Notional Machines and Introductory Programming Education

Article *in* ACM Transactions on Computing Education · June 2013

DOI: 10.1145/2483710.2483713

1 author:

Juha Sorva
Aalto University
31 PUBLICATIONS **247** CITATIONS

Notional Machines and Introductory Programming Education

JUHA SORVA, Aalto University

This article brings together, summarizes, and comments on several threads of research that have contributed to our understanding of the challenges that novice programmers face when learning about the runtime dynamics of programs and the role of the computer in program execution. More specifically, the review covers the literature on programming misconceptions, the cognitive theory of mental models, constructivist theory of knowledge and learning, phenomenographic research on experiencing programming, and the theory of threshold concepts. These bodies of work are examined in relation to the concept of a "notional machine"—an abstract computer for executing programs of a particular kind. As a whole, the literature points to notional machines as a major challenge in introductory programming education. It is argued that instructors should acknowledge the notional machine as an explicit learning objective and address it in teaching. Teaching within some programming paradigms, such as object-oriented programming, may benefit from using multiple notional machines at different levels of abstraction. Pointers to some promising pedagogical techniques are provided.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms: Languages, Human Factors, Theory

Additional Key Words and Phrases: Notional machine, introductory programming education, CS1, misconceptions, mental models, constructivism, phenomenography, threshold concepts, literature review

ACM Reference Format:
Sorva, J. 2013. Notional machines and introductory programming education. ACM Trans. Comput. Educ. 13, 2, Article 8 (June 2013), 31 pages.
DOI: http://dx.doi.org/10.1145/2483710.2483713

1. INTRODUCTION

This article reviews the literature on learning the runtime dynamics of computer programs in the context of introductory programming education. The review is structured around the concept of a "notional machine", an abstract computer responsible for executing programs of a particular kind. I use the notional machine concept to discuss the applicability of learning theory to introductory programming education and as a lens through which to view research findings.

A Multiparadigmatic View. According to the philosopher of science Charles Sanders Peirce, "reasoning should not form a chain which is no stronger than its weakest link, but a cable whose fibers may be ever so slender, provided they are sufficiently numerous and intimately connected" [Menand 1997, quoting the 1868 original]. More recently, but much in this pragmatist spirit, the restrictiveness of strict paradigmatic dogmas on research approaches and theories is being recognized; paradigm pluralism and mixed-methods research are increasingly being advocated both generally

Author's address: J. Sorva, Department of Computer Science and Engineering, Aalto University, P.O. Box 15400, FI-00076 AALTO, Finland.

DOI: http://dx.doi.org/10.1145/2483710.2483713

[Morgan 2007; Tashakkori and Teddlie 2010] and within computing education research (CER) in particular [Thota et al. 2012]. Where findings from different research traditions point in the same direction, they strengthen each other. Where they point in different directions, they give us food for thought and remind us that learning is complex and multilayered. Often, different theoretical perspectives complement rather than conflict with each other, and something may be learned from points of conflict, too.

This review encompasses work that comes from multiple threads within several disciplines of research—psychology, education, and computing. These threads vary greatly in their epistemological foundations, research foci, and typical methodology, yet all have contributed in different ways to what we know about beginners' struggles with program dynamics and the notional machine. My aim is to collect what is known about this topic and to synthesize a coherent whole from the various threads.

Article Structure. Section 2 introduces the concept of a notional machine, which the following sections then relate to different (although not disjoint) bodies of research: studies of misconceptions (Section 3), mental model theory (Section 4), constructivism (Section 5), phenomenographic research on the different ways in which learners understand programming (Section 6), and the theory of threshold concepts (Section 7). Section 8 considers the pedagogical implications of the literature for teaching students about notional machines. Section 9 is reserved for the role of notional machines within object-oriented programming. Section 10 contains a few concluding remarks.

This article, which focuses on notional machines, has been edited down from a more extensive literature review of research on introductory programming education, which appears in the author's doctoral dissertation [Sorva 2012]. The philosophical foundations of the learning theories that feature in this article—and the tensions between them—are also discussed in that work. Parts of Sections 7 and 8 have originally been published in a conference paper [Sorva 2010].

2. NOTIONAL MACHINES—WHAT ARE THEY?

The term *notional machine* was introduced to CER by Benedict du Boulay, who used it to refer to "the general properties of the machine that one is learning to control" as one learns programming [du Boulay 1986]. A notional machine is an idealized computer "whose properties are implied by the constructs in the programming language employed" [du Boulay 1986], but which can also be made explicit in teaching [du Boulay et al. 1981].

Abstractions are formed for a purpose; the purpose of a notional machine is to explain program execution. A notional machine is a characterization of the computer in its role as executor of programs in a particular language or a set of related languages. A notional machine encompasses capabilities and behaviors of hardware and software that are abstract but sufficiently detailed, for a certain context, to explain how programs get executed and what the relationship of programming language commands is to such executions. The following quote from Bruce-Lockhart and Norvell [2007] illustrates this idea well.

> As we struggled to impart to our students that each instruction they wrote was meaningful, we had an important insight. The machine (or system) T we were programming (and which we wanted the students to understand), was not really a computer, at least in the classic, hardware, sense. Consider the following simple C code:

```
int x=5;
int y = 12;
int z;
z = y/5 + 3.1;
```

> In the language of programming, we say, there are four instructions to be executed. Instructions to what and to be executed by what? T of course, but T is certainly not the CPU. The first three "instructions" are actually to the compiler. . . . The fourth is a minefield. There's a truncation and two automatic type conversions. . . . T is at least partly defined by the language. In the case of C++ and Java languages, T is an abstraction combining aspects of the computer, the compiler and the memory management scheme. Our T is not nearly as "knowable" [as some other systems]. That does not relieve us of the responsibility of at least trying to define it.

2.1. Many Machines

Since a notional machine is tied to a way of programming, different kinds of programming languages will have different notional machines. An object-oriented Java notional machine can be quite different from a functional Lisp notional machine. Most notional machines that execute Prolog are likely to be quite different again. Similar languages may be associated with similar or even identical notional machines. Some notional machines may not be very 'machine-like' at all if they are based on, for example, mathematics and lambda calculus.

Not only are there different notional machines for different languages and paradigms, but even a single language can be associated with different notional machines. After all, there is no one unique abstraction of the computer for describing the execution of programs in a language. Let us consider, for instance, the following ways of understanding the execution of Java programs.

One notional machine for single-threaded Java programs could define the computer's execution-time behavior in terms of abstract memory areas such as the call stack and the heap and control flow rules associated with program statements. The notional machine embodies ideas such as "the computer is capable of keeping track of differently named variables, each of which can have a single value," "a frame in the call stack contains parameters and other local variables," "the computer goes through the lines of the program in order except when it encounters a statement that causes it to jump to a different line," and so on. A Java notional machine at a higher level of abstraction could define the computer as a device that is capable of keeping track of objects that have been created and passing messages between these objects as instructed by method calls in a Java program. Objects take turns at performing their defined behaviors and stop to wait for other objects whose methods they call. The computer stores the objects and makes sure each object gets its "turn to act" when appropriate. A third Java notional machine could define the role of the computer on a relatively low level of abstraction in terms of bytecodes and the components of the Java Virtual Machine.

2.2. Characteristics of Notional Machines

To summarize, a notional machine:

—is an idealized abstraction of computer hardware and other aspects of the runtime environment of programs;
—serves the purpose of understanding what happens during program execution;
—is associated with one or more programming paradigms or languages, and possibly with a particular programming environment;
—enables the semantics of program code written in those paradigms or languages (or subsets thereof) to be described;
—gives a particular perspective to the execution of programs; and
—correctly reflects what programs do when executed.

It is also instructive to consider what a notional machine is *not*, as I use the term.

A notional machine is not a mental representation that a student has of the computer, that is, someone's notion of the machine. Students do form mental models of notional machines, however, as discussed in the following text.

A notional machine is not a description or visualization of the computer either, although descriptions and visualizations of a notional machine can be created by teachers for students, for instance.

Finally, a notional machine is often not a general, language- and paradigm-independent abstraction of the computer. Although some notional machines are generic enough to cover many languages, a whole programming paradigm, or even all programming languages, the prototypical notional machine is limited to a single language or a few similar languages. From the perspective of the typical monolingual CS1 course, the monoglot programming beginner, and the present review, it is the paradigm- and language-specific notional machines that are generally of greater interest.

The next five sections relate threads of work within CER to the concept of a notional machine. The focus of the existing literature is on imperative programming (either procedural or object-oriented) and so is that of my review. Imperative object-oriented programming is a mainstream CS1 paradigm within which there are several different ways to think about notional machines, which is discussed separately in Section 9. I gloss over "the third CS1 paradigm" of functional programming; the empirical results and pedagogical implications discussed do not necessarily apply to functional/declarative programming.

3. NOTIONAL MACHINES AND THE MISCONCEPTIONS LITERATURE

> "On two occasions I have been asked,—"Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" . . . I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question" [Babbage 1864].

In some disciplines, concepts and phenomena are largely negotiable and up for interpretation. Students may be encouraged to interpret things in a personal way and to develop alternative conceptual frameworks. Certainly, many computing concepts are like this, too. However, computing also features many concepts that are precisely defined and implemented within technical systems. Students are expected to reach particular ways of understanding what the assignment statement in Java does, of what an object is, and of how a given C program executes. Sometimes a novice programmer "doesn't get" a concept or "gets it wrong" in a way that is not a harmless (or desirable) alternative interpretation. Incorrect and incomplete understandings of programming concepts result in unproductive programming behavior and dysfunctional programs.

Unfortunately, misconceptions of even the most fundamental programming concepts, which are trivial to experts, are commonplace among novices and challenging to overcome. Recent studies measuring students' conceptual knowledge suggest that introductory programming courses are not particularly successful in teaching students about fundamental concepts and that the problems are not limited to a single institution nor caused by the use of a particular programming language [Elliott Tew 2010; Kunkle 2010].

3.1. Evidence of Misconceptions

Over the past few decades, many researchers have reported on the ways in which programming students struggle with fundamental concepts and the kinds of incomplete and incorrect understandings that students have exhibited. A run-through of this work appears in the following text. A more detailed catalog of specific misconceptions

reported in the literature can be found in the doctoral thesis on which the present article is based [Sorva 2012].

I use the word "misconception" here in a very broad sense to refer to understandings that are deficient or inadequate for many practical programming contexts. What I have lumped together as "misconceptions," the original researchers have variously called "misconceptions," "partial understandings," "incorrect understandings," "student-constructed rules," "difficulties," "mistakes," "bugs," and so forth.[1]

Early Work on Imperative Programming. Bayman and Mayer [1983] studied beginners' interpretations of statements in the BASIC language by asking students to write plain English explanations of programs. They list a number of misconceptions about BASIC semantics, for example, `LET` statements are understood as storing equations instead of assigning to a variable. Around the same time, Soloway, Bonar, and their colleagues also explored novice misconceptions and bugs, and discussed how they may be caused by knowledge from outside of programming, particularly by analogies with the everyday semantics of natural language [Soloway et al. 1982; Bonar and Soloway 1985; Soloway et al. 1983; du Boulay 1986].

Samurçay [1989] studied the answers that programming beginners gave to three program completion tasks and reported that variable initialization in particular was difficult for students to grasp. Putnam, Sleeman, Baxter, and Kuspa [Putnam et al. 1986; Sleeman et al. 1986] analyzed students' answers to code comprehension tests and subsequent interviews. They list numerous errors—surface and deep—that students make with variables, assignment, print statements, and control flow.

Parameter Passing and Recursion. Fleury [1991] and Madison and Gifford [1997] interviewed and observed students to discover various conceptions of parameter passing. Their results suggest that even students who are sometimes capable of producing working code that uses parameters may misunderstand the concepts involved in different ways.

Kahney [1983] discovered that students have various flawed models of recursion, such as the "looping model," in which recursion is understood to be much like iteration. Kahney's work has since been elaborated on by various authors [Bhuiyan et al. 1990; George 2000a; 2000b; Götschi et al. 2003]. Recursion is also one of the phenomena investigated by Booth in her phenomenographic work on learning to program [Booth 1992]. Booth identified three different ways of experiencing recursion: as a programming construct, as a means for repetition, and as a self-reference; students are not always able to grasp all of these aspects.

Recent Themes: OOP and Java. Since the 1990s, interest in CER has shifted from procedural programming toward object-oriented programming. Several studies have reported ways in which students misunderstand object-oriented concepts and features of OO languages. Holland et al. [1997] noted several misconceptions students have about objects. For instance, students sometimes conflate the concepts of object and class, and may confuse an instance variable called `name` with object identity. More novice misconceptions about OOP were reported by Détienne [1997] as part of her review of the cognitive consequences of the object-oriented approach to teaching programming.

Fleury [2000] reported that students form their own, unnecessarily strict rules of what happens in programs and what works in Java programming. For instance, some

[1]Some of these differences in terminology are superficial, others are motivated by learning theory—many phenomenographers, for instance, adopt a perspective in which poorer ways of understanding concepts can be seen as limited (rather than mistaken) versions of richer understandings (see Section 6).

of the students she studied thought that the dot operator could only be applied to methods and that the only purpose of a constructor was to initialize instance variables.

Hristova et al. [2003] list a number of common errors students make when programming in Java. Many of these are on a superficial syntactic level (e.g., confusing one operator with another), but some suggest deeper-lying misconceptions. Ragonis and Ben-Ari [2005a] report the results of a wide-scope, long-term, action research study of high school students learning object-oriented programming. They uncovered an impressive array of misconceptions and other difficulties students have with object-oriented concepts, the Java language, and the BlueJ programming environment.

Teif and Hazzan [2006] observed students of introductory programming in two high school courses and discuss students' conceptual confusion regarding classes and objects. For instance, students may incorrectly think that the relationship between a class and its instances is partonomic, that is, that objects are parts of a class. Eckerdal and Thuné [2005] also studied the conceptions that students have of these fundamental object-oriented concepts. Their results highlight the fact that not all students learn to appreciate objects and classes as dynamic execution-time entities or as modeling tools that represent aspects of a problem domain.

Vainio [2006] used interviews to elicit students' understandings of programming concepts and reports a number of misconceptions about fundamental concepts, for example, the idea that the type of a value can change on the fly (in Java).

Several complementary reports have affirmed the existence of a number of incorrect understandings of assignment, variables, and the relationships between objects and variables [Ma 2007; Sorva 2007; 2008; Doukakis et al. 2007]. For instance, Ma gave a large number of volunteer CS1 students a test with open-ended and multiple-choice questions about assignment in Java. He analyzed the results both qualitatively and quantitatively to understand students' mental models of assignment and reference semantics.

Sajaniemi et al. [2008] elicited the understandings that novice programmers have of program state by having CS1 students draw and write about how they perceived given Java programs' state at specific stages of execution. They discovered numerous misconceptions relating to parameter passing and object-oriented concepts.

As part of a project to develop a concept inventory for CS1, Kaczmarczyk et al. [2010] interviewed students in order to identify misconceptions. Four themes were identified: the relationship between language elements and memory, while loops, the object concept, and code-tracing ability.

Even more misconceptions about object-oriented Java programs have been uncovered by two recent studies. The interviews of students carried out by Chen et al. [2012] suggested several forms of conceptual confusion (e.g., between user-defined versus built-in types, and static members versus constant members). Shmallo et al. [2012] used questionnaires and interviews to produce a list of misconceptions that either overexpand on definitions (e.g., "an identifier may refer to two objects or more") or incorrectly constrain the properties of constructs (e.g., "it is impossible to access a static attribute through the relevant class name").

3.2. Misconceptions of the Hidden Machine

The "hidden, internal changes" that happen within the (notional) machine have been noted as being problematic for novices [du Boulay 1986]. Consider, for instance, the notion that the object assignment `a = b` (in Java) copies the values of an object's instance attributes to another object. Overcoming this misunderstanding requires the concept of a reference to an object, which is something that is not apparent in code. Many misconceptions, if not most of them, have to do with aspects that are not readily

visible, but hidden within the execution-time world of the notional machine: references, objects, automatic updates to loop control variables, and so forth.

Subtle, "hidden" aspects of programs also top various polls on difficult CS1 topics. Milne and Rowe [2002] surveyed students' and tutors' opinions of the difficulty of programming concepts. They conclude that the most difficult concepts, such as pointers, have to do with the execution-time use of memory, and that "these concepts are only hard because of the student's inability to comprehend what is happening to their program in memory, as they are incapable of creating a clear mental model of its execution." A Delphi survey of computing educators identified references, pointers, and an overall memory model as some of the most difficult topics in introductory programming [Goldman et al. 2008]. The students from various educational institutions that were surveyed by Lahtinen et al. [2005] found pointers and recursion to be the most difficult topics.

Computer Capabilities and the Superbug. Some generic misconceptions regarding the the computer and the nature of programs may lie behind many of the other more specific misconceptions that have been discovered. Pea [1986] suggested on the basis of his analysis of novice bugs that many of them are rooted in a "superbug," that is, the assumption that there is a hidden, intelligent mind within the computer that helps the programmer to achieve their goals. Evidence of unrealistic expectations of the reasoning capabilities of the computer can be seen also in the more recent literature; for instance, the students studied by Ragonis and Ben-Ari [2005a] sometimes expected that the computer can draw conclusions from the logical context of statements or the real-world semantics of identifiers .

It is not just what the computer does behind the scenes that needs to be understood. The novice must also realize what the notional machine does *not* do, unless specifically instructed by the programmer. People do not naturally describe processes in the way programmers need to. The equivalents of `else` clauses, for instance, are conspicuous by their absence in non-programmers' process descriptions, as people tend to forget about alternative branches and may consider them too obvious to merit consideration [Miller 1981; Pane et al. 2001]. The novice needs to learn what the notional machine does for them on the one hand, and what their own responsibility as a programmer is on the other.

Section Summary. The literature on misconceptions suggests that many of the problems of novice programmers are related to inadequate understandings of the notional machine, and especially to the "hidden" processes that are not directly apparent from program code. Some of these problems are specific to how the machine deals with individual constructs, whereas others are more generic and involve broader misconceptions of the notional machine and its capabilities.

4. NOTIONAL MACHINES AND MENTAL MODEL THEORY

The theoretical construct of a mental model has been used within the CER literature to explain (among other things) how people perceive the notional machine and how people trace programs mentally. This section reviews these two threads and brings them together.

4.1. An Introduction to Mental Models

A *mental model* is a mental structure that represents some aspect of one's environment. Of particular interest of most mental model theorists have been the interactions between humans and causal systems such as electrical circuits, process control mechanisms, and software [Schumacher and Czerwinski 1992]. It is posited that people use

mental models of such systems to describe the purpose and underlying mechanisms of the systems to themselves and to predict future system states.[2]

Typical Characteristics. According to Norman's [1983] seminal description, mental models:

—reflect people's beliefs about the systems they use and about their own limitations and include statements about the degree of uncertainty people feel about different aspects of their knowledge;
—provide parsimonious, simplified explanations of complex phenomena;
—often contain only incomplete, partial descriptions of operations, and may contain huge areas of uncertainty;
—are "unscientific" and imprecise, and often based on guesswork and naïve assumptions and beliefs, as well as "superstitious" rules that "seem to work" even if they make no sense;
—are commonly deficient in a number of ways, perhaps including contradictory, erroneous, and unnecessary concepts;
—lack firm boundaries so that it may be unclear to the person exactly what aspects or parts of a system their model covers—even in cases where the model is complete and correct;
—evolve over time as people interact with systems and modify their models to get workable results;
—are liable to change at any time; and
—can be "run" to mentally simulate and predict system behavior, although people's ability to run models is limited.

People commonly confuse and combine mental models of similar systems with each other. One may also have multiple mental models of a single system. Multiple models may cover different parts of the system in a nonoverlapping and complementary way, or they may be parallel—perhaps contradictory—models of the same parts.

Norman [1983] suggested that people rely on their mental models to develop behavior patterns that make them feel more secure about how they interact with systems, even when they know what they are doing is not necessary. Mental models need to be only minimally viable to be maintained, and they do not even need to be accurate for some system users to feel they are fully satisfactory—an "ignorance is bliss" approach [Westbrook 2006].

The Growth of Expertise. While mental models are clearly useful, they are also potentially dangerous. An inaccurate mental model will lead to mistakes. Kempton's [1986] research contributes the example of the thermostat: Mental models based on ill-fitting analogies to, say, car accelerators, lead people to think that turning the thermostat up to 'full throttle' will heat the home faster. A poor mental model of a computer programming environment will result in bugs. Even though people themselves do not require their mental models to be complete and accurate in order to be used, they "certainly function with varying levels of efficiency and effectiveness as they employ mental models that are inaccurate and/or incomplete" [Westbrook 2006].

The quality of mental models has been argued to be part of what sets the expert apart from the novice. Experts rely on analogies based on existing mental models as they encounter new situations that require them to form new models—as novices do. However, experts' mental models are robust, based on a principled understanding of system

[2]In this review, when I write of mental models, I refer only to so-called *causal mental models*—long-term mental models of causal systems. Another body of research, less germane for present purposes, is concerned with *logical mental models* [Johnson-Laird 1983; Markman and Gentner 2001].

components, and allow for unanticipated situations to be dealt with. Because uncertainty about system capabilities can lead to trying multiple approaches, novices tend to rely more on multiple inconsistent causal mental models of systems, while experts are less likely to do so. Compared to the ad hoc naïve models often employed by novices, experts' mental representations are relatively stable as the result of lengthy experience [Schumacher 1987; Schumacher and Czerwinski 1992; Gentner and Stevens 1983].

Learning About a System. Mental models are often not the product of deliberate reasoning; they can be formed intuitively and quite unconsciously. Prior knowledge plays a key role in the early stages of a mental model of a system, as model formation often draws on metaphors and analogies [Gentner and Gentner 1983; Schumacher and Czerwinski 1992]. To create initial models of unfamiliar systems, people typically retrieve experiences of superficially similar systems (e.g., other systems having a similar-looking GUI). With experience, understanding of causal relationships emerges through prolonged exposure to the system, but it is unrealistic to expect the transfer of a mental model before it is well ingrained. According to Schumacher and Gentner [1988], the less superficially similar two systems are, the worse transfer is, even when the systems are functionally isomorphic.

Problematic from a learning point of view is that, although models can be developed or corrected through practice and instruction, people tend to cling to emotionally comfortable and familiar existing models. Moray, for instance, found that changing one's mental model took significantly longer than it took to originally form an initial model of similar complexity [Schumacher and Czerwinski 1992]. Making matters worse is that mere coincidences can reinforce people's confidence in their existing yet flawed models [Besnard et al. 2004].

4.2. Models of the Machine

The CS1 student needs to construct a mental model of a notional machine in order to program.

Many authors have discussed the role of the machine in introductory programming in terms of mental models. Some "attribute students' fragile knowledge of programming in considerable part to a lack of a mental model of the computer" [Perkins et al. 1990]. It is claimed that novices' difficulties in developing and debugging their programs stem from the fact that "their mental model of how the computer works is inadequate" [Smith and Webb 1995]. "It is widely accepted that programming requires having access to some sort of 'mental model' of the system" [Cañas et al. 1994].

A mental model of a notional machine—together with an understanding of the program (see later discussion)—allows a programmer to make inferences about program behavior and to envision future changes to programs they are writing. A beginner will only have a mental model of the specific system that they are using for programming (a specific language-dependent notional machine), but as he gains in experience, he forms mental models of other notional machines and increasingly general schemas of computer behavior.

Mental model theory tells us that people often use analogies based on surface features when forming mental models of new systems they encounter. There is evidence of this in programming education as well. The machine's properties are implicit in the constructs of the corresponding programming language—the visible façade of the notional machine. Indeed, program code is a fertile basis for constructing a mental model as "novices make inferences about the notional machine from the names of the instructions" [du Boulay et al. 1981]. On the surface, many programming languages resemble natural language and the language of mathematics. Misconceptions (see Section 3) are brought about by unsuccessful analogies with these realms, such as

when students conclude that the Java statement `a = b + 1` defines a mathematical equation.

A novice's mental model of a notional machine is likely to be—typically of mental models in general—incomplete, unscientific, deficient, lacking in firm boundaries, and liable to change at any time. It may be based on guesswork that draws on superficial program characteristics such as keywords and identifiers. Despite such shortcomings, the learner can feel comfortable with the model and rely on it while developing behavioral patterns for programming. Novices may also use multiple, possibly contradictory models to deal with different situations. Assignment statements with integer variables might be explained with one mental model, for example, and assignment using record types with an entirely different one.

By contrast, experts' mental models are more stable and accurate, and draw on general principles rather than superficial characteristics. A challenge of programming education is to facilitate the evolution of students' models so that they have these features. Teaching about an explicit notional machine may decrease the level of freedom that learners allow themselves as they form mental models and may result in better models (see Section 8). Aiding mental model formation as early as possible is important, as changing an ingrained but flawed mental model is more difficult than helping a model to be constructed in the first place. Mental model research further predicts that novice programmers can be expected to have trouble transferring their mental models of a notional machine to an even superficially different programming language unless the original model is well ingrained through a substantial amount of practice.

4.3. Mental Models and Tracing

> "To understand a program you must become both the machine and the program" [Perlis 1982].

Mental model theory has also been applied to introductory programming education by considering the roles of mental models in program tracing.

Running a Mental Model. According to Norman's description, mental models are "runnable." That is, people can use mental models to reason about systems in particular situations, to envision with the mind's eye how a system works, and to predict the behavior and states of a system given a set of initial conditions [Gentner and Stevens 1983; Markman and Gentner 2001]. For instance, people can run their mental models to predict the trajectories of colliding balls in a physical system, the behavior of an existing software system under given parameters, or the behavior of a computer program which they are presently designing.

Mental simulation is performed in *working memory*—the main bottleneck of the human cognitive apparatus [Tuovinen 2000]. It often involves visual imagery and may have a motor component. Since the capacity of working memory is very limited, it comes as no surprise that researchers have found that mental simulations involve only a very small number of factors. According to Klein [1999], for instance, even experts' mental simulations rarely involve more than three factors (or "moving parts") and six transition states (stages). Simulating a system's behavior at a low level of abstraction can fail as a result of too many variables or states. Simulation at an excessively high level of abstraction will not produce working solutions to problems either, and even when the overall level of abstraction is appropriate, people tend to neglect or abstract out important information. To solve problems successfully, it is crucial to simulate systems at a level of abstraction that is just right for the problem at hand and to focus exactly on those factors that are important to produce the kind of prediction or solution aimed for. To do so is difficult and requires experience.

Tracing the Program and the Machine. In the context of programming, simulations of program execution are often referred to as *tracing*. Tracing is a key programming skill that expert programmers routinely use during both design and comprehension tasks [Adelson and Soloway 1985; Soloway 1986; Détienne and Soloway 1990].

One might view program tracing as running a mental model of a program with some input, while also running a separate mental model of a notional machine for which the program itself serves as input. However, when we run a program, "the computer effectively becomes the mechanism described by the program" [du Boulay 1986]. When we speak of program execution, we use expressions such as "the program does X" and "the computer does X" to mean effectively the same thing. As noted earlier, the borders of mental models are often vague. Even though the models of the machine and program can be viewed analytically as separate, they may be inextricably entwined during mental tracing and are not necessarily distinct in the programmer's memory. In the following, I write of mental tracing as the running of a single mental model that encompasses both the notional machine and the program that is being traced.

Tracing and Novices. Programmers prefer symbolic tracing that uses nonspecific values—this is indeed typical of mental simulations in general. Experts tend to only use concrete tracing in challenging situations where other methods fail. Novices, who do not perceive the kinds of patterns in code that allow for abstraction, and whose programs are buggier than those of experts, need concrete tracing often.

Unfortunately, novice programmers often struggle greatly with tracing.

In a well-known study, Lister et al. [2004] measured students' ability to trace through a given program's execution. They gave a multiple-choice questionnaire to students in a number of educational institutions around the world. The questions required the students, who were near the end of CS1, to predict the values of variables at given points of execution and to complete short programs by inserting a line of code chosen from several given alternatives. Lister et al. concluded that many students were unable to trace. While there was obviously some variation in students' ability between institutions, the results were disappointing across the board.

Other studies have produced similar results. For instance, an earlier multi-institutional study by Sleeman et al. [1986] analyzed code-driven interviews to conclude that "at least half of the students could *not* trace through programs systematically" upon request, and instead "often decided what the program would do on the basis of a few key statements." Adelson and Soloway [1985] found that novices were unable to mentally trace interactions within the system they were themselves designing. Kaczmarczyk et al. [2010] report an inability to "trace code linearly" as a major theme of novice difficulties. A series of studies has recently indicated that many students fail to understand statement sequencing to the extent that they cannot grasp a simple three-line swap of variable values [Corney et al. 2011; Simon 2011; Teague et al. 2012; Murphy et al. 2012]. In one study, the problem existed even among students taking a third programming course [Simon 2011].

Moreover, Perkins et al. [1986] found that many novices do not even try to trace the programs they write, even when they need to in order to progress. In their study, "students seldom tracked their programs without prompting." The disinclination to trace one's programs, Perkins et al. argue, may be due to reasons such as a failure to realize the importance of tracing, a lack of belief in one's tracing ability, a lack of understanding of the programming language, or a focus on program output rather than on what goes on inside.

Status Representations. Tracing a program requires the programmer to keep track of the state of program execution, that is, to simulate the job of the notional machine. A description of program state, which Perkins et al. [1986] call a *status representation*, must

be dynamic, changing as execution proceeds. A status representation is based on the elements of (one's mental model of) the notional machine used: variables, objects, references, function activations, and so on. A status representation of a complex program involves an amount of information that often exceeds the capacity of working memory, which is why we use external aids such as scraps of paper and debugging software.

Novices need concrete tracing often, but are not experienced at selecting the right "moving parts" to keep track of in the status representation, causing them to fail as a result of excessive cognitive load [Vainio and Sajaniemi 2007; Fitzgerald et al. 2008]. An example of a novice coping mechanism may be the "single-value tracing" observed by Vainio and Sajaniemi, which limits the number of nontrivial variable values in one's mental status representation to one: Only a single unnamed "slot" is used to store the value of whichever variable was most recently assigned a value that is not directly visible in code. Single-value tracing is based on an incorrect mental model of the notional machine, but the novice may not even realize this, as the approach "works fine with small programs that are typical to elementary programming courses."

Robustness. One theory of mental models was put forward by de Kleer and Brown [1981; 1983], who argue that only certain kinds of mental models, which meet certain "esthetic principles," lend themselves to answering unanticipated questions about systems. De Kleer and Brown contended that using such high-quality mental models, which the authors termed *robust*, is characteristic of expert behavior. Robustness arises out of submodels that represent components of the system: The components of a robust model are understood in terms of general knowledge that pertains to those components rather than specific knowledge that pertains to the particular configuration of the components. A nonrobust model may serve in the context of a particular system under normal circumstances. A robust model is needed for transfer to a similar but novel problem and for dealing with exceptional situations such as component malfunctions. This makes robust models highly desirable.

If de Kleer and Brown's theory holds, and assuming that computer programs are causal mechanisms of the sort that their theory applies to, then for students to transfer their understanding of one program to other programs, they need a robust model of the original program. A robust mental model would also be needed for debugging. The components of a program are the programming constructs used in the program code. In a robust model, these program components are understood in terms of their general semantics, that is, in terms of what they mean to the notional machine.

Vainio, who applied de Kleer and Brown's theory to programming [Vainio and Sajaniemi 2007; Vainio 2006], discusses the following code fragment:

```
for (i = 0; i < 10; i++) {
    System.out.println(i);
}
```

A reasonable description of this `for` loop is: "First, the variable—`i` in this case—is set to zero. Then the loop iterates over all the values of `i` from 0 to 9 and prints them out." A novice programmer may form a mental model of the code that matches this description and is viable when it comes to dealing with this specific program, but is not robust. For instance, some of the students in Vainio's study thought that *whichever variable* is used within the body of a `for` loop is always set to zero at the start of the loop. Such an understanding is clearly not generally viable. It is also not robust: The semantics of a `for` statement (a program component) are mixed up with what the `for` statement is used for in the particular context. Vainio and Sajaniemi [2007] describe such violations of robustness principles as common, attributing this in part to the tendency in CS1 courses to associate each type of problem with only a single

kind of programming construct, and each programming construct with a single kind of problem.

Section Summary. Mental model theory has contributed to our understanding of the kinds of intuitive mental representations that novice programmers form of notional machines, often initially based on superficial language features and analogies. Moreover, the theory suggests that substantial experience with a particular notional machine is needed before transfer to superficially dissimilar machines is likely to succeed. The key skill of program tracing can be viewed as the "running" of a mental model that encompasses both program and machine. Two challenges to the successful running of a mental model are keeping track of program state in working memory, and the difficulty of forming mental models that are robustly founded on context-free runtime semantics of each construct.

5. NOTIONAL MACHINES AND CONSTRUCTIVISM

The central tenet of the educational paradigm known as constructivism is that people actively construct knowledge rather than passively receiving and storing ready-made knowledge. Knowledge is not taken in as is from an external world, and it is not a copy of what a textbook or teacher said. Instead, knowledge is unique to the person or group that constructed it. Learning occurs as existing knowledge and the learners' interests interact with new experiences. On such grounds, constructivists commonly promote pedagogies in which learners have active roles.

Section 5.1 is a broad overview of constructivism (based primarily on Phillips [1995], Steffe and Gale [1995], Larochelle et al. [1998], Phillips [2000], and Tobias and Duffy [2009]). In Section 5.2, I present and comment on what has been said in the CER literature concerning the relationship between constructivism and the notional machine.

5.1. Different Constructivisms

It is nigh on impossible to find a present-day educational researcher who believes that learning simply involves the transmission, or "pouring," of pre-existing knowledge from a teacher or a book into students. Calling oneself a constructivist is politically correct; denying the active role of learners in building knowledge is to invite scorn. "There is a very broad and loose sense in which all of us these days are constructivist" [Phillips 1995]. However, we vary in how constructivist we are and in how we are constructivists.

Flavors of Constructivism. Constructivists differ among themselves as to whether individual minds or social groups (or both) are seen as the knowledge-constructing agents. *Personal constructivists* emphasize the idiosyncratic construction of knowledge by individuals. *Social constructivists* instead emphasize the importance of the social and cultural nature of individuals' knowledge construction and tend to see knowledge as something that is defined through social collaboration and language use.

Some constructivists are satisfied with a very abstract notion of knowledge construction; others draw on various theories to explain the specifics. For instance, personal constructivists draw variously on mental model theory, schema theory, and conceptual change theory, among others.

There are also many "pedagogical constructivists," who do not take a stance on theoretical issues and focus on the use of constructivist pedagogies such as problem-based or inquiry-based learning.

Degrees of Constructivism. Purists of various constructivist traditions adopt extreme epistemological stances. They argue that what we call knowledge should be discussed independently of ontology, as a real world—if one exists—if unknowable. According to these strong forms of constructivism, what we know is only our own (or our social

group's) construction and cannot be said to be true or false with respect to an external reality. The middle ground between the purists and traditional realists is occupied by moderates who agree with the constructivists that knowledge is not securely founded on an external reality and that we cannot be sure about the truth value of knowledge claims. Nevertheless, moderate constructivists argue that reality impacts what knowledge people construct and affects how useful that knowledge is and, therefore, has a place in epistemology [Phillips 1995; 2000].

5.2. Constructivisms in the Notional Machine Debate

Although forms of constructivism have impacted on computing education for decades, it is only more recently that constructivism has become a clearly visible force in the CER literature [Greening and Kay 2001]. Constructivism of one kind or another is increasingly referred to by computing educators—whether as a buzzword or as a genuine theoretical framework—as they seek to justify and inform pedagogical interventions in programming education, to motivate research questions or approaches, or as an analytical tool that retroactively justifies existing pedagogy. In this review, I focus on what has been said about the notional machine in constructivist terms; the discussion in two publications (Greening [1999] and Ben-Ari [2001]) forms the spine of this part of the review.

Just as constructivism in general is far from being a single well-defined position, scholars within CER differ greatly in their how they perceive the implications of (variants of) constructivism to introductory programming education.

Greening: Multiple Perspectives. Greening [1999] advocates a form of strong constructivism in pedagogy. He argues that the main challenge of learning programming, from the constructivist point of view, is not the acquisition of knowledge about programming languages, syntax, and semantics. Rather, learners must come to see programming as an essentially creative pursuit involving the skills of synthesizing and problem solving. Genuine skill in programming involves being able to tackle ill-structured, complex problems in authentic contexts. Greening further contends that strong constructivism is much needed in modern computer science education as students and teachers are forced to cope with the information explosion and "multiple world views in flux." Greening promotes *problem-based learning* [Savery and Duffy 1995] as an example of a pedagogy suitable for achieving these goals.

When it comes to various human aspects of computing, such as software engineering practices, good design, and usability, Greening is no doubt right to stress the usefulness of learning to cope with multiple viewpoints. However, when it comes to learning how to cope with the computer itself, a different line of constructivism-inspired thinking is suggested by Ben-Ari [2001]. The computer does not negotiate. . .

Ben-Ari: Prior Knowledge of the Computer. Ben-Ari [2001] discusses the application of constructivism to computing education, focusing primarily on what could be characterized as a moderate personal constructivism of a cognitivist bent. He emphasizes how constructivism highlights the importance of learners' prior knowledge for learning, and draws a number of conclusions that "seem to follow directly from constructivist principles." Ben-Ari's thesis rests on four claims.

—Knowledge of the computer is a prerequisite for understanding computing as we know it.
—Beginners to computing lack effective models of the computer.
—Learners necessarily construct knowledge about the phenomena they encounter, for better or worse.
—The computer forms an "accessible ontological reality".

Ben-Ari argues that, according to (cognitive) constructivist principles, when learners come across a system, they construct a mental model of it. Constructing knowledge is inevitable and will happen regardless of whether the learner has been taught a normative conceptual model of the phenomenon, although instruction can have an effect on the resulting model [Ben-Ari 2001; Ben-Ari and Yeshno 2006]. Ben-Ari marshals evidence from the literature (see Sections 3 and 4) to support his argument that when faced with abstractions such as program code, students will necessarily construct their own knowledge of the notional machine. Unfortunately, "intuitive models of computers are doomed to be nonviable."

In stressing "accessible ontological reality," Ben-Ari contends that computing (the parts of it that directly involve computers, at least) does have an ontology that is reflected in useful knowledge; this goes against the radical constructivist rejection of ontology as an epistemological basis. The computer is an artifact that behaves in a certain way, and unless the learner's understandings of the computer are a close enough match with this reality, there will soon be consequences. Feedback on nonviable understandings is often immediate and brutal in the shape of error messages, crashes, and bugs. Moreover, while the specifics of people's constructed understandings of the computer are unique, all viable understandings must match the normative model fairly closely, leaving little room for disagreement. As Ben-Ari [2001] points out, "there is not much point negotiating models of the syntax or semantics of a programming language" once the decision to use a particular programming language has been made. So much for the "spectrum of views" [Greening 1999] that constructivist educators generally hope students will explore!

The Role of the Machine. Greening is skeptical about the claim that a model of the computer is needed for learning computing, and notes that (strong) constructivism is less about prescribing what prerequisite knowledge is needed than it is about acknowledging that prior knowledge plays a part in knowledge construction. He further argues that a model of the computer will be less important in the (near?) future.

> "Increasingly, perhaps as a sign of maturity of the discipline(s) of computing, the need to understand the machine will dissipate. This statement will surely horrify some readers" [Greening 1999].

Greening appears to be talking about the importance of understanding the actual machine at a fairly low level and, in this respect, has an unhorrifying point. Levels of abstraction in programming do appear still to be rising, which may indeed mean that the low-level machine is becoming less and less important for introductory programming education. However, unless computing and the nature of program execution change in a dramatic way, it remains easy to agree with Ben-Ari's [2001] argument that students need an understanding "one level beneath" the primary targeted level (of program code) that is viable for the purpose of explaining phenomena at the targeted level. The level of abstraction of the notional machine that is needed may increase in the future, along with the level of abstraction of programming languages, but a notional machine will nevertheless be required by would-be computer programmers.

Section Summary. A moderate form of constructivism that underlines the need to identify prerequisite knowledge may be interpreted as lending support to the claim that it is crucial for beginner programmers to form an understanding of the computer that matches a normative model. This model does not need to be the actual computer, but a notional machine that operates just beneath the abstractions on which the learners primarily focus. Stronger form of constructivism, on the other hand, question educational norms that are dependent on ontology; proponents of these views may see learning about the computer as less important.

6. NOTIONAL MACHINES AND PHENOMENOGRAPHY

Phenomenography is a primarily qualitative tradition of empirical research, which has evolved from a research method used in the 1970s into a theoretically laden, methodologically varied approach to empirical research on human experience. Section 6.1 first introduces some fundamentals of phenomenographic research (primarily on the basis of Marton and Booth [1997], Marton [2000], and Pang [2003]), which Section 6.2 then relates to the notional machine.

6.1. Phenomenography

Phenomenographers explore the educationally critical variation in how people experience, perceive, or understand various phenomena. A phenomenon of interest can be specific, such as number in kindergarten mathematics, matter in physics, or variable in computing, or more generic, such as learning to program or even learning in general.

Central to the phenomenographic approach is the concept of a way of experiencing a phenomenon.

Ways of Experiencing. People experience the same phenomenon differently. Even the same person experiences the same phenomenon differently at different times and in different contexts. Are there countless significantly different ways of experiencing the same phenomenon? Phenomenographers posit that it is not so, and a researcher can describe ways of experiencing a phenomenon in an abstract way that captures the telling differences between the few qualitatively different ways of experiencing the phenomenon.

A phenomenon, viewed from a particular perspective, is characterized by a limited number of *critical aspects* that distinguish it from other phenomena. Each way of understanding a phenomenon is defined by which critical aspects are discerned; contrasts in what is discerned and what is not define the qualitative differences between ways of experiencing.

To take an example from programming, one study discovered three different ways in which CS1 students understand what an object is [Eckerdal and Thuné 2005]. The first category is very simple: An object is a piece of code. Here, only the critical aspect of program text is in focus. In a richer understanding, objects are additionally seen as active entities during a program run; this relationship between objects and execution-time events is another critical aspect of objects. In the third category, a world-modeling aspect is also attributed to objects. The results of this study are typical of phenomenography in that the categories form a logical hierarchy in which some understandings are richer, that is, additional critical aspects are discerned in them.

Learning as Change in Experience. The most important form of learning, from the phenomenographic point of view, involves becoming able to discern critical aspects of a phenomenon so that the phenomenon as a whole is experienced in a new, qualitatively different way.

Phenomenographers emphasize that the challenges of learning are intrinsically tied to the content that is being learned about and the specific learning goals. They deemphasize or even question the very ideas of generic mental representations (e.g., mental models), generic processes of learning, and generic pedagogical aids. Instead, it is recommended that researchers and teachers focus on the relationships between learners and the phenomena that they learn about, and consider pedagogy in terms of the specific critical aspects that learners are expected to learn to discern.

6.2. Experiencing Introductory Programming

Some of the phenomenographic work within CER relates to program dynamics and the notional machine.

Table I. Different Ways of Experiencing Programming

Code	Description
A	Computer programming is experienced as using a programming language for writing program texts.
B	Computer programming is experienced as above, and as a way of thinking that relates instructions in the programming language to what will happen when the program is executed.
C	Computer programming is experienced as above, and as producing applications of the kind familiar from everyday life.
D	Computer programming is experienced as above, and as a "method" of reasoning that enables problems to be solved.
E	Computer programming is experienced as above and as a skill that can be used outside the programming course and for other purposes than computer programming.

Adapted from Thuné and Eckerdal [2010].

Experiencing Specific Concepts. Some studies have investigated the ways in which programming students experience specific concepts and constructs. An emerging theme in the results of these studies is that novice programmers sometimes focus exclusively on the "obvious" static aspect of concepts that is visible in program text. Eckerdal and Thuné's study on conceptions of objects [Eckerdal and Thuné 2005], cited above, is an example of this kind of work; the authors also report an analogous categorization of understandings of the concept of class. An earlier study on ways of experiencing recursion similarly discovered that recursion sometimes only perceived as a code construct rather than in terms of runtime behavior [Booth 1992].

Experiencing Programming in General. Even more interesting for present purposes are the phenomenographic studies in which the phenomenon of interest has been broader. Developing on the pioneering work of Booth, various researchers have studied how novice programmers perceive the phenomena of *programming* and *learning to program* [Booth 1992; Bruce et al. 2004; Stoodley et al. 2004; Eckerdal et al. 2005; Thuné and Eckerdal 2009; 2010]. This body of work largely points in the same general direction, with studies supporting Booth's findings and elaborating on some themes. As a representative example, Table I shows Thuné and Eckerdal's categorization of students' ways of understanding what programming is. As in the studies of specific programming concepts, the first category describes a relatively poor way of understanding in which runtime behavior is ignored.

A beginner programmer who perceives learning to program merely as learning to write program text according to syntactic rules will inevitably miss out on many opportunities to learn until their overarching view of programming changes significantly. Phenomenographic results have highlighted the critical aspects of programming that constitute challenges for programming education. One of these challenges—the execution-time aspect which is discerned in Category B of Table I but not in Category A—corresponds roughly to the notional machine; in Category A, programming is not perceived as writing instructions for the machine to execute.

Section Summary. Novice programmers sometimes fail to perceive programming constructs as more than code and programming as more than the production of code. One of the things that is lacking from such conceptions is the action that takes place when programs are run. The limited understandings limit programming ability and opportunities for further learning. Phenomenographic work within CER can be interpreted as supporting the need for beginners to learn about a notional machine that relates program text to runtime activity within the computer.

7. NOTIONAL MACHINES AND THRESHOLD CONCEPT THEORY

Meyer and Land [2003; 2006] propose that embedded within academic disciplines there are troublesome barriers to student understanding, which they term *threshold concepts*. These "jewels in the curriculum" represent transformative points in students' learning experiences that allow them to view other concepts in a different light. Proposed threshold concepts in programming include program dynamics, information hiding, and object interaction.

Threshold concepts (TCs) are still a fairly young theoretical framework that requires better empirical support. However, they are obviously a fruitful basis for discussion and pedagogical explorations, as evidenced by the rapid growth of TC literature over the past few years. The ongoing work on threshold concepts may help us gain further insights into why some students seem not to learn "any of the stuff," while other students seem to get "all of the stuff"—a phenomenon familiar from CS1 courses. The answer may lie in the curriculum itself: TC theory suggests that some particularly transformative and integrative "stuff" leaves many students stuck and unable to proceed until they are able to see the connections between related concepts.

7.1. Characteristics of Threshold Concepts

A threshold concept is not a mere "core concept."[3] To qualify as a TC, a concept must meet a stricter definition, albeit one which is still being debated. Proposed characteristics of threshold concepts include the following [Meyer and Land 2006; Land and Meyer 2008; see also Sorva 2010].

—A threshold concept significantly *transforms* how the student perceives a subject or part thereof, and perhaps even occasions a shift of personal identity.
—A threshold concept widely *integrates* other content by exposing its interrelatedness.
—A threshold concept *leads to a new way of thinking and practicing* within a subject area; the learner becomes able to perceive concepts, problems, solutions, and the subject itself like a fully fledged member of the disciplinary community.
—Such a transformation is probably *irreversible*, that is, unlikely to be forgotten or undone.
—A threshold concept is potentially *very troublesome* to students for any of a variety of reasons including conceptual complexity, tacitness in expert practice, apparent meaninglessness, and counterintuitivity.
—Learning about a threshold concept commonly involves a potentially lengthy *state of liminality*, during which the learner oscillates between "knowing" and "not knowing," may attempt to mimic presumably correct behavior and often experiences strong negative emotions.
—Threshold concepts tend to involve the transformation of one or more "obvious" *everyday ideas* into discipline-specific forms (e.g., the general idea of abstraction into information hiding).
—A threshold concept may *mark boundaries* in "conceptual space" between disciplines or schools of thought.

7.2. Program Dynamics as a Threshold Concept

A crucial distinction in programming is the one between the existence of a program as code and its existence as a dynamic execution-time entity within a computer. Code is tangible and its existence is easy to perceive. The latter aspect of a program—

[3]In fact, threshold concepts may not be individual concepts with well-established names at all, but something broader; this is an issue that is being debated by the TC research community. My use of the word "concept" in "threshold concept" may be read as shorthand for "way of understanding certain curricular content."

the program as run by a notional machine—is much less so. An argument has been presented that program dynamics constitutes a major transformative threshold that beginner programmers must cross. I restate this argument in the following text (largely reproduced from Sorva [2010], see also Vagianou [2006], Shinners-Kennedy [2008], and Zander et al. [2008]).

Integration. A dynamic view of a program brings together program code, the state of the program, and the process that changes it, as well as the computer on which the program runs (if not the actual hardware, at least a notional machine). The ability to view programs as dynamic is required to genuinely understand a legion of other concepts and distinctions: variables and values; function declarations versus function calls; classes, objects, and instantiation; expressions and evaluation; static type declarations versus execution-time types; scope versus lifetime; and so on. The dynamic use of memory to keep track of program state is central to much of this integrative power.

Transformation. As the phenomenographic work reviewed in Section 6 has illustrated, learning to program involves a qualitative shift in how learners become able to perceive programming constructs and programming itself not only in terms of code but in relation to the notional machine.

Program dynamics transforms the universal, everyday notion of state—which might be termed a "fundamental idea" of computing [Schwill 1994; Sorva 2010]—into something that is pivotal to how the programmer thinks. A dynamic view of programs leads to what Perkins [2006] calls a new *episteme*, a new way of reasoning about programs that is impossible unless the student has ingrained the notion of program dynamics into their thoughts and practices. In particular, thinking in terms of program dynamics makes the key skill of program tracing possible (see Section 4.3).

Trouble. Program dynamics are troublesome. Practically any student of programming can pay lip service to the idea that programs are executed step by step, making things happen within the computer. However, not all of those students genuinely internalize this notion and make it work for them. The preceding sections have provided plentiful references to studies demonstrating the difficulties that novices have with understanding what happens when a program is run.

Part of the troublesomeness of program dynamics lies in its tacitness. As noted in Section 4.3, programmers rarely make explicit the dual nature of programs, which is obvious to them—when we speak of a "program," we refer to either the code, to what the code does upon execution, or to both at once. The centrality of the previously unproblematic notion of state to program dynamics may also be counterintuitive to novices [Shinners-Kennedy 2008].

Boundaries. End users and programmers have different stances toward computer programs. Only the latter group has a sense of being directly involved in what happens when a program gets executed by a computer. This is one way in which program dynamics serves as a boundary marker, separating computer programmers from nonprogrammers. A student who does not trace programs or think about the dynamics of their execution is not really thinking and practicing like a computer programmer.

Program dynamics also demarcates two schools of thought within computing: It lies at the border of computer programming and programming-as-mathematics. The latter episteme, which Dijkstra famously and controversially advocated as the perspective of choice for CS1 courses, is ruled by formal logic and proofs, and the former by testing, mental tracing, and operational reasoning [Dijkstra et al. 1989].

Irreversibility. There is little research-based evidence of the irreversibility of learning about program dynamics. However, at least the present author has never heard

of a programmer forgetting how to see programs as dynamic, traceable entities once they have made that concept their own, nor does he expect to hear of one. A sign of irreversibility may also be the difficulty that some experienced programmers have in perceiving how programming appears to the beginner who has not yet crossed this early "obvious" threshold.

Section Summary. Program dynamics—the realm of the notional machine—is a plausible threshold concept in computer programming, as it appears to have the main characteristics of such concepts. It is integrative, transformative, troublesome for many learners, sits at a disciplinary boundary, and opens up to a way of reasoning about programs as programmers do. Failing to cross this threshold is a serious obstacle to further learning.

8. PEDAGOGICAL IMPLICATIONS

In this section, I consider the pedagogical implications of what has been said, drawing eclectically on the theories discussed in the preceding five sections and their associated pedagogical recommendations.

8.1. The Notional Machine as a Learning Objective

Transforming Learners' Perspectives. Both phenomenography and the theory of threshold concepts emphasize that there are different ways of perceiving curricular content. Correspondingly, teachers should take as their goal the transformation of learners' perspectives to key content. Unless learners have access to the appropriate perspective, further teaching will be inefficient if not entirely fruitless. The threshold concepts framework suggests "a less is more approach" to teaching [Cousin 2006]: The teacher should concentrate their efforts on the selected content that transforms the student's view of the discipline and makes learning other concepts easier, rather than burying these conceptual jewels within a vast bulk of knowledge where they may go all but unnoticed.

Recursion, reference parameters, and object instantiation, for example, are concepts that are hard enough to grasp even for a novice who is capable of thinking about programs in terms of their dynamic aspect and tracing execution step by step. Without the "lens" of program dynamics, mastering those other important concepts becomes next to impossible. A programming teacher must not fall into the trap of assuming that students think as the teacher does. Instead, teachers need to help students uncover the nature of the tacit "underlying game." Neither the teacher nor the student should be allowed to settle for mere lip service to the idea that programs run step by step and use memory. A student who passes a programming course without having developed a dynamic perspective on program execution has not really learned very much about computer programming, no matter how many concept definitions they may have memorized or how many code templates they may be capable of applying. Conversely, someone who has crossed the threshold is well positioned to learn more with relative ease.

Teaching About a Machine Model. Every introductory programming course involves a notional machine; the machine is implicit in the programming language used. However, it is probably better to have learning about a notional machine as an explicit goal than an implicit one.

According to mental model researchers, it is useful to use a *conceptual model*—an explanation crafted for the purpose of explaining a system's structure and internal workings—in teaching. A conceptual model may be just a simple metaphor or analogy, or a more complex explanation of the system. Conceptual models have been found to be useful in many contexts. According to at least one review, the typical study of mental models concludes that learners taught "how it works" using a conceptual model

demonstrate better performance than those provided with "how to" instructions on controlling a system [Schumacher and Czerwinski 1992].

The use of a conceptual model is also advocated by Ben-Ari in connection with his moderate constructivism (see Section 5). Ben-Ari [2001] argues that to avoid the ill-fated construction of intuitive knowledge about the computer and programming language semantics, instruction should start with the underlying model before proceeding to the abstractions founded on it. This is in line with the findings by mental model researchers that changing an initial model tends to be more difficult than forming one in the first place. Another concern is that a seriously flawed model of a notional machine may work for explaining the behavior of some program examples, even though it fails generally, which further deepens the learner's belief in their present understanding. While students obviously do not come in as blank slates, no matter what teachers do, getting in as early as possible seems a good idea. A conceptual model can make explicit the ways in which machine behavior differs from human thought, and it can underline how a programming language differs from natural language and familiar mathematics.

Furthermore, a conceptual model of a notional machine may serve as a basis for teaching about program tracing. The conceptual model can suggest a suitable perspective for learners to adopt as they trace programs. It may also provide a platform for teachers and learners to discuss what to keep track of while tracing and the role of external representations.

The Alternative: An Implicit Machine. By no means do all CS1 teachers explicitly teach their students about a notional machine. Not everyone agrees that it even makes sense to do so. As discussed in Section 5, strong forms of constructivism may lead to the rejection of ontology as an epistemological basis and to the rejection of the importance of explicitly teaching a normative model of the computer. Greening [1999] claims that with the employment constructivist practices such as problem-based learning, serious problems will not arise because students will discover viable understandings of the computer as they work on large, complex projects: "A constructivist environment would not find students writing the trivial code fragments needed to allow such misconceptions to escape." If this is the case, the implications for introductory programming education are significant; however, the CER literature presently appears to provide little in the way of concrete evidence supporting this argument.

8.2. Techniques for Teaching

Assuming we wish to teach explicitly about program dynamics and a notional machine, what should we do in practice?

Pedagogy for Threshold Concepts. The threshold concepts literature makes many pedagogical suggestions [Meyer and Land 2006; Land and Meyer 2008]. For instance, teachers should seek to inform students about the existence of threshold concepts and liminality, increase students' metacognition about their liminal states, and help students deal with uncertainty and the emotional issues involved. Students need to be engaged in actively and consciously manipulating each threshold concept in order to internalize it. Students should be helped to become aware of the ways in which they presently think and practice, and motivated to transform those ways. The kinds of mimicry that are motivated by a genuine attempt to cross a threshold should be seen as positive rather than negative.

To follow this advice, CS1 teachers should try to help their students become aware of the importance of the (candidate) threshold concept of program dynamics and its troublesomeness to numerous students. They should encourage students to become self-aware of how they think about programs and reason about what programs do. One way to accomplish this may be through visualizations and metaphors that concretize

the dynamic aspects of programs. By making program dynamics tangible, the teacher may not only help the student to think about programs dynamically but make the student more conscious of what they are doing. A student who is metacognitively aware of thinking about program execution as a dynamic step-by-step process may find it easier to grasp the general principles embodied in the threshold concept and to relate them to multiple contexts. Being exposed to other students' struggles with the same issue may help learners cope with the affective challenges of deep learning; peer teaching techniques and groupwork may be helpful here.

Program Visualization. A relatively popular means for making a notional machine concrete is to visualize it and its behavior. This may be accomplished by drawing on paper or blackboard, or in presentation software; another alternative is to use visualization software that is built to visualize program execution. The most familiar example of such software is the visual debugger, which provides a step-by-step trace of an input program as it is executed, making explicit the flow of control, the values of variables, frames on the call stack, and so on.

Regular visual debuggers are designed with the experienced programmer in mind. Computing educators have come up with various more learning-oriented software systems to visualize aspects of the machine to beginner programmers. Some of these systems are meant exclusively for visualizing runtime behavior related to a specific concept, such as pointers, parameter passing, objects, or assignment; other systems are more generic. A review of generic tools for visualizing program dynamics in CS1 can be found in a companion paper [Sorva et al. to appear].

The Impact of Learning Activities. Learning tasks that activate students are supported by many learning theories, currently perhaps most prominently by various forms of constructivism. The goal of such tasks is to cognitively activate students; many constructivists, for instance, argue that often this may be accomplished by requiring concrete actions on the part of the students rather than having them listen to a lecture or view a visualization that a teacher made.

Wickens and Kessel's work on the development of mental models provides an interesting perspective to the impact of learning activities on the knowledge learners construct about causal systems [Kessel and Wickens 1982; Schumacher and Czerwinski 1992]. They studied the performance of people trained alternatively as "monitors," who supervise a complex technological system, or as "controllers," who control the system manually. As one would expect, Wickens and Kessel found that training in system monitoring improves people's monitoring skills, and training in controlling a system improves controlling skills. However, and significantly, they also found that the controllers could transfer their skills to monitoring tasks, while the reverse was not true of the monitors. The controllers were also found to be better at detecting system faults from subtle cues that escaped the attention of the monitors. Their result demonstrates that people doing similar yet different tasks on the same machine develop different kinds of knowledge about it. In particular, a more passive task resulted in worse learning. It seems likely that the novice programmer who merely studies visualizations of the computer running programs is less likely to become a good "notional machinist" than the another who actively engages with the program runtime.

Program visualization for beginner programmers, like software visualization within computing education in general, has a credible track record but is not a panacea. During the past decade, the role of learning activities has surfaced as a major theme within the CER literature on visualization, particularly in algorithm visualization but also in the context of visualizing notional machines for beginners. In particular, research has suggested that the way learners engage with software visualizations is more important from a learning point of view than the visualization techniques

used [Hundhausen et al. 2002; Naps et al. 2003; Sorva et al. to appear]. Some recent visualization systems especially seek to engage learners in answering questions about programs or in controlling execution manually through a visualization. Having students draw visualizations on paper or role-play a notional machine in class are alternative pedagogies that may be useful in getting students to engage with program dynamics [Sorva et al. to appear]. Even where such activities do not explicitly feature a "machine," they may help students understand program execution within a notional machine [Andrianoff and Levine 2002; Börstler and Schulte 2005].

Variation in Critical Aspects. In the phenomenographic tradition (Section 6), the teacher's job is seen primarily as one of helping students discern critical aspects and associate them with the phenomenon. The teacher must create an appropriate "space of learning" [Marton et al. 2004]—a learning situation in which the targeted critical aspects are present, thereby providing an opportunity to learn about the phenomenon. In many cases, just spelling out critical aspects is not sufficient, however. Students should be placed in situations where they feel they need a new perspective to the phenomenon they are learning about. Marton and Booth [1997] write of building a *relevance structure* for a learning situation—defined as a person's experience of what a situation calls for—and encourage teachers to "stage situations for learning in which students meet new abstractions, principles, theories, and explanations through events that create a state of suspense." Furthermore, it is suggested that challenging phenomena may be best learned about by first focusing on individual critical aspects separately and then "fusing" aspects together to discern their relationships and gain a holistic feel for the phenomenon [Marton et al. 2004].

In the context of program dynamics, pedagogy based on phenomenography and the associated variation theory of learning is being explored by Thuné and Eckerdal [2009]. Their approach is to expose students to examples in which changes to program text are minimal but have dramatic impact on what the program does when it is run. Once the dynamic aspect of a program has been brought into focus in this way, students explore the relationship between program text and the resulting behavior using software tools (e.g., a visual debugger).

The Impact of Environments. Programming paradigms and programming environments can make a difference to learning about program dynamics. Some existing environments, and perhaps especially the programming environments of the future, blur the line between development time and program runtime—consider, for instance, the Smalltalk environment, the BlueJ IDE [Kölling 2008], the DISCOVER tutor [Ramadhan et al. 2001], and the recent work of Victor [2012]. Such environments bring many exciting benefits to both novice and expert programmers. They may also introduce some pedagogical challenges. For instance, as Ragonis and Ben-Ari [2005a, 2005b] studied high school students learning object-oriented programming, they "became aware of serious learning difficulties on program dynamics," as "students find it hard to create a general picture of the execution of a program that solves a certain problem." They suggest that object-oriented modeling and pedagogical tools that involve direct manipulation of objects while authoring a program (such as BlueJ) may exacerbate this difficulty.

The Impact of Curricular Ordering. How well students learn about the notional machine may be affected in various ways by the ordering of topics in instruction. It has been argued both on theoretical grounds [Ben-Ari 2001] and on the basis of direct empirical evidence [Mayer 1976, 1981] that it could be a better idea to help students form a viable mental model of the computer before they are taught to program in a high-level language. Delaying programming assignments may help prevent the rise of misconceptions as learners then have prior knowledge to found their understandings

of programming constructs on (see Section 5.2). However, such an approach involves a major trade-off, as students need to spend more time on fundamental concepts before they get to work with code and motivating programming projects.

We may gain another insight to topic ordering from an experimental study by Kessler and Anderson [1986] in which a group of novices learned recursive programming first, followed by iterative programming, and another group was introduced to the topics in the reverse order; neither group was explicitly taught about a notional machine (see also the similar study by Wiedenbeck [1989]). The students who started with iteration initially formed better mental models of control flow, which they were later able to transfer to the novel context of recursive programming. In contrast, the students who started with recursive programming tended toward a template-based programming style in which they tried to match the surface features of problems to the surface features of known program examples. Consequently, the recursion-first group managed to solve certain kinds of recursive problems but failed to transfer what they knew to iterative programming, becoming "overwhelmed by the surface differences between recursion and iteration." A protocol analysis suggested that the recursion-first group did not construct a model of the implicit principles of control flow underlying the example programs they saw; in other words, they had failed to understand the required notional machine. Kessler and Anderson's results highlight a pitfall in transfer from recursion to iteration, meaning that either one should start with iteration or alternatively pay particular attention to the execution model of programs as one moves from recursion to iteration. More generally, these results emphasize the importance for CS1 teachers to examine the subtle ways in which topic ordering may impact on learning about program dynamics.

9. NOTIONAL MACHINES AND OBJECT-ORIENTED PROGRAMMING

An ongoing debate among practitioners concerns whether object-oriented programming is an appropriate paradigm for beginner programmers [Bruce 2004; Lister et al. 2006]. Since the 1990s, objects-early approaches have become common, but more traditional procedural-first approaches also continue to be popular. In this section, I review what has been said regarding notional machines in the context object-oriented programming in CS1.

To follow the advice given in the previous sections, a teacher using OOP in CS1 should provide a conceptual model of a notional machine that explains the dynamic behavior of object-oriented programs at a reasonable level of abstraction. But what is an appropriate object-oriented notional machine like? Different views have been expressed in the literature.

An Extended Imperative Machine. Some have argued that a notional machine suitable for object-oriented programming is an extension of a procedural or imperative notional machine. Sajaniemi and Kuittinen [2008] compare two notional machines for object-oriented programming and procedural programming, both of which describe execution at the same level of abstraction. They argue that a notional machine for very simple imperative programs needs to feature (only) variables, I/O devices, and a program counter. It can be seamlessly expanded into a richer notional machine by adding pointers, a call stack, and mechanisms for parameter passing and returning values once students reach programs involving pointers and functions. In contrast, Sajaniemi and Kuittinen argue, any object-oriented notional machine must be more complex, featuring objects, object references, a call stack, mechanisms for parameter passing and return values, variables, I/O devices, and a program counter. According to Sajaniemi and Kuittinen, "not only is the size of the required notional machine much larger than in the procedural case, but the initial notional machine needed in order to understand

the first programs is much more complicated, as well." They draw on the literature (see Section 3) to further claim that "the OO notional machine is even more poorly understood by students than the imperative notional machine."

Schulte and Bennedsen [2006] briefly mention an object-oriented notional machine that "comprises traditional imperative aspects of the programming language as well as an understanding of the interaction among objects that take place during run-time." This phrasing also suggests an object-oriented notional machine that is an extension of an imperative one but also involves higher-level aspects.

A High-Level OO Machine. Sajaniemi and Kuittinen's object-oriented notional machine operates on the same level of abstraction as their procedural notional machine, dealing with primitives such as variables and stack frames. In contrast, Bergin [2000] emphasizes the dramatically different way computation is thought about in OOP.

> One fairly typical component of a beginning course of programming using the procedural paradigm is a discussion of the von Neumann machine architecture. . . . This simple machine model fits well with the procedural paradigm, but less well with other, more abstract, ways of looking at computation. There is a simple relationship between the physical level provided by the von Neumann architecture and the virtual level provided by most procedural languages. This is just not the case with the functional or object-oriented paradigm. The functional paradigm, of course, completely hides the underlying physical architecture. The object-oriented one does not hide it, but turns it on its head. Instead of the data being moved to the CPU for processing, a very common metaphor in OOP is that the CPU moves inside the objects.

Elsewhere, Gries [2008] criticizes the use of low-level abstractions in teaching about OOP.

> But many programming texts fail to use abstraction appropriately, e.g., by describing variables and assignment in terms of computers . . . Introducing computing concepts in terms of the computer can create unnecessary and confusing detail, especially when OO concepts are described in terms of a computer, with discussions of pointers to objects in memory, heaps, and other implementation-related terms.

Gries prefers to scrap difficult terminology (e.g., "reference," "pointer") as "with appropriate abstraction away from the computer, these terms become unnecessary." He suggests a high-level metaphorical conceptual model for program execution in terms of objects and classes.

In a similar vein, Caspersen and his colleagues have sought to represent an object-oriented notional machine on a higher level of abstraction [Caspersen 2007; Bennedsen and Schulte 2006; Henriksen 2007]. They argue that object-oriented programmers need to understand program execution in terms of a notional machine that deals with interacting object structures, and envision the use of a conceptual model that would abstract out details such as expression evaluation and concentrate on interactions. Caspersen [2007] outlines a future system in which students could "play" with the visualization of an object model, stepping forward and backward, and making changes to program state at will.

Two Machines. Berglund and Lister [2007, 2010] found through phenomenographic analysis that amongst participants in the objects-early debate, objects early is experienced in different ways: as learning an extension of imperative programming or as learning something conceptually quite distinct from imperative programming. This qualitative divide is (in my interpretation) reflected in the literature on object-oriented

notional machines. Sajaniemi and Kuittinen's OO notional machine is an example of the former perception, while Bergin represents the other line of thinking.

Most writers, perhaps for simplicity's sake, talk of a single procedural/imperative notional machine and of a single, more complex, object-oriented notional machine. Another way to think about the matter is that while a single notional machine may be enough to understand procedural programs, object-oriented programming effectively requires (at least) two different notional machines. One can be seen as an object-enabled extension of a procedural notional machine à la Sajaniemi and Kuittinen, and another describes message-passing between interacting objects. The two notional machines operate on different levels of abstraction and give two different perspectives on object-oriented programming. This is consistent with the idea that object-oriented programming is simultaneously an extension of imperative programming and something conceptually different from it. Objects-early students need to learn about both notional machines—and their relationship—early.[4] This makes an object-oriented CS1 more challenging to teach successfully—more is demanded of the teacher so as not to demand too much of the students. When you succeed, however, you have accomplished more.

10. CONCLUSIONS

Schulte and Bennedsen [2006] surveyed the opinions of programming teachers on the relative importance of various CS1 topics and made an interesting observation. They report that even though programming teachers found some specific notional-machine-related topics (e.g., references) to be relatively important compared to other specific topics, the notional machine more generally was seen as relatively unimportant compared to learning about notation and pragmatics, among other things. Schulte and Bennedsen's result may be affected by the teachers not agreeing on what exactly is meant by "notional machine"—the term was used and briefly explained in the survey—but probably also reflects the status quo in CS1 teaching, in which the big picture of how programs work at runtime does not get quite the attention it deserves in the light of learning theory and empirical evidence.

This article has brought together several perspectives on the role of notional machines in introductory programming education. Misconceptions catalogs, theories of mental models and constructivism, research on learners' ways of experiencing programming, and the theory of threshold concepts all lend support to the idea that beginner programmers need to learn about one or more notional machines. A notional machine of some sort is present within every programming-first CS1, whether it is made explicit or not: It is implied by the programming language and the paradigm used. Notional machines are not a singular bottleneck responsible for students' struggles—other challenges include learners' lack of common solution patterns, the need for syntactical precision, and motivational issues, for instance—but they do represent one of several main sources of difficulty, one that programming instructors should explicitly address as they plan their teaching.

[4]There is some limited evidence from program comprehension studies that suggests that novices who have been taught object-oriented programming have more difficulty with forming so-called "program models" (mental representations of a program text, the elementary operations it performs, control flow, etc.) whereas within the procedural paradigm novices struggle relatively more with forming "domain models" (involving, among other things, a program's goals and subgoals) [Wiedenbeck and Ramalingam 1999; Wiedenbeck et al. 1999]. A conjecture from these studies is that object-oriented novices may need additional help with the lower-level notional machine compared to procedural novices.

REFERENCES

ADELSON, B. AND SOLOWAY, E. 1985. The role of domain experience in software design. *IEEE Trans. Softw. Eng. 11,* 11, 1351–1360.

ANDRIANOFF, S. K. AND LEVINE, D. B. 2002. Role playing in an object-oriented world. *SIGCSE Bull. 34,* 1, 121–125.

BABBAGE, C. 1864. *Passages from the Life of a Philosopher*. Longman Green.

BAYMAN, P. AND MAYER, R. E. 1983. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Comm. ACM 26,* 9, 677–679.

BEN-ARI, M. 2001. Constructivism in computer science education. *J. Comput. Math. Sci. Teach. 20,* 1, 45–73.

BEN-ARI, M. AND YESHNO, T. 2006. Conceptual models of software artifacts. *Interact. Comput. 18,* 6, 1336–1350.

BENNEDSEN, J. AND SCHULTE, C. 2006. A competence model for object interaction in introductory programming. In *Proceedings of the 18th Workshop of the Psychology of Programming Interest Group (PPIG'06).* 215–229.

BERGIN, J. 2000. Why procedural is the wrong first paradigm if OOP is the goal. http://csis.pace.edu/~bergin/papers/Whynotproceduralfirst.html.

BERGLUND, A. AND LISTER, R. 2007. Debating the OO debate: where is the problem? In *Proceedings of the 7th Baltic Sea Conference on Computing Education Research (Koli Calling'07)*. Australian Computer Society, 171–174.

BERGLUND, A. AND LISTER, R. 2010. Introductory programming and the didactic triangle. In *Proceedings of the 12th Australasian Computing Education Conference (ACE'10)*. Australian Computer Society, 35–44.

BESNARD, D., GREATHEAD, D., AND BAXTER, G. 2004. When mental models go wrong: co-occurrences in dynamic, critical systems. *Int. J. Hum. Comput. Stud. 60,* 1, 117–128.

BHUIYAN, S. H., GREER, J. E., AND MCCALLA, G. I. 1990. Mental models of recursion and their use in the SCENT programming advisor. In *Proceedings of the International Conference on Knowledge Based Computer Systems (KBCS'89)*. Springer, 135–144.

BONAR, J. AND SOLOWAY, E. 1985. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Hum. Comput. Interact. 1,* 2, 133–161.

BOOTH, S. 1992. Learning to program: A phenomenographic perspective. Doctoral dissertation, University of Gothenburg.

BRUCE, C., BUCKINGHAM, L., HYND, J., MCMAHON, C., ROGGENKAMP, M., AND STOODLEY, I. 2004. Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *J. Inf. Technol. Educ. 3*, 143–160.

BRUCE, K. B. 2004. Controversy on how to teach CS 1: A discussion on the SIGCSE-members mailing list. *SIGCSE Bull. 36,* 4, 29–34.

BRUCE-LOCKHART, M. P. AND NORVELL, T. S. 2007. Developing mental models of computer programming interactively via the Web. In *Proceedings of the 37th Annual Frontiers in Education Conference (FIE'07)*. IEEE, S3H–3–S3H–8.

BÖRSTLER, J. AND SCHULTE, C. 2005. Teaching object oriented modelling with CRC cards and roleplaying games. In *Proceedings of the 8th IFIP World Conference on Computers in Education (WCCE'05).* IFIP TC-3.

CAÑAS, J. J., BAJO, M. T., AND GONZALVO, P. 1994. Mental models and computer programming. *Int. J. Hum. Comput. Stud. 40,* 5, 795–811.

CASPERSEN, M. E. 2007. Educating novices in the skills of programming. Doctoral dissertation, Department of Computer Science, University of Aarhus.

CHEN, C.-L., CHENG, S.-Y., AND LIN, J. M.-C. 2012. A study of misconceptions and missing conceptions of novice Java programmers. In *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS'12)*.

CORNEY, M., LISTER, R., AND TEAGUE, D. 2011. Early relational reasoning and the novice programmer: swapping as the "*Hello World*" of relational reasoning. In *Proceedings of the 13th Australasian Conference on Computing Education (ACE'11)*. Australian Computer Society, 95–104.

COUSIN, G. 2006. An introduction to threshold concepts. *Planet 17*, 4–5.

DE KLEER, J. AND BROWN, J. S. 1981. Mental models of physical mechanisms and their acquisition. In *Cognitive Skills and Their Acquisition*, J. R. Anderson, Ed., Lawrence Erlbaum, 285–309.

DE KLEER, J. AND BROWN, J. S. 1983. Assumptions and ambiguities in mechanistic mental models. In *Mental Models*, D. Gentner and A. L. Stevens, Eds., Lawrence Erlbaum, 155–190.

DIJKSTRA, E. W., ET AL. 1989. A debate on teaching computing science [in response to Dijkstra's On the Cruelty of Really Teaching Computing Science]. *Comm. ACM 32,* 12, 1397–1414.

Doukakis, D., Grigoriadou, M., and Tsaganou, G. 2007. Understanding the programming variable concept with animated interactive analogies. In *Proceedings of the The 8th Hellenic European Research on Computer Mathematics & Its Applications Conference (HERCMA'07)*.

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. *Interact. Comput. 9,* 1, 47–72.

Détienne, F. and Soloway, E. 1990. An empirically-derived control structure for the process of program understanding. *Int. J. Man. Mach. Stud. 33,* 3, 323–342.

du Boulay, B. 1986. Some difficulties of learning to program. *J. Educ. Comput. Res. 2,* 1, 57–73.

du Boulay, B., O'Shea, T., and Monk, J. 1981. The black box inside the glass box: presenting computing concepts to novices. *Int. J. Man. Mach. Stud. 14*, 237–249.

Eckerdal, A. and Thuné, M. 2005. Novice Java programmers' conceptions of "object" and "class", and variation theory. *SIGCSE Bull. 37,* 3, 89–93.

Eckerdal, A., Thuné, M., and Berglund, A. 2005. What does it take to learn 'programming thinking'? In *Proceedings of the International Workshop on Computing Education Research (ICER'05)*. ACM, 135–142.

Elliott Tew, A. 2010. Assessing fundamental introductory computing concept knowledge in a language independent manner. Doctoral dissertation, School of Interactive Computing, Georgia Institute of Technology.

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

Fleury, A. E. 1991. Parameter passing: the rules the students construct. *SIGCSE Bull. 23,* 1, 283–286.

Fleury, A. E. 2000. Programming in Java: student-constructed rules. *SIGCSE Bull. 32,* 1, 197–201.

Gentner, D. and Gentner, D. R. 1983. Flowing waters or teeming crowds: mental models of electricity. In *Mental Models*, D. Gentner and A. L. Stevens, Eds., Lawrence Erlbaum, 99–130.

Gentner, D. and Stevens, A. L. 1983. *Mental Models*. Lawrence Erlbaum.

George, C. E. 2000a. EROSI—visualising recursion and discovering new errors. *SIGCSE Bull. 32,* 1, 305–309.

George, C. E. 2000b. Experiences with novices: the importance of graphical representations in supporting mental models. In *Proceedings of the 12th Workshop of the Psychology of Programming Interest Group (PPIG'00)*. 33–44.

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., and Zilles, C. 2008. Identifying important and difficult concepts in introductory computing courses using a delphi process. *SIGCSE Bull. 40,* 1, 256–260.

Greening, T. 1999. Emerging constructivist forces in computer science education: shaping a new future. In *Computer Science Education in the 21st Century*, T. Greening, Ed., Springer, 47–80.

Greening, T. and Kay, J. 2001. Editorial. *Comput. Sci. Educa.* 11, 3, 189–202.

Gries, D. 2008. A principled approach to teaching OO first. *SIGCSE Bull. 40,* 1, 31–35.

Götschi, T., Sanders, I., and Galpin, V. 2003. Mental models of recursion. *SIGCSE Bull. 35,* 1, 346–350.

Henriksen, P. 2007. SIGCSE 2007 DC Application. http://www.cs.kent.ac.uk/archive/people/staff/ph53/SIGCSE2007DCApplication–PoulHenriksen.html.

Holland, S., Griffiths, R., and Woodman, M. 1997. Avoiding object misconceptions. *SIGCSE Bull. 29,* 1, 131–134.

Hristova, M., Misra, A., Rutter, M., and Mercuri, R. 2003. Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE Bull. 35,* 1, 153–156.

Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. 2002. A meta-study of algorithm visualization effectiveness. *J. Visual Lang. Comput. 13,* 3, 259–290.

Johnson-Laird, P. N. 1983. *Mental Models: Towards a Cognitive Science of Language, Inference and Consciousness*. Harvard University Press.

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., and Herman, G. L. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)*. ACM, 107–111.

Kahney, H. 1983. What do novice programmers know about recursion? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'83)*. ACM, 235–239.

Kempton, W. 1986. Two theories of home heat control. *Cognit. Sci. 10*, 75–90.

Kessel, C. J. and Wickens, C. D. 1982. The transfer of failure-detection skills between monitoring and controlling dynamic systems. *Hum. Factors 24,* 1, 49–60.

KESSLER, C. M. AND ANDERSON, J. R. 1986. Learning flow of control: recursive and iterative procedures. *Hum. Comput. Interact. 2,* 2, 135–166.

KLEIN, G. A. 1999. *Sources of Power: How People Make Decisions*. MIT Press.

KÖLLING, M. 2008. Using BlueJ to introduce programming. In *Reflections on the Teaching of Programming: Methods and Implementations*, J. Bennedsen, M. E. Caspersen, and M. Kölling, Eds., Springer, 98–115.

KUNKLE, W. M. 2010. The impact of different teaching approaches and languages on student learning of introductory programming concepts. Doctoral dissertation. Drexel University.

LAHTINEN, E., ALA-MUTKA, K., AND JÄRVINEN, H.-M. 2005. A study of the difficulties of novice programmers. *SIGCSE Bull. 37,* 3, 14–18.

LAND, R. AND MEYER, J. H. F., EDS. 2008. *Threshold Concepts within the Disciplines*. Sense Publishers.

LAROCHELLE, M., BEDNARZ, N., AND GARRISON, J., EDS. 1998. *Constructivism and Education*. Cambridge University Press.

LISTER, R., ADAMS, E. S., FITZGERALD, S., FONE, W., HAMER, J., LINDHOLM, M., MCCARTNEY, R., MOSTRÖM, J. E., SANDERS, K., SEPPÄLÄ, O., SIMON, B., AND THOMAS, L. 2004. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull. 36,* 4, 119–150.

LISTER, R., BERGLUND, A., CLEAR, T., BERGIN, J., GARVIN-DOXAS, K., HANKS, B., HITCHNER, L., LUXTON-REILLY, A., SANDERS, K., SCHULTE, C., AND WHALLEY, J. L. 2006. Research perspectives on the objects-early debate. *SIGCSE Bull. 38,* 4, 146–165.

MA, L. 2007. Investigating and improving novice programmers' mental models of programming concepts. Doctoral dissertation, Department of Computer & Information Sciences, University of Strathclyde.

MADISON, S. AND GIFFORD, J. 1997. Parameter passing: The conceptions novices construct. Res. rep. http://eric.ed.gov/PDFS/ED406211.pdf.

MARKMAN, A. B. AND GENTNER, D. 2001. Thinking. *Ann. Rev. Psychol. 52*, 223–247.

MARTON, F. 2000. The structure of awareness. In *Phenomenography*, J. A. Bowden and E. Walsh, Eds., RMIT University Press, 102–116.

MARTON, F. AND BOOTH, S. 1997. *Learning and Awareness*. Lawrence Erlbaum.

MARTON, F., RUNESSON, U., AND TSUI, A. B. M. 2004. The space of learning. In *Classroom Discourse and the Space of Learning*, F. Marton and A. B. M. Tsui, Eds., Lawrence Erlbaum, 3–40.

MAYER, R. E. 1976. Some conditions of meaningful learning for computer programming: advance organizers and subject control of frame order. *J. Educ. Psychol. 68*, 143–150.

MAYER, R. E. 1981. The psychology of how novices learn computer programming. *ACM Comput. Surv. 13,* 1, 121–141.

MENAND, L. 1997. *Pragmatism: A Reader*. Vintage.

MEYER, J. H. F. AND LAND, R. 2003. Threshold concepts and troublesome knowledge: linkages to ways of thinking and practising within the disciplines. In *Improving Student Learning—Ten Years On*, C. Rust, Ed., Oxford Centre for Staff and Learning Development.

MEYER, J. H. F. AND LAND, R., EDS. 2006. *Overcoming Barriers to Student Understanding: Threshold Concepts and Troublesome Knowledge*. Routledge.

MILLER, L. A. 1981. Natural language programming: styles, strategies, and contrasts. *IBM Syst. J. 20,* 2, 184–215.

MILNE, I. AND ROWE, G. 2002. Difficulties in learning and teaching programming—views of students and tutors. *Edu. Inf. Technol. 7,* 1, 55–66.

MORGAN, D. L. 2007. Paradigms lost and pragmatism regained. *J. Mixed Methods Res. 1,* 1, 48–76.

MURPHY, L., MCCAULEY, R., AND FITZGERALD, S. 2012. "Explain in plain english" questions: implications for teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)*. ACM, New York, 385–390.

NAPS, T. L., RÖSSLING, G., ALMSTRUM, V., DANN, W., FLEISCHER, R., HUNDHAUSEN, C., KORHONEN, A., MALMI, L., MCNALLY, M., RODGER, S., AND VELÁZQUEZ-ITURBIDE, J. Á. 2003. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bull. 35,* 2, 131–152.

NORMAN, D. A. 1983. Some observations on mental models. In *Mental Models*, D. Gentner and A. L. Stevens, Eds., Lawrence Erlbaum, 7–14.

PANE, J. F., RATANAMAHATANA, C. A., AND MYERS, B. A. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Hum. Comput. Stud. 54,* 2, 237–264.

PANG, M. F. 2003. Two faces of variation: on continuity in the phenomenographic movement. *Scand. J. Educ. Res. 47,* 2, 145–156.

PEA, R. D. 1986. Language-independent conceptual "bugs" in novice programming. *J. Educ. Comput. Res. 2,* 1, 25–36.

PERKINS, D. 2006. Constructivism and troublesome knowledge. In *Overcoming Barriers to Student Understanding: Threshold Concepts and Troublesome Knowledge*, J. H. F. Meyer and R. Land, Eds., Routledge, 33–47.

PERKINS, D. N., HANCOCK, C., HOBBS, R., MARTIN, F., AND SIMMONS, R. 1986. Conditions of learning in novice programmers. *J. Educ. Comput. Res. 2,* 1, 37–55.

PERKINS, D. N., SCHWARTZ, S., AND SIMMONS, R. 1990. Instructional strategies for the problems of novice programmers. In *Teaching and Learning Computer Programming: Multiple Research Perspectives*, R. E. Meyer, Ed., Lawrence Erlbaum, 153–178.

PERLIS, A. J. 1982. Epigrams on programming. *SIGPLAN Not. 17,* 9, 7–13.

PHILLIPS, D. C. 1995. The good, the bad, and the ugly: the many faces of constructivism. *Educ. Res. 24,* 7, 5–12.

PHILLIPS, D. C., ED. 2000. *Constructivism in Education: Opinions and Second Opinions on Controversial Issues*. National Society For The Study Of Education.

PUTNAM, R. T., SLEEMAN, D., BAXTER, J. A., AND KUSPA, L. K. 1986. A summary of misconceptions of high school BASIC programmers. *J. Educ. Comput. Res. 2,* 4, 459–72.

RAGONIS, N. AND BEN-ARI, M. 2005a. A long-term investigation of the comprehension of OOP concepts by novices. *Comput. Sci. Educ. 15,* 3, 203–221.

RAGONIS, N. AND BEN-ARI, M. 2005b. On understanding the statics and dynamics of object-oriented programs. *SIGCSE Bull. 37,* 1, 226–330.

RAMADHAN, H. A., DEEK, F., AND SHILAB, K. 2001. Incorporating software visualization in the design of intelligent diagnosis systems for user programming. *Artif. Intell. Rev. 16*, 61–84.

SAJANIEMI, J. AND KUITTINEN, M. 2008. From procedures to objects: a research agenda for the psychology of object-oriented programming in education. *Hum. Technol. 4,* 1, 75–91.

SAJANIEMI, J., KUITTINEN, M., AND TIKANSALO, T. 2008. A study of the development of students' visualizations of program state during an elementary object-oriented programming course. *J. Educ. Res. Comput. 7,* 4, 1–31.

SAMURÇAY, R. 1989. The concept of variable in programming: its meaning and use in problem-solving by novice programmers. In *Studying the Novice Programmer*, R. E. Mayer, Ed., Lawrence Erlbaum Associates, 161–178.

SAVERY, J. R. AND DUFFY, T. M. 1995. Problem based learning: an instructional model and its constructivist framework. In *Constructivist Learning Environments: Case Studies in Instructional Design*, B. Wilson, Ed., Educational Technology Publications, 135–150.

SCHULTE, C. AND BENNEDSEN, J. 2006. What do teachers teach in introductory programming? In *Proceedings of the 2nd International Workshop on Computing Education Research (ICER'06)*. ACM, 17–28.

SCHUMACHER, R. M. 1987. Acquisition of mental models. In *Proceedings of the 4th Annual Mid-Central Human Factors/Ergonomics Conference*. Springer, 142–148.

SCHUMACHER, R. M. AND CZERWINSKI, M. P. 1992. Mental models and the acquisition of expert knowledge. In *The Psychology of Expertise: Cognitive Research and Empirical AI*, R. R. Hoffman, Ed., Springer, 61–79.

SCHUMACHER, R. M. AND GENTNER, D. 1988. Transfer of training as analogical mapping. *IEEE Trans. Syst. Man Cybern. 18,* 4, 592–600.

SCHWILL, A. 1994. Fundamental ideas of computer science. *Bull. Eur. Assoc. Theor. Comput. Sci. 53*, 274–274.

SHINNERS-KENNEDY, D. 2008. The everydayness of threshold concepts: state as an example from computer science. In *Threshold Concepts within the Disciplines*, R. Land and J. H. F. Meyer, Eds., Sense Publishers, 119–128.

SHMALLO, R., RAGONIS, N., AND GINAT, D. 2012. Fuzzy OOP: expanded and reduced term interpretations. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'12)*. ACM, New York, 309–314.

SIMON. 2011. Assignment and sequence: why some students can't recognize a simple swap. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research (Koli Calling'11)*. ACM, 16–22.

SLEEMAN, D., PUTNAM, R. T., BAXTER, J., AND KUSPA, L. 1986. Pascal and high school students: a study of errors. *J. Educ. Comput. Res. 2,* 1, 5–23.

SMITH, P. A. AND WEBB, G. I. 1995. Reinforcing a generic computer model for novice programmers. In *Proceedings of the 7th Australian Society for Computer in Learning in Tertiary Education Conference (ASCILITE'95)*.

SOLOWAY, E. 1986. Learning to program = learning to construct mechanisms and explanations. *Comm. ACM 29,* 9, 850–858.

Soloway, E., Bonar, J., and Ehrlich, K. 1983. Cognitive strategies and looping constructs: an empirical study. *Comm. ACM 26,* 11, 853–860.

Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. 1982. What do novices know about programming? In *Directions in Human–Computer Interactions*, A. Badre and B. Shneiderman, Eds., Ablex Publishing, 27–54.

Sorva, J. 2007. Students' Understandings of Storing Objects. In *Proceedings of the 7th Baltic Sea Conference on Computing Education Research (Koli Calling'07)*. Australian Computer Society, 127–135.

Sorva, J. 2008. The same but different—students' understandings of primitive and Object Variables. In *Proceedings of the 8th Koli Calling International Conference on Computing Education Research (Koli Calling'08)*. 5–15.

Sorva, J. 2010. Reflections on threshold concepts in computer programming and beyond. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling'10)*. ACM, 21–30.

Sorva, J. 2012. Visual program simulation in introductory programming education. Doctoral dissertation. Department of Computer Science and Engineering, Aalto University.

Sorva, J., Karavirta, V., and Malmi, L. A review of generic program visualization systems for introductory programming education. *ACM Trans. Comput. Educ.* To appear.

Steffe, L. P. and Gale, J. E., Eds. 1995. *Constructivism in Education*. Lawrence Erlbaum.

Stoodley, I., Christie, R., and Bruce, C. 2004. Masters students' experiences of learning to program: an empirical model. In *Proceedings of the International Conference on Qualitative Research in IT & IT in Qualitative Research (QualIT'04)*.

Tashakkori, A. and Teddlie, C., Eds. 2010. *Sage Handbook of Mixed Methods in Social & Behavioral Research*, 2nd Ed., Sage.

Teague, D., Corney, M., Ahadi, A., and Lister, R. 2012. Swapping as the "Hello World" of relational reasoning: replications, reflections and extensions. In *Proceedings of the 14th Australasian Conference on Computing Education (ACE'12)*. Australian Computer Society, 87–93.

Teif, M. and Hazzan, O. 2006. Partonomy and taxonomy in object-oriented thinking: junior high school students' perceptions of object-oriented basic concepts. *SIGCSE Bull. 38,* 4, 55–60.

Thota, N., Berglund, A., and Clear, T. 2012. Illustration of paradigm pluralism in computing education research. In *Proceedings of the 14th Australasian Conference on Computing Education (ACE'12)*. Australian Computer Society, 103–112.

Thuné, M. and Eckerdal, A. 2009. Variation theory applied to students' conceptions of computer programming. *Euro. J. Eng. Educ. 34,* 4, 339–347.

Thuné, M. and Eckerdal, A. 2010. Students' conceptions of computer programming. Tech. rep. 2010-021, Department of Information Technology, Uppsala University.

Tuovinen, J. E. 2000. Optimising student cognitive load in computer education. In *Proceedings of the Australasian Conference on Computing Education (ACE'00)*. ACM, 235–241.

Vagianou, E. 2006. Program working storage: a beginner's model. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research (Koli Calling'06).* 69–76.

Vainio, V. 2006. Opiskelijoiden mentaaliset mallit ohjelmien suorituksesta ohjelmoinnin peruskurssilla. Master's thesis. Department of Psychology, University of Helsinki.

Vainio, V. and Sajaniemi, J. 2007. Factors in novice programmers' poor tracing skills. *SIGCSE Bull. 39,* 3, 236–240.

Victor, B. 2012. Inventing on principle (video). http://vimeo.com/36579366 Accessed February 2012.

Westbrook, L. 2006. Mental Models: A theoretical overview and preliminary study. *J. Inf. Sci. 32,* 6, 563–579.

Wiedenbeck, S. 1989. Learning iteration and recursion from examples. *Int. J. Man Mach. Stud. 30,* 1, 1–22.

Wiedenbeck, S. and Ramalingam, V. 1999. Novice comprehension of small programs written in the procedural and object-oriented styles. *Int. J. Hum. Comput. Stud. 51,* 1, 71–87.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., and Corritore, C. L. 1999. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interact. Comput. 11,* 3, 255–282.

Zander, C., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., and Sanders, K. 2008. Threshold concepts in computer science: a multi-national empirical investigation. In *Threshold Concepts within the Disciplines*, R. Land and J. H. F. Meyer, Eds., Sense Publishers, 105–118.

Received May 2012; revised September 2012; accepted January 2013