's simple hypothesis is no longer tenable. Chase and Ericsson [12] have showed that purposeful training, based upon metacognitive mnemonic strategies [1], can triple the apparent working memory capacity. However, the fundamental theory of chunking and schema acquisition still applies.

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

Cognitive load theory is a universal set of learning principles that are proven to result in efficient instructional environments as a consequence of leveraging human cognitive learning processes [14].

John Sweller [61] suggests that novices who are unable to recognize a schema to solve a problem must resort to ineffective problem solving strategies like means-ends analysis [44]. Sweller suggests that problem solving by means-ends analysis requires a relatively large amount of cognitive processing capacity, which may not be devoted to schema construction. Instead of problem solving, Sweller suggests that instructional designers limit cognitive load by designing instructional materials like *worked-examples*. We return to these later.

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

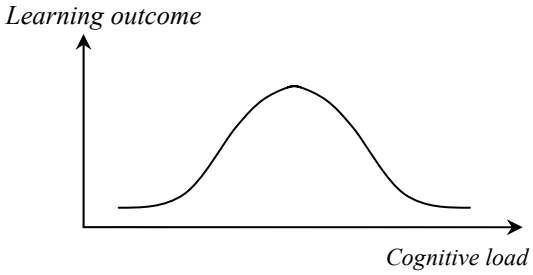

Figure 2: *Learning outcome as a function of cognitive load*

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

Extraneous cognitive load (E) is caused by instructional procedures that interfere with, rather than contribute to, learning. Extraneous cognitive load might impede learning, since it requires working memory resources that can no longer be devoted to cognitive processes associated with learning. Furthermore, cognitive resources required by extraneous cognitive load might result in an overall cognitive load that exceeds the limits of working memory capacity.

Germane cognitive load (G) is a non-intrinsic cognitive load that contributes to, rather than interferes with, learning by supporting schema acquisition [64]. Germane cognitive load is imposed by adding higher-level cognitive processes that aid schema acquisition and automation.

Intrinsic cognitive load (I) is cognitive load intrinsic to the problem that cannot be reduced without reducing understanding. Intrinsic cognitive load depends on the *relational complexity* of the

to-be-learned content (so-called *element interactivity*) and the learner's degree of prior knowledge.

Informally, we can express the relationship between L, E, G, and I as: $L = E + G + I$.

In these terms, the challenge of balancing cognitive load for optimal learning becomes a question of minimizing E and maximizing G. Cognitive load theory have developed several techniques (called effects) to help minimize E and maximize G.

Most of the research in cognitive load theory is focused on identifying so-called *effects* with associated instructional techniques. In the following we present four such effects.

3.2.1 Worked examples effect

Alternation of worked examples and problems increase learning outcome and transfer (a worked example is a demonstration of problem solving by the instructor) [63].

A worked example is, as the name suggest, a description of how to solve a problem. It focuses on problem states and the steps needed for the solution, thereby helping student to form schemas. As Sweller et al. conclude: "learners often view worked examples, rather than explanatory texts, as the primary and most natural source of learning material" [64] (p. 274).

The relevance of worked examples to programming education is explicitly expressed by several authors: "The worked examples literature is particularly relevant to programs of instruction that seek to promote skill acquisition, e.g. music, chess, and programming" [2] and "Frequently studied tasks include [...] computer programming" [73]. Trafton and Reiser found that college students performed better after studying worked examples (when learning LISP) than from solving traditional problems [65].

3.2.2 Example completion effect

Worked examples only work if the student studies the example. Chi et al. [13] found that "poor" students only study worked examples when they resemble conventional problems. Consequently, only using worked examples may be beneficial to "good" students but "poor" students may not create schemas. To overcome this, van Merriënboer and Krammer [72] suggested the use of *completions problems* in the approach to teaching programming called the reading approach: "This approach emphasizes the reading, modification and amplification of nontrivial, well-designed and working programs" (p. 257). Merriënboer have made two controlled experiments in order to evaluate the effect of using completions problems. One where high school students should learn COMAL-80 [70] and one where undergraduate students should learn turtle graphics [71]. In both experiments, the completion group outperformed the group generating programs from scratch.

3.2.3 Variability effect

Worked examples with high variability increase cognitive load *and* learning (provided that intrinsic cognitive load is sufficiently low); identification of this effect lead to the notion of germane cognitive load [47].

Using variations of practice is traditionally seen as a positive element in an instructional design because it fosters transfer. The rationale is that students notice the similarities between different situations; as Detterman notes: "transfer occurs, when it occurs, because of the common elements in the two situations" [18] (p.6).

Quilici and Mayer [52] demonstrated that students learning statistics in a high variability setting performed better than students in a low variability setting.

3.2.4 Expertise-reversal and guidance-fading effect

The previous effects were demonstrated using novices. In the late 1990s, effects were tested under the new conditions of students that had developed some expertise. It was found that effects gradually disappear as students develop expertise. But it turned to be worse than that; it was demonstrated that with further expertise, effects reverse, i.e. the learning outcome was reduced [29].

The expertise-reversal effect was used to demonstrate the guidance-fading effect: complete-examples followed by partially completed examples followed by full problems is superior to any of the three used in isolation [54].

3.3 Cognitive Apprenticeship

The theory of cognitive apprenticeship holds that masters of a skill often fail to take into account the implicit processes involved in carrying out complex skills when teaching novices. To combat these tendencies, cognitive apprenticeship is designed, among other things, to bring these tacit processes into the open, where students can observe, enact, and practice them with help from the teacher [15, 16].

Collins et al. describe cognitive apprenticeship as follows: "We call this rethinking of teaching and learning in school *cognitive apprenticeship* to emphasise 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 emphasises their uses in solving problems and carrying out tasks. Second, our term, cognitive apprenticeship, refers to the learning-through-guided-experience on cognitive and meta-cognitive, 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." [15] (p.457).

According to [16], traditional apprenticeships have 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" [28] (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" [16] (p. 7). Fading is, as the word suggests, the master gradually pulling back, 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, challenging and encouraging the apprentice, etc.

Collins, Brown and Holum have described how these aspects can be used in a more traditional, school based educational system: "In order to translate the model of traditional apprenticeship to cognitive apprenticeship, teachers need to: identify the process of the task and make them 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" [16] (p. 8).

3.4 Worked Examples

Studies of students in a variety of instructional situations have shown that students prefer learning from examples rather than learning from other forms of instruction (e.g. [13, 33, 51]). Students learn more from studying examples than from solving the same problems themselves [8, 17]. The relevance of worked examples to programming education is explicitly expressed: "The worked examples literature is particularly relevant to programs of instruction that seek to promote skill acquisition, e.g. music, chess, and programming" [2].

Atkinson et al. [2] emphasize three major categories that influence learning from worked examples; we present the categories as *how-to* principles of constructing and applying examples in education: (1) How to construct examples, (2) How to design lessons that include examples, and (3) How to foster students' thinking process when studying examples.

How to construct examples. *Accentuate subgoals*. Structuring worked examples so that they include cues or beacons that highlight meaningful chunks of information reflecting a problem's and its solution's underlying conceptual structure and meaning significantly enhances learning [10]. Catrambone demonstrates that formatting an example's solution to accentuate its subgoals can assist a learner in actively inducing the example's underlying goal structure, and that this cognitive activity presumably helps promote induction of deeper structure representing domain principles, or schemas [2]. Two techniques have particular efficacy: labels (e.g. verbal specification) and visual separation of steps. Catrambone found that it is the presence of a label, not its semantic content, which influences subgoal formation.

How to design lessons that include examples. *At least add a second example*. Educators must decide how many examples to provide for each problem type. The number may be constrained by external factors, but Reed and Bolstad [53] indicates that one example may be insufficient for helping students to induce a usable idea, and that the incorporation of a second example, especially one that is more complex than the first, increases students' learning outcome significantly. Others have found similar results: education that helps to develop schemas helps in solving problems, and multiple examples of the same schema improves learning and transfer [22, 25].

Vary form of problem type. Novices categorize problems according to surface features of the problem statement itself, whereas experts categorize problems according to features and structural similarities of their solution [74]. Variation of form (e.g. cover story) can help novices to realize that there is a many-to-one relationship between form and problem type and vice versa: "when students see the same battery of cover stories used across problem types, they are more likely to notice that surface features are insufficient to distinguish among problem types" [52]. This principle is a supplement to the variability-effect of section 3.2.3.

Alternate examples and practice problems. Lessons that pair each worked example with a practice problem and intersperse examples throughout practice will produce better outcomes than lessons in which a blocked series of examples is followed by a blocked series of practice problems [65].

How to foster students' thinking process when studying examples. *Induce self-explanations in example-based instruction*. The message from the large amount of self-explanation literature is clear: students who self-explain outperform students who do not. Furthermore, there are different forms of self-explanation, and students often fail to self-explain successfully; most learners self-explain in a passive and superficial way [13, 73]. A good deal of self-explanation research has been conducted in the context of programming education, e.g. [49, 50].

Beware of social incentives. Social incentives rarely work. Due to the fact that most learners are passive and superficial self-explainers, Researchers have made controlled experiments of initiatives aiming at increasing the quality of degree and quality of self-explanation in various social contexts. In one experiment, students were assigned the role of teacher. The hypothesis was that teaching expectancy would motivate learners to thoroughly self-explain worked examples. In another experiment, students were paired and told to explain examples to each other. Surprisingly, the result of all these experiment was counter-intuitive: Neither teaching expectancy nor peer explanations improved performance; in fact it appeared to hamper learning partly because of increased stress and reduced intrinsic motivation on the part of the students [2].

4. MODEL-BASED PROGRAMMING

In this section we provide a short characterisation of the notion of model-based programming —the approach to object-oriented programming applied in the introductory programming course under consideration. Furthermore, provide a brief characterisation of the course; we describe duration, aims and goals (expressed as expected outcome), prerequisites, and examination form.

4.1 Model-Based Programming

In [34], the object-oriented perspective on programming is defined as follows: "A program execution is regarded as a physical model simulating the behaviour of either a real or imaginary part of the world." The real or imaginary part of the world being modeled is called the referent system, and the program execution constituting the physical model is called the model system.

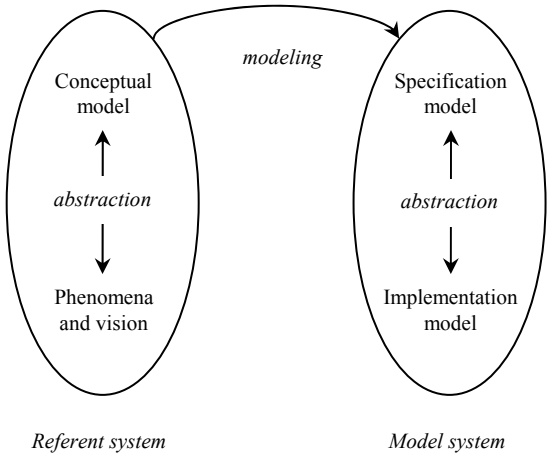

Figure 3: *Programming as a modeling process*

The programming process (initiated by a vision of a new system) involves identification of relevant concepts and phenomena in the

referent system and representation of these concepts and phenomena in the model system. This process consists of three sub-processes: abstraction in the referent system, abstraction in the model system, and modeling (no particular ordering is imposed among the sub-processes). Figure 3 illustrates the programming process as a modeling process between a referent system and a model system.

In this course, specification models are expressed as a static class model and (informal) functional specifications of the public methods of each class of the models.

We adopt an incremental approach to programming education in which novices are provided with worked examples and initially do very simple tasks and then gradually do more and more complex tasks, including design-in-the-small by adding new classes and methods to an already existing design. In [7], the authors argue that "traditional approaches to CS1 and CS2 are not in congruence with cognitive learning theory" and provide arguments for a reversed order of topics based on Bloom's classification of educational objectives [6]. The title of Buck and Stucki's paper is "Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development", and the message of the paper is that the ordering of topics that best matches Bloom's hierarchy of cognitive development is the reverse of the order of activities in the classical software lifecycle model. The students first do implementation of methods within an existing design; later they move to design; and analysis and requirements are covered in later courses.

In our current course design, we more or less ignore two of the sub-processes described in Figure 3 and restrict ourselves to the task of implementing and expanding specification models expressed as class diagrams, informal functional specifications of methods, and test suites. (The students do design-in-the-small, but the major emphasis is on implementing designs provided by the teacher.)

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

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

The loop invariant is a prime software engineering tool that allows understanding each independent part of the loop —initialization, termination, condition, progressing toward termination— without having to look at the other parts [24]. Similarly, the class invariant is a prime software engineering tool that allows us to separate consideration and evaluation of alternative representations from implementation of the methods of the class and to implement each method without worrying about the others. And the same story goes for class modeling, which is a prime software engineering tool that allows us to separate specification of each class from specification of the relationship between classes. The fundamental and recurring principle of separation of concerns permeates the techniques of all three activities. Figure 4 summarize activities and associated programming techniques for implementation of specification models.

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

Figure 4: *Activities and associated programming techniques for implementation of specification models*

In the next section, we present the instructional design of the introductory programming course where an incremental programming process and practise of these activities and techniques constitutes the primary contents of the course. Increasingly complex specification models define the course progression, not constructs of the programming language, as is the custom. We focus on teaching aspects of implementing inter-class structure.

4.2 Course Description

The course *Introduction to Programming* lasts seven weeks (one quarter) with four lecture hours and four practical lab hours with a TA per week; the course supposedly takes up one third of the students' time of the quarter. There are weekly mandatory assignments except for the first week of the course. The students are recruited from a large variety of study programs including computer science; in week six there is a subject specific project where the students work on a simple programming project relevant to their primary field of study (e.g. economy, mathematics, multimedia, or nano science). The final exam is in week eight or nine.

The formal description of the course *Introduction to Programming* is as follows:

Aims: The participants will after the course 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 participants 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.

Prerequisites: None.

Evaluation: Each student is evaluated through a practical examination where the student alone solves a concrete programming task at a computer.

For more details about the course and examination form, see [4].

4.3 An Overview of a Concrete Course

In the course *Introduction to Programming*, we focus on teaching aspects of implementing classes, inter-class structure, and methods with loops using the simple standard algorithmic pattern, *sweep* (iteration through a data set). The course is organized in six phases: (1) Getting starter, (2) Learning the basics, (3) Conceptual framework and coding recipes, (4) Programming method, (5) Subject specific assignment, and (6) Practice. The focus, goal and contents of the six phases are captured in Figure 5. In the table, we use {*method*, *class*}{*use*, *extend*, *create*} as a terminology of progression of programming assignments; the terminology is discussed further in section 5.1.1.

In the next section we present details of the instructional design of the third phase, *Conceptual framework and coding recipes*.

Phase	Week(s)	Goal/Content
1	1.5	*Getting started* (*method use*) Overview of fundamental concepts. Learning the basics of the IDE.
2	1.5	*Learning the basics* (*method extend and method create*) Class (access modifiers), object State (type, variable, value, integer) Behaviour (instantiation, constructor, method declaration, signature, formal parameter, return type, method body assignment, invoking a method, actual parameter, returning a value) Control structures (sequence, iteration)
3	1	*Conceptual framework and coding recipes* (*class extend*) Control structure (selection, more iteration) Data structure (Collections) Class relationship (aggregation, association) Schemas for implementing structure (class relations)
4	1	*Programming method* (*class extend and class create*) The mañana principle Schemas for implementing functionality (how and in which order)
5	1.5	*Subject specific assignments* (*class create*) Practice on harder and more challenging tasks (problems) Motivation: tasks/problems are picked from the domain of the students' major subject (bioinformatics, business, chemistry, computer science, economy, geology, math, multimedia, nano science, etc.)
6	0.5+	*Practice* (*class create*) Achieve routine in solving standard tasks (UML2Java)

Figure 5: *Sub-goals and progression*

The learning theories described in section three plays a major role in the overall instructional design. Worked examples (section 3.2.1) and guidance fading (section 3.2.4) are used throughout the course. In phase 1 and 2, the emphasis is on reducing extraneous cognitive load while phase 3 and 4 focus on germane cognitive load through a pattern-based approach to programming and explicit teaching of the process of programming. In phase 2, 3, and 4, we focus on revealing the programming process by applying the theory of cognitive apprenticeship. In phase 6 we focus on cognitive skill acquisition and automation through repeated practice of programming patterns.

The detailed instructional design of each phase is also based upon the learning theories; in the next section we unfold the detailed instructional design of phase 3.

5. INSTRUCTIONAL DESIGN

Section 5.1 presents a few fundamental principles of programming education that guides us in organizing a programming course. In section 5.2, we discuss organization of a small part of an introductory programming course. In doing so, we apply results of cognitive science and educational psychology in general and cognitive load theory in particular to ensure an instructional design that balances the cognitive load in order to optimize learning. In particular, we focus on programming patterns —the concrete representations of schemas in program and algorithm design.

5.1 Principles of Programming Education

In this section we briefly present four fundamental principles we use to guide the instructional design of programming courses.

The principles are (1) *Consume before produce*, (2) *Worked, exemplary examples*, and (3) *Reinforce patterns and conceptual frameworks*. We lean on more principles, but these three are the ones that most directly relates to the learning theories of section 3.

5.1.1 Consume before produce

In [48], the author introduces the *call before write* approach to teaching introductory programming, arguing that it "allows students to write more interesting programs early in the course and it familiarizes them with the process of writing programs that call subprograms; so it is more natural for them to continue writing well structured programs after they learn how to write their own subprograms". Pattis points out that the "call before write" approach requires the linguistic ability to cleanly separate a subprogram's specification from its implementation.

In [57], the author briefly mentions the notion of *consuming before producing* by providing three specific examples. One example is: "BlueJ allows beginning with an object "system" with just one class where students just interactively use instances of this class (they *consume* the notion of interacting with an object via its interface). *Producing* the possibility of interacting with an object, on the other hand, requires more knowledge about class internals and should thus be done after the principle of interaction with objects is well understood".

We rely heavily upon the principle of *Consume-before-Produce*. The principle is applicable to a wide number of topics, e.g. code, specifications, class libraries, design patterns, and frameworks. We employ the principle with respect to the way students write code at three levels of abstractions: method level, class level, and class model level as follows: (1) *Use methods* (as indicated above, BlueJ allows interactive method invocation on objects without

writing any code). At this early stage, students can perform experiments with objects in order to investigate the behaviour and determine the actual specification of a method. (2) *Modify methods* by altering statements or expressions in existing methods. (3) *Extend methods* by writing additional code in existing methods. (4) *Create methods* by adding new methods to an existing class. This may also be characterised as *extend class*. (5) *Create class* by adding new classes to an existing model. This may also be characterised as *extend model*. (6) *Create model* by building a new model for a system to be implemented.

5.1.2 Worked, exemplary examples

Examples are considered very important for learning in general. Novice programmers even think they learn programming best from examples [32]. However, computer science educators use many examples that might do more harm than good (see for example [26, 27, 35, 75]). It is therefore mandatory that these examples are not only correct but can also serve as a template for "good" design and style in any reasonable aspect.

Exemplary can mean many things depending on purpose, perspective, and point of view. Our concern is to address the topic from a didactical/pedagogical perspective.

All examples must follow all the definitions, and "rules" we have introduced, i.e. we must say as we do and do as we say. This requires very careful planning and development. According to our own experience, shortcuts or examples constructed "on-the-fly" will almost certainly introduce unintended problems.

Consequently, follow accepted principles, rules and guidelines. However, make sure to keep the focus on OOP novices. Many principles, rules, and guidelines are targeted toward professionals. They might not be applicable or even meaningful for novices. Accepted principles, rules, and guidelines encompass (1) general coding guidelines and style, like naming of identifiers, indentation, categorization of methods, like accessors, mutators, etc.; (2) common principles, like the ones summarized in [36, 39]; and (3) object-oriented design heuristics, like the ones described in [21, 55].

Finally, it pays off to get to know your students to be able to give them relevant and challenging examples. In courses such as ours, where students come from a large number of study programmes, it is vital to ensure that the examples are meaningful to all. With help from faculty members of other departments, we have developed subject specific assignments targeted at students from all study programmes that officially include the introductory programming course.

5.1.3 Reinforce patterns and conceptual frameworks

The fundamental motivation for a pattern-based approach to teaching programming is that patterns capture chunks of programming knowledge. According to cognitive science and educational psychology, explicit teaching of patterns reinforces schema acquisition as long as the total cognitive load is "controlled" (see section 3).

We reinforce patterns at different levels of abstraction including elementary patterns, algorithm patterns, and design patterns, but equally important, we provide a conceptual framework for object-orientation that qualifies modeling and programming and increases transfer [31, 34] (ch.18). Furthermore, we stress coding patterns for standard relations between classes as we shall see in the next section.

5.2 Conceptual Frameworks and Patterns

In this phase of the course (week 3-4) we introduce a subset of the conceptual framework for object-orientation developed by Knudsen et al. [31, 34]. According to Madsen et al., the object-oriented perspective on programming is defined as follows: "A program execution is regarded as a physical model simulating the behaviour of either a real or an imaginary part of the world". From the object-oriented perspective, concepts are modeled as classes and phenomena as objects. A basic understanding of phenomena, concepts, and abstraction forms the basis of the conceptual framework that provides well-defined characterizations of classification, aggregation (decomposition), and generalisation (specialisation) as ways of forming concepts from phenomena or other concepts. Object-oriented programming languages support these abstractions mechanisms in different but similar ways; thus, the conceptual framework provides knowledge and understanding that carries across different object-oriented programming languages.

The conceptual framework provides guidance for a disciplined use of components in modeling languages (e.g. UML) and abstraction mechanisms in object-oriented languages. We supplement this guidance with coding recipes for the fundamental types of relations between concepts (classes): generalisation/specialisation, aggregation/decomposition, and association. In popular terms, generalisation is known as *is-a*, aggregation as *has-a*, and association as *x-a* for any verb *x* different from *is* and *has*.

5.2.1 Model patterns

In this phase, the programming tasks are described by class models such as *ClockDisplay* and *NumberDisplay* (a *ClockDisplay* with two *NumberDisplay* objects), *DieCup* and *Die* (a *DieCup* with two *Die* objects and later a *DieCup* with an arbitrary number of *Die* objects, a *Notebook* with many *Note* objects (each with many *Keyword* objects associated), a *Playlist* with associated *Track* objects (each with associated *Picture* objects), *Account* with *Transaction* objects, etc. The generic models the students learn to implement in this phase are sketched in Figure 6.

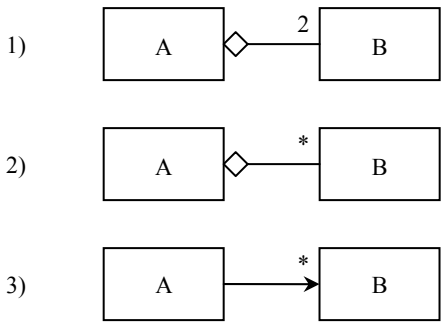

Figure 6: *Generic class models*

A typical sequence of worked examples and problems is as follows: (1) a worked example is introduced in a lecture (implementation of a simple class model through live programming) and (2) a supplement in the form of a video presentation of the same or a similar example (screen capture of narrated programming session,

see [5]), (3) a lab-assignment, (4) another assignment for the class later the same week, and (5) a mandatory assignment in the following week. This scheme repeats (partially overlapping) in the following week with a new sequence of examples of increased complexity. In the following we present two concrete examples of such sequences of examples and problems.

Example A: (A1) In a lecture we present an example of a program consisting of two classes: *ClockDisplay* and *NumberDisplay*. A *ClockDisplay* has two *NumberDisplays* (showing hours and seconds respectively). This example is from chapter 3 in Barnes and Kölling's textbook [3].

(A2) A video presentation of a partial development of the example with *ClockDisplay* and *NumberDisplay* is made available.

(A3) A1 and A2 are followed by a lab-exercise where the students interact with, modify, and extend the clock example.

(A4) A follow-up exercise where the students are provided with a partial implementation of a project with two classes: *DieCup* and *Die*. In this example a die cup always contains two die. To the students, this is a completely different example than the clock example; however, structurally they are identical (isomorphic), and the students realize this —sooner or later.

(A5) In the following week we give a mandatory assignment where the students implement a project modeling a parent relation between *Person* objects. Although this is a recursive relation (i.e. a relation from a class to itself), it is conceptually and structurally similar to the relation between *ClockDisplay* and *NumberDisplay* and to the relation between *DieCup* and *Die*. The difference, of course, is that there is only one class in play.

In a lecture following this sequence of activities related to examples of the aggregation-with-multiplicity-2 relation, we reveal the structural similarity between the three examples. Some students have already realized the similarity, but during the lecture, almost all of the students realize that the seemingly very different examples actually have a lot in common beneath the surface. This realization results in valuable schema acquisition and construction of more general competencies and knowledge.

Example B: (B1) In a lecture we present an example of a program with two classes: *Playlist* and *Track*, a *Playlist* object may be associated with any number of *Track* objects (a 0-many association).

(B2) A video presentation of a partial development of a similar example (*Account* and *Transaction*) is made available.

(B3) B1 and B2 are followed by a lab-exercise where the students interact with, modify, and extend both examples.

(B4) We provide a follow-up exercise where the students extend the *Playlist-Track* example by adding a new class representing a *Picture*. A *Track* object may have any number of *Picture* objects associated; (the idea is that the pictures associated to a track are shown in turns while the track is playing).

(B5) Again we give a mandatory assignment where the students implement a system of three classes: *Notebook*, *Note*, and *Keyword*. A notebook may contain any number of notes and a note may have any number of keywords associated (allowing notes to be searched and categorised by keyword).

The structural similarity is revealed to the students in a following lecture. In a follow-up exercise, the students are asked to develop generic coding recipes for the zero-to-many association between two classes. For example, the standard implementation of a zero-to-many association is to declare a collection object at the origin of the association and two methods to add/remove elements to/from the association (see Figure 7, compare with Figure 6-3).

```
class B { ... }
class A {
  ...
  private List<B> bs;
  public void add(B b) { bs.add(b); }
  public void remove(B b) { bs.remove(b); }
}
```

Figure 7: *Pattern for implementation of zero-to-many association*

These activities strongly support schema acquisition and hence transfer of programming competencies.

As is evident from example A and B, worked examples, example completion, and guidance fading play a key role in the organization of the student's learning process in phase 3. In activity 3, the students interact with, modify, and extend the example from activity 1 and 2. In activity 4, the students complete a new but similar example. In activity 5, the students implement a specification model. The progression through the five activities illustrates how the teacher's guidance fades.

Schema acquisition is supported by variation of cover stories of structurally similar programming tasks (e.g. *Playlist-Track* and *Account-Transaction*).

Cognitive apprenticeship occurs in activity 1 and 2 where live programming in class and videos illustrating the programming process helps revealing the tacit knowledge and implicit processes involved in program development.

The pattern-based approach to programming reveals standard solutions to recurring class structures and hence supports the goal of maximizing germane cognitive load in order to acquire the relevant cognitive schemas.

The pattern-based approach is also utilized with respect to algorithmic structures; this aspect is described in the following.

5.2.2 Algorithmic Patterns

Sweeping through a data set is a standard algorithmic pattern; zero-to-many associations invite methods with a select-like functionality, e.g. *findOne* or *findAll* associated object(s) satisfying a certain predicate. In the case of *Account-Transaction* it could be all transactions within a certain timeframe or all transactions of at least a certain amount. In the case of *Playlist-Track* it could be all tracks with a certain rating or (one of) the most popular track.

Through several similar examples, we urge the students to identify algorithmic patterns to solve these kinds of standard problems (inductive); occasionally we provide the patterns up-front (deductive). Figure 8 shows the two algorithmic patterns for finding one or all associated objects satisfying a certain criteria (of a zero-to-many association).

```
class B { ... }
class A {
  ...
  private List<B> bs;
  public B findOneX() {
    B res= bs.get(0);
    for ( B b : bs ) {
      if ( ”b is a better X than res” ) {
        res= b;
    } }
    return res;
  }
  public List<B> findAllX() {
    List<B> res= new ArrayList<B>();
    for ( B b : bs ) {
      if ( ”b satisfies criteria X” ) {
        res.add(b);
    } }
    return res;
  }
  ...
}
```

Figure 8: *Patterns for implementing findOne and findAll*

Worked examples that the students complete before the embark on similar problems, and the faded guidance as described in example A and B above help focus on the essential aspects of a programming task, and the specific details of the programming language becomes means to an end instead of a goal in itself.

6. CONCLUSIONS AND FUTURE WORK

We have provided an overview of selected learning theories: cognitive load theory, cognitive apprenticeship, and worked examples (a key area of cognitive skill acquisition) and we have described the instructional design of a model-based, object-oriented introductory programming course according to effects and guidelines of the aforementioned learning theories. The particular effects and techniques applied are: worked examples, scaffolding, faded guidance, cognitive apprenticeship, and emphasis of patterns to aid schema creation and improve learning.

We have presented an overview of the instructional design of the complete course and argued for the design according to the learning theories. Furthermore, we have provided a detailed presentation of one of six phases of the course where we discuss the application of cognitive load theory, cognitive apprenticeship, and worked examples in a pattern-based approach to programming education.

The instructional design described in this paper has been successfully used for more than four years with more than 400 students per year. We have not yet conducted any formal evaluation of the instructional design. It would be relevant to do so, e.g. by running controlled experiments and by applying the design at other institutions thus providing the opportunity for multi-institutional and multinational studies of the effect of (elements of) the instructional design. So far, we have indications from colleagues at universities in Israel, U.K. and U.S.A. expressing interest in testing (elements of) our instructional design.

7. ACKNOWLEDGEMENT

It is a pleasure to thank David Gries for comments and suggestions for improvements of an earlier version of the paper.

8. REFERENCES

[1] Allsopp, D.H. "Metacognitive Strategies", http://coe.jmu.edu/mathvidsr/metacognitive.htm, last accessed 25 January 2007.

[2] R. K. Atkinson, S. J. Derry, A. Renkl and D. Wortham, "Learning from Examples: Instructional Principles from the Worked Examples Research," Review *of Educational Research,* vol. 70, pp. 181-214, 2000.

[3] D. J. Barnes and M. Kölling, *Objects First with Java: A Practical Introduction using BlueJ.* 3rd ed. New York: Prentice Hall, 2006.

[4] J. Bennedsen and M. Caspersen, "Assessing process and product — A practical lab exam for an introductory programming course," in *Proceedings of the 36th Annual Frontiers in Education Conference,* 2006, pp. M4E-16-M4E-21.

[5] J. Bennedsen and M. E. Caspersen, "Revealing the programming process," in *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education,* 2005, pp. 186-190.

[6] B. S. Bloom, D. R. Krathwohl and B. B. Masia, *Taxonomy of Educational Objectives. The Classification of Educational Goals. Handbook I: Cognitive Domain.* New York: Longmans, Green, 1956.

[7] D. Buck and D. J. Stucki, "Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development," in *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education,* 2000, pp. 75-79.

[8] W. M. Carroll, "Using worked examples as an instructional support in the algebra classroom," *Journal of Educational Psychology,* vol. 86, pp. 360-367, Sep. 1994.

[9] Caspersen, M.E. "Educating Novices in the Skills of Programming," DAIMI PhD Dissertation PD-07-4, University of Aarhus, 2007.

[10] R. Catrambone, "The subgoal learning model: Creating better examples so that students can solve novel problems," *J. Exp. Psychol.: Gen.,* vol. 127, pp. 355-376, Dec. 1998.

[11] P. Chandler and J. Sweller, "Cognitive Load Theory and the Format of Instruction," *Cognition and Instruction,* vol. 8, pp. 293-332, 1991.

[12] W. G. Chase and K. A. Ericsson, "Skilled memory," in *Cognitive Skills and their Acquisition* J. R. Anderson, Ed. Hillsdale, NJ: Erlbaum, 1981, pp. 141-190.

[13] M. T. H. Chi, M. Bassok, M. W. Lewis, P. Reimann and R. Glaser, "Self-explanations: How students study and use examples in learning to solve problems," *Cognitive Science,* vol. 13, pp. 145-182, 1989.

[14] R. Clark, F. Nguyen and J. Sweller, *Efficiency in Learning: Evidence-Based Guidelines to Manage Cognitive Load.* John Wiley & Sons, 2006, pp. 390.

[15] A. Collins, J. S. Brown and S. E. Newman, "Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics," in *Knowing, Learning and Instruction: Essays in Honour of Robert Glaser* L. B. Resnick, Ed. Hillsdale, NJ: Erlbaum, 1989.

[16] A. M. Collins, J. S. Brown and A. Holum, "Cognitive apprenticeship: Making thinking visible," *American Educator,* vol. 15, pp. 6-11, 38-46, 1991.

[17] G. Cooper and J. Sweller, "Effects of schema acquisition and rule automation on mathematical problem-solving transfer," *J. Educ. Psychol.,* vol. 79, pp. 347-362, Dec. 1987.

[18] D. K. Detterman, "The case for the prosecution: Transfer as an epiphenomenon," in *Transfer on Trial: Intelligence, Cognition and Construction* D. K. Detterman and R. J. Sternberg, Eds. Ablex publishing, 1993.

[19] J. P. East, S. R. Thomas, E. Wallingford, W. Beck and J. Drake, "Pattern-based programming instruction," in 1996.

[20] S. Feinberg and M. Murphy, "Applying cognitive load theory to the design of web-based instruction," in *Proceedings of IEEE Professional Communication Society International Professional Communication Conference and Proceedings of the 18th Annual ACM International Conference on Computer Documentation,* 2000, pp. 353-360.

[21] C. A. Gibbon and C. A. Higgins, "Towards a learner-centred approach to teaching object-oriented design," in *APSEC '96: Proceedings of the Third Asia-Pacific Software Engineering Conference,* 1996, pp. 110.

[22] M. L. Gick and K. J. Holyoak, "Schema induction and analogical transfer," *Cognitive Psychology,* vol. 15, pp. 1-38, 1983.

[23] J. G. Greeno, A. M. Collins and L. B. Resnic, "Cognition and learning," in *Handbook of Educational Psychology* D. C. Berliner and R. C. Calfee, Eds. New York: Macmillan, 1996, pp. 15-46.

[24] D. Gries, "What Have We Not Learned about Teaching Programming?" *IEEE Computer,* vol. 39, pp. 81-82, 2006.

[25] B. Hesketh, S. Andrews and P. Chandler, "Opinion-Training for Transferable Skills: The Role of Examples and Schema," *Education and Training Technology International,* vol. 26, pp. 105-156, 1989.

[26] S. Holland, R. Griffiths and M. Woodman, "Avoiding object misconceptions," *SIGCSE Bull,* vol. 29, pp. 131-134, 1997.

[27] C. Hu, "Dataless objects considered harmful," *Commun. ACM,* vol. 48, pp. 99-101, 2005.

[28] S. Järvelä, "The cognitive apprenticeship model in a technologically rich learning environment: interpreting the learning interaction," *Learning and Instruction,* vol. 5, pp. 237-259, 1995.

[29] S. Kalyuga, P. Ayres, P. Chandler and J. Sweller, "The Expertise Reversal Effect," *Educational Psychologist,* vol. 38, pp. 23-31, 2003.

[30] P. A. Kirschner, "Cognitive load theory: implications of cognitive load theory on the design of learning," *Learning and Instruction,* vol. 12, pp. 1-10, 2002.

[31] J. L. Knudsen and O. L. Madsen, "Teaching object-oriented programming is more than teaching object-oriented programming languages," in *ECOOP '88 European Conference on Object-Oriented Programming,* 1988, pp. 21-40.

[32] E. Lahtinen, K. Ala-Mutka and H. Järvinen, "A study of the difficulties of novice programmers," in *ITiCSE '05: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* 2005, pp. 14-18.

[33] J. LeFevre and P. Dixon, "Do Written Instructions Need Examples?" *Cognition & Instruction,* vol. 3, pp. 1, 1986.

[34] O. L. Madsen, B. Møller-Pedersen and K. Nygaard, *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, 1993.

[35] K. Malan and K. Halland, "Examples that can do harm in learning programming," in *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications,* 2004, pp. 83-87.

[36] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices.* Upper Saddle River, NJ: Prentice-Hall, 2003, pp. 529.

[37] A. McGettrick, R. Boyle, R. Ibbett, J. Lloyd, G. Lovegrove and K. Mander, "Grand Challenges in Computing: Education--A Summary," *The Computer Journal,* vol. 48, pp. 42-48, 2005.

[38] J. Mead, S. Gray, J. Hamer, R. James, J. Sorva, C. S. Clair and L. Thomas, "A cognitive approach to identifying measurable milestones for programming skill acquisition," in *ITiCSE-WGR '06: Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education,* 2006, pp. 182-194.

[39] B. Meyer, *Object-Oriented Software Construction. 2nd Ed.*, Upper Saddle River, New Jersey: Prentice Hall, 1997,

[40] G. A. Miller, "The magical number seven, plus or minus two: some limits on our capacity for processing information," *Psychol. Rev.,* vol. 63, pp. 81-97, Mar. 1956.

[41] O. Muller, "Pattern oriented instruction and the enhancement of analogical reasoning," in *ICER '05: Proceedings of the 2005 International Workshop on Computing Education Research,* 2005b, pp. 57-67.

[42] O. Muller and B. Haberman, "Guidelines for a multiple-goal CS introductory course: Algorithmic problem-solving woven into OOP," in *ITiCSE '05: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* 2005a, pp. 356-356.

[43] O. Muller, B. Haberman and H. Averbuch, "(An almost) pedagogical pattern for pattern-based problem-solving instruction," in *ITiCSE '04: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* 2004, pp. 102-106.

[44] A. Newell and H. Simon, *Human Problem Solving.* Englewood Cliffs, NJ: Prentice-Hall, 1972.

[45] A. Newell, P. S. Rosenbloom and J. E. Laird, "Symbolic architectures for cognition," in *Foundations of Cognitive Science* M. I. Posner, Ed. MIT Press, 1989, pp. 93-131.

[46] F. Paas, A. Renkl and J. Sweller, "Cognitive Load Theory and Instructional Design: Recent Developments," *Educational Psychologist,* vol. 38, pp. 1-4, 2003.

[47] F. G. W. C. Paas and J. J. G. Van Merriënboer, "Variability of worked examples and transfer of geometrical problem-

solving skills: A cognitive-load approach," *J. Educ. Psychol.,* vol. 86, pp. 122-133, Mar. 1994.

[48] R. E. Pattis, "A philosophy and example of CS-1 programming projects," in *SIGCSE '90: Proceedings of the Twenty-First SIGCSE Technical Symposium on Computer Science Education,* 1990, pp. 34-39.

[49] P. Pirolli, "Effects of Examples and Their Explanations in a Lesson n Recursion: A Production System Analysis," *Cognition & Instruction,* vol. 8, pp. 207, 1991.

[50] P. Pirolli and M. Recker, "Learning Strategies and Transfer in the Domain of Programming," *Cognition and Instruction,* vol. 12, pp. 235-275, 1994.

[51] P. Pirolli and J. R. Anderson, "The role of learning from examples in the acquisition of recursive programming skills," *Canadian Journal of Psychology,* vol. 39, pp. 240-272, 1985.

[52] J. L. Quilici and R. E. Mayer, "Role of examples in how students learn to categorize statistics word problems," *J. Educ. Psychol.,* vol. 88, pp. 144-161, Mar. 1996.

[53] S. K. Reed and C. A. Bolstad, "Use of examples and procedures in problem solving," *Journal of Experimental Psychology: Learning, Memory, and Cognition,* vol. 17, pp. 753-766, Jul. 1991.

[54] A. Renkl and R. K. Atkinson, "Structuring the Transition From Example Study to Problem Solving in Cognitive Skill Acquisition: A Cognitive Load Perspective," *Educational Psychologist,* vol. 38, pp. 15-22, 2003.

[55] A. J. Riel, *Object-Oriented Design Heuristics.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 1996.

[56] A. Robins, J. Rountree and N. Rountree, "Learning and Teaching Programming: A Review and Discussion," *Journal of Computer Science Education,* vol. 13, pp. 137-172, 2003.

[57] A. Schmolitzky, "Towards complexity levels of object systems used in software engineering education," in 2005.

[58] J. Segal and K. Ahmad, "The Role of Examples in the teaching of Programming Languages," *Journal of Educational Computing Research,* vol. 9, pp. 115-129, 1993.

[59] J. E. Sims-Knight and R. L. Upchurch, "Teaching Object-Oriented Design Without Programming: A Progress Report," *Journal of Computer Science Education,* vol. 4, pp. 135-156, 1993.

[60] J. Sweller, "Cognitive technology: Some procedures for facilitating learning and problem solving in mathematics and science," *J. Educ. Psychol.,* vol. 81, pp. 457-466, 1989.

[61] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cognitive Science,* vol. 12, pp. 257-285, 1988.

[62] J. Sweller and P. Chandler, "Why Some Material Is Difficult to Learn," *Cognition and Instruction,* vol. 12, pp. 185-233, 1994.

[63] J. Sweller and G. A. Cooper, "The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra," *Cognition and Instruction,* vol. 2, pp. 59-89, 1985.

[64] J. Sweller, J. J. G. Van Merriënboer and F. G. W. C. Paas, "Cognitive Architecture and Instructional Design," *Educational Psychology Review,* vol. 10, pp. 251-296, 1998.

[65] J. G. Trafton and B. J. Reiser, "The contributions of studying examples and solving problems to skill acquisition," in *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society,* 1993, pp. 1017-1022.

[66] A. B. Tucker, "Strategic directions in computer science education," *ACM Comput. Surv.,* vol. 28, pp. 836-845, 1996.

[67] J. E. Tuovinen, "Optimising student cognitive load in computer education," in *Proceedings of the Fourth Australian Computing Education Conference,* 2000, pp. 235-241.

[68] R. L. Upchurch and J. E. Sims-Knight, "Integrating software process in computer science curriculum," in *Proceedings of the 27th Frontiers in Education Conference,* 1997.

[69] D. W. Valentine, "CS educational research: A meta-analysis of SIGCSE technical symposium proceedings," in *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* 2004, pp. 255-259.

[70] J. J. G. Van Merriënboer, "Strategies for programming instruction in high school: Program completion vs. program generation," *Journal of Educational Computing Research,* vol. 6, pp. 265-285, 1990.

[71] J. J. G. Van Merriënboer and M. B. M. Croock, "Strategies for computer based programming instruction: program completion vs. program generation," *Journal of Educational Computing Research,* vol. 8, pp. 365-394, 1992.

[72] J. J. G. Van Merriënboer and H. P. M. Krammer, "Instructional strategies and tactics for the design of introductory computer programming courses in high school," *Instructional Science,* vol. 16, pp. 251-285, 1987.

[73] K. VanLehn, "Cognitive Skill Acquisition," *Annual Review of Psychology,* vol. 47, pp. 513-539, 1996.

[74] K. VanLehn, "Problem solving and cognitive skill acquisition," in *Foundations of Cognitive Science* M. I. Posner, Ed. MIT Press, 1989, pp. 527-579.

[75] R. Westfall, "Technical opinion: Hello, world considered harmful," *Commun. ACM,* vol. 44, pp. 129-130, 2001.

[76] Yam San Chee, "Cognitive apprenticeship and its application to the teaching of Smalltalk in a multimedia interactive learning environment," *Instructional Science,* vol. 23, pp. 133-161, 1995.