Aalto University publication series
DOCTORAL DISSERTATIONS 61/2012

Visual Program Simulation in Introductory Programming Education

Juha Sorva

Doctoral dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the School of Science for public examination and debate in Auditorium T2 at the Aalto University School of Science (Espoo, Finland) on the 30th of May 2012 at 12 noon.

Aalto University
School of Science
Department of Computer Science and Engineering
Learning + Technology Group (LeTech)

Supervisor
Professor Lauri Malmi

Preliminary examiners
Professor Mark Guzdial, Georgia Institute of Technology, USA
Professor Emeritus Veijo Meisalo, University of Helsinki, Finland

Opponent
Associate Professor Mordechai Ben-Ari, Weizmann Institute of Science, Israel

Aalto University publication series
DOCTORAL DISSERTATIONS 61/2012

© Juha Sorva

ISBN 978-952-60-4625-9 (printed)
ISBN 978-952-60-4626-6 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Unigrafia Oy
Helsinki 2012

Finland

The dissertation can be read at http://lib.tkk.fi/Diss/

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Juha Sorva

Name of the doctoral dissertation
Visual Program Simulation in Introductory Programming Education

Publisher School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 61/2012

Field of research Computing Education Research

Manuscript submitted 27 February 2012 **Manuscript revised** 25 April 2012

Date of the defence 30 May 2012 **Language** English

[✓ **Monograph**] ☐ **Article dissertation (summary + original articles)**

Abstract

This thesis formulates and evaluates a pedagogical technique whose goal is to help beginners learn the basics of computer programming. The technique, visual program simulation (VPS), involves the learner in interactive simulations in which the learner takes on the role of the computer as the executor of a program. The student uses a given visualization of a so-called notional machine, an abstract computer, to illustrate what happens in memory as the computer processes the program. The purpose of these simulations is to help the beginner learn to reason about program execution, a skill whose development has been identified as a major challenge in introductory programming education. VPS promotes effective learning by seeking to cognitively engage the learner with a visualization. It can be made practical through visualization software. VPS software may also automatically assess students' simulations and provide personal feedback, which is a valuable asset especially in the large classes that are typical of introductory courses.

The thesis contributes to VPS in four ways. First, it formulates the concept of visual program simulation and outlines its underpinnings in terms of learning theory. Second, it presents a new software prototype that facilitates the use of VPS in practice. Third, it reports on a preliminary empirical evaluation of VPS and the software in the context of an introductory programming course. Fourth, it makes recommendations on the use of VPS in teaching and the further development of VPS tools, which arise from the empirical work.

The findings from a mixed-methods evaluation of VPS suggest that it is a promising pedagogical approach that helps many students learn programming. At the same time, the evaluation highlights certain important weaknesses. The purpose of VPS is not obvious to many students. Care must be taken to ensure that students develop a rich understanding of what VPS is and what they stand to gain from it. For best results, it is recommended that VPS be tightly integrated into the teaching and learning environment. The results from a controlled experiment further indicate that the short-term learning benefits of a VPS assignment are heavily dependent on which interactions the assignment demands from students. This implies that extreme care must be taken in the design of VPS systems and specific assignments so that required user interactions are aligned with intended learning goals.

On a more general level, the thesis serves as an example of educational tool development that is grounded in learning theory and informed by empirical evaluations. A fairly broad review of the literature on learning and teaching introductory programming is also contributed.

Keywords visual program simulation, introductory programming education, novice programmers, program visualization, interactive visualization for education

ISBN (printed) 978-952-60-4625-9 **ISBN (pdf)** 978-952-60-4626-6

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2012

Pages 428 **The dissertation can be read at** http://lib.tkk.fi/Diss/

Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä Juha Sorva	
Väitöskirjan nimi Visuaalinen ohjelmasimulaatio ohjelmoinnin alkeisopetuksessa	
Julkaisija Perustieteiden korkeakoulu	
Yksikkö Tietotekniikan laitos	
Sarja Aalto University publication series DOCTORAL DISSERTATIONS 61/2012	
Tutkimusala Tietotekniikan opetustutkimus	
Käsikirjoituksen pvm 27.02.2012	**Korjatun käsikirjoituksen pvm** 25.04.2012
Väitöspäivä 30.05.2012	**Kieli** Englanti
☒ **Monografia**	☐ **Yhdistelmäväitöskirja (yhteenveto-osa + erillisartikkelit)**

Tiivistelmä

Visuaalinen ohjelmasimulaatio (engl. visual program simulation, VPS) on tekniikka, jolla pyritään tukemaan aloittelijoita tietokoneohjelmoinnin oppimisessa. Siinä ohjelmoinnin oppija ottaa tietokoneen roolin ohjelman suorittajana. Hän käyttää vuorovaikutteista visualisaatiota abstraktista tietokoneesta kuvatakseen, mitä tietokone tekee, kun se käsittelee ohjelman askel askelelta. Tällaiset simulaatiot voivat opettaa aloittelijaa järkeilemään ohjelmien suoritusvaiheista ja näin selviytymään eräästä ohjelmoinnin opintojen varhaisvaiheen keskeisestä haasteesta. VPS pyrkii tehostamaan oppimista edellyttämällä opiskelijalta aktiivista visualisaation käyttöä sen sijaan, että tämä vain katselisi annettua kuvamateriaalia. Toimiakseen käytännössä VPS tarvitsee tuekseen tarkoitukseen laaditun apuohjelman. VPS-järjestelmän avulla voidaan myös automaattisesti arvioida opiskelijoiden suoriutumista ja tarjota henkilökohtaista palautetta. Tämä on arvokas etu erityisesti suurilla massakursseilla, jollaisia ohjelmoinnin johdantokurssit usein ovat.

Väitöskirja edistää VPS:ää neljällä tavalla. 1) Väitöskirjassa muotoillaan visuaalisen ohjelmasimulaation käsite ja sen oppimisteoreettinen pohja. 2) Väitöskirja esittelee uuden järjestelmäprototyypin, joka mahdollistaa VPS:n käytännön opetuksessa. 3) Väitöskirjassa raportoidaan tuloksia alustavista empiirisistä tutkimuksista, joissa arvioidaan VPS:ää ja mainittua järjestelmää erään ohjelmoinnin peruskurssin yhteydessä. 4) Näihin tuloksiin nojaten väitöskirjassa esitetään suosituksia siitä, miten VPS:ää kannattaa käyttää opetuksessa sekä siitä, millaisiksi VPS-järjestelmiä tulisi jatkossa kehittää.

Tutkimustapoja yhdistelevän arvioinnin tulokset viittaavat siihen, että VPS on lupaava opetustekniikka, joka auttaa monia opiskelijoita oppimaan ohjelmointia. Toisaalta arviointi nostaa esille myös tärkeitä heikkouksia. VPS:n merkitys ei ole opiskelijoille itsestäänselvää. Opetuksessa on pidettävä huoli siitä, että opiskelijoille muodostuu rikas ymmärrys siitä, mitä VPS on ja miten he voivat siitä hyötyä. Parhaat oppimistulokset saavutettaneen integroimalla VPS tiukasti muuhun oppimiskontekstiin. Esitetyt tutkimustulokset puoltavat näkemystä, jonka mukaan VPS:n lyhyen aikavälin oppimisvaikutukset riippuvat vahvasti siitä, mitkä asiat korostuvat tehtävän opiskelijalta vaatimassa vuorovaikutuksessa. Näinollen simulointitehtäviä ja -järjestelmiä suunniteltaessa on erinomaisen huolellisesti sovitettava yhteen tehtävien oppimistavoitteet ja simulaation vaatimat toimenpiteet.

Yleisemmällä tasolla väitöskirja toimii esimerkkinä oppimistyökalun kehittämishankkeesta, joka pohjautuu oppimisteoriaan ja kokemusperäiseen tutkimukseen. Väitöskirja sisältää laajan kirjallisuuskatsauksen ohjelmoinnin alkeiden oppimisesta ja opetuksesta.

Avainsanat visuaalinen ohjelmasimulaatio, ohjelmoinnin perusopetus, ohjelmoinnin aloittelijat, ohjelmavisualisaatio, vuorovaikutteinen opetusvisualisaatio

ISBN (painettu) 978-952-60-4625-9	**ISBN (pdf)** 978-952-60-4626-6	
ISSN-L 1799-4934	**ISSN (painettu)** 1799-4934	**ISSN (pdf)** 1799-4942
Julkaisupaikka Espoo	**Painopaikka** Helsinki	**Vuosi** 2012
Sivumäärä 428	**Luettavissa verkossa osoitteessa** http://lib.tkk.fi/Diss/	

Acknowledgements

Lauri Malmi first hired me as an inexperienced programming teacher, and some years later introduced me to computing education research. This thesis results from Lauri's recognition of the need for sound cross-disciplinary research on computing and engineering education, a goal towards which he has steered his research group in recent years. Thank you, Lauri, also for allowing me the freedom to pursue my interests, while nevertheless gently guiding this seemingly ever-growing project to completion.

Teemu Sirkiä's contribution to this thesis has been invaluable. It was with Teemu that we designed the UUhistle software, which Teemu has implemented single-handedly. It has been a great pleasure to work on this project together.

Anders Berglund introduced me to phenomenographic research and has gone to great trouble to help me experience it in ever richer ways. Anders's enthusiasm for computing education research is infectious, as is his eagerness to learn about and find applications for educational theory. In many ways, what I have learned through Anders has helped me understand what it means to be a researcher.

I almost certainly would not have written a thesis on visual program simulation if it were not for a conversation that I had with Ari Korhonen and Ville Karavirta back in 2007, in which we brainstormed ideas on the use of visualization in courses on computing. It was from this seed that this thesis eventually grew.

The research that I report in this book has been conducted in collaboration with several people. Jan Lönnberg, Ville Karavirta, Kimmo Kiiski, and Teemu Koskinen contributed to parts of this work. I am indebted to Kerttu Pollari-Malmi for allowing us to use her course as a guinea pig, and to her students for participating in our research.

Otto Seppälä, Juha Helminen, Päivi Kinnunen, and Petri Ihantola provided useful references, for which I am very grateful. I further thank all the members of the Learning+Technology research group for their feedback and for stimulating discussions on program visualization, computing education, the thesis manuscript, and what not.

I am grateful to the pre-examiners, Professor Mark Guzdial and Professor Emeritus Veijo Meisalo, for taking the time to read my work and for their valuable observations and suggestions concerning it.

I would like to thank the Koli Calling research community for providing a friendly platform for presenting and discussing ideas. And for the hotel room parties.

I gratefully acknowledge the indirect influence of Professor Jorma Sajaniemi on this work. Saja's inspiring work on the roles of variables catalyzed my interest in the psychology of programming.

I thank my parents, in particular, for their help during this busy time, and, in general, for everything.

I would like to mention that my kids, Rafa and Juju, are totally awesome.

Last, and most importantly, I thank my wife Rosana for her patience, love, and support. I'm sorry for all the late nights. Thank you. Smooch!

Contents

List of Figures

List of Tables

Chapter 1

Here is How to Make Sense of This Thesis

1.1 This work is about a pedagogical technique for improving programming education

In this thesis, I formulate and evaluate a pedagogical technique whose goal is to help beginners learn the basics of computer programming. This technique, *visual program simulation* (VPS for short), involves the learner in interactive simulations in which the learner assumes the role of the computer as executor of a program. Such simulations can help the beginner learn to reason about program execution, a skill whose development has been identified as a major challenge in introductory program education.

Visual program simulation promotes effective learning by seeking to cognitively engage the learner with a visualization of computer memory. It can be made practical through visualization software. Such software may also automatically assess students' simulations and provide feedback, which is a valuable asset especially in the large classes that are typical of introductory courses.

1.2 This is computing education research on software visualization

Denning et al. (1989) defined *computing* as "the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application." In other words, computing deals with the theory and practice of computers, including computer software and hardware. The work presented in this thesis relates to computing in the dual sense that it involves the design and use of a computing application – a software system that supports visual program simulation – for the purpose of learning about computing. Within computing research, the field of *software visualization* (SV) studies the creation and use of visual representations of computer software for various purposes; the present work is an example of educationally motivated research on software visualization.

This work also has significant elements of social science: I study the relationships between people and VPS, and the effects that VPS has on people as they learn. As dissertations for the degree of "Doctor of Science in Technology" go, mine is a relatively 'soft' one – although perhaps instead of hard and soft sciences we should speak of hard and difficult sciences, respectively, as rigorous research on human thought and behavior is anything but easy (cf., e.g., Diamond, 1987). My work falls in the domain of *computing education research* (CER), the multidisciplinary field that investigates the learning and teaching of computing. In addition to being a subfield of computing and education, CER draws on the theories and methods of psychology. Figure 1.1 illustrates these relationships.

The main application of computing education research is to help computing educators develop what Shulman (1986) called *pedagogical content knowledge* – knowledge of particular content from the point of view of teaching it – and *curricular knowledge* – knowledge of the various alternative approaches and techniques for teaching about a subject. Indirectly, the beneficiaries of computing education research include students learning about computing – an increasingly varied group – and, by extension, everyone whose life is or could be affected by the products of computing, that is, nearly everybody.

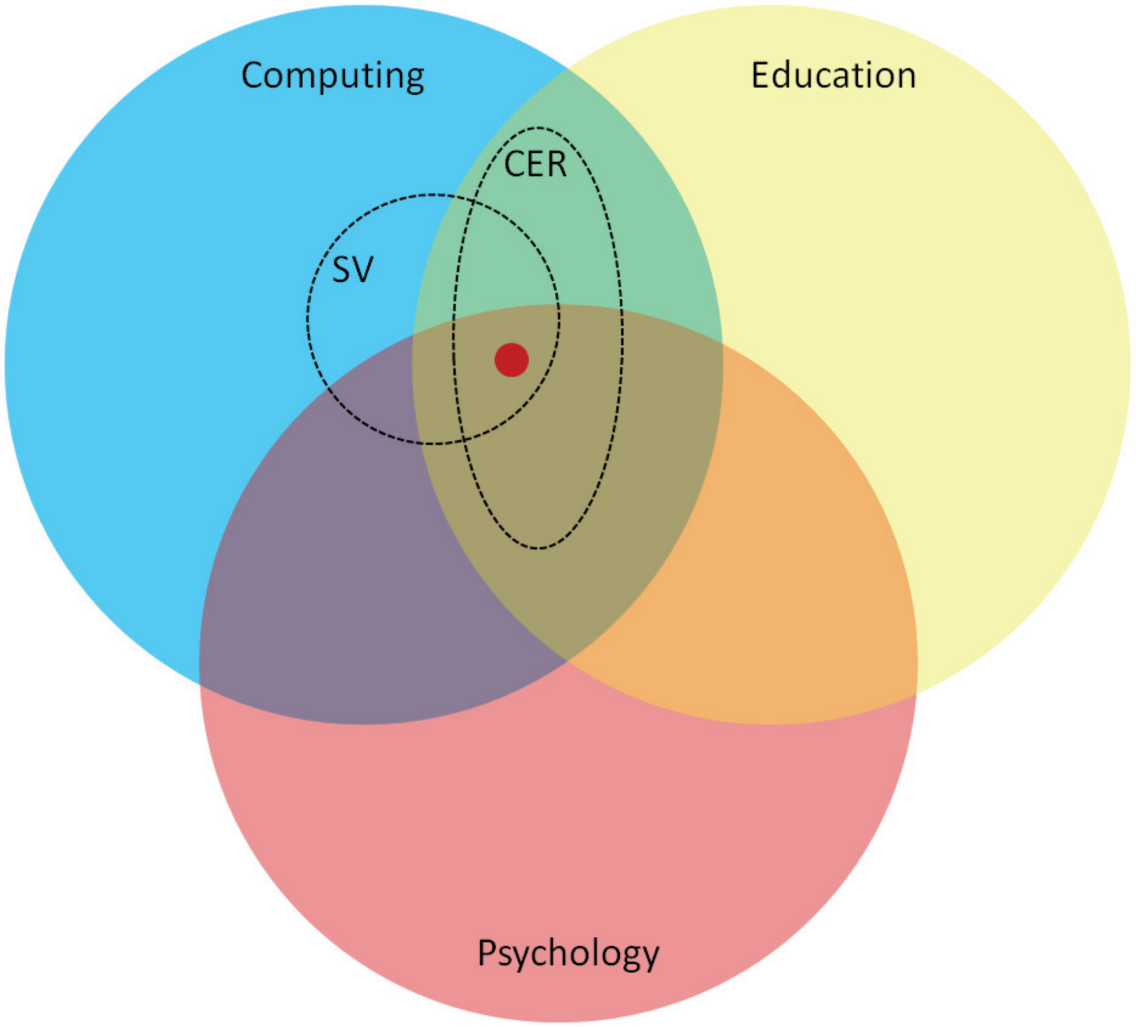


Figure 1.1: This thesis – the red dot – in the context of three disciplines of research: computing, education, and psychology. CER stands for computing education research, SV for software visualization. The size of each area is unimportant.

On research traditions

Each research tradition tends to think particularly highly of its own way of seeing and doing things. In the words of one of association football's erudite minds,

> *Everyone thinks they have the prettiest wife at home.* (Arsène Wenger, quoted by BBC Sport, 2004)

My relationship with research traditions, with theories and methods, is polyamorous. This thesis builds on multiple traditions – primarily schema theory and mental model theory in cognitive psychology, the phenomenographic tradition within educational research, and the software visualization tradition. Further influences come from the constructivist paradigm of education and educational research on threshold concepts. The influence of different traditions is reflected in the theoretical groundwork of visual program simulation, as well as in my empirical work, which I undertake from a pragmatic mixed-methods perspective.

1.3 This thesis consists of six main parts

The first three parts of this dissertation constitute a literature survey on the learning and teaching of programming. I review what has been written, and comment on it, in an attempt to synthesize a coherent picture of pertinent aspects of introductory programming education. Part I, *The Challenge of Introductory Programming Education*, establishes that there is a problem: introductory programming courses are not working nearly as well as educators – and students – would like. Part II, *Learning Introductory Programming*, considers learning to program from the perspective of several general theories of learning as well as prior research within CER. Part III, *Teaching Introductory Programming*, reviews approaches to teaching introductory programming courses and software visualization tools designed to help with this task.

Part IV, *Introducing Visual Program Simulation*, presents visual program simulation as a pedagogical technique, and our particular implementation of it in a software system. Part V, *Empirical Investigations of Visual Program Simulation*, moves us from constructive research to evaluative empirical research as I report the results from a set of interrelated studies in which my colleagues and I explored the use of VPS in the context of an introductory programming course. Part VI, *Conclusions*, looks back at what has been achieved, and forward to future work.

Each of these parts is prefaced with an introduction that provides an overview of the upcoming chapters.

Part VII is just for the end bits: appendices and bibliography.

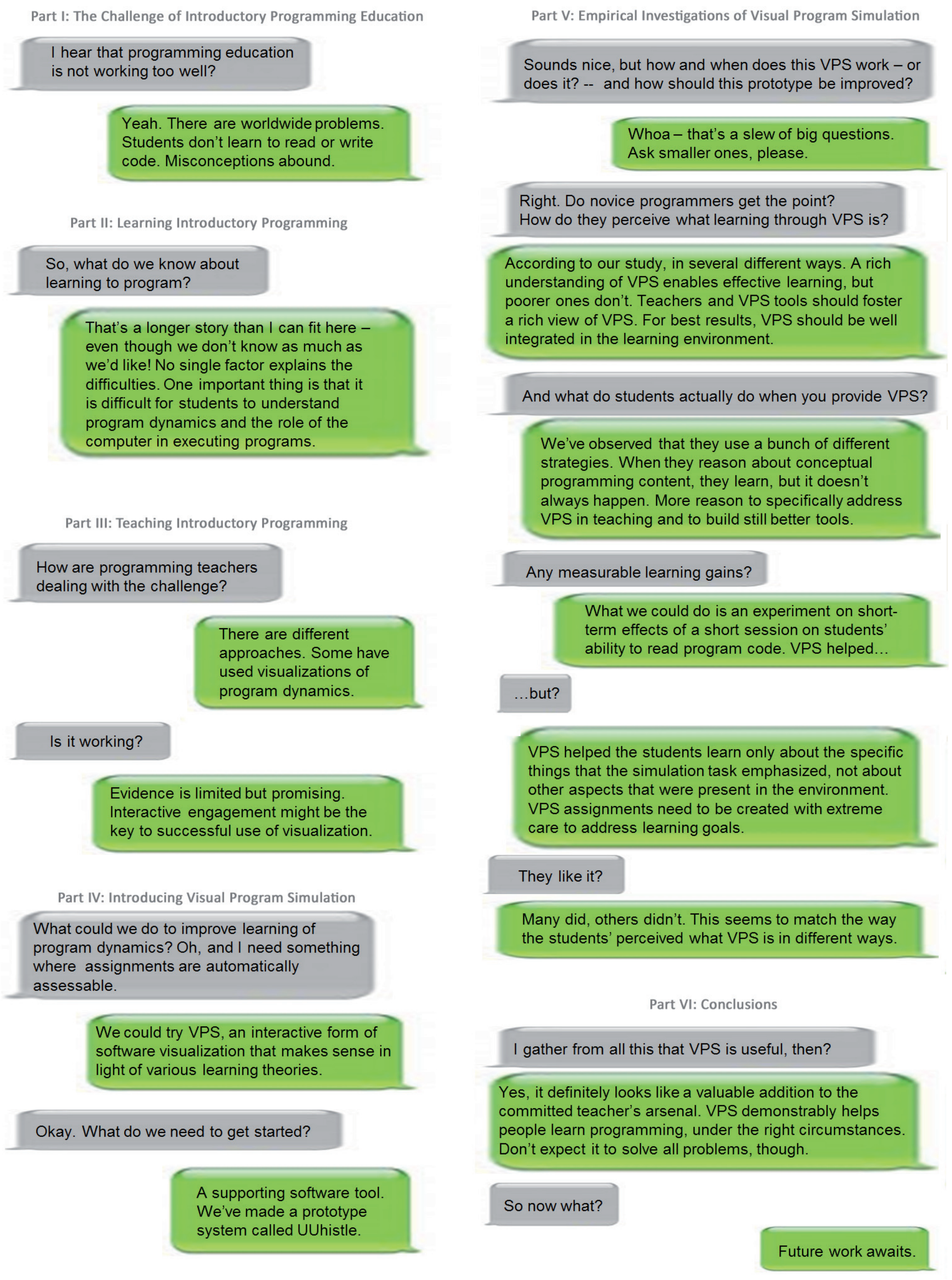
1.3.1 The thesis is driven by a sequence of questions and answers

The present work can be expressed as a sequence of questions and answers. The first chapters chart the terrain by putting questions to the existing literature. The answers provoke more questions, which produce more answers. Eventually, the answers motivate the formulation of visual program simulation in Part IV, and lead to the research questions posed to empirical data in Part V. Figure 1.2 summarizes some of the main contents of this book from a question-setting perspective.

For a more traditional exposition of my research questions for empirical research, see Chapter 16.

1.3.2 There are different ways of reading the thesis

I expect that most readers of this thesis will have some expertise in computing. Throughout this work, I refer to fundamental programming concepts, common programming languages, and programming paradigms under the assumption that they will be familiar to the reader. However, I do cover in some detail even the well-known theories from education and psychology that I build on. This is to help readers that do not have a background in these other fields. The lengthier treatment of educational and psychological theory also reflects my own learning process. Writing about these topics has been a device for learning as I have approached these other disciplines during my postgraduate studies in computing.



Part I: The Challenge of Introductory Programming Education

I hear that programming education is not working too well?

Yeah. There are worldwide problems. Students don't learn to read or write code. Misconceptions abound.

Part II: Learning Introductory Programming

So, what do we know about learning to program?

That's a longer story than I can fit here – even though we don't know as much as we'd like! No single factor explains the difficulties. One important thing is that it is difficult for students to understand program dynamics and the role of the computer in executing programs.

Part III: Teaching Introductory Programming

How are programming teachers dealing with the challenge?

There are different approaches. Some have used visualizations of program dynamics.

Is it working?

Evidence is limited but promising. Interactive engagement might be the key to successful use of visualization.

Part IV: Introducing Visual Program Simulation

What could we do to improve learning of program dynamics? Oh, and I need something where assignments are automatically assessable.

We could try VPS, an interactive form of software visualization that makes sense in light of various learning theories.

Okay. What do we need to get started?

A supporting software tool. We've made a prototype system called UUhistle.

Part V: Empirical Investigations of Visual Program Simulation

Sounds nice, but how and when does this VPS work – or does it? -- and how should this prototype be improved?

Whoa – that's a slew of big questions. Ask smaller ones, please.

Right. Do novice programmers get the point? How do they perceive what learning through VPS is?

According to our study, in several different ways. A rich understanding of VPS enables effective learning, but poorer ones don't. Teachers and VPS tools should foster a rich view of VPS. For best results, VPS should be well integrated in the learning environment.

And what do students actually do when you provide VPS?

We've observed that they use a bunch of different strategies. When they reason about conceptual programming content, they learn, but it doesn't always happen. More reason to specifically address VPS in teaching and to build still better tools.

Any measurable learning gains?

What we could do is an experiment on short-term effects of a short session on students' ability to read program code. VPS helped…

…but?

VPS helped the students learn only about the specific things that the simulation task emphasized, not about other aspects that were present in the environment. VPS assignments need to be created with extreme care to address learning goals.

They like it?

Many did, others didn't. This seems to match the way the students' perceived what VPS is in different ways.

Part VI: Conclusions

I gather from all this that VPS is useful, then?

Yes, it definitely looks like a valuable addition to the committed teacher's arsenal. VPS demonstrably helps people learn programming, under the right circumstances. Don't expect it to solve all problems, though.

So now what?

Future work awaits.

Figure 1.2: This thesis as dialogue.

The thesis should ideally be read from beginning to end. There are threads of thought that are gradually developed throughout the book. However, readers with different backgrounds and goals may wish to do something different.

Readers with a strong background in computing education research may wish to skip what is familiar in Parts I to III, and focus on Parts IV to VI, which deal directly with visual program simulation.

The reader in a hurry to find out something about VPS may wish to look at just the first two chapters (12 and 13) of Part IV, and the conclusions in Chapter 21.

The table of contents is designed to read like an extended abstract of sorts; some readers may find this useful for obtaining a quick overview.

Or, if you are one of my relatives with a limited interest in computing education, or otherwise stuck at my doctoral defense with a copy of this book in front of you, you may prefer to try the surface approach provided by the crossword in Figure 1.3. (This one might take a while to complete.)

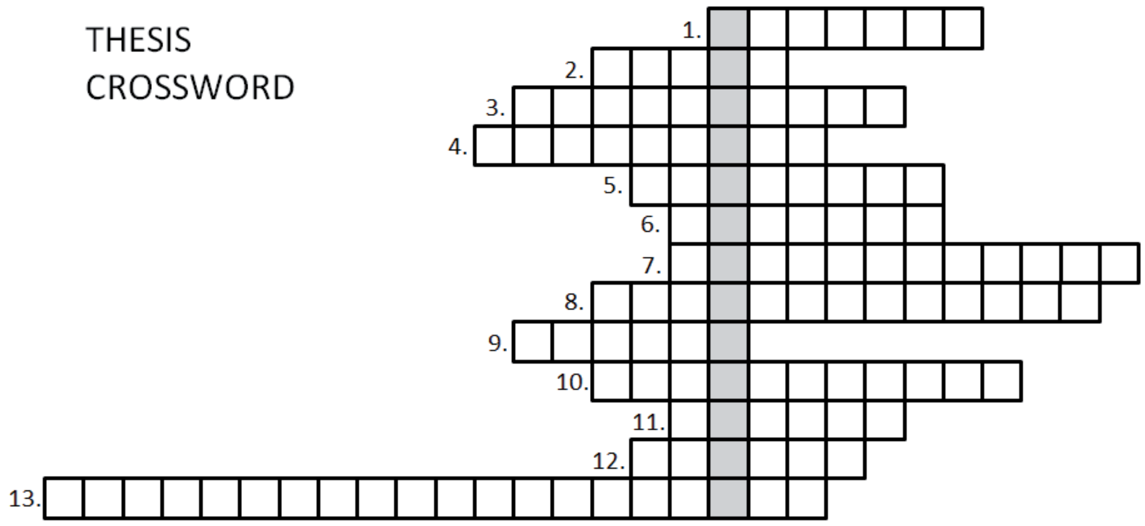


Figure 1.3: 1. The most common word of five or more letters in this thesis.
2. The first name of the supervisor of the thesis.
3. The key to the success of educational visualization?
4. A type of cognitive load.
5. Describing the machine that students control in visual program simulation.
6. The most common surname within the list of references.
7. An abstraction of validity and credibility.
8. "Assignment moves a value from a variable to another", for one.
9. Describing the kind of mental model suitable for fixing problems.
10. What threshold concepts tend to be for learners.
11. The more abstract of UUhistle's parents.
12. Any word with A as its third letter might be considered to be, as an answer to this question?
13. The longest unhyphenated English word in this thesis.
Down: The kind of understanding of this thesis that this crossword is likely to engender.

Part I

The Challenge of Introductory Programming Education

Introduction to Part I

Introductory programming courses around the world are failing to teach students how to program. While many students certainly do succeed, too many others fall far short of the goals set by curriculum planners and teachers. Evidence from computing education research shows that the problem is significant and not just local to a few institutions.

Programming is an activity that is central to the field of computing. This is evidenced in computing education by the dominance of programming-first approaches to teaching introductory computing courses (ACM and IEEE Computer Society, 2001). The primary goal of a typical introductory computing course – commonly called a *CS1 course* or simply *CS1* – is that students learn to create programs in some programming language. Unfortunately, it turns out that this is a demanding goal for an introductory course, and one that is often not met. In Part I, I try to substantiate these claims.

Part I consists of two chapters. Chapter 2 examines the goals of programming education from the viewpoint of two influential educational taxonomies, Bloom's taxonomy and SOLO. Chapter 3 is about the unwelcome evidence: many students do not acquire even rudimentary programming skills in CS1, and the problem is widespread. These chapters set the scene for a more detailed look at what it takes to learn to program, which will follow in Part II.

Chapter 2

Introductory Programming Courses are Demanding

This chapter takes a look at the goals of introductory programming education in the light of two educational taxonomies: Bloom's taxonomy and the SOLO taxonomy. These taxonomies enable us to characterize programming tasks in terms of their cognitive and structural complexity, which gives us an idea of what is commonly expected of beginner programmers.

In Section 2.1 below, I introduce Bloom's taxonomy and its applications to programming. Section 2.2 deals with SOLO.

2.1 Bloom's taxonomy sorts learning objectives by cognitive complexity

Some learning objectives are harder to achieve than others. It is much harder to learn to evaluate the quality of computer programs than to list programming keywords, for instance. In the 1950s, a group of educators led by Benjamin Bloom defined a taxonomy that divides learning into three domains: the cognitive, affective, and psychomotor (Bloom, 1956). In the same work, they further detailed the cognitive branch of the taxonomy by presenting a hierarchy of learning objectives ranked according to their expected cognitive complexity (see Figure 2.1). Bloom's work has received wide acclaim and remains highly influential.

The name "Bloom's taxonomy" is used in two different ways. It sometimes refers to the overall taxonomy, with cognitive, affective, and psychomotor branches. It is also commonly used to refer to the taxonomy of learning objectives within the cognitive domain. In this thesis, "Bloom's taxonomy" is shorthand for "Bloom's taxonomy of learning objectives for the cognitive domain".

2.1.1 The taxonomy distinguishes between six types of cognitive process

The six levels of the original 1956 taxonomy, from lowest to highest, are:

1. *Knowledge*: the student can recall specific facts or methods. This level is characterized by verbs such as enumerate, name, and define;

2. *Comprehension*: the student understands the meaning of facts or concepts. This level is characterized by verbs such as explain, discuss, and paraphrase;

3. *Application*: the student can solve problems by applying knowledge to new concrete situations. This level is characterized by verbs such as produce, implement, and solve;

4. *Analysis*: the student can break down information into its parts to determine motives or causes, or to make inferences. This level is characterized by verbs such as analyze, discriminate, and infer;

5. *Synthesis*: the student can combine elements in new ways to produce novel wholes. This level is characterized by verbs such as create, compose, and invent;

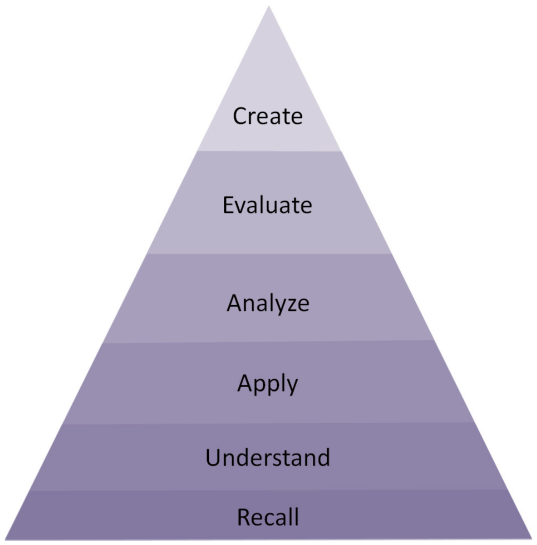


Figure 2.1: Bloom's taxonomy of learning objectives for the cognitive domain (the level names shown are from the revised taxonomy by Anderson et al., 2001).

6. Evaluation: the student can make judgments about material in light of selected criteria. This level is characterized by verbs such as appraise, critique, and compare.

Many variants of the taxonomy have been proposed in the literature. An influential variant – which I will refer to as the *revised Bloom's taxonomy* – was defined by an interdisciplinary group of experts led by Anderson and Krathwohl (Anderson et al., 2001). The revised Bloom's taxonomy has two dimensions – a cognitive process dimension similar to that of the original taxonomy, and a knowledge dimension that specifies the type of content being processed – factual, conceptual, procedural, or metacognitive. Anderson et al. also exchanged the places of the two last levels of the cognitive process dimension to produce a revised hierarchy: recall, understand, apply, analyze, evaluate, and create, as shown in Figure 2.1.

2.1.2 Applying Bloom to programming is tricky but increasingly popular

In comparison to its esteemed status in other subfields of education, Bloom's taxonomy had received relatively little attention in programming education until recent years. However, in the past decade or so, a growing body of work has emerged that relates the taxonomy – either the original or the revised version – to introductory programming. Many programming educators have reported on how they have used Bloom's taxonomy to motivate improvements to the instruction or assessment of programming courses (e.g., Buck and Stucki, 2000; Lister and Leaney, 2003; Scott, 2003; Thompson et al., 2008; Starr et al., 2008; Khairuddin and Hashim, 2008; Alaoutinen and Smolander, 2010). The recent CER literature also features a thread that applies Bloom's taxonomy to study the learning of programming and discuss the appropriateness of forms of assessment in CS1 (e.g., Johnson and Fuller, 2006; Fuller et al., 2007; Whalley et al., 2006, 2007; Meerbaum-Salant et al., 2010). The revised Bloom's taxonomy is being used as a guideline by ACM and IEEE's work on developing their computer science curriculum (ACM and IEEE Computer Society, 2008). Several variants of Bloom's taxonomy have been proposed to be particularly suitable for programming education (Shneider and Gladkikh, 2006; Fuller et al., 2007; Bower, 2008).

No general consensus has emerged on precisely how to map the goals of programming education onto Bloom's taxonomy. Code-tracing skills, for instance, have been variously classified within the literature as understand or analyze, and many interpretations have been presented as to how to 'Bloom rate' program-writing assignments of different kinds. Gluga et al. (2011) found that academics untrained in use of Bloom's taxonomy for the classification of programming assignments in a particular way produced a variety of different classifications.

There is some convergence of opinion, too. The ability to create a program to solve an unfamiliar problem, the literature widely agrees, belongs at the *synthesis* level in the original taxonomy, or *create* in the revised version. Scholars within CER, as well as Bloom's original group, have stressed the relevance of students' prior knowledge in determining the cognitive demands of an activity (e.g., Johnson and Fuller, 2006; Thompson et al., 2008; Gluga et al., 2011). For instance, Thompson et al. (2008) consider that *applying* programming knowledge involves solving familiar problems with new data or solving unfamiliar problems that match a familiar pattern or require an algorithm that is known to the student. To come up with a previously unknown kind of solution is to *create*.

Despite these challenges of interpretation, Bloom's taxonomy can tell us something about CS1 courses:

2.1.3 The goals of introductory programming courses are cognitively challenging

Bloom's taxonomy is intended to be used by teachers as a tool for analyzing and designing courses and curricula. In particular, the taxonomy was created to emphasize that learning objectives should not be set only at the lowest levels, as was being done in many traditional educational settings, but at all levels of the taxonomy. David Krathwohl, a member of both Bloom's original group and the one that revised it decades later, looks back on how Bloom's taxonomy has been applied across disciplines:

> *One of the most frequent uses of the original Taxonomy has been to classify curricular objectives and test items in order to show the breadth, or lack of breadth, of the objectives and items across the spectrum of categories. Almost always, these analyses have shown a heavy emphasis on objectives requiring only recognition or recall of information, objectives that fall in the Knowledge category. But it is objectives* [. . .] *in the categories from Comprehension to Synthesis that are usually considered the most important goals of education. Such analyses, therefore, have repeatedly provided a basis for moving curricula and tests toward objectives that would be classified in the more complex categories.* (Krathwohl, 2002)

It is interesting to observe that in the field of computer programming, applying Bloom's taxonomy to course evaluation has not precipitated the kind of shift that Krathwohl describes, from *knowledge* towards more complex educational goals. In fact, what effect there has been, has been largely in the opposite direction. Applying Bloom's taxonomy to introductory programming education has highlighted the fact that even introductory courses set the 'cognitive bar' very high for would-be programmers.

Lister and Leaney (2003) note that the traditional problem-solving goal of a CS1 course – to be capable of developing a (small) program which solves a given problem that has been expressed vaguely in non-programming terms – corresponds to *synthesis* (or *create*) high up in the taxonomy. Typical introductory programming exams emphasize writing code, that is, *application* and/or *synthesis*, depending on the question and the interpretation of Bloom (Scott, 2003; Petersen et al., 2011; Simon et al., 2012).

Oliver et al. (2004) examined computing courses in terms of their 'Bloom rating' (a weighted average of the numbered Bloom levels of each type of assessment). They discovered that programming courses, including introductory ones, have high Bloom ratings, whereas courses on other computer science topics had much lower ratings.[1]

The findings of Whalley et al. (2006) suggest that the higher up in the revised Bloom's taxonomy a programming task is, the more difficult it is for students succeed in it – as Bloom's group hypothesized more generally.

These Bloom-driven analyses show the goals of a typical introductory programming course to be quite demanding. Another influential taxonomy, SOLO, lends weight to this conclusion.

[1]Although Oliver et al. (2004) analyzed only computing courses, one suspects that typical introductory courses in many non-computing subjects also tend to have significantly lower 'Bloom ratings' than introductory programming courses.

2.2 The SOLO taxonomy sorts learning outcomes by structural complexity

Some educators have questioned the appropriateness of Bloom's taxonomy for the design of learning activities and assessments.

> *When using Bloom's taxonomy, the supposition is that the question leads to the particular type of Bloom response. There is no necessary relationship, however, as a student may respond with a very deep response to the supposedly lower order question: "Describe the subject matter of Guernica?" Similarly, a student may provide a very surface response to "What is your opinion of Picasso's Guernica?"* (Hattie and Purdie, 1998, p. 161)

One solution to such mismatches between activity and outcome is to categorize outcomes. This is the focus of the *Structure of the Observed Learning Outcome*, or *SOLO*, a taxonomy formulated by Biggs and Collis (1982) from empirical analyses of students' responses to learning tasks.

2.2.1 SOLO charts a learning path from disjointed to increasingly integrated knowledge

SOLO's five levels can be used to categorize learner responses in terms of their structural complexity. Paraphrased from Biggs and Tang (2007, pp. 77-78), the levels of SOLO are:

1. Prestructural: a response at this level misses the point or consists of empty phrases, which may be elaborate but show little evidence of actual learning;

2. Unistructural: this kind of response meets only a single part of a given task or answers only one aspect of question. It misses other important attributes entirely;

3. Multistructural: the response is 'a bunch of facts'. It expresses knowledge of various important aspects, but does not connect them except possibly on a surface level. The learner sees 'the trees' but not 'the forest';

4. Relational: the response relates and integrates facts into a larger whole that has a meaning of its own. It is no longer a list of details; rather, facts are used by the learner to make a point;

5. Extended abstract: a response at this level goes beyond what is given and applies it to a broader domain.

SOLO describes a systematic progression in performance as an individual learns. First, from the prestructural through to the multistructural level, the learner makes quantitative progress, increasing the amount of knowledge they have. Progressing to the relational and extended abstract levels involves a qualitative change as meaning emerges from the increasingly well perceived connections between elements of knowledge (Figure 2.2).

SOLO is intended to be used by teachers both for analyzing responses to learning activities (e.g., answers to questions) and for setting learning objectives. SOLO and Bloom's taxonomy have somewhat different perspectives – Bloom classifies learning objectives (skills), while SOLO classifies learning outcomes (responses to activities) – and they are, to a certain extent, complementary. Both can be used to characterize the objectives set for learners.

2.2.2 SOLO has been used to analyze programming assignments

A research project called BRACElet has recently applied SOLO to code-reading and code-writing tasks in the context of introductory programming (Lister et al., 2006b; Sheard et al., 2008; Clear et al., 2009; Lister et al., 2009a; Whalley et al., 2011). They present an interpretation of how SOLO applies to simple code comprehension problems of the form "in plain English, explain what the following segment of Java code does", and characterize the four main levels as follows (Lister et al., 2009a):

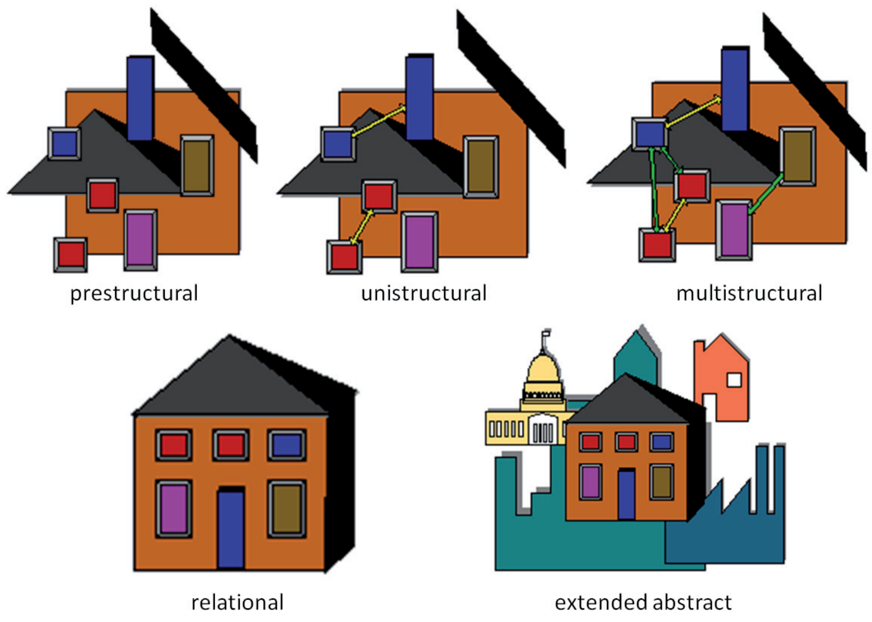


Figure 2.2: Atherton's (n.d.) metaphorical illustration of SOLO's five levels. At the unistructural and multistructural levels, links between previously unconnected pieces are increasingly perceived. A qualitative change happens at the relational level, at which point the whole becomes genuinely meaningful.

1. Prestructural: substantially lacks knowledge of programming constructs or is unrelated to the question;
2. Unistructural: a description of one part of the code;
3. Multistructural: a line-by-line description of all the code (the 'trees');
4. Relational: a summary of what the code does in terms of its purpose (the 'forest').

The results of Sheard et al. (2008) suggest that the degree of structuredness of students' responses to a code-reading task measured on such a SOLO-based scale correlates significantly and positively with their ability to write program code.

As for writing code, one suggestion for categorizing responses was sketched out by Clear et al. (2009); I paraphrase:

1. Prestructural: inability to write correct code;
2. Unistructural: ability to write a single small piece of code, e.g., to increment the value of a variable.
3. Multistructural: ability to write combine a few statements to write a multi-line solution based on a detailed specification or pseudocode. E.g., completing a method so that it returns false if a given book is on loan but true otherwise;
4. Relational: ability to write code to solve a problem which has not been specified to the extent that the problem represents pseudocode for the solution. E.g., writing a class to represent library books.

Like Bloom's taxonomy, SOLO provides us with a lens through which to examine the goals of introductory programming education.

2.2.3 The expected outcomes of programming courses are structurally complex

In terms of the SOLO taxonomy, successfully writing programs requires an understanding of programs that reaches the relational level. Program design tasks may even require students to transfer programming concepts beyond what they have encountered or learned about and to the extended abstract level. In a vein of work similar to the 'Bloom rating' measurements of Oliver et al. (Section 2.1.3 above), Brabrand and Dahl (2009) analyzed the stated requirements of the computer science, mathematics, and natural science courses of a Danish university where all teachers are required to use the SOLO taxonomy as they specify course goals. They found that computer science courses in general had significantly higher SOLO levels than natural science courses and (even more clearly) than mathematics courses. Typical programming-related competencies desired were relational – at a high level.

Both Bloom's taxonomy and SOLO illuminate the challenge of introductory programming education: the learning objectives are cognitively challenging, the expected learning outcomes structurally complex. The emphasis on synthesis skills and relational knowledge in introductory programming education is not surprising in light of the history of programming education. Early programming courses introduced programming as a tool for practitioners of other sciences rather than as a facet of computer science (ACM and IEEE Computer Society, 2001, p. 22). The goal was that these practitioners – a different demographic group than today's CS1 students – would take perhaps only a single course in programming and would then be able to apply what they learned to difficult problems within their main field.

There is no question that programming continues to be a key skill within computing, and is an ever more important tool for non-computer-scientists. Developing learners' ability to create novel programs will continue to be the central goal of programming education. Nevertheless, analyses of the cognitive demands of introductory programming courses have led researchers to ring a warning bell (Whalley et al., 2006):

> *It appears likely that programming educators may be systemically underestimating the cognitive difficulty in their instruments for assessing programming skills of novice programmers. For non-elite institutions it is likely that some proportion of the high failure rate in introductory programming may be attributed to this difficulty in setting fair and appropriate assessment instruments.* [...] *The level of difficulty of programming assessments at introductory levels, whether or not inherent in the subject itself, presents a significant and possibly unfair barrier to student success.*

Chapter 3

Students Worldwide Do Not Learn to Program

The previous chapter made the theoretical observation that programming courses have demanding goals. This is borne out by concrete evidence: poor results in introductory programming education have been widely reported. Section 3.1 below reviews research on learning outcomes, which indicates that many students are not learning to write programs in CS1. Section 3.2 considers the relationships between code-reading skill and code-writing skill, but sadly, it turns out in Section 3.3 that many students do not even learn to read program code reliably in CS1. Finally in Section 3.4, I review research on the various specific problems that novice programmers have with understanding fundamental programming concepts.

3.1 Many students do not learn to write working programs

> *Few teachers of programming in higher education would claim that all their students reach a reasonable standard of competence by graduation. Indeed, most would confess that an alarmingly large proportion of graduates are unable to 'program' in any meaningful sense.* (Carter and Jenkins, 1999)

This is not recent news. According to the review of CER literature from the 1980s by Robins et al. (2003), "an observation that recurs with depressing regularity, both anecdotally and in the literature, is that the average student does not make much progress in an introductory programming course". Linn and Dalbey (1985) report that most students struggled to get past learning language features and never got to learning about higher-order skills of program planning and general problem-solving strategies for programming. Kurland et al. (1986) concluded that after two years of programming instruction, many high-school students had only a rudimentary understanding of programming. Guzdial reviews the work of Soloway and others:

> *One of the first efforts to measure performance in CS1 was in a series of studies by Elliot Soloway and his colleagues at Yale University. They regularly used the same problem, called "The Rainfall Problem": Write a program that repeatedly reads in positive integers, until it reads the integer 99999. After seeing 99999, it should print out the average. In one study, only 14% of students in Yale's CS1 could solve this problem correctly. The Rainfall Problem has been used under test conditions and as a take-home programming assignment, and is typically graded so that syntax errors don't count, though adding a negative value or 99999 into the total is an automatic zero. Every study that I've seen (the latest in 2009) that has used the Rainfall Problem has found similar dismal performance, on a problem that seems amazingly simple.* (Guzdial, 2011, see also Soloway et al., 1982; Venables et al., 2009)

A common topic of conversation among computing education researchers is the 'Bactrian' grade distribution of many introductory programming courses: students either fail miserably or pass with flying colors, with few 'just doing okay' (Dehnadi and Bornat, 2006). Many explanations have been offered and many studies have been conducted, but the phenomenon remains unexplained (see, e.g., Bornat et al., 2008; Robins, 2010).

Multi-institutional studies

In the past decade, several international working groups have looked into the skill levels of students at the end of CS1 courses, or, in some cases, at the end of a degree program (McCracken et al., 2001; Lister et al., 2004; Eckerdal et al., 2006b). These oft-cited studies have been influential as they produced concrete evidence of the mismatch between the goals of programming education and the actual skills gained.

In 2001, a multi-institutional, multi-national working group chaired by Michael McCracken gave a set of program-writing problems of varying difficulty to students completing CS1 or CS2 in several countries. The students only got an average score of approximately 23 out of 110 points. The authors state that their "first and most significant result was that the students did much more poorly than we expected" and that "the disappointing results suggest that many students do not know how to program at the conclusion of their introductory courses" (McCracken et al., 2001).

Another international working group study explored program design skills. When Eckerdal et al. (2006b) analyzed the designs produced by students, they found "poor performance from students who are near graduation: over 20% produced nothing, and over 60% communicated no significant progress toward a design". They conclude that "the majority of graduating students cannot design a software system". A recent follow-up study by Loftus et al. (2011) produced similar results.

It is clear by now that learning to write programs is a challenging goal. Teachers aware of this have tried to find ways of easing the burden of students as they gradually develop code-writing ability. The question then becomes: what are the relationships between the various goals of programming education? Or more specifically: what other skills does the skill of writing code build on? There is some limited evidence of dependencies between programming skills, so that learning certain skills builds on learning others first.

3.2 Code-writing skill is (loosely?) related to code-reading skill

The ability to write program code is what we aim to teach, so anything else that we can discover about students' acquisition of skills must ultimately be considered in the light of their ability to write code. (Lister et al., 2009a)

Many programming teachers find it intuitive, even self-evident, that one must learn to read code before one can write code, just as one learns to read their first natural language before learning to write in it. Similarly, the ability to trace a program's execution steps would seem to be a prerequisite both for explaining what given code accomplishes and for writing code. However, students of programming tend to give examples and reading tasks little attention compared to writing; some also say right out that writing code is easier than reading (Simon et al., 2009).

Does learning to explain what a piece of code does precede learning to write code? Can people learn to write code successfully without being able to trace its execution (and code of what kind)? What comes before the ability to explain code? More generally, is there evidence of a general learning path of programming skills?

From taxonomy to learning path?

Bloom's taxonomy (Section 2.1) is not of much assistance in the search for a general learning path. Even if we assume that the cognitive categories form a hierarchy of increasingly complex learning objectives and assessments, there is no evidence in the general case that learning a higher-ranking skill requires the lower-ranking skills to be learned first. It has been argued that Bloom's taxonomy does not match the path(s) of skill progression that learners take, either in general or for programming in particular (e.g., Fuller et al., 2007; Anderson et al., 2001; Eckerdal et al., 2007; Biggs and Tang, 2007).

Other frameworks may match the learning process better. A substantial body of empirical research that seeks to discover how programmers' skills develop has recently been produced by the BRACElet project (for an overview, see Clear et al., 2011). This empirical work builds on SOLO (Section 2.2), a taxonomy that was designed to reflect stages of learning.

BRACElet: some evidence of skill dependencies

In terms of one interpretation of SOLO (Section 2.2), code-tracing skills (that is, the ability to step through a program's execution) rank below code-explaining skills (that is, the ability to determine and state the overall purpose of a piece of code), as the former requires multistructural learning outcomes while the latter requires relational ones. If SOLO levels represent a learning path, students would need to learn to trace code of a particular kind and complexity before they learn to explain code of a similar kind and complexity. The BRACElet members hypothesized that learners generally first learn to trace programs, then to explain them, and finally to write them.

A number of BRACElet publications have produced empirical evidence that links the skills of tracing, explaining, and writing code. Students who write programs successfully tend to be able to produce correct overall explanations of what a given piece of code does (Whalley et al., 2006). Students' abilities to explain and write correlate positively (Sheard et al., 2008). Students who cannot trace code are usually also unable to explain what a given piece of code does (Philpott et al., 2007). Summarizing what given code does appears to be an intermediate-level skill, which programming experts use naturally in lieu of line-by-line traces of programs, but which many novices struggle with (Lister et al., 2006b).

Lopez et al. (2008) report that performance level in a code-tracing task accounts for some of the variation in code-explaining performance. Further, they found that code-explaining ability alone, or code-tracing ability alone, appears to account for only a little of the variation in code-writing skills. However, explaining and tracing abilities together account for a substantial amount of the variation in writing ability. They conclude that if one posits that a causal model exists between the skills, then their findings support a hierarchy where tracing is lower than explaining, which is again lower than writing. The later results of Lister et al. (2009b) and Venables et al. (2009) are consistent with the analysis of Lopez et al. In sum, the BRACElet research points "to the possibility of a hierarchy of programming related tasks where knowledge of programming constructs forms the bottom of the hierarchy, with 'explain in English', Parson's problems, and the tracing of iterative code forming one or more intermediate levels in the hierarchy" (Clear et al., 2011).

Simon et al. (2009) followed up on these studies. They found that – contrary to expectations – no meaningful relationships could be established when comparing the marks of a writing task and a comparable reading task (a 'Parson's problem', which requires the sorting of lines of code, and has been shown to assess similar skills to traditional code-writing questions; see Denny et al., 2008). This result may be explained by: 1) the lack of established criteria for comparing the complexity of two comparable programs from a learning point of view; 2) the differences between marking code-reading problems on the one hand and code-writing problems on the other (Simon et al., 2009), and 3) the ambiguities of code-explaining questions in general (Simon, 2009).

Other researchers have found some evidence to support the claim that novices can produce code based on familiar templates even without being good at tracing it (Anderson et al., 1989; Thomas et al., 2004). It can be questioned, however, whether such novices can reliably produce bug-free code and whether they can fix all the bugs in the code they produce without successfully tracing the programs.

As the BRACElet authors themselves have noted, the results from their line of research are not straightforward to interpret, and the matter of a learning hierarchy of programming skills remains unsolved. Furthermore, research has so far has focused on simple tracing, code-explaining, and writing tasks, and little has been shown about the relationship of these skills to other programming skills such as debugging, program design, or dealing with larger amounts of code. Nevertheless, it is easy to agree at least with the cautious conclusion of Venables et al.:

> *In arguing for a hierarchy of programming skills, we merely argue that some minimal competence at tracing and explaining precedes some minimal competence at systematically writing code. Any novice who cannot trace and/or explain code can only thrash around, making desperate and ill-considered changes to their code – a student behavior many computing educators have reported observing.* (2009, p. 128)

3.3 But many students do not learn to read code, either

In the light of the previous chapter, it is not very surprising that learners fail to learn to write and design programs. After all, these are cognitively complex skills that require relational understandings of structurally complex content. What about presumably less complex programming skills, such as tracing and explaining? Even though a strict learning hierarchy may not exist, the skill of program writing is partially dependent on these less complex skills. Are introductory programming courses succeeding in teaching students to read code?

Following up on the McCracken investigation described in Section 3.1, 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. A quantitative analysis of the multiple-choice questions was complemented by student interviews and an analysis of the 'doodles' students made while tracing. Lister et al. found 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. The analyses of quiz questions by Corney et al. (2011), Simon (2011), Teague et al. (2012), and Murphy et al. (2012) indicate that many students fail to understand statement sequencing to the extent that they cannot grasp a simple three-line swap of variable values. In one study, the problem existed even among students taking a third programming course (Simon, 2011).

It is clear from these results that students' foundational programming skills commonly do not develop as well as teachers expect. These results are discouraging, but cannot be ignored; of particular significance is the fact that many of the findings have been produced by international, multi-institutional studies which demonstrate that the problem is not local to a particular university, country, or type of institution.

3.4 What is more, many students understand fundamental programming concepts poorly

So far in this chapter, we have focused on skills. What about conceptual understanding? We gain another perspective on the challenge of introductory programming education by looking at what is known about novices' conceptions of programming concepts.

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).

Over the past few decades, many researchers have catalogued ways in which programming students struggle with fundamental concepts, and the kinds of incomplete and incorrect understandings that students have exhibited. Variables, assignment, references and pointers, classes, objects, constructors and recursion are among the CS1 concepts most commonly reported as problematic. Many students appear to have problematic understandings about the capabilities of computers and programs in general.

What follows is a review of research on the misconceptions and limited understandings that novice programmers have been found to have about fundamental programming concepts. Some examples of misconceptions appear below; there is a more comprehensive list in Appendix A. The reader can find a longer review of the earlier work on misconceptions within CER in a book chapter by Clancy (2004).

Research on programming misconceptions

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, e.g., `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 (e.g., Soloway et al., 1982; Bonar and Soloway, 1985; Soloway et al., 1983).

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 et al. (1986) and Sleeman et al. (1986) analyzed students' answers to code comprehension tests and subsequent interviews. They listed numerous errors – surface and deep – that students make with variables, assignment, print statements, and control flow. Du Boulay (1986) likewise listed a number of novice misconceptions with variables, assignment, and other fundamental programming concepts.

Pea (1986) listed a number of common novice bugs and suggested that they are rooted in a 'superbug', the assumption that there is a hidden, intelligent mind within the computer that helps the programmer to achieve their goals.

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 with parameters may misunderstand the concepts involved in different ways.

Recursion has been the focus of several studies. 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 Bhuiyan et al. (1990), George (2000a,c), and Götschi et al. (2003). Recursion is also one of the phenomena investigated by Booth (1992) in her phenomenographic work on learning to program. 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 '90s, interest in CER has shifted from procedural programming towards 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 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) listed 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) reported the results of a wide-scope, long-term, action research study of high-school students learning object-oriented programming. Their study 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, i.e., 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' mental models of programming concepts. Among his results are a number of misconceptions about fundamental concepts, e.g., the idea that the type of a value can change on the fly (in Java).

Several recent, complementary studies affirmed the existence of a number of incorrect understandings of assignment, variables, and the relationships between objects and variables. Ma (2007) 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. I myself explored students' understandings of storing objects and of Java variables through phenomenographic analyses of interviews (Sorva, 2007, 2008). Doukakis et al. (2007) combined findings from the literature and anecdotal evidence to list introductory programming misconceptions, which, the authors argue, arise as a result of students' prior knowledge of mathematics.

Sajaniemi et al. (2008) elicited the mental models that novice programmers have of program state by having CS1 students draw and write about how they perceived given Java programs' state at a specific stage 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) used interviews to identify student misconceptions about concepts and grouped the misconceptions thematically. Four themes were identified: the relationship between language elements and memory, while loops, the object concept, and code-tracing ability.

Viability

An understanding – even a misconception – is usually not universally useless. It may be viable for a particular purpose but non-viable in the general case. People may be entirely satisfied with their misconceived notions, even for a considerable amount of time. Nevertheless, the kinds of understandings of fundamentals reviewed here and in Appendix A are worrisome because they will cause problems for the novice programmer usually sooner rather than later. Those non-viable understandings need addressing; failure to do so is a failure of programming education.

The literature is effectively unanimous. Introductory programming education is struggling to cope with the challenging task of teaching beginners to create programs. Even teaching students to read programs is a challenge, as is helping students form viable understandings of fundamental programming concepts.

Part II

Learning Introductory Programming

Introduction to Part II

What is involved in learning to program? Under what circumstances do students succeed? What are the major obstacles to learning programming?

The following chapters delve more deeply into what learners go through as they study programming. I review what several learning theories have to say about learning in general, and what computing education research has to say about learning to program in particular. I also consider, in fairly generic terms, the pedagogical recommendations that arise from the theories. This review lays the foundations for the discussion of approaches to teaching CS1 in Part III.

Part II consists of seven chapters.

Chapter 4 focuses around schema theory, which explains how people learn to solve complex problems despite the limitations of the human cognitive architecture. I stay with cognitive psychology in Chapter 5, which deals with the mental models that people form of the systems they interact with; this leads to a discussion of how programming requires a viable mental model of how the computer executes programs. In Chapter 6, I turn to the family of educational theories known as constructivism, whose emphasis on learner-driven pedagogy has grown to be influential in many fields of education, including computing education. Chapter 7 is concerned with the phenomenographic tradition of empirical research on education. In the phenomenographic view, the most important form of learning involves becoming able to experience particular content – such as computer programs – in new ways.

There is, of course, a vast number of other traditions and theories within psychology, education, and CER; I cannot possibly discuss more than a few. The theories and research traditions that I have selected for Part II are particularly relevant because they each help us understand learning from a different perspective. Each of them is also relatively influential in programming education research: the cognitive perspective is well established, constructivism has recently greatly grown in influence, and a body of phenomenographic work is also emerging in the CER literature.

Although there is common ground, the different theories and theorists are not in full agreement with each other. Especially since I draw on multiple perspectives in a single thesis, it is important to consider the merits and weaknesses and compatibilities and incompatibilities of each point of view. Chapters 5, 6, and 7 each conclude with a section in which I review some of the main criticisms that have been leveled at cognitivism, constructivism, and phenomenography, respectively. The short interlude that is Chapter 8 compares and contrasts what these perspectives have to say about learning and programming.

Chapter 9 rounds off Part II with a brief discussion of the theory of threshold concepts, an emerging framework which seeks to explain why learners sometimes get 'stuck' at particular points of a curriculum, and which has eclectically drawn from many sources, including the ones discussed in the preceding chapters.

Chapter 4

Psychologists Say: We Form Schemas to Process Complex Information

Contrary to popular belief, the brain is not designed for thinking. It's designed to save you from having to think, because the brain is not actually very good at thinking. (Willingham, 2009, p. 3)

A massive, multi-threaded literature on learning has emerged from cognitive psychology. Theories of working memory, schemas, cognitive load, and mental models – among others – have contributed to our understanding of what it means and what it takes to learn. There is also a body of research on the psychology of programming that has applied general psychology to computer programming and come up with theoretical models that explain how people program and how they learn to program.

This chapter is the first of two chapters on cognitive psychology. Section 4.1 provides a short introduction to the idea of mental representation and the multi-store model of human memory, two cornerstones of modern psychology. Section 4.2 is an introduction to schema theory, followed by a discussion in Section 4.3 of the role of mental schemas in problem solving and in the growth of expertise. In Section 4.4 I turn from general schema theory to the psychology of programming: I review what is known about how people apply and construct schemas as they write programs. Section 4.5 introduces cognitive load theory, an influential 'spin-off' of schema theory with important and concrete implications for instructional design. Finally, in Section 4.6, I discuss existing research on program comprehension models, that is, theoretical models of how people make sense of computer programs. We will get to the theory of mental models in the next chapter.

4.1 Cognitive psychology investigates mental structures

A large body of psychological research deals with the ways in which knowledge is represented in the human cognitive system and the way people process these *mental representations*. Our use of mental representations is constrained by how human memory works – another topic fundamental to much of cognitive psychology. A sound theory of mental representation helps explain why we can function effectively despite the significant limitations of the architecture of our memory.

4.1.1 Inside minds, there are mental representations

Markman (1999, pp. 5-11) defines a *representation* as something that: 1) is about and captures aspects of a *represented world*; 2) exists in a *representing world* that is an abstraction of the represented world; 3) relies on rules that map elements in the represented world to elements in the representing world, and 4) is accompanied by *processes* that allow people or intelligent systems to make sense of the representation. The height of mercury in a thermometer, for instance, is a (non-mental) representation of a certain aspect of our physical world in the more abstract representing world of the thermometer. Representing rules map the possible heights of the mercury to different temperatures, and humans use an interpretative process associated with the representation to read the temperature. In Markman's definition, the difference

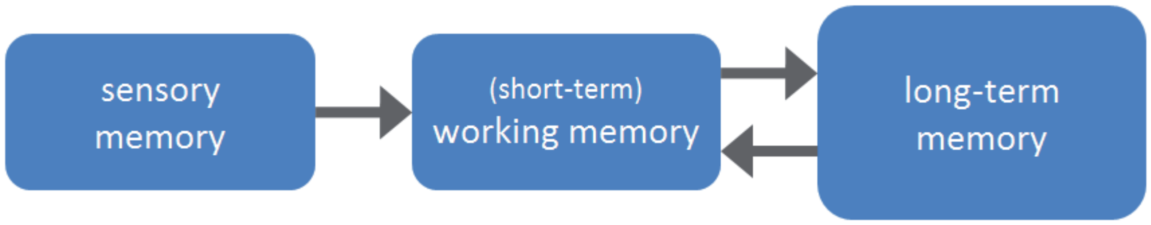


Figure 4.1: A commonly used basic architecture of human memory.

between a cognitive (mental) representation and other representations is a disciplinary one: a mental representation needs to be explained primarily by psychology rather than, say, the physical sciences.

There is a vast number of theories of mental representation. Some of the theories complement each other, some are in competition. The mental representations postulated by the various theories vary in terms of the duration of their existence, their discreteness, and their degrees of genericity and abstraction (Markman, 1999, pp. 14-16). The theories also vary in terms of which aspects of human behavior they seek to explain. For instance, schema theory (this chapter) is primarily concerned with how people gradually become able to deal with increasingly complex situations, while mental model theory (Chapter 5) primarily characterizes the human ability to deal with novel situations and causal systems. The program comprehension models of Section 4.6 are examples of programming-specific theories of mental representation.

4.1.2 Working memory can only hold a handful of items at a time

Since the 1960s, educational psychology has been greatly influenced by the distinction between short-term and long-term memory, two parts of the so-called multi-store model that was seminally formulated by Atkinson and Shiffrin (1968) and later extended in various ways by others. Figure 4.1 shows a typical diagram of the multi-store model, which consists of three parts.

Sensory memory is transient and stores sensory input extremely briefly, for less than a second. Its contents serve as input to the other memory systems.

Long-term memory is a memory system that is capable of storing large amounts of information for very long times. Its capacity is vast. However, we are oblivious to most of what is in our long-term memory most of the time. In order to use it, we must retrieve it for processing.

Such processing happens in *working memory*, an intermediate storage that is very limited in duration and capacity. People use working memory as they actively manipulate information that has just come in through the senses or that has been retrieved from long-term memory.[1] The most well known theory of working memory is Baddeley's. In his model, working memory consists of a *central executive* system which controls three other subsystems: a *visuospatial sketch-pad*, where we 'see' what we are currently thinking about visually, a *phonological loop*, where we 'hear' what we are currently articulating or hearing, and an *episodic buffer* that is capable of short-term integration of information from multiple sources (see, e.g., Baddeley and Hitch, 1974; Baddeley, 2000). Without rehearsal, information in working memory is lost in a matter of seconds (Peterson and Peterson, 1959).

George Miller's famous paper estimated the number of items that working memory can simultaneously hold to be "the magical number seven plus or minus two" (Miller, 1956). Later estimates have generally been even lower. Some theories do not count items; for instance, the concept of working memory in the influential ACT-R model of cognition is based on the "degrees of activation" of items in long-term memory – it is the total degree of simultaneous activation that is strictly limited (Anderson et al., 1996). It has also been suggested that the auditory part of working memory is temporally constrained not by a magic number but a "magic spell" (Schweickert and Boruff, 1986). Although the theories differ in their

[1]Working memory is often equated with *short-term memory*. Some theories distinguish between the two concepts, however. Ericsson and Kintsch (1995), for instance, argue for the existence of *long-term working memory*, a domain-specific extension of short-term memory within long-term memory that is created through extensive practice. This distinction between short-term memory and working memory is unimportant for my present purposes. Where I write "working memory", one may read "short-term working memory" instead.

details, there is a general agreement that working memory capacity is very limited, and that this limitation is a key feature of the human cognitive system.

4.1.3 Chunking and automation help overcome the limitations of working memory

How can we accomplish anything at all sophisticated with such a limited working memory? The answer, according to cognitive psychology, lies in the mechanisms of *chunking* information (Miller, 1956) and *automation* of processing (Schneider and Shiffrin, 1977).

To remember a phone number or date, we do not memorize a sequence of individual integers such as 0-4-0-0-5-1-5-3-8-8 or 0-6-0-3-2-0-1-0, but group the integers so that they form bigger *chunks*: 0400-515-388 or 06-03-2010. Chunks often, but not always, carry meaning on their own (e.g., month, operator code). A chunk, although composite, can be treated as a single item by working memory, allowing us to process larger amounts of information simultaneously. As we become increasingly familiar with information, we process it increasingly automatically, without paying attention to its components and without having to use our working memory. Tuovinen (2000) uses fluent readers as an example: "they do not try to read out individual letters, but process larger groups, words or groups of words, without attending to individual letters or even words separately."

What does this mean for learning? We learn something when we store it in long-term memory. Working memory is a bottleneck that limits our access to long-term memory. Cognitive science suggests that increasing one's ability to deal with information in large chunks and ever more automatically is key to learning and the growth of expertise. The following sections elaborate on this claim.

4.2 We store our generic knowledge as schemas

> *Schemas represent knowledge as stable patterns of relationships between elements describing some classes of structures that are abstracted from specific instances and used to categorize such instances.* (Kalyuga, 2010, p. 48)

A *schema* is a mental structure that contains generic conceptual knowledge (see, e.g., Rumelhart and Ortony, 1977; Rumelhart and Norman, 1978; Rumelhart, 1980; Anderson, 1977; Kalyuga, 2010). People make use of schemas constantly, both as they reason about everyday objects and situations, and as they solve problems. A person can have a schema of what a house is, of what going to a restaurant typically involves, or of how to write a program that computes an average of given numbers. A schema of houses, for example, may include the knowledge that a house is a type of building, that it consists of rooms, has walls and windows, is typically made of wood, brick, or stone, probably has a rectilinear shape, normally functions as a human dwelling, and is likely to be between 100 and 10,000 square feet in size (example from Anderson, 2009, pp. 134–135).

A schema contains knowledge of the stereotypical properties and parts of the concept, process, or situation it represents. The properties of a schema may involve other schemas, creating hierarchical structures or networks of schemas; the concept of house is linked to the concepts of walls and windows, for instance. Schemas may link to themselves as well, forming recursive structures (see, e.g., Rumelhart and Ortony, 1977). Schemas reside in long-term memory; the human mind is capable of storing countless complex schemas.

A generic schema allows us to make inferences about specific instances. For example, if someone speaks of their own house, we can – and do – fairly safely assume certain things about it, such as its shape (but we may be wrong, too). A schema "determines expectancies, organizes encoding, and systematically distorts retrieval [of knowledge from memory] in the direction of internal consistency." (Hintzman, 1986, p. 424). Schemas do allow for unusual properties in specific instances. If we see a house made of glass, we can still easily consider it a house, albeit untypical. Just how untypical an instance is allowed to be to still be considered a member of a schema-based category depends on the schema and the individual, and possibly even the time of day, as people do not always categorize instances consistently (see, e.g., Anderson, 2009, p. 136–138).

Schemas are created, extended, and modified through experience. Schema theorists vary on the details, but they all make the same central claim: previously existing schemas in long-term memory play a decisive role in how and where new experiences are integrated into people's mental representations. Rumelhart and Norman (1978) identify three modes of learning. *Accretion* is the common, straightforward accumulation of new information into existing schemas as more and more instances of a familiar concept are encountered. *Tuning* means the continual minor adjustment of one's existing schemas by constraining and generalizing the generic conceptual categories that they represent and adjusting the typicality of the values of schema properties. Finally, *restructuring* is a more dramatic change in mental representation that results in novel conceptualizations. It is triggered by unfamiliar experiences and involves the creation of new schemas, the reorganization, modification, or deletion of old schemas, and possibly abstraction from known concepts. People tend to resist such radical restructuring, and dramatic changes in schemas may take great time and effort. In all three forms of learning, the schemas that we already have determine how we deal with new experiences.

Having a schema does not mean you 'got it right'. New information is encoded – via accretion, tuning and restructuring – in terms of existing domain-specific schemas, but not necessarily the ones the teacher intended. Misconceptions (such as those discussed in Section 3.4; see also Appendix A) form when the learner integrates new information into the wrong existing schema or uses a non-viable analogy to motivate the creation of a new schema.

Broad vs. narrow definitions of schemas

The development of schema theory and the differing definitions given for schemas in the literature have been discussed by Brewer (2002), among others; my short review below draws primarily on Brewer.

The origin of schema theory is typically attributed to Bartlett's 1932 book *Remembering*, which predates modern cognitive psychology by a couple of decades. Bartlett found that people filled in stories with imaginary details they assumed to be true on the basis of the mental frameworks they used for understanding and remembering information. Later, Ausubel's (1968) subsumption theory of learning, which involves hierarchical memory structures and emphasizes the role of prior knowledge, foreshadowed the development of schema theory, as did the work of Jean Piaget. In the 1970s, Bartlett's ideas were revitalized by Minsky (1974), who sought to model humans' cognitive structures in computer memory as a part of the artificial intelligence movement. Schema theory was then established as a bona fide theory of learning within the research communities of psychology and education in seminal papers by Rumelhart (e.g., 1980) and Anderson (e.g., 1977). Around this time, it was noticed that schema theory explained the results of many previous experiments, and it led to a proliferation of empirical studies. Since the turn of the '80s, schema theory has made a substantial impact on educational psychology and CER. Over the past two decades, cognitive load theory (Section 4.5 below) has emerged as an influential offshoot of schema theory.

Perhaps because varied groups of people have been interested in the concept over a number of decades of early development, the word "schema" is not consistently used in the literature. There is a myriad of ways in which schemas are described, but uses of the term appear to fall roughly in one of two categories. The first corresponds to the way I have used the word above. In this usage, a schema is a mental representation for 'old', generic conceptual knowledge. The knowledge is old in the sense that it is based on previous experience and stored in long-term memory (rather than new in the sense of being currently experienced and constructed in working memory), and generic in the sense that it deals with conceptual categories rather than specific instances or contexts. The other, broader way to use "schema" is to umbrella all kinds of knowledge structures with it. For instance, Derry (1996) describes schemas as "virtually any memory structures", examples including generic schemas (as per the narrower definition above), situation-specific mental models, and phenomenological primitives (see Section 6.5).

Some authors have addressed this confusing terminological issue explicitly (e.g., Brewer, 1987, 2002; Choi and Sato, 2006). Brewer notes that even the original authors of seminal schema theory papers were inconsistent in their use of the term. He suggests that the narrow meaning is preferable, since the broad interpretation of "schema" is redundant and problematic with respect to the explanatory power of schema theory. I tend to agree with Brewer, and will continue throughout this thesis to use the word "schema" in the narrower sense to mean structures for old generic knowledge.

4.3 Schemas keep the complexity of problem solving in check

The discovery that the major difference between people who differed in ability was in terms of what they held in long-term memory changed our view of cognition. [. . .] *Long-term memory was not just used by humans to reminisce about the past but, rather, was a central component of problem solving and thought.* (Sweller, 2010a, p. 32)

4.3.1 Scripts and problem-solving schemas tell us what to do

The literature discusses various more specific types of schemas. An often-cited construct is the *script* or *event schema* formulated by Schank and Abelson (1977). Scripts are schemas that encode knowledge of recurring events and their stereotypical stages and other properties, and allow us to act comfortably and efficiently in familiar situations. The event of a visit to a restaurant, which consists of typical stages like ordering, eating, and paying, is a classic example. Similarly, a *problem-solving schema* tells us what to do to accomplish a goal in a particular kind of situation. Sweller and Chandler provide a good example:

Someone who is competent at algebra will have a schema for multiplying out a denominator. The schema will tell that person which of the infinite variety of algebraic equations is amenable to multiplying out a denominator and the procedure for doing so. When faced with a problem such as a/b = c, *solve for* a, *we can immediately solve such a problem, despite the many forms in which it could be presented, because our schema for this type of algebra problem informs us, for example, that the solution requires multiplying out the denominator on the left-hand side, irrespective of the complexity of the term on the right-hand side. Schemas, stored in long-term memory, permit us to ignore the variety that would otherwise overwhelm our working memory.* (Sweller and Chandler, 1994, p. 187)

Schemas allow us to draw on our prior experience and general knowledge so as not to be overwhelmed by the details of what we are currently experiencing. When we see an unfamiliar person – a specific instance of a familiar, automated schema – we can easily tell that the person is a human and act accordingly, without having to store or process every tiny detail of the person that our senses barrage us with. Schemas are crucial to our success in whatever we do as they "allow us to carry out everyday activities with minimum effort and to capitalize on the regularities of events and situations" (Preece et al., 1994, p. 128).

When faced with a complex problem to solve, we need to process various aspects of the problem and its solution in working memory simultaneously. A problem-solving schema such as the algebra schema described above allows us to ignore the specific details of the problem and recognize a more general pattern for which we have a schema. When a schema is available, it allows the entire problem or a part of the problem to be dealt with as a single chunk (see Section 4.1 above). In relation to one of their experimental studies, Sweller and Chandler (1994) discuss the solving of a simple problem: marking a point with given coordinates in a two-dimensional graphical coordinate system. They argue that for a complete novice, solving this problem requires an understanding of roughly seven distinct items, e.g., the fact that x in $P(x, y)$ refers to a location x on the x-axis. Where the complete beginner may be overwhelmed, someone who has experience with such problems can incorporate the entire problem into a single chunk that does not burden working memory greatly – or at all, if automated through practice.

4.3.2 Schemas are a key ingredient of expertise

Educational psychologists have long sought to understand the nature and development of expertise. On the one hand, the goal has been to characterize the mental representations that experts have and the ways in which experts put their knowledge to use when solving problems. On the other hand, researchers have investigated the mental representations and associated processes of novices, and contrasted them with those of experts. Through such comparisons, it is argued, we may better understand the changes that take place as we learn, and thereby improve teaching and learning.

An easy explanation for the superior performance of experts in problem solving would be in their superior 'cognitive hardware' such as a larger short-term working memory or superior intelligence. Early cognitive psychology also sought to explain expertise through general, domain-independent problem-solving skills and thinking strategies. Undoubtedly, talent does play a part in the development of expertise, and there is some transfer of expertise between related domains. However, many present-day cognitive scientists argue that the main factor that sets experts apart from non-experts is not their 'hardware' or any generic strategy that they possess, or their innate aptitude for a domain, but a readily accessible and usable domain-specific knowledge base in the form of schemas, acquired through lengthy practice.

The results of Chi et al. (1982; Glaser and Chi, 1988), who studied experts and novices solving physics problems, provide evidence of the key role that schemas play in the development of expertise. Chi et al. found that both experts and novices use schemas of relevant types of physical objects (inclined planes, for instance), but only experts use schemas which deal with more general principles such as the conservation of energy, and which subsume object schemas. Correspondingly, novices classified problems (that is, chose which schemas to activate) on the basis of the surface features of problems while experts were more concerned with the underlying patterns and principles. As schemas form organized hierarchies, experts could search for the required knowledge efficiently and quickly.

According to the psychological literature reviewed by Glaser and Chi (1988) and Winslow (1996), the growth of expertise is a process that involves the formation of many mental representations, deep and interlinked knowledge hierarchies, sophisticated strategies, and a rich arsenal of problem-solving patterns. Experts perceive – within their own domain of expertise – large, meaningful patterns that permit fast and error-free problem solving. They analyze problems in terms of principles rather than surface structures and tend to spend a lot of time analyzing problems qualitatively. In contrast, novices' untransferable 'fragile knowledge' (Perkins and Martin, 1986), lack of adequate mental representations, and tendency to use generic and inefficient problem-solving strategies have been widely observed. These characteristics of novices and experts have been documented in many disciplines, from chess to sports to programming.

The capacity of experts' working memory is as limited as that of novices, but experts compensate for this through efficient schema-based chunking. As experience grows, one forms schemas at ever higher levels of abstraction, which allows representing ever larger parts of a problem or its solution as individual chunks. Experts' high degree of automation further alleviates the load on working memory.

A corollary of the schema theory view of expertise is that since expertise is tied to the extent and quality of one's knowledge base, expertise in one domain is not readily transferable to another. A number of studies corroborate this claim (see, e.g., Glaser and Chi, 1988; Anderson, 2009, pp. 263–265).

4.3.3 An introductory course starts the novice on a long road of schema-building

Degree programs do not generally create fully-fledged experts. Dreyfus and Dreyfus (2000) identify five stages in the development of expertise: 1) a *novice* learns context-independent facts and rules; 2) an *advanced beginner* starts to recognize more abstract patterns in situations; 3) a *competent* problem-solver is capable of considering wholes and consciously choosing plans for achieving goals; 4) the level of *proficiency* is characterized by the automation of planning, which is no longer completely conscious, and 5) an *expert* has a mature and practiced understanding that allows them to 'see' what to do. Reaching expertise is a long process; cognitive psychologists speak of the "ten-year rule" in reference to the roughly ten years, or 10,000 hours, of deliberate practice that it takes to become an expert in a domain (see Willingham, 2009; Ericsson and Kintsch, 1995; Palumbo, 1990, and references therein). The goal of an undergraduate program, then, is not (or should not be) to create experts but to help students take some steps towards expertise and to prepare them to complete the journey on their own. As Winslow (1996) says, "most of us would probably settle for a graduate who ranks between competent and proficient".

Does it even make sense to speak of expertise in the context of introductory courses, the focus of my present work? I believe it does. 'Full expert status' may not be a reasonable goal for entire degree programs, and certainly is not one for introductory programming courses. Nevertheless, studies of expertise do give us a sense of the direction in which novice learners should progress. Schema theory suggests that the path to expertise is a path of knowledge-building. Novices should be helped to form fundamental schemas that can serve as the building blocks of more complex ones, and gain practice in their use so that they do not need to worry about the smallest details while solving increasingly complex problems.

Now let us turn to programming.

```
count = 0
sum = 0

number = float(raw_input('Enter a number:'))
while number < 99999:
    sum = sum + number
    count = count + 1
    number = float(raw_input('Enter a number:'))

if count > 0:
    average = sum / count
    print average
else:
    print 'No numbers input: average undefined.'
```

Figure 4.2: This program, which determines the average of given numbers, features several programming plans (adapted from Soloway, 1986).

4.4 People form and retrieve schemas when writing programs

Plenty of past research on how novices and experts program is based on cognitive psychology. Schema theory has been particularly influential in this work.

4.4.1 Plan schemas store general solution patterns

Expertise in programming, as in other fields, is marked by an improved knowledge base of generic solutions. Familiar problems for which a schema is available can be dealt with using more efficient strategies than unfamiliar problems, enabling experts to program effectively. Compared to expert programmers, novice programmers lack general schemas of 'canned solutions' to recurring problems and subproblems, which hinders problem solving.

Plans and plan schemas

Research on problem-solving schemas in programming was particularly active in the 1980s, when it revolved to a great extent around the pivotal and prolific figure of Elliot Soloway. Soloway and his colleagues used the term "plan" for the stereotypical action structures that programmers use as they design, write, and read programs. Consider the short program in Figure 4.2. Soloway (1986) presents an analysis of this program in terms of the plans that have contributed to its construction. Plans at various levels meet the subgoals of the programmer and combine to meet the overall goal of determining the average of given numbers. A running-total-loop plan has been used to compute the sum of input values. A counter-loop plan has been used to count the inputs. A division plan is used to compute a division. These three plans combine to meet the goal of computing the average. A print plan is used to print out the result. The running-total-loop plan and the counter-loop plans involve a stopping condition; a sentinel-controlled-loop plan caters for this subgoal. A skip-guard plan is associated with the division plan to prevent division by zero.

The example illustrates how, in order to produce a program, the programmer has applied schematic knowledge at many different levels of abstraction, from an overall program plan to lower-level plans to ever lower levels that are realized as just a single statement or expression. Plans are combined through abutment (one after the other, e.g., print after calculating), nesting (e.g., printing within the guard plan) and merging (e.g., the running-total-loop plan interleaves with the counter-loop plan) (Soloway, 1986).

The term "plan" requires a bit of clarification. Soloway and his colleagues used the word both as a term for generic problem-solving schemas within programming and to refer to specific instantiations of those generic patterns within solutions to particular problems. Rist (1989) makes note of this terminological issue – which recalls the 'broad vs. narrow schema' debate from Section 4.2 – and proceeds to use separate terms for the two meanings. Following Rist's lead, I will use the term *plan schema* to mean an abstract problem-solving schema that mentally represents a generic solution to a generic problem, and *plan* to refer to specific solutions in specific contexts (which may be instances of a generic plan schema known to the programmer).

Plan schemas and programming expertise

There is considerable empirical evidence that plans and plan schemas are the basic cognitive chunks used in designing and understanding programs.

Many publications have demonstrated how researchers can analyze programs in terms of their constituent plans (e.g., Soloway et al., 1982, 1983; Spohrer et al., 1985; Soloway, 1986; Rist, 2004). A number of empirical studies support the idea that programmers make use of plan schemas when working on programming tasks (see, e.g., McKeithen et al., 1981; Soloway and Ehrlich, 1986; Soloway et al., 1988a; Rist, 1989; Détienne, 1990). These studies have not only shown that novices are limited by their relative lack of schemas but that experts, too, have difficulty with 'unplanlike' programs that do not follow the schematic patterns the expert is accustomed to. It has been argued that plan schemas are the single most important feature of the programming expert. On the basis of their empirical results, Spohrer and Soloway (1986a,b) contended that novices' lack of problem-solving schemas is a greater challenge for educators than misconceptions about particular language features.

A prominent thread of research investigating the relationships between plan schemas, strategies, and expertise runs through the CER literature. Immediately below, I will focus on the roles of schemas in program writing. Program comprehension will be dealt with in Section 4.6.

4.4.2 Programming strategy depends on problem familiarity (read: schemas)

From the literature, three related dichotomous pairs of program implementation strategies rise to the fore: top-down vs. bottom-up, forward vs. backward, and breadth-first vs. depth-first (Rist, 1989).

Top vs. bottom, forward vs. backward, breadth vs. depth

A *top-down* design strategy starts with an overall problem at a high level of abstraction and decomposes it into subproblems. The subproblems are further decomposed until at the lowest level of the hierarchy, solutions to subproblems are written as program code. Conversely, *bottom-up* design starts at a lower level of abstraction to produce pieces of the solution, which are then used as a basis for reasoning about and solving a higher-level problem, and joined together hierarchically until they fulfill the overall purpose of the program.

Forward development means producing a program in the order in which it appears in the program code. This is made possible by what Rist (1989) calls *schema expansion*: a previously known plan schema is activated that specifies the solution steps that are needed. It is instantiated as a specific plan, and applied to the particular situation so that the coding process reflects the order in which the solution is mentally represented in the plan schema. In *backward development*, on the other hand, the programmer goes back to earlier sections of code to make additions or modifications according to a plan that is created on the fly during coding.

A *breadth-first* strategy is one in which all the subgoals and solutions at one level of abstraction are dealt with uniformly before moving on to another level. For instance, a programmer using a top-down, breadth-first strategy would solve all the subproblems at one level of abstraction, decomposing them into smaller subsubproblems, before considering the solutions of the subsubproblems at an even lower level of abstraction. Contrast this with a programmer using a top-down, *depth-first* strategy, who first exhaustively solves a single subproblem by successively decomposing it into ever smaller problems until a concrete solution is found, and only then considers how to solve the other subproblems.

Early studies: mixed findings

The early literature on empirical studies of experts' and novices' programming strategies has been reviewed by Rist (1989; see also Visser and Hoc, 1990). I will summarize briefly.

Jeffries et al. (1981) concluded that both novice and expert programmers used a decompositional top-down strategy and forward development to create programs. The difference they found is that novices used a depth-first strategy that concretized a subsolution as program code as soon as possible, whereas experts used a breadth-first strategy that kept all the parts of the design equally abstract at any given time. Anderson et al. (1984) also concluded that novices used template-based strategies that can be characterized as top-down, forward-developing, and depth-first.

The difference between expert and novice programming strategies is not quite as straightforward as these early studies might suggest, however. Guindon et al. (1987) report a study of expert designers in which top-down design was rarely seen and expert strategies were instead characterized by exploratory and serendipitous behavior suggestive of a bottom-up strategy. Visser (1987) found that an expert may combine bottom-up and top-down strategies and shift between the two during the program authoring process. Rist (1989) pointed out that when writing a program, novices tend to flounder and search for a solution with little overall plan or organization. He observes on theoretical grounds that novice programmers cannot possibly always use top-down strategies since they simply do not have the abstract plan schemas that would allow a problem to be matched with a solution and decomposed into a set of connected pieces. This observation is supported by the prior finding that novices have only low-level representations of programming knowledge, whereas experts have representations at both the abstract and concrete levels.

The gist of Rist: mixed strategies

The work of Adelson and Soloway (1985) hints at a solution to these mixed findings. They found that when working with a familiar problem domain, expert programmers mentally simulated a solution at each successively lower level of abstraction according to a top-down, forward-developing, breadth-first strategy. However, in an unfamiliar domain where they lacked complete solutions, the experts reverted to a bottom-up strategy involving simpler local models which they tested before combining such pieces to form a full solutions.

Building on Adelson and Soloway's work, Rist set out to explain the partially conflicting results on programmer strategies. He presented a sophisticated, empirically supported theory of how people use and create plan schemas during programming. I will describe the theory in brief; for more detail, see the original publications (Rist, 1986, 1989, 1995, 2004).

According to Rist, people use top-down, forward-developing, breadth-first strategies to solve programming problems whenever they can, that is, whenever a problem is familiar and they have a suitable plan schema available. When confronted with unfamiliar or particularly difficult problems, people revert to bottom-up, backward-developing, depth-first strategies in order to develop new solutions. Since a problem and its solution may have parts that are familiar and others that are not, programmers alternate between the two kinds of strategies. When a plan schema can be retrieved for an overall problem on a high level of abstraction, it is used to produce a high-level solution in the order suggested by the schema. Then each of the subproblems is addressed at a lower level of abstraction, retrieving and using a plan schema for each, if possible. When a problem does not have a retrievable solution in memory, one needs to be created. This is where the programmer reverts to a bottom-up, depth-first strategy, forming solutions first at a very low, concrete level and combining them until the unfamiliar subproblem gets solved. This process is characterized by backward development. When the previously unfamiliar problem is solved, a new plan schema is formed and stored for later retrieval. With further practice, plan schemas become increasingly abstract and their use increasingly automatic.

An implication of Rist's theory is that none of the strategies are exclusive to novices or experts. The key difference is that experts have more plan schemas, which are furthermore more abstract and apply to a wider range of cases. It is precisely because experts have knowledge of more kinds of problems that they can rely more on top-down, forward-developing, breadth-first strategies. For a complete beginner, any programming problem will be unfamiliar and involve relatively slow, difficult and error-prone bottom-up, backward, depth-first development.

By means of a longitudinal study of novices' programming behavior, Rist (1989) found that as students gained experience with programs of a particular kind, they shifted from bottom-up backward development to top-down forward development. From an educational point of view, Rist's finding is very interesting.

It suggests that the growth of expertise is marked not by adding top-down strategies to one's arsenal, as has sometimes been assumed, but by being able to use top-down strategies as a result of growing familiarity with problem types and their solutions. Furthermore, Rist's theory explains other researchers' findings that suggest that expert programmers spend significant amounts of time on analyzing problem descriptions and planning what to do while novices tend to rush to concrete code and make local changes rather than work on the big picture (Adelson and Soloway, 1985; Linn and Dalbey, 1985; Robins et al., 2003, and references therein). It is worthwhile for experts to spend time on figuring out which schemas to apply, but novices' lack of schemas forces them to work bottom-up from a low abstraction level.

Variable-related schemas and roles of variables

Many plan schemas in programming are related to the use of variables. For instance, a simple programming schema serves to explain the use of variables as counters whose values start at zero and are then repeatedly incremented by one. Commonly, the ways in which a variable is used in a program are not defined by a single line of code or even by consecutive lines; references to each variable are spread throughout the program code. In the terminology of Letovsky and Soloway (1986), the plan for such a variable is *delocalized*. Delocalization of a plan increases the cognitive load of a programmer trying to comprehend it, since multiple separate units have to be kept in working memory simultaneously in order to figure out the plan (more on cognitive load in Section 4.5 below). Novice programmers may find this cognitive load very difficult to cope with.

Attempts to clarify delocalized plans have included documentation (Soloway et al., 1988b), software tools (Sajaniemi and Niemeläinen, 1989), and techniques that make plans explicit. An example of the latter line of work is that of Sajaniemi (2002) who studied the nature of variable-related plan schemas. He proposed a set of *roles of variables*, which capture stereotypical patterns of variable use; an example is the 'most-wanted holder', which stores a value that best matches a particular criterion amongst a number of candidates (e.g., the largest integer encountered so far in a loop). Sajaniemi concluded that 99% of the variables used in novice-level programs can be described using a small set of roles. He further showed that his role set can be identified in expert programmers' thinking and therefore can be said to be an explication of experts' tacit schematic knowledge (Sajaniemi and Navarro Prieto, 2005). Byckling and Sajaniemi (2006a,b) found that role-based teaching makes a difference to novice programmers' programming strategy: students taught using the roles of variables tended to use forward-development more than students taught in a more traditional way, suggesting an improvement in schema formation.

What about pedagogy?

The literature on programming strategies suggests that the formation of problem-solving schemas is a key challenge of programming education. Novice programmers need to form schemas at multiple levels of abstraction, from the basic building blocks to increasingly complex abstractions that 'put the pieces together'. It is through an improving knowledge base of schemas that the novice gradually becomes able to employ more sophisticated strategies and work on programs in an increasingly top-down, forward-developing way.

It has been suggested that novices' programming strategy can depend on the tools used; Meerbaum-Salant et al. (2011) report that Scratch – a visual programming environment for beginners (MIT Media Lab, n.d.) – fosters an extremely bottom-up approach to program writing in which novices put visual code components together with barely a thought for the big picture. On the other hand, Hu et al. (2012) report positive results from a pedagogical reform in which students were taught to explicitly analyze goals and form and merge plans in Scratch programs.

Many schema-theory-based recommendations for CS1 pedagogy call for plan schemas to be made explicit in instruction. The roles of variables, mentioned above, are one concrete initiative; for more, see Chapter 10 on CS1 teaching strategies. The importance of lower-level schemas of particular kinds of statements and expressions should not be underestimated either; I will return to this point in Section 5.6 below.

The implications of schema theory have been expounded on by cognitive load theory, discussed next.

4.5 Cognitive load strains the human working memory during learning

Cognitive load theory is a framework for investigating the relationships between the human cognitive architecture (from Section 4.1), schema formation (Section 4.2), and the structure of the information that one learns about.

The central theses of cognitive load theory are that people learn best when their working memory is not strained too much or too little, and when as much as possible of the working memory load is directed towards the formation of schemas. The requirements imposed on working memory – that is, one's cognitive load – depend on the structure of the information that needs to be processed, which may in turn be manipulated through instructional design.

What is now known as cognitive load theory originates in Sweller's (e.g., 1988) investigations of problem solving. It builds on earlier theories of working memory and schema acquisition, and on the concept of subjectively experienced 'mental load' from human factors science. The volumes edited by Paas et al. (2003) and Plass et al. (2010) give excellent introductions to cognitive load theory; my summary below draws especially on these sources.

4.5.1 Some cognitive load is unavoidable, some desirable, some undesirable

Traditional "triarchic" cognitive load theory divides working memory load into three components: intrinsic, germane, and extraneous.

Intrinsic cognitive load is imposed upon the learner's working memory by the material that is to be learned and other aspects of the learning task. Intrinsic load hinges on *element interactivity*, that is, the degree to which learning task involves interacting elements that must be held in working memory simultaneously in order to succeed. Paas et al. (2003) use an example from photo editing: learning to understand the interactions between changing color tones, darkness, and contrast requires the consideration of each element simultaneously. Different materials and learning goals differ in element interactivity. Element interactivity also crucially depends on the learner's existing schemas, which determine what constitutes an element. Intrinsic cognitive load is reduced by the learner's prior knowledge of the subject matter: as described in Section 4.1, a complex existing schema may be treated as a single chunk, reducing the demand on working memory. A schema automated through practice may not impose any cognitive load at all. Intrinsic cognitive load cannot be reduced without compromising the learning outcomes of the present learning activity.

Germane cognitive load refers to working memory usage that is non-essential in the sense that it is possible to carry out the task at hand without it, but that is nevertheless needed for learning. Germane load contributes to the creation and enhancement of schemas in long-term memory, and the automation of those schemas. For instance, the cognitive effort involved in noticing underlying similarities and principles in superficially dissimilar examples constitutes germane cognitive load. In other words, germane cognitive load is the cognitive load that is devoted to effortful learning.

Unnecessary content in learning materials, content that is not easy to access while solving a problem but needs to be kept in mind, disturbing background noise, redundant instructions that need to be scanned for new content, instruction that introduces unnecessarily many new topics at once, and verbose lists of examples of factors causing extraneous cognitive load which appear at the beginnings of sentences that fail to start by explaining what it is that they list, are examples of factors causing *extraneous cognitive load*. Extraneous cognitive load is non-essential load that is unhelpful for learning about what is intended to be learned about – and consequently often harmful. It is brought about by materials, instructional setups, and other environmental factors.

Within cognitive load research, the types of load have generally been considered to be distinct and additive, with the total cognitive load being the sum of the three components (Figure 4.3). Instruction should be designed so that germane cognitive load is high. This implies that the sum of intrinsic and extrinsic loads should not exceed working memory capacity. The theory warns that when intrinsic cognitive load is high (as a result of insufficient prior knowledge and – consequently – high element interactivity), the learner is likely to be overwhelmed by any extraneous load, and hard pressed to find capacity for germane load. On the other hand, when intrinsic load is low, extraneous load is less of an issue; however, the lack of germane cognitive load may still present a problem if the learner fails to put their mind to work (because of lack of motivation, for instance). Both scenarios, which are illustrated in Figure 4.3, present challenges for learners and teachers.

While the threeway additive formulation of cognitive load theory may be overly simplistic – and has been questioned in the recent literature[2] – it serves to explain many of the interesting interactions between types of cognitive load that have been documented and is sufficient for my present purposes in this thesis.

4.5.2 Cognitive load theory has many recommendations for instructional design

Cognitive load theory has also contributed more specific points concerning the conditions under which meaningful learning is likely to take place. Researchers have found empirical evidence for a number of conclusions – often termed 'effects' or 'principles' – regarding the impact of cognitive load on learning. I summarize some of the main ones here; for more, see Plass et al. (2010) and references therein.[3]

Reducing extraneous load: the worked-out example effect and the completion effect

Some of the main findings of cognitive load theory concern learning by problem solving.

Problem-solving assignments are popular in education. However, cognitive load theory suggests that just solving problems is not the best way to learn to solve problems. Learning through problem solving is taxing for working memory, as resources have to be devoted to searching the problem for information relevant to solving it. Novices lack powerful schemas that suggest solution strategies, so they resort to weak generic strategies such as means-ends analysis, which require numerous elements to be kept in working memory at a time (see, e.g., Sweller, 1988; Paas et al., 2003). Little to no capacity is left for forming schemas in long-term memory to inform later problem solving. Some of this load is extraneous as it can be reduced by using different modes of instruction.

According to the *worked-out example effect*, extraneous cognitive load is reduced by studying *worked-out examples* of problems rather than solving the same problems oneself (see, e.g., Renkl, 2005). Worked-out examples are expert-produced solutions to a problem, often including explanations of the steps that were used to produce the solution. They are a staple of the cognitive-load-informed educational setting. Studying worked-out examples enables learners with limited schemas to allocate more of their cognitive capacity to germane load.[4]

A weakness of worked-out examples is that they may not engage the learner enough to induce germane load. The effectiveness of the examples is negated if the learner does not study them attentively and explain the given solution to themselves. The literature suggests that many learners are not naturally inclined towards effective spontaneous self-explanation of worked-out examples (see Renkl, 1997; Renkl et al., 1998; Renkl and Atkinson, 2003; Mayer and Alexander, 2011, and references therein).

Van Merriënboer et al. (e.g., 2003, and see Section 4.5.3 below) have noted a related *completion effect* according to which learning is enhanced by doing not full-blown problem solving but *completion problems* that provide the learner with a partial solution to be completed.

[2]Leading cognitive load theorists have recently come to emphasize the difficulty of telling apart intrinsic and germane cognitive load and to question the existence of germane load as a distinct source of cognitive load (Schnotz and Kürschner, 2007; Sweller, 2010b; Kalyuga, 2011). Along these lines, Sweller (2010b) and Kalyuga (2011) have advanced a reformulation of cognitive load theory which differentiates between two different aspects. The first aspect, *element interactivity*, corresponds to the working memory demands imposed by a learning task on a fully motivated learner (with a particular extent of prior knowledge) who utilizes their entire cognitive capacity. Overall element interactivity is the sum of intrinsic and extraneous element interactivity. The second aspect is the *actual working memory usage* of a (possibly unmotivated) learner when engaged in the learning task. This latter aspect depends on the former and also on factors such as motivation, which determine how or whether the learner actually processes the material at hand and what elements of the material they focus on. Working memory usage that is directed at processing intrinsic element interactivity is germane; other working memory usage is extraneous. I feel that this is a very useful clarification of the fundamental concepts of cognitive load theory for the future. However, in the present work, I remain with the better-established concepts of the traditional theory.

[3]I will discuss two more cognitive load effects, which are related to how information is presented – the modality effect and the split-attention effect – in the context of a visualization system in Chapter 15.

[4]The worked-out example effect, like most of the cognitive load effects, has been documented in the context of individual learning. Recent research on groupwork undertaken from a cognitive load perspective suggests that groups of novice learners can deal with more complex tasks than individuals, as their joint information-processing capacity exceeds that of the individual, despite the communication overheads (Kirschner, 2009). Some of the instructional recommendations from cognitive load theory may only hold for individual learning.

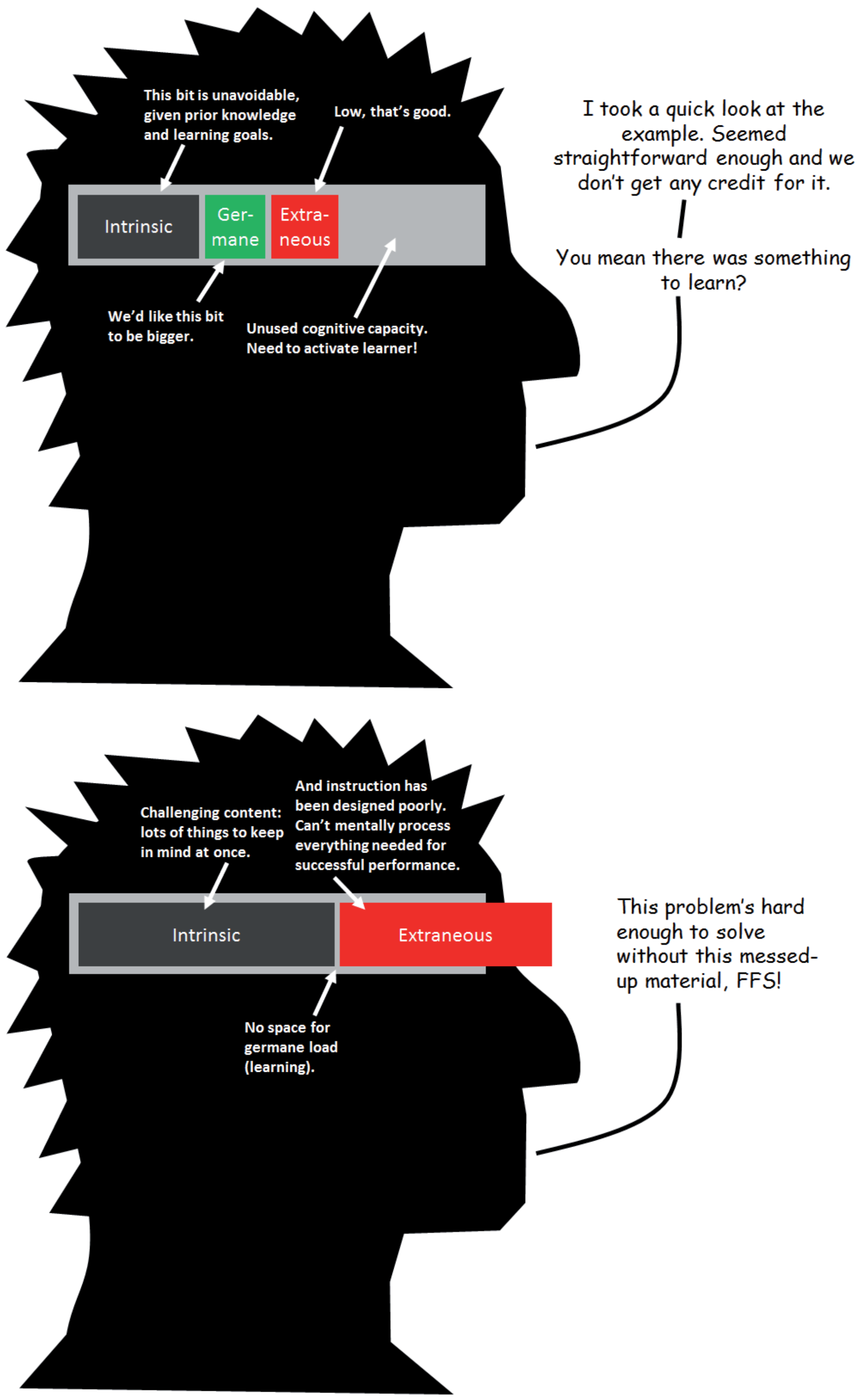

This bit is unavoidable, given prior knowledge and learning goals.
Low, that's good.
Intrinsic
Ger-mane
Extra-neous
We'd like this bit to be bigger.
Unused cognitive capacity. Need to activate learner!
I took a quick look at the example. Seemed straightforward enough and we don't get any credit for it.
You mean there was something to learn?
Challenging content: lots of things to keep in mind at once.
And instruction has been designed poorly. Can't mentally process everything needed for successful performance.
Intrinsic
Extraneous
No space for germane load (learning).
This problem's hard enough to solve without this messed-up material, FFS!

Figure 4.3: The different types of cognitive load in human working memory (as envisioned by traditional cognitive load theory). Two different scenarios are illustrated which have resulted in too little germane cognitive load.

The effect of prior knowledge: the guidance-fading, expertise reversal, and redundancy effects

Cognitive load theorists often recommend using multiple worked-out examples or completion problems as practice before a problem-solving task on the same topic. This emphasis on worked-out examples, completion problems, and the like does not mean that students should not solve problems. On the contrary, lower-load tasks serve to make problem-solving tasks better: they help in initial schema formation so that subsequent problem-solving tasks can be tackled effectively. "Guidance can be relaxed only with increased expertise as knowledge in long-term memory can take over from external guidance" (Kirschner et al., 2006, p. 80). According to the *guidance-fading effect*, decreasing support which takes into account learners' increasing experience is superior to worked-out examples alone (see, e.g., Renkl, 2005). With reference to the four-stage model of skill development of Anderson et al. (1997), Atkinson et al. (2000) have argued that worked-out examples are likely to work best in the early phases of the development of a cognitive skill, during which learners rely on analogies, have little practice, and are only starting to construct abstract problem-solving rules.

The guidance-fading effect is a manifestation of the more general phenomenon of the *expertise reversal effect*, which states that learning activities that are suitable for novices may become ineffective, or even harmful, as experience grows. Conversely, activities that are unsuitable for novices are often suitable for more advanced students. Several of the other cognitive load effects have been shown to be dependent on expertise (Kalyuga et al., 2003; Kalyuga, 2005, and references therein).[5]

Part of the reason why the expertise reversal effect occurs is that guidance given to experts may in fact hinder them, as they have to relate and compare the (to them unnecessary) guiding framework to their existing knowledge to find out what is relevant. This is an example of the yet more general *redundancy effect*, which states that presenting unnecessary information interferes with learning (see, e.g., Sweller, 2005).

Portioning intrinsic load: the isolated/interacting elements effect

The *isolated/interacting elements effect* states that "learning is enhanced if very high element interactivity material is first presented as isolated elements followed by interacting elements versions rather than as interacting elements form initially" (Plass et al., 2010, p. 30). This is essentially an endorsement of a part-whole approach to learning in which learners initially deal with small parts of a topic, and are not subjected to the complexity of the big picture until they have formed rudimentary schemas, however incomplete, of the parts.

Part-whole approaches run the risk of resulting in poorly integrated piecemeal knowledge, but may be worth the risk in the case of very highly interactive material, assuming that care is taken to foster the integration of the parts initially learned.

Increasing germane load: the variability effect and the imagination effect

Two cognitive load effects deal in particular with how certain kinds of activities can increase germane cognitive load.

The *variable examples effect* states that examples whose surface features are dissimilar to each other are more effective than examples that look similar on the surface (Paas and van Merriënboer, 1994). The variation encourages learners to identify the similarities between situations and thereby helps with the creation of generic schemas that are suitable for transfer.

The *imagination effect* states that having learners imagine themselves carrying out procedures related to worked-out examples enhances learning compared to having them simply carefully study the examples for the same length of time. The imagination task encourages learners to make use of their available cognitive capacity to mentally practice relevant skills (Cooper et al., 2001). However, to be effective, imagination appears to require significant working memory resources available for germane load (that is, the mental manipulation of and learning from the intrinsic content). Consequently, the imagination effect

[5]The expertise reversal effect is not a novelty discovered by cognitive load theorists. As discussed by Plass et al. (2010), the studies on expertise reversal are effectively an extension of the aptitude–treatment interaction studies carried out since the 1970s to examine the relationships between instructional methods and students' personal characteristics.

depends on prior knowledge in accordance with the expertise reversal effect: the literature suggests that imagination tasks work best for learners with some experience and existing schemas, and poorly if at all for beginners (Cooper et al., 2001; Kalyuga et al., 2003). On a related note, Renkl (1997) found that anticipating solution steps while studying a worked-out example was an effective way of learning, but only when the learner already had some prior knowledge.

Taking the effects into account

Van Merriënboer et al. (2003; van Merriënboer and Kirschner, 2007) present a general model for the design of learning environments that is mindful of the various cognitive load effects. In their model (dubbed 4C/ID) students are given learning activities that are divided into "task classes" of increasing complexity. Within each task class, the nature of the material covered is the same (that is, each activity has the same element interactivity and relies on the same set of generic knowledge in the form of schemas). To combat extraneous cognitive load while also catering for the expertise reversal effect, guidance fades within each task class: learning within each task class begins with a strongly guided activity such as studying a worked-out example, and ends with a considerably more challenging task such as solving a problem. Intermediate activities may include completion tasks, for instance. As a result of the simple-to-complex sequencing of task classes, the formation of schemas within a task class serves to reduce intrinsic load as the learner tackles the next task class on the same topic. To increase germane cognitive load, the tasks within each task class are designed to be variable in their surface features. To avoid overload during task performance, information about the general principles that pertain to the task class is presented to students before they tackle the activities of the task class (and is kept accessible throughout). However, procedural information that helps with specific learning activities is integrated into the learning environment and presented in a just-in-time fashion during the activities. Finally, part-task practice ('etudes' or 'drills' on a decontextualized aspect) may be used for selected parts of complex tasks for which the learners urgently need automated schemas.

The 4C/ID model is in many ways representative of cognitive load theorists' advice on instructional design. It underlines the need for deliberate practice and carefully designed guidance. It calls attention to the efficacy of activities other than problem solving on the way towards building problem-solving ability. It attends, via a managed learning environment, to the interplay between the element interactivity inherent in the content of learning and the learner's prior knowledge.

The design of learning environments is traditionally in the hands of the teacher. The impact of learners' personal goals on cognitive load is an open question that has not been much discussed in connection with cognitive load theory (Gerjets and Scheiter, 2003). Cognitive load theory does, however, highlight challenges in the use of highly learner-controlled learning environments. For instance, can a learner make reliable judgments on the element interactivity of content they are yet to learn about?

4.5.3 Some effects of cognitive load on introductory programming have been documented

There is a small body of cognitive-load-related research on the learning of introductory programming. Shaffer et al. (2003) discuss the potential of cognitive load theory in computing education, review earlier work, and observe that "the connection between cognitive load theory and the challenges faced by novice computer science students has not been fully addressed".

Van Merriënboer (1990; van Merriënboer and de Croock, 1992) found empirical support for the completion effect in the context of introductory programming education. In two separate experiments, students who modified and extended existing programs learned to write programs better than did others in a control group that wrote programs from scratch. In a similar study with third-year students as subjects, Nicholson and Fraser (1996) found no similar effect; Shaffer et al. (2003) speculate that the level of prior knowledge of programming may be the reason behind these mixed findings.

A study of the impact of reading assignments was conducted by Linn and Clancy (1992), who compared the performance of novices who designed and wrote their own programs to that of other novices who instead studied expert commentaries on how to solve the same problem (worked-out examples, in other words). Linn and Clancy conclude that such expert commentaries are eminently useful, and that in fact "writing a computer program is less helpful than having expert commentary for developing design skills, even when test questions require application of templates that were used for writing the program".

The results from these studies are in line with cognitive load theorists' recommendations to use worked-out examples and completion problems in addition to problem-solving tasks. These activities emphasize the pedagogical value of reading code, as opposed to merely designing and writing it. In this respect, cognitive load theory is compatible with the analyses of programming courses reviewed in Chapter 2, according to which the goals of introductory programming courses are cognitively demanding, and with the BRACElet project's hypotheses as to how reading and writing skills are interdependent in terms of their development (Chapter 3).

Robins (2010) discusses a related issue as he presents his hypothesis of why learners fail to cope with introductory programming courses. One of the underlying reasons, Robins argues, is that basic programming concepts are particularly densely connected to each other. In other words, introductory programming has an unusually high intrinsic cognitive load. A related point was made earlier by du Boulay (1986), who observed that programming is difficult to learn because the novice needs to deal with multiple interwoven challenges at once: programming notation, runtime dynamics, the need to mentally represent programs and their domain, tool use and the programming process, problem-solving schemas, and the notion of programming in general.

The theory of threshold concepts (more on which in Chapter 9) suggests that certain perspectives are particularly challenging to develop, in part because of the way they require multiple concepts to be integrated. Such learning thresholds may involve particularly high intrinsic cognitive load. Program dynamics, information hiding, and object interaction have been proposed as thresholds in introductory computer programming.

As noted, advice on pedagogy that draws on cognitive load theory tends to emphasize the need for teachers to carefully design direct, explicit guidance for learners (as opposed to student-directed, free exploration of complexity). The advice concerning introductory programming pedagogy is no different. I will return to cognitive-load-inspired pedagogies for CS1 in Chapter 10.

Cognitive load theory highlights how useful it is for novices to read plenty of program code. This brings us to the topic of the last section in this chapter, program comprehension studies.

4.6 Both program and domain knowledge are essential for program comprehension

A *program comprehension model* is a theoretical model of the mental representations that a programmer forms as they familiarize themselves with a program. A program comprehension model may describe the formation, use, structure, and/or content of those mental representations. Many different program comprehension models have been proposed in the psychology of programming literature since the 1970s, usually drawing on empirical investigations of novice and/or expert programmers. My review in this section draws on the original studies cited below and on previous reviews of program comprehension models, in particular those by von Mayrhauser and Vans (1995), Corritore and Wiedenbeck (2001), and Robins et al. (2003); see also Schulte et al. (2010). I comment primarily on three prominent themes within the program comprehension literature: the distinction between program and domain models, the distinction between top-down vs. bottom-up comprehension strategies, and the effects of programming paradigm on program comprehension.

4.6.1 Experts form multiple, layered models of programs

Program models and domain models

At least since the early program comprehension model presented by Brooks (1983), most program comprehension models have posited two components within people's mental representations of programs:

a program model and a domain model (in addition to various other structures posited by different theorists). These constructs are sometimes called by different names, and the definitions differ in their details, but the basic concepts are widely shared. A *program model* is a mental representation of the program itself such as the program text, the elementary operations the code performs (e.g., assignments, function calls), and the control flow of the program. A *domain model* is a mental representation of the problem that the programmer is trying to solve and its solution. It contains knowledge about the 'world of the problem', e.g., its components, their relationships, the data flow between components, and the purpose of the program overall, as well as the goals of its subprograms. Both program and domain models can contain knowledge of both static and dynamic aspects of a program.

The division between program and domain models in CER was influenced by the text comprehension studies of van Dijk and Kintsch (1983), who distinguished between a *textbase* (a representation of the semantics of a text that has been read) and a *situation model* (a representation of the world that the text is about). Program and domain models are not schemas in the sense of general knowledge, but their formation is guided by schemas and they incorporate elements of instantiated schemas.

Top-down or bottom-up? (The answer is in schemas.)

In Section 4.4, we saw a mix of early research results suggesting top-down program authoring strategies on the one hand and bottom-up strategies on the other. A similarly mixed set of findings characterized early program comprehension studies. Consequently, some early models emphasized the role of hypotheses of program behavior at a high level, recognizing common patterns in code and applying the corresponding plan schemas in a *top-down* way to make sense of a program (Brooks, 1983; Soloway and Ehrlich, 1986). Others found evidence for a *bottom-up* view of program comprehension, where a detailed program model is formed before a more abstract domain model (Pennington, 1987a,b; Corritore and Wiedenbeck, 1991; Bergantz and Hassell, 1991).

In program authoring studies, Rist's theory of schema formation (Section 4.4.2 above) explains how programmers mix top-down and bottom-up strategies depending on the availability of retrievable plan schemas. Similarly, researchers have presented program comprehension models according to which comprehenders use a mix of top-down and bottom-up strategies in opportunistic ways (von Mayrhauser and Vans, 1995; Corritore and Wiedenbeck, 2001, and references therein). When programmers seek to understand a program in an unfamiliar domain, they tend towards bottom-up comprehension strategies in which they first construct a program model. The program model allows the programmer to reason about the program's control flow, and is used to develop a domain model that deals with data flow and the purpose of program components. A top-down approach is used when constructing hypotheses about program behavior is possible, that is, when the domain is familiar enough. The direction of comprehension may change frequently during a comprehension process if parts of the program differ from each other in terms of familiarity or difficulty.

Wiedenbeck et al. (1993) studied the characteristics of experts' and novices' mental representations of programs and concluded that experts' mental representations are hierarchical and multilayered, contain explicit mappings between the different layers, are founded on the recognition of basic patterns (schemas), and are well connected internally and well grounded in the program text. Novices' mental representations exhibited these characteristics only to a much lesser extent. As in the case of program writing, experts can rely more on top-down comprehension strategies since they are familiar with more domains, problems, and solutions.

Paradigm matters for comprehension strategy

Research suggests that the programming paradigm used can make a difference in program comprehension. In particular, the paradigm influences the order in which program and domain knowledge is formed during the comprehension process.

Within the paradigms of procedural programming (Pennington, 1987a,b; Corritore and Wiedenbeck, 1991) and logic programming (Bergantz and Hassell, 1991), the literature suggests that when studying a program, programmers tend to construct first a program model, then a domain model. Results from object-oriented programming have been different, however. Burkhardt et al. (1997, 2002) extended Pennington's model of program comprehension to object-oriented programming. They found that people studying object-oriented programs formed a domain model early on during the comprehension process. This, the authors argue, is due to the nature of object-orientation, which emphasizes modeling the domain as objects and classes at the expense of program model aspects such as control flow. The results of Corritore and Wiedenbeck (2001) point in the same direction, and also suggest that object-oriented experts tend to use top-down strategies more than procedural experts do. Khazaei and Jackson (2002) found that, from the program comprehension perspective, event-driven programming resembles OOP rather than procedural programming in its emphasis on domain knowledge.

4.6.2 Novices need to learn to form program and domain models

Let us now consider program comprehension from the perspective of introductory programming education. Some research results point at a learning hierarchy of program comprehension skills, which, interestingly, may be different in different programming paradigms.

Paradigm matters for learning to comprehend

Corritore and Wiedenbeck (1991) studied novices' comprehension of procedural programs. Their study suggests that novices first learn to analyze programs in terms of program models, and later learn to develop domain models as they gain expertise. That is, the order in which people learn to form each model is the same as the order in which they form these models while studying a particular program, which was discussed above. Follow-up work on procedural programming tends to support Corritore and Wiedenbeck's findings: Wiedenbeck and her colleagues found that when studying procedural programs, novices formed mental representations that were stronger in terms of detailed program knowledge than in terms of knowledge of program function (Wiedenbeck and Ramalingam, 1999; Wiedenbeck et al., 1999).

Some work has been done to investigate the relationship between the program and domain models of novice programmers taught in an object-oriented way. Wiedenbeck and Ramalingam (1999) studied students taking their second course in programming. They found that when reading short object-oriented programs, the students developed mental representations that were strong in terms of knowledge of program function (the domain model) but weaker in terms of detailed program knowledge (the program model). This is in contrast to the models that the same students constructed when studying procedural programs, where the relative strengths and weaknesses of the mental representations were the opposite. More specifically, Wiedenbeck and Ramalingam report that the novices who performed better overall did equally well with both programs of both paradigms, with a similar pattern of errors in both program and domain knowledge. However, the novices who performed worse were different: the lower-performing half developed a better program model than domain model of the procedural programs they read, but a better domain model than program model of the object-oriented programs. In another paper, Wiedenbeck et al. (1999) compare novices taught using the object-oriented paradigm to novices taught procedurally. Both groups performed more or less equally well when reading short programs, but with different patterns of errors.

One plausible interpretation of the results of Wiedenbeck and her colleagues is that object-orientation changes the learning path of novice programmers: learning proceeds from being able to construct a domain model towards being able to construct a program model. This contrasts with the results from procedural program comprehension studies.

Pedagogical implications

Recent work by Schulte (2008; Schulte et al., 2010) has sought to translate the psychology of program comprehension into educational practice. This initiative is timely, as despite there being a sizable body of theoretical work on program comprehension, few educators have drawn substantially on this work (as reviewed by Schulte et al., 2010).

Schulte et al. (2010) suggest that CS1 teachers should make it their business to foster students' ability to glean multiple kinds of information from reading a program, to form connections between pieces of program-related information, and thereby to develop a holistic understanding of the program. They further point out that the direction of the teaching and learning sequence is a pedagogical choice: should one start with learning about programs' structure in terms of code and its runtime dynamics (to build a program model) or does one initially rely on domain knowledge? This question can be considered from the point of view of what is known about learning to comprehend programs in different programming paradigms. Where novices initially (naturally?) focus on building a program model – as seems to be the case in procedural programming – does this mean that they have less trouble with that aspect and teaching should focus on helping them form a domain model? Conversely, do object-oriented novices need additional help with the program model, as their paradigm primarily stresses the domain model?

Existing research does not provide unambiguous answers to these questions. Perhaps the most important message that we can take from program comprehension studies at the present time is that any introductory programming course should – in one way or another – teach students to build both program models and domain models when reading programs, and to relate the two.

Chapter 5

Psychologists Also Say: We Form Causal Mental Models of the Systems Around Us

In interacting with the environment, with others, and with the artefacts of technology, people form internal, mental models of themselves and of the things with which they are interacting. These models provide predictive and explanatory power for understanding the interaction. (Norman, 1983)

In this chapter, I stay with cognitive psychology as I turn to theories of mental models and their implications for learning to program.

Section 5.1 below outlines the general properties of what are known as mental models. Section 5.2 elaborates on what the literature has to say about mental model formation and quality. Section 5.3 introduces the notion of a 'conceptual model' that may be employed as a pedagogical device for learning about a system. Section 5.4 brings us to programming, and one of the pivotal concepts of this thesis: the 'notional machine', that is, the runtime mechanism which novice programmers must learn to control via programming, and which they need effective mental models of. Section 5.5 discusses the challenges of the step-by-step tracing of programs, a key skill that novices struggle with, in part because of poor mental models of the notional machine.

In Section 5.6, I ask whether we should be more concerned about the lack of problem-solving skills and schemas, about misconceptions concerning fundamental concepts, or about the lack of tracing ability. To foreshadow my conclusion, each issue is important, and there is an argument to be made that understanding the role in program execution of the computer – the notional machine – is fundamental to each of them.

The final section, 5.7, is more generic. I wrap up the review of theories of cognition from the previous chapter and this one by pointing out and briefly discussing some criticisms of cognitivist approaches to educational research.

5.1 Mental models help us deal with our complex environment

Schema theory deals with the growth of expertise and the forming of generic knowledge through experience. Various scholars (e.g., Preece et al., 1994; Brewer, 2002) have argued that the generic concepts embodied in schemas are not sufficient in themselves to explain how humans deal with objects and systems – including ones that are unfamiliar to them. This is where mental model theory steps in.

A *mental model* is a mental structure that represents some aspect of one's environment. A mental model is often about a specific thing or system rather than a generic concept. For instance, I have mental models of myself, of my wife, and of the TeXlipse software system that I am using to write this text. These mental models, which I have formed in part consciously and in part unconsciously, allow me to reason about how these specific aspects of my environment function in different circumstances. The concept of mental model has been applied to various disciplines, such as human–computer interaction, physics, the design of everyday objects, computer programming, ecology, and astronomy (see, e.g., references in Rouse and Morris, 1986; Schumacher and Czerwinski, 1992; Markman and Gentner, 2001). In particular, mental model theorists have been interested in the interactions between humans and causal systems.

5.1.1 We use mental models to interact with causal systems

Causal vs. logical mental models

It is necessary at this point to differentiate between two influential threads of research on constructs called mental models.

The term "mental model" received wide recognition after its introduction to cognitive psychology in the early 1980s. The two main threads of research on mental models are often attributed to the near-simultaneous publication of two books titled *Mental Models*, which elaborated in two different ways on Craik's (1943) earlier notion of "models of reality". The articles in the volume edited by Gentner and Stevens (1983) greatly influenced research into the kinds of mental representations people store about physical and software systems. The book by Johnson-Laird (1983) is seminal to a body of work that investigates a certain kind of situation-specific mental representation that people create in working memory to help them reason about logical problems. Markman and Gentner (2001) term mental models as described in the Gentner and Stevens book *causal mental models* and those in Johnson-Laird's research tradition *logical mental models*. Logical mental models are not of interest for my present purposes; whenever I write about "mental models" in this thesis, I refer to causal mental models.[1]

Mental models of systems

People form causal mental models of all manner of things. Schumacher and Czerwinski (1992) note that while one can have a mental model of a marriage or a social environment, not all topics are equally well represented on the research agenda. Much of the research on mental models has focused on the way people interact with and think about complex physical and software systems that involve causal mechanisms (e.g., an electrical circuit, a word processor). The mental model is a theoretical construct posited to explain how people describe the purpose and underlying mechanisms of such systems to themselves and how they predict future system states. This emphasis stems partially from arbitrary historical reasons that arise from research tradition[2] and partially from practical reasons: "We would argue that experts in computer systems are easier to define than experts in marriage." (Schumacher and Czerwinski, 1992)

Research does exist on causal mental models that is not concerned with humans' relationships with technical systems. However, I will limit my discussion to causal mental models of technical systems.

Researchers have further sought to identify the features that mental models of systems have in general, the characteristics that distinguish between useful and not-so-useful mental models, the transferability of the knowledge stored in mental models, and the relationships between learning and mental model construction. I will comment on each of these topics below.

5.1.2 Research has explored the characteristics of mental models

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;

[1]The body – or rather, bodies – of research on mental models make up a complex terminological and conceptual tangle. As with the word "schema" (Section 4.2), some authors use "mental model" as an umbrella term for all knowledge, a practice that has been criticized by authors such as Rouse and Morris (1986). There are numerous other more or less idiosyncratic ways of using the term in the literature, which I will not cover here.

[2]The focus on causal systems and devices can be traced back to human–machine interaction studies in the 1960s and a body of research that investigates process control: the manual control of complex machines and, later, the supervisory control of increasingly automatic machines (Rouse and Morris, 1986; Wickens, 1996, and references therein).

- 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 or 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 non-overlapping and complementary way, or they may be parallel – perhaps contradictory – models of the same parts. Parallel models may also operate on different levels of abstraction with each other (e.g., one model describes the physical aspects of a system and another its functional purpose).

Mental models evolve in long-term memory. When we reason about an object or a situation, we use a mental model of it; when necessary, we construct new mental models. Schumacher and Czerwinski distinguish between studying *stable* mental models that predate the need to use them and *derived* mental models, which are created when a situation calls for them. A derived mental model can be stored or forgotten right after it has been processed. If we have related schemas, activating them contributes to the kind of mental model we construct as our prior generic knowledge affects the way in which we see new instances. Conversely, aspects of the mental models of similar instances may be abstracted into a more general schema. Metaphors and analogies can also play a part in constructing and evolving a mental model (see, e.g., Gentner and Gentner, 1983; Schumacher and Czerwinski, 1992). Mental models are often not the product of deliberate reasoning; they can be formed intuitively and quite unconsciously.

Like schemas, mental models have 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 components, and allow for unanticipated situations to be dealt with (de Kleer and Brown, 1981). Because uncertainty about system capabilities can lead to trying out multiple approaches, novices tend to rely more on multiple inconsistent causal mental models of systems, while experts are less likely to do so (see Schumacher and Czerwinski, 1992, and references therein). Compared to the *ad hoc* naïve models often employed by novices, experts' mental representations are relatively stable as the result of lengthy experience.

What does a mental model consist of? Researchers vary in how they describe the internal structure of mental models, with some emphasizing their role as collections of knowledge, others the role of metaphors and analogies, and yet others the role of procedural knowledge. Ways of describing the structure of mental models include "topologies of device models" (de Kleer and Brown, 1983) and "homomorphs of physical systems" (Moray, 1990). On the other hand, many authors do not concern themselves with the internal structure of mental models. Jonassen and Henning (1996) argue that mental models are inherently epistemic – that is, the mental models themselves affect how we understand and express them – and that therefore their actual contents are not readily knowable by others. Still, Jonassen and Henning continue, researchers can gain information about these models by eliciting people's structural and procedural knowledge of systems, as well as the metaphors and visualizations people use. For the purposes of the present thesis, the internal structure of mental models is not important.

5.1.3 Mental models are useful but fallible

> *One of the reasons that mental models are so important is that for the individuals who hold them, those models have a value and reality all their own. Individuals believe in them, often without direct reference to their accuracy or to their level of completeness, and are reluctant to give them up.* (Westbrook, 2006)

Mental models allow users to be comfortable with complex systems. 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. Markman (1999, p. 266) further points out that people do not assess their mental models for completeness, but are content with partial knowledge and may not notice the limits of what they know. 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, as Westbrook (2006) puts it.

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). A further complication is that although models can be developed or corrected through practice and instruction, people often cling to emotionally comfortable and familiar existing models.

'Running' a model

According to Norman's description above, mental models are 'runnable'. This means that 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 future (or past) behavior and states of a system given a set of initial conditions (see, e.g., de Kleer and Brown, 1981, 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.

Running a mental model of a system is often called *mental simulation* of the system. I will use this term below.

Mental simulation is performed in working memory. It often involves visual imagery and may have a motor component. Since working memory capacity 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.

Researchers (e.g., de Kleer and Brown, 1981; Markman and Gentner, 2001) have emphasized the often qualitative nature of mental simulations, meaning that simulations tend to be based on relative properties rather than specific quantities. People do not calculate the specific values of the variables involved in a simulation. Rather, they reason about relative properties such as relative speed, and relative mass in a physical system. Qualitative simulation does not require the significant computation that would be necessary to carry out detailed quantitative simulations. Since the simulation process is demanding, people shift to using learned rules and cached results as they gain experience with the system.

While the level of abstraction must not be too low, 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 considerable experience (Klein, 1999; Markman and Gentner, 2001).

5.2 Eventually, a mental model can store robust, transferable knowledge

In this section, I describe three different theoretical frameworks that give insights into the transferability of the knowledge stored in mental models. Schumacher's theory describes mental model formation as a three-stage sequence that culminates in abstraction from specific models to generic knowledge at the expert stage. De Kleer and Brown characterize mental models in terms of their adherence to certain robustness principles, and suggest that robust models allow better transfer to new situations. Finally, Wickens and Kessel's studies show that certain kinds of learning activities are better than others at fostering the creation of mental models that transfer to similar tasks.

Schumacher: stages of model formation

Schumacher proposed a theory of forming mental representations of causal systems, which is empirically based and consistent with a number of related theories (Schumacher, 1987; Schumacher and Czerwinski, 1992). A three-stage sequence describes the acquisition of both specific and generic knowledge, with the latter arising from the recognition of common features in the former.

1. During the *pretheoretic stage*, an initial mental model of a specific system is formed by a user of the system. The initial model is a collection of retrieved experiences of superficially similar systems (e.g., other systems having a similar physical appearance). If no such experiences are found in memory, performance in using the system is reduced.

2. During the *experiential stage*, some understanding of causal relationships emerges through prolonged exposure to the system, even if the understanding is not supported by any superficial similarities to other systems. Knowledge embodied in the mental model is not yet readily transferable to other systems unless they are superficially very similar to the known system. During the experiential stage, the mental model becomes increasingly well ingrained.

3. During the *expert stage*, generic information is abstracted from multiple system representations. The user easily recognizes systemic patterns of behavior and effortlessly retrieves old knowledge about systems. Knowledge is easily transferred across instantiations of a system type, even when the system instances are superficially dissimilar.

This learning sequence provides a platform for making several points.

First, since previously known instances are used as a basis for learning, prior knowledge plays a key role in the acquisition of knowledge, both about specific systems and when abstracting to the general case at the expert stage.

Second, the early stages of learning about a system depend greatly on the superficial characteristics of systems. Many writers on mental models have noted the role played in mental model formation by social norms and schemas related to the surface structure of the system. Similar GUI controls in software applications create superficial similarities between systems, for instance.

Third, the experiential stage, which is often quite long, highlights the difficulty of transferring knowledge from one mental model to another. According to Schumacher and Gentner (1988), the less superficially similar two systems are, the worse the transfer is, even when the systems are functionally isomorphic. It is unrealistic to expect the transfer of a mental model before it is well ingrained.

Fourth, it is commonly easier to form some kind of pretheoretical mental model based on surface similarities than it is to substantially alter one's existing model during the experiential stage. As mentioned earlier, people tend to cling to their existing models. Moray (whose work is reviewed by Schumacher and Czerwinski, 1992) found that changing one's mental model took significantly longer than it took to originally form an initial model of similar complexity. Making matters worse is that mere coincidences can reinforce people's confidence in their existing yet flawed models (Besnard et al., 2004).

De Kleer and Brown: robustness

De Kleer and Brown's (1981; 1983) work on mental models provides another perspective on knowledge transfer. Using electrical devices as examples, they studied the mental models that people construct. De Kleer and Brown stressed the importance of knowledge that can be used to understand various systems, as "any single device however complex is of no fundamental importance". They argued that to be as useful as possible, a mental model should be internally consistent and correspond behaviorally to the actual device under consideration. Further, they argued that only certain kinds of mental models, meeting certain "esthetic principles", lend themselves to answering unanticipated questions or predicting the consequences of novel situations. They contended that using such mental models, which they termed *robust*, is characteristic of expert behavior.

De Kleer and Brown's theory defines mental models as topologies of submodels that represent components of the system. Each submodel is a collection of rules that describe the causal behavior of a component. Robustness arises out of the component models. The better a mental model's component models meet the following principles, the more robust the overall model is.

- The *no-function-in-structure principle*: the rules that specify the behavior of a system component are context free. That is, they are completely independent of how the overall system functions. For instance, the rules that describe how a switch in an electric circuit works must not refer, not even implicitly, to the function of the whole circuit. This is the most central of the principles that a robust model must follow.
- The *locality principle*: the rules that specify the behavior of a system component are represented only in terms of the internal aspects of the component and its connections to other components, not in terms of the internal aspects of other components. For instance, the rules that describe a switch in an electrical circuit must not depend on the internal state of any other component in the circuit. The locality principle helps ensure that the no-function-in-structure principle is met.
- The *weak causality principle*: the rules of the mental model attribute each event in the system to a direct cause. The reasoning process involved in determining the next state does not depend on any "indirect arguments". For instance, what happens next to a component in an electrical circuit must be directly attributable to some local cause rather than indirectly inferred by elaborately reasoning about other components. The weak causality principle is important for the efficient running of the mental model.
- The *deletion principle*: the mental model should not predict that the system will work properly even when a vital component is removed.

An overarching aspect of robust models is that the components of the 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 non-robust model may serve for mental simulations of a particular system under normal circumstances. A robust model is needed for transferring the knowledge embodied in a mental model to a similar but novel problem. A robust model is also needed to mentally simulate exceptional situations such as when a component malfunctions or a change to the system is either made or planned. This makes robust models highly desirable.

Wickens and Kessel: transferable models through active learning

Wickens and Kessel's work (see Kessel and Wickens, 1982; Wickens, 1996; Schumacher and Czerwinski, 1992) provides another perspective on mental model formation. 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. Wickens and Kessel inferred that the two kinds of training led to different kinds of internal models being formed. Process control researchers evoke worrying images of supervisors of automated nuclear power plants trained to monitor rather than to control, and of airplane pilots whose training is excessively based on autopiloting.

Wickens and Kessel's results are important as they show how people doing similar yet different tasks on the same system develop different kinds of mental representations and different kinds of expertise. In particular, a more passive task resulted in worse learning. This is not the last time that we will run into this thought on these pages.

5.3 Teachers employ conceptual models as explanations of systems

Designing, learning about, and using a system such as a software or household application involve several different models. The users and the designers of a system have their own understandings of it, the designer has an idea of what the system's users are like, and explanatory material can be presented to users to help them interact with the system. In the field of human–computer interaction, these different models, mental and otherwise, have been called by such a variety of names that it prompted Turner and Bélanger (1996) to write a paper specifically to "escape Babel" by sorting out terminological issues regarding causal mental models. Their take is paraphrased in Table 5.1.

Table 5.1: Terms related to mental models of systems, adapted from Turner and Bélanger (1996)

Term		**Definition**
target system	T	A system that is designed, used, or learned about, e.g., a piece of software.
design model	$M_D(T)$	A designer's mental model of the target system.
user model	$M_D(U)$	A designer's mental model of the (stereotyped) user of the system.
system image	$I(T)$	The parts of the target system that are visible to its users, e.g., displays, controls, help files.
user's model	$M_U(T)$	A user's mental model of the target system.
conceptual model	$C(T)$	An explanation of how the target system works.

A *conceptual model* is not a mental model but an explanation of a system deliberately created by a system designer, a teacher, or someone else. Its purpose is to explain a system's structure and workings to potential users. A conceptual model may be just a simple metaphor or analogy, or a more complex explanation of the system. The aim is usually to give an accurate and consistent account of the system, but conceptual models can be incomplete and even somewhat inaccurate if the author of the model deems it appropriate for present purposes. In other words, the purpose of a conceptual model is to present some or all of the content of a design model in a pedagogically motivated way to facilitate the creation of a viable mental model. A conceptual model may be worked into the system itself, attached to it as documentation, or presented to users separately.

Conceptual models have been shown to be useful in many contexts. Schumacher and Czerwinski (1992) describe what a typical study of mental models is like: one group of learners is given procedural instructions on how to control a system ('what to do') while another group is given 'how-it-works' knowledge (a conceptual model). According to Schumacher and Czerwinski's review, the typical study concludes that those taught using a conceptual model demonstrate better performance.

Let us now take the general psychology of this chapter into the context of programming education.

5.4 The novice programmer needs to tame a 'notional machine'

The discussion in Chapter 4 highlighted the fact that writing and reading programs requires mental representations of problem-solving patterns as well as models of the program itself and the problem domain. There is another important entity that needs to be mentally represented: the computer that executes the programs.[3] But what aspects of a computer are relevant and what can be abstracted away? Let us first read about the reasoning of Bruce-Lockhart and Norvell (2007), who describe what they are trying to teach novice programmers about.

> *The essence* [of Norman's work on mental models] *is that given a system T, a mental model of T can be defined as M[T]. Norman's work, however, was based on a very well defined T (a simple calculator). In teaching high-level (as opposed to machine language) programming it is much harder to define T. 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. We view them as requests for allocation of memory, in the stack, if they are internal declarations, in the static store if external. The fourth is a minefield. There's a truncation and two automatic type conversions. If you really want students to understand it they need to be able to interpret the expression and see the conversions, but these are normally done by the compiler. CPU operations include fetching the value for y (whether in a register or memory), carrying out the separate calculations, one in the integer arithmetic unit and one in the floating-point processor, and writing the final value back to z. We define T to be the system to which we are giving instructions. That is, 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 Norman's. That does not relieve us of the responsibility of at least trying to define it. We developed* [our software tool] *the Teaching Machine to provide students with a visual representation of the T that we believe approximates the one most professional programmers program to.*

5.4.1 A notional machine is an abstraction of the computer

Benedict du Boulay was probably the first to use the term *notional machine* for "the general properties of the machine that one is learning to control" as one learns programming. A notional machine is an idealized computer "whose properties are implied by the constructs in the programming language employed" but which can also be made explicit in teaching (du Boulay et al., 1981; du Boulay, 1986).

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 the computer executes programs and what the relationship of programming language commands is to such executions.

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[4]. 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.

[3] If you share the extremist views of Edsger W. Dijkstra, you may disagree with this claim and argue for a computer-free CS1. See Section 14.5.3 for further discussion.

[4] A functional programming notional machine may not be very 'machine-like' at all if it is grounded in mathematics and lambda calculus. Nevertheless, I consider that such a mathematical perspective on how computer programs work when executed also falls under the term "notional machine".

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", etc. 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 to pass 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.

Notional machine: a definition

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.

An obverse of the definition

I have heard computing education researchers use the expression "notional machine" to mean different things; there is also some variation in how the term is used in the literature. It is also instructive to consider what a notional machine is *not*, in my definition.

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 below.

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). Notional machines are implicitly defined by many visualizations of program execution.

Finally, a notional machine is not a general, language- and paradigm-independent abstraction of the computer. At least it is not that by definition, although notional machines can be generic enough to cover many languages, a whole programming paradigm, or even all programming languages. However, 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 this thesis, it is less generic notional machines that are usually of greater interest.

5.4.2 Students struggle to form good mental models of notional machines

In Section 5.3 above, I distinguished between a user's mental model of a target system and a conceptual model of the target system that is used for teaching purposes. In those terms, a notional machine is a target system that the CS1 student (a user) needs to construct a mental model of in order to program. A teacher, on the other hand, may wish to facilitate the creation of viable mental models of a notional machine by employing conceptual models of it.

A notional machine reflects the runtime semantics of statements. A mental model of a notional machine 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.

On the importance of the machine

There is plenty of agreement in the CER literature on the importance of a model of the computer for CS1 students. Du Boulay (1986) writes of the "bizarre" ideas that students have of how the computer executes programs, and identifies the notional machine as one of the main areas of difficulty that the programming novice needs to come to grips with. According to Cañas et al. (1994), "it is widely accepted that programming requires having access to some sort of "mental model" of the system". Perkins et al. (1990) "attribute students' fragile knowledge of programming in considerable part to a lack of a mental model of the computer". Smith and Webb (1995a) state that novices' difficulties in developing and debugging their programs stem from the fact that "their mental model of how the computer works is inadequate". Ben-Ari (2001a) concludes from the literature that "intuitive models of computers are doomed to be non-viable" and that novices' lack of an effective model of a computer can be a serious obstacle to learning about computing. And so on.

The lack of a viable model of the computer can lead to misconceptions, difficulties with understanding program state, and problems in knowledge transfer. I will comment on each of these topics in turn.

On misconceptions and hidden processes

> *A running program is a kind of mechanism and it takes quite a long time to learn the relation between a program on the page and the mechanism it describes.* (du Boulay, 1986)

A computer program has two forms: static and dynamic. The static aspect of a program is visible in code, but the dynamic aspect is usually implicit. The hidden nature of program dynamics has been linked to the multitude of misconceptions about programming concepts that students have.

In Section 3.4, I gave a lengthy list of studies that have uncovered misconceptions of programming concepts. Many of the inadequate understandings and difficulties discovered in those studies can be explained by the lack of a viable mental model of the notional machine that one is learning to control. Sleeman et al. (1986) concluded as much after reporting numerous misconceptions about Pascal programs: "even after a full semester of Pascal, students' knowledge of the conceptual machine underlying Pascal can be very fuzzy". With reference to the literature on misconceptions, Sajaniemi and Kuittinen (2008) likewise attribute student difficulties with basic concepts to a lack of understanding of a notional machine. Kaczmarczyk et al. (2010) identify, from their empirical data, "the relationship between language elements and underlying memory usage" as a major theme in students' misconceptions.

The list of misconceptions in Appendix A provides many examples of the kind of "hidden, internal changes" within the notional machine that du Boulay (1986) noted as being problematic for students. 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 computer: references, objects, automatic updates to loop control variables, and so forth.

Some generic misconceptions may lie behind many of the other more specific misconceptions. A few such generic misconceptions are listed at the beginning of Appendix A – for instance, it is thought that the computer can carry out deductions regarding what the programmer intends to do. These generic misconceptions concern the capabilities of the computer and/or the execution-time behavior of programs. In other words, they indicate problems with students' mental models of notional machines.

Subtle, 'hidden' aspects of programs also top some 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.

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.

Difficulties with program state

How the computer keeps track of program state is one of the central aspects of most notional machines. However, execution-time state is generally not explicit in program code. This is a source of confusion for novices, who "sometimes [forget] that each instruction operates in the environment created by the previous instructions" (du Boulay, 1986). Concrete examples of student difficulties can be found in the work of Sajaniemi et al. (2008), who studied student-created visualizations of the states of object-oriented programs. Their results showcase the variety of misconceived ways in which students envision state.

A finding from program comprehension experiments (Section 4.6) that has received relatively little attention is the difficulty of comprehending state-related aspects of programs. Corritore and Wiedenbeck (1991) compared novices' ability to answer questions about elementary operations, control flow, data flow, program function, and program state after reading a small program. They found that questions about state had a dramatically and unexpectedly high error rate. A comparison of upper and lower quartile novices showed that they differed particularly in their ability to answer questions about program state (although both had the most difficulty with state questions compared to other kinds). According to another study reported in the same paper, the difference between upper and lower quartile novices' ability to answer questions about state was even more pronounced when dealing with longer programs. Corritore and Wiedenbeck contrast their findings on novices with Pennington's (1987b) comparable study on experts, which showed no such highly elevated error rate on state questions. They conclude that understanding state is an area where novices differ from experts.

Notional machines and transfer

> *Programming should not be taught as a copy-paste art that only incidentally results in a correctly functioning program, but as a clearly defined activity that deals with unambiguous constructs.* (Sajaniemi and Kuittinen, 2008)

The importance of the notional machine is emphasized in situations where knowledge needs to be transferred to new contexts or where creative solutions, rather than familiar templates, are needed. Furthermore, research shows that pedagogy can have a significant impact on how well students can transfer their knowledge.

Kessler and Anderson (1986) conducted a sequence of studies on novice learners' iteration and recursion. One group of novices learned recursive programming first, followed by iterative programming, and another group were introduced to the topics in the reverse order. Neither group was explicitly taught about a notional machine. In this study, 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 towards 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 notional machine. In terms of Schumacher's theory of mental model formation (Section 5.2 above), it might be said that the students in Kessler and Anderson's studies who started with iteration soon reached the experiential stage with respect to possessing a mental model of a notional machine that supports repetition, whereas those who started with recursion struggled to get past the pretheoretical stage. A similar study by Wiedenbeck (1989) produced results compatible with Kessler and Anderson's.

For present purposes, what is most significant about these studies is that while even novice programmers manage to solve certain kinds of problems without being taught about a notional machine, a viable mental model of a notional machine is needed to understand programs on a deeper level and to transfer that understanding to new contexts for which ready-made templates are not available. No machine model is needed if the goal is to produce learners who will not use the programming language creatively, but if and when the goal is to produce creative problem solvers, then learning about the machine early on is "quite useful", as Mayer (1981) mildly puts it.

The birth of naïve models

Unfortunately, novices' mental models are often based on mere guesses.

Mental model theory tells us that people rely heavily on analogies based on surface features when forming mental models of new systems they encounter (Section 5.2). There is evidence of this in programming education as well. As du Boulay points out, a notional machine's properties are implicit in the constructs of the corresponding programming language (which is, in terms of Table 5.1, a key aspect of the system image 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). Many programming languages, on the surface, resemble natural language and the language of mathematics. Misconceptions (Section 3.4 and Appendix A) 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.

Mental model theory further explains that novices may have several contradictory mental models of a notional machine that they use to deal with different scenarios, whereas an expert's mental model is more generic and stable. Assignment statements with integer variables might be explained with one mental model, for example, and assignments using record types with an entirely different one.

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 developments 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,b) 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.

Therefore: teach early and teach long

Studies of mental models suggest that people cling to initial models, so that fixing a 'broken' model can be more work than constructing a viable model in the first place. This is one reason why it is important to help students form a workable mental model of a notional machine early on in their programming studies. 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.

Starting early does not mean ending early, either. Programming teachers often expect students to be able to switch between programming languages (and notional machines!) fairly early on in programming curricula, commonly after a first programming course. While transfer on the basis of superficial features is commonplace, transfer to even similar notional machines that look dissimilar on the surface (i.e., in program code) requires a deeply ingrained mental model of the original notional machine. As forming such models tends to require a substantial amount of experience, teachers need to make sure that students get plentiful practice with the original notional machine before expecting them to transfer to other languages and paradigms.

Learning about a notional machine can draw on a conceptual model that makes explicit the way in which the machine works and underlines how a programming language differs from natural language and familiar mathematics. A conceptual model of a notional machine can take the form of a drawn visualization, a verbal or textual description, or a software application that visualizes the notional machine. It makes explicit the hidden effects of commands in program code. In these terms, what Bruce-Lockhart and Norvell (2007) describe (p. 59 above) as their target system T is a notional machine for C programming, and the visualization in the Teaching Machine software serves as a conceptual model for teaching about this notional machine.

I will discuss conceptual models of notional machines in more detail in Part III. Visual program simulation, which is introduced in Part IV, is a way of engaging students in interactions with a conceptual model of a notional machine.

The notional machine is also involved in the activity of program tracing.

5.5 Programmers need to trace programs

Tracing a program means analyzing its execution to determine what operations occur and how its state changes. The human act of tracing a program can be viewed as a form of mental simulation (cf. Section 5.1.3). Tracing is a key programming skill that expert programmers routinely use during both design and comprehension tasks (Adelson and Soloway, 1985; Soloway, 1986). As discussed in Section 3.3, developing the ability to trace programs is linked – at least to some extent – to the development of program comprehension and program writing skills.

5.5.1 Novices especially need concrete tracing

Concrete vs. symbolic tracing

Détienne and Soloway (1990) distinguish between two different techniques that experienced programmers use when trying to comprehend a program: symbolic and concrete tracing (also known as symbolic and concrete simulation, respectively).

Symbolic tracing means using generic values while tracing a program's execution, e.g., "it loops ten times, reads a number, compares the number to maximum and sets Max to Num if something is true" (Détienne and Soloway, 1990). This was the default strategy used by the experienced programmers studied by Détienne and Soloway when they first encountered a new program. They used it whenever possible, that is, as long as they thought that the program matched their existing plan schemas and no problems were evident. *Concrete tracing*, on the other hand, uses specific values: "the number should be five, so the average should be five, I put five in Sum, add 1, Num is not 99999 [the sentinel value], enter 99999 to get out, Sum is now five, Count is one." (ibid.) Experienced programmers used concrete tracing, in

Détienne and Soloway's parlance, "to evaluate the external coherence between plans, i.e., to check for unforeseen interactions". In other words, concrete tracing was useful for studying programs which merged familiar plans in an unfamiliar way. For experts, concrete tracing is a "last resort in cases that are complex, hard to understand, or when the program does not work the way that symbolic tracing suggests" (Vainio and Sajaniemi, 2007).

Concrete tracing and debugging

Concrete tracing is a vital skill for any programmer. It is especially useful in the fight against bugs: the programmer may know what the various code constructs do in isolation, but not what happens when they are combined in the buggy way they currently are, and need to "check for unforeseen interactions".

Concrete tracing is even more important for novices. Vainio and Sajaniemi (2007; Vainio, 2006) argue that the novice programmer needs to trace code carefully in order to understand the causal relationships between statements, and cannot use symbolic tracing as the default strategy. The novice lacks plan schemas that provide solutions to common problems, which hinders symbolic tracing and can force the novice to work at the concrete level. Before they can raise the level of abstraction during tracing, novices need to automate – through experience and practice – the processing of syntactic and semantic details.

Some important skills are pedagogically unproblematic. Unfortunately, as we have already seen in Section 3.3, there is plenty of evidence that many CS1 students fail to learn how to trace programs. 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". Failure 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.

5.5.2 Tracing programs means running a mental model

One of Perlis's (1982) famous epigrams states that "to understand a program you must become both the machine and the program". Similar sentiments have been expressed in the psychological research literature.

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", as well as "the computer does X", to mean effectively the same thing. As noted earlier in the chapter, 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. For present purposes, it is convenient to speak of mental tracing as the 'running' of a single mental model that encompasses both the notional machine and the program that is being traced.

Two challenges to the successful running of a mental model during tracing are the difficulty of keeping track of program state in working memory, and the robustness of the mental model.

Status representations

Tracing a program requires keeping track of the state the execution of the program. Such a description of state, which Perkins et al. (1986) call a *status representation*, must be dynamic, changing as execution proceeds. A status representation consists of the elements of (one's mental model of) the notional machine used: variables, objects, references, function activations, etc.

As discussed in Section 5.1.3, mental tracing is taxing for working memory, and success is predicated on carefully choosing a level of abstraction and the 'moving parts'. People tend to run their mental models qualitatively rather than with specific values, and correspondingly prefer symbolic to concrete tracing. However, as noted above, programming tasks like debugging call for concrete tracing. 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 in particular 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 (see, e.g., Vainio and Sajaniemi, 2007; Fitzgerald et al., 2008). Vainio and Sajaniemi (2007, p. 239) discuss how failure to abstract can lead to overwhelming cognitive load: "Tracing a sorting algorithm without [the abstraction of a swap operation] may result in too many objects taking part in the simulation, which may cause the simulation to exceed the limits of working memory and become error-prone or too slow to be practical.[5] Vainio and Sajaniemi discovered a novice strategy (commonness unknown) for concrete tracing which is motivated by its low load on working memory. This strategy, which they call "single-value tracing" limits the number of non-trivial variable values in one's mental status representation to one.[6] That is, only a single unnamed 'slot' is used to store the value of whichever variable was most recently non-trivially assigned to. Single-value tracing does not work in the general case, but the novice may not even realize this as "it works fine with small programs that are typical to elementary programming courses".

Despite problems with mental status representations, novices tend not to use external aids as much as they need to. Lister et al. (2004) report that the most common type of 'doodle' on paper used by students during a tracing task was no doodle at all. Thomas et al. (2004) experimented with object diagrams that describe runtime state, essentially an external visual aid for object-oriented status representation. They found that novice programmers were not helped by object diagrams that they were given, and were distinctly lacking in eagerness to draw diagrams themselves. The results of Vainio and Sajaniemi (2007) suggest that there are two kinds of difficulties: novices not only struggle to produce state diagrams, but also fail to make use of such diagrams when they have produced them. Moreover, anecdotal evidence from CS1 courses suggests that students do not voluntarily use debuggers – another form of external status representation – as much as their teachers would like them to (this is my own experience; see also, e.g., Isohanni and Knobelsdorf, 2010).

Robust models needed

According to de Kleer and Brown's theory (Section 5.1.3 above), a mental model of a causal system needs to be robust if it is to be transferred to novel component configurations. 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. The components of a program are the programming constructs used in the program code and the corresponding mechanisms within the notional machine. To stay with de Kleer and Brown's theory, models of these program components ideally follow certain "esthetic principles" of robustness. De Kleer and Brown also contended that a model must be robust to be usable in unexpected contexts where the behavior of the system does not follow one's prediction. A robust mental model of both the program and the associated notional machine would therefore be needed to debug programs.

Vainio (2006; Vainio and Sajaniemi, 2007) applied de Kleer and Brown's theory to programming. They focused in particular on the no-function-in-structure principle and the locality principle that supports it (Section 5.1.3). Consider the following code fragment.

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

[5]When the program is sufficiently complex, experts will also be overwhelmed by cognitive load. Parnas (1985) criticized tracing as a software development tool: "As we continue in our attempt to "think like a computer," the amount we have to remember grows and grows [along with program complexity]. The simple rules defining how we got to certain points in a program become more complex as we branch there from other points. [...] Eventually, we make an error." However, as Soloway (1986) pointed out in response, tracing is the best way invented so far for understanding the dynamic aspect of programs, and therefore should be learned by students. Programmers today have access to ever better software tools that support mental tracing and sometimes even eliminate the need for it, but Soloway's observation still appears to hold largely true.

[6]'Trivial variable values' are those that can be seen directly from the program code, such as literals assigned to variables.

Figure 5.1: Robust models transferred into an unexpected context (Rubens, ca. 1623).

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 this program 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 had developed an understanding according to which *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.

A problem with an understanding such as the above is that the student has formed a component model of the `for` statement (within their model of the program) that does not meet the principles set out by de Kleer and Brown. The understanding violates the no-function-in-structure principle, as it mixes up the definition of a `for` statement (its structure) with what the `for` statement is used for in a particular context (its function, which here includes assigning zero to a variable). A related violation is that of the locality principle. The component model of the `for` construct is not independent of the specifics of other constructs; it is dependent on the idea that somewhere within the body of the loop there is at least one other statement that makes use of a variable that affects what the `for` statement does.

Vainio and Sajaniemi (2007) describe violations of the no-function-in-structure principle 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. Their work emphasizes the need for novices to separate what a construct is and what it is typically used for.

To summarize this section, tracing a program requires the ability to mentally simulate programs running in a notional machine. Novices especially need concrete tracing at a low level of abstraction to make sense of programs, but often lack the ability or even the inclination to trace programs. Ideally, mental simulations are based on robust, generalizable mental models of programs, and rely on abstraction and external status representations if and when the limits of working memory are met.

5.6 Where is the big problem – misconceptions, schemas, tracing, or the notional machine?

Let us reflect on what this and the previous chapters have told us about learning to program. My review of learning programming has so far centered around five challenges that face the novice programmer.

1. Creating programs imposes a great cognitive load on novice programmers.
2. Programmers need plan schemas which represent generic solutions to common problems, but novices have few.
3. Novice programmers have misconceptions about basic programming concepts, which give them trouble when reading and writing programs.
4. Many creative and unexpected programming tasks require mental tracing of programs, something that novices are not always capable of.
5. Novices need to form a viable mental model of a notional machine to be able to understand program execution.

Some pertinent and difficult questions are as follows: Which of these challenges are the most important? Is there a particular bottleneck for learning? How do the challenges depend on each other?

Different researchers have given different answers to these questions.

A legion of misconceptions

Research on misconceptions has a long history which demonstrates that non-viable understandings of programming concepts have been considered an important and pedagogically fertile area of computing education research by numerous authors. Clancy enthuses:

To me, this [research on misconceptions and mistaken attitudes] *is the most interesting and, for my teaching, the most relevant education research. It helps me interpret wrong answers – which I see a lot of! – and also guides me toward activities that address student problems.* (Clancy et al., 2001, p. 330)

As a teacher, it is easy to agree with Clancy that knowing about the kinds of mistaken understandings that students have is very useful in practice, and can help in the design of better programming courses and communicating better with students. On the other hand, some of my colleagues have called research on misconceptions passé or pointless, as there are presumably an infinite number of incorrect ways to understand a concept and charting out all these ways is a futile effort.[7] While I certainly do not consider such research pointless, it is true that the multitude of misconceptions is an issue. If we manage, through research, to get at the general causes underlying some of the more specific misconceptions, we gain better insight into learning programming and can perhaps swat out entire families of novice bugs in one stroke.

Some phenomenographers have argued that rather than misconceptions, we should be concerned with the different correct but partial ways in which people understand programming concepts. Research à la Eckerdal and Thuné (2005) can point out the educationally critical aspects of programming concepts, which – according to the constitutionalist perspective on learning (Chapter 7) – are limited in number. A different approach that seeks to explain why certain kinds of understandings are not viable is Vainio's application of de Kleer and Brown's 'esthetic principles' to programming (Section 5.5 above). Soloway and Bonar's 'bug generators' and Pea's 'superbug' of the anthropomorphic computer (Section 3.4) also aim to get at the general reasons behind various specific misconceptions.

My last-not-least example of a principled way to address numerous misconceptions at once was already discussed in Section 5.4: the notional machine. To recapitulate, it seems that many non-viable understandings can be explained by the root cause of novice programmers not understanding program execution and the role the computer plays in it. While little empirical work exists that has investigated whether or how particular misconceptions can be helped by learning about a notional machine, there is plenty of general evidence and agreement in the literature that learning how the machine works is important.

Soloway & Spohrer: schemas $>$ misconceptions $+$ syntax

A line of thinking influentially argued for by Spohrer and Soloway (1986a,b) is that syntax and misconceptions about language constructs are not as significant a problem for learning programming as the lack of plan schemas that enable novices to solve common problems. Spohrer et al. (1985) attribute many novice mistakes to an inability to successfully merge plans. Winslow (1996) claims that "study after study has shown that students have no trouble generating syntactically valid statements once they understand what is needed. The difficulty is knowing where and how to combine statements to generate the desired result."

Prerequisites of high-level schemas

What is required to use programming schemas to solve problems? Misconception-free understandings of syntax and semantics, and some skill at tracing, at least.

When emphasizing the importance of plan schemas, Spohrer and Soloway referred especially to higher-level plan schemas (e.g., the find-average schema) rather than low-level ones (e.g., the assign-to-variable schema). As discussed in Section 4.4.1, high-level schemas build on low-level ones. To be useful, these low-level problem-solving schemas must contain viable, non-fragile understandings of the semantics of programming constructs and the associated notional machine. Misconceptions about fundamental programming concepts lead the novice to ignore pertinent aspects of a situation and miss links between examples, reducing germane cognitive load and hindering the formation of higher-level schemas. Misconceptions can also lead to the formation of 'buggy schemas' at higher levels.

As discussed in Section 3.3, many novices lack tracing skills, which are important for many debugging tasks and bug-free problem solving using plan schemas. One study explicitly concludes that their findings

[7]Personal communication.

are “somewhat contrary to the classic work of Spohrer and Soloway”: students are sometimes able to create buggy template-based (schema-based) code, but do not necessarily understand the code that they themselves produce, and cannot trace its execution to fix the bugs (Thomas et al., 2004).

Overall, recent research does not disagree with Spohrer and Soloway’s identification of problem-solving schemas as important, but does emphasize that lower-level issues are also worthy of attention.

Garner, Haden, and Robins categorized the problems that CS1 students needed help with in class and examined the resulting distributions (Garner et al., 2005; Robins et al., 2006). Many of the problems had to do with program design, but many also involved various construct-related issues. Trivial mechanical problems (e.g., missing semicolons) were also common. The distribution pattern was similar across course offerings; high-, medium-, and low-achieving students also all exhibited similar patterns. Fitzgerald et al. (2008) report that novice programmers with 15 to 20 weeks of programming experience behind them had more trouble finding non-construct-related bugs than construct-related ones. Ko and Myers (2005) presented a framework for analyzing the causes of software errors; applying it, they found that many bugs were rooted in cognitive difficulties with particular language constructs, although other causes of difficulty were common as well. Denny et al. (2011) demonstrated that novices have great trouble in producing even small programs that are syntactically correct. McCauley et al. (2008) point out that construct-related bugs feature even in a revered pedant’s classification of the bugs in TeX (Knuth, 1989).

Drawing on the BRACElet studies (Chapter 3), Lister (2011b,c) has recently emphasized that although struggles with syntax initially contribute to students’ cognitive load – and are important to address – the great cognitive load inherent in program writing remains a major concern even when those earliest problems have been overcome. Kranch (2011) studied novice programmers taught using three different topic sequencings, including one that started with higher-level schemas before moving to their elements, and one that did the opposite. He found that irrespective of ordering, the higher-level schemas were always the most difficult topic (both in terms of achievement and student-given difficulty ratings), a finding that he attributes to higher intrinsic cognitive load. Kranch argues that novices should be given ample opportunity to practice lower-level fundamentals before they are taught higher-level schemas.

The studies from the 1980s that pointed to the relatively minor importance of misconceptions used simple imperative programs. It is not obvious to what extent the findings are generalizable to object-oriented programming, which has brought a slew of new concepts and misconceptions into many CS1 courses. Many OOP misconceptions are about the fundamental nature of pivotal concepts such as object and class (see Appendix A).

Conclusion: they all matter

My conclusion from the literature is that all five challenges – cognitive load, plan schemas, misconceptions, tracing skills, and notional machines – are significant. Furthermore, none of these challenges is independent of the others. Of particular interest to the present work is the fact that the other four challenges appear to be affected by the fifth: the novice’s ability to understand the notional machine. Understanding the role of the machine in program execution:

- prevents and corrects numerous misconceptions;
- serves as a basis for the formation of low-level schemas which in turn form the basis for the successful formation and application of higher-level ones;
- thereby indirectly reduces cognitive load as one is able to rely on increasingly complex problem-solving schemas, and
- is key to the skill of tracing programs, which is useful in both program authoring and comprehension and in finding bugs whether they be misconception-related or the results of high-level schema failures.

The notional machine is not a singular bottleneck responsible for students’ struggles, but it does appear to be one of several main sources of difficulty.

5.7 The internal cognition of individual minds is merely one perspective on learning

Cognitive psychology has made useful, empirically warranted contributions to the study of learning, but it has also come in for criticism from various quarters. Researchers from a number of overlapping camps – phenomenologists and phenomenographers, proponents of situated learning, qualitative researchers, philosophers, and neo-behaviorists, as well as cognitive psychologists themselves – have pointed out the limitations and possible fundamental problems of cognitive science. Before we turn to other traditions of educational research in the following chapters, let us consider some of the complaints concerning the cognitive tradition, or *cognitivism* as it is sometimes dubbed.[8]

Controversy 1: internal representations

Cognitivism tends to see reasoning in terms of the manipulation of symbols. A famous argument against internal mental representations is the problem of the *homunculus* (see, e.g., Marton and Booth, 1997, pp. 9–10): it is claimed that to operate on an internal representation of the world one needs something other than the representation itself – a so-called homunculus, or a "little human in the head". Following the same logic, the homunculus represents the representation internally, requiring another homunculus within it. This leads to an infinite regress. Many solutions to the problem have been proposed. Marton and Booth's self-statedly 'non-psychological' solution sidesteps the little human by choosing to research not mental representations at all but the relationships of people to phenomena as embodied in dialogue, actions, and artifacts. I will explain their phenomenographic approach further in Chapter 7.

Another way to answer the conundrum is to assume that the cognitive apparatus that operates on internal representations is innately capable of manipulating entities of the general form that internal representations have. That is, operating on an internal representation does not require a representation of the representation but only the ability to operate on representations of that general type. The traditional cognitivist way to phrase such an answer is to liken the human cognitive system to a computational device which contains both mental representations (data) and procedures that manipulate the representations (see, e.g., Markman, 1999, p. 13). This answer brings us to our second controversial theme.

Controversy 2: mind as machine

Cognitivism is associated with the idea that the human cognitive system resembles a general-purpose computer. Within this apparatus, information is stored within memory and processed algorithmically. Because of this analogy, cognitive science has been criticized for "vastly oversimplifying both the human mind and the human brain" (Westbrook, 2006), "for imposing severe constraints on potential descriptions and explanatory models" (Marton and Booth, 1997), and for committing so heavily to the analogy of a computer as to ignore consciousness and intuition (Searle, 1992; Dreyfus, 1992; Klein, 1999).

Perhaps the most fruitful way to address this point is to accept that the information-processing view can serve – as it has – to provide insights into the human mind, but that – like all analogies – it has its limits. Other perspectives are also needed.

Controversy 3: individual, context-free knowledge

Most cognitivists view reasoning primarily as a mental operation performed by individuals. Exploring the social situatedness of learning – more on which in the next chapter – has not traditionally placed especially high in the cognitivist agenda. According to proponents of some social theories of learning, cognitivists tend to ignore or pay too little attention to the relationship between individual thought and social context, and the way knowledge is shaped and problems are solved socially rather than individually. Researchers into situated cognition (cognition embedded in physical/social/cultural contexts) have been critical of the dominant role of internal memory in much of cognitivist research, arguing instead for a focus on situated perception and knowledge that is created on the fly, in activity, rather than retrieved from

[8]It is my impression that the term "cognitivism" is used more often by detractors of cognitivism than by its proponents. I do not use the word in a pejorative sense.

memory storage (see, e.g., Greeno, 1994; Anderson et al., 2000a, and references therein). An example of a moderate position is that of Anderson et al. (2000a), who seek to mediate the conflict. They sensibly point out that the individual and the social perspectives are not incompatible but provide two complementary views on the study of learning and thinking.

Controversy 4: content-independent cognition

A related issue is that many cognitivists seek to discover generally applicable principles and forms of representation (e.g., schemas) that explain activities in any field of knowledge and in any context. This universalist goal has been criticized by phenomenologists (e.g., Dreyfus, 1992) and phenomenographers (e.g., Marton and Booth, 1997), who prefer to investigate specific phenomena and people's relationships to specific phenomena. Such critics see generic cognitive psychology as flawed because the specific content that is processed is of fundamental importance. In seeking to differentiate their work from cognitive psychology, Marton and Booth (1997) argue that "the general can only be revealed through the specific" and that "ideas and principles need to be developed anew in specific contexts and contents of learning and teaching". This criticism does not acknowledge the fact that results in cognitive psychology, too, can arise from specific contexts, and sometimes apply only to a specific domain. For example, some of the work presented in this chapter and the previous one has "developed anew" general ideas from psychology, using programming education for context and content (e.g., programming plans and the roles of variables). Nevertheless, Marton and Booth's criticism serves as a reminder to moderate the eagerness for wide generalization that typifies cognitivist work.

Controversy 5: accessibility

Cognitivism is based on the notion that mental entities are accessible to researchers. Behaviorists – and others – have accused cognitivism of overestimating researchers' ability to elicit people's mental representations and analyze them objectively. A related claim is that in seeking to explain thought, cognitivists are as likely to invent one internal mechanism as another. To Uttal (2000, p. 12), "it seems that many cognitive psychologists are not deeply enough concerned with the fundamental issue of accessibility. Usually, the issue is finessed and ignored." He is concerned that the inaccessibility of mental processes leads to poor experiments with "embarrassingly transient" results that tend to be soon contradicted.

> *As I studied and reviewed the "high level" literature, I came to a rather surprising general conclusion. The reliability, durability, and presumably the validity of the data from the sample of experiments with which I was concerned seemed to evaporate. Data, as well as conclusions, seemed to last only for a few issues of the journal in which they had been published before some criticism of it emerged. Or, when the experiment was repeated, slight differences in the design produced qualitatively distinct, not just marginally different quantitative results. My summary statement on this issue still expresses my conviction that the high level, cognitive aspects of perception and other aspects of mentation are far less accessible than many mentalists would accept simply because the very database is so fragile.* (Uttal, 2000, p. 77)

The results of Uttal's review are worrying and must be taken seriously. Still, there also certainly exists evidence within cognitive psychology of the convergence of results. In some cases, mixed results may be later clarified by an improved, unifying theory; we have seen examples of this in Section 4.4.2, in which Rist's theory of schema formation helped make sense of earlier mixed results, and in Section 4.5, in which the discovery of the expertise reversal effect helped make sense of confusing results concerning several other effects of cognitive load.[9]

Controversy 6: dualism

Cognitivism has been tagged with the "original sin of dualism" by critics from different backgrounds (e.g., Uttal, 2004, 2000; Uljens, 1996; Marton, 1993; Marton and Booth, 1997). A dualist view of ontology

[9]Problems of the sort Uttal mentions nevertheless continue to concern cognitive load theorists, as discussed by Moreno (2006).

maintains that physical and mental objects are fundamentally disparate and 'inhabit different worlds' or consist of different substances. Dualism is commonly criticized, among other things, for failing to explain how it is possible for mind and matter to interact. Many cognitivists, however, denounce dualism and prefer a monist, materialist position in which the mental is viewed as reducible to physical phenomena. Another alternative to traditional dualism is *property dualism*, which draws a dichotomy not between substances of mind and matter but rather between objective physical and subjective mental phenomena that arise out of a single physical matter but are fundamentally distinct in character from each other. These philosophical positions, and others, have been reviewed, e.g., by Searle (1992, 2002), who himself promotes *biological naturalism*, the "fairly simple and obvious" solution that mental phenomena are physical but additionally have higher-level, emergent characteristics that are irreducible to physics.

Mind–matter dualism has been a source of bother and employment for philosophers for a long time, without a generally accepted solution in sight.

Controversies 7 and 8: quantitative "positivist" research

Finally, since cognitivism has traditionally been characterized by quantitative, experimental, "positivist"[10] research, it has come in for criticism from various quarters that are opposed to those paradigms. "Positivism" and quantitative research have been criticized on ontological, epistemological, and practical grounds by qualitative researchers (e.g. Lincoln and Guba, 1985; Patton, 2002) and others. Claimed weaknesses include:

- a reliance on a correspondence theory of truth, that is, the idea that facts correspond to a one real world,
- a naïve belief in objectivity; ignoring the context-, subject- and theory-dependence of facts,
- equating the natural and social sciences,
- the belief that (all) science is transcultural,
- losing richness of data and validity of research through the reductionist attempt to operationalize complex phenomena into simple variables,
- being good only for hypothesis testing rather than exploration,
- a focus limited to laws, causes, and predictions as opposed to multiple interpretations,
- ignoring the societal embeddedness of phenomena,
- invalid results that ignore the humanness of research subjects and researchers,
- the idea that inquiry can be value-free, or at least should be as value-free as possible,
- vain attempts at wide or universal generalization, and
- an obsession with the reproducibility of experiments.

Obviously, the extent to which these criticisms are applicable to particular cognitivists and studies within cognitive psychology varies. Controlled laboratory experiments in the quantitative tradition do have a credible track record of success in motivating educational reform (see, e.g., references in Atkinson et al., 2000). They are also hardly the only form of cognitivist research. Many of the CER studies in the cognitivist tradition that I have reviewed in this and the previous chapter are actually qualitative in character (e.g., much of the work on identifying programming misconceptions and plans).

A detailed review of this long and complicated debate is well beyond the scope of this thesis. For the present, I will only note that cognitivist research and its competing paradigms have different strengths and weaknesses. In my own empirical work for this thesis in Part V, I take a pragmatist approach that mixes paradigms.

[10]The word "positivism" has been defined in many different ways, and is often not very well defined at all when employed as a bludgeon (Phillips, 2004). I use it here, in quotes, to refer loosely to various criticized views that have been assigned this label. Typically, "positivists" are associated with the view that all scientific knowledge must be founded on value-free empirical testing of hypotheses.

Chapter 6

Constructivists Say: Knowledge is Constructed in Context

The central tenet of the educational paradigm known as *constructivism* is that people actively construct knowledge rather than passively receive and store ready-made knowledge. Knowledge is not taken in as is from an external world, and is not a copy of what a textbook or teacher said. Instead, knowledge is unique to the person or group that constructed it – constructivists differ among themselves as to whether individual minds or social groups (or both) are the constructing agents. From these premises, constructivist thinkers have derived pedagogical recommendations which tend to promote active, learner-centered education.

Proponents of constructivism often claim a positive, even revolutionary impact on education; skeptics either point at perceived flaws in constructivist reasoning or dismiss constructivism as not so much a revolution as a rephrasing of prevalent wisdoms. These issues notwithstanding, many critics, too, agree that the currently extremely influential constructivist movement has done good by bringing epistemological issues to the forefront of educational discussions, by advancing the increasingly widespread recognition of the social aspects of learning and the importance of learners' prior knowledge, and by emphasizing active learning (Phillips, 1995, 2000).

To get an overall feel, let us begin with a list of selected constructivist claims (my phrasings based on von Glasersfeld, 1982; Phillips, 1995, 2000; Steffe and Gale, 1995; Greening, 1999; Ben-Ari, 2001a; Larochelle et al., 1998; Rasmussen, 1998; Anderson et al., 2000b; Patton, 2002; Kirschner et al., 2006; Tobias and Duffy, 2009). These claims range from the epistemological to the pedagogical. Some are considerably more controversial than others. Not all of the claims are equally, or at all, accepted by all constructivists.

1. Knowledge is constructed by learners, it is not (and cannot be) transmitted as is.
2. The knowledge that people – or groups of people – have is different from the knowledge of other people who have ostensibly 'learned the same thing'.
3. Knowledge construction takes place as prior knowledge interacts with new experience.
4. Knowledge is not derived from, is not about, and does not represent an external, observer-independent natural reality, but an experiential, personal (or socially shared) 'reality'.
5. If an extraexperiential reality exists, it is not rationally accessible.
6. There is no objectively correct or incorrect, true or untrue knowledge, only knowledge that is more or less viable for a purpose.
7. The social and cultural context mediates the construction of knowledge.
8. Effective learning features the learner as an intellectually active constructor of knowledge; good teaching means facilitating and motivating such construction.
9. Allowing students to leverage their prior knowledge is key to good teaching.
10. Learners should be presented with minimal information and allowed to discover principles and rules for themselves.
11. Hands-on learner-directed exploration is an effective form of learning.
12. Effective learning requires complex, authentic learning situations. Learners should solve ill-structured, open-ended problems similar to those that experts solve.

13. Effective learning has a social dimension, e.g., groupwork.
14. Education should not impose learner-independent norms or goals; the learner is in charge.
15. It is not possible to, or does not make sense to, apply standard evaluations to assess learning.

The following sections elaborate on these claims. Section 6.1 below gives an overview of the main types of constructivism. In Section 6.2, I describe in more detail some of the main features of constructivist epistemology. The discussion in that section revolves largely around the more extreme forms of constructivism; I outline a moderate position in Section 6.3. Section 6.4 provides a brief overview of constructivist pedagogy. Sections 6.5 and 6.6 introduce two relatives of constructivism, conceptual change theory and situated learning, respectively. In Section 6.7, we get to computing education and the impact of constructivism on CER. Finally, Section 6.8 reviews some of the not insignificant criticism that constructivism has drawn.

6.1 There are many constructivisms

The roots of constructivism go well back in time. John Locke was an early influence, and Immanuel Kant is sometimes mentioned as one of the first real constructivists. Many of John Dewey's progressivist ideas are echoed in present-day constructivism. Especially through the seminal work of Jean Piaget and Lev Vygotsky, constructivism has become a major force in education since the 1900s. Subsequently, an ever increasing number of different interpretations of constructivism have come into existence. Gunstone (2000, p. 254) even suggests on the basis of an internet search that "there are no areas of human activity to which the label "constructivist" is not currently being applied in some form!". Phillips (2000, p. 7), who sets out to describe the "constructivist landscape", characterizes the view as "nightmarish" in reference to the daunting task of making sense of the myriad interrelated, sometimes complementary, sometimes contradictory constructivisms in the literature.

6.1.1 Nowadays, everyone is a constructivist(?)

It is nigh on impossible to find a present-day educational researcher that 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, p. 7). However, we vary in how constructivist we are, and in how we are constructivists.

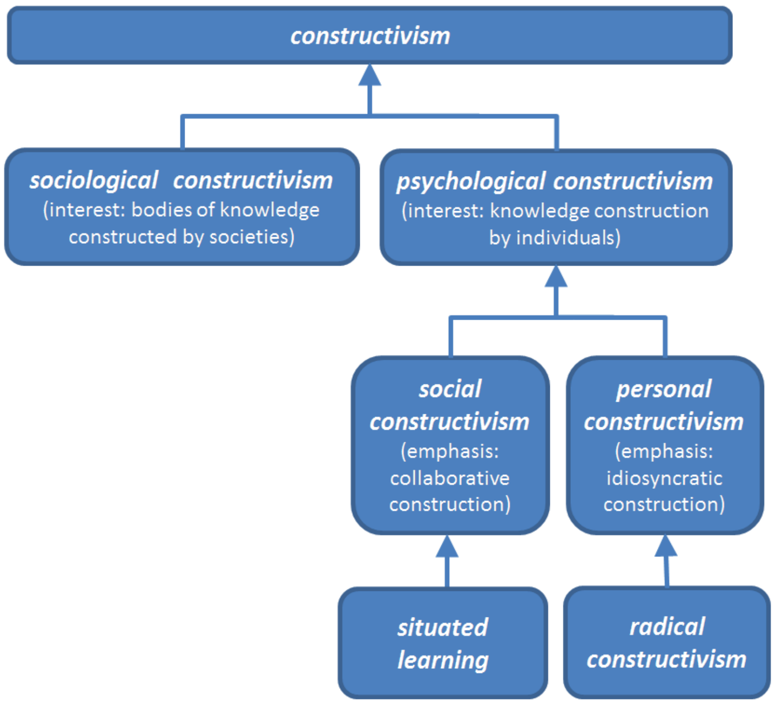
How constructivist are you?

Phillips (1995, 2000) characterizes constructivists by considering their positions along a dimension that is essentially a measure of how much one emphasizes the idea of knowledge as a construction. That is, to what extent does one consider knowledge to be a reality-independent, perspectival human construction as opposed to something that is dictated by nature itself, i.e., by what is real? In this scheme, those who espouse extreme versions of constructivism are at one end, and non-constructivists at the other, with the middle of the constructivist landscape occupied by "a marsh of wishy-washy scholars". Wishy-washy constructivists prefer watered-down, non-radical versions of constructivist epistemology.[1] Like Phillips, I, too, find myself in these wetlands. (Section 6.8 discusses the reasons.)

A popular buzzword

Constructivism has become a common buzzword that features in numerous publications whose basis in constructivist learning theory or epistemology is debatable. A colleague of mine recently made what I think was a more-than-half-serious comment to the effect that "Nowadays, you may need to put some

[1]This should not be interpreted as being wishy-washy about the importance of epistemological questions.



constructivism
sociological constructivism (interest: bodies of knowledge constructed by societies)
psychological constructivism (interest: knowledge construction by individuals)
social constructivism (emphasis: collaborative construction)
personal constructivism (emphasis: idiosyncratic construction)
situated learning
radical constructivism

Figure 6.1: A classification of constructivisms, roughly based on Phillips (1995, 2000).

learning theory into CER papers. But don't worry – just mention constructivism and the reviewers will be happy."[2]

Matthews (2000) observes that some educationalists apply the term "constructivism" to any non-behaviorist learning theory, or to any view that recognizes social, cultural, and historical aspects of cognition. Many practitioners are not aware of the differences between forms of constructivism, or indeed about the underlying epistemological and ontological assumptions of whichever form of constructivism they profess allegiance to (Prior McCarty and Schwandt, 2000). These practitioners are not decidedly wishy-washy; they are uncommitted and may retain a traditional epistemology merely by default.

These developments make the term increasingly ambiguous and hinder the debate on the deeper issues involved. This has understandably made many authors unhappy, as they would prefer to reserve the term for epistemologically explicit variants of constructivism. This group includes critics of constructivism, wishy-washy constructivists, and proponents of extreme positions such as Ernst von Glasersfeld, who coined the term *radical constructivism* specifically to differentiate his views from "naïve constructivism" that does not question traditional epistemology.

6.1.2 Constructivism comes in a few main flavors

Buzzwords aside, there are dozens of variants of epistemologically grounded constructivism (see, e.g., Matthews, 2000; Ernest, 1995, and references therein) that I will not even attempt to cover. However, constructivisms can be roughly grouped by their main emphasis (Phillips, 1995, 2000). Figure 6.1 is a breakdown of some of the main branches of constructivism.

A constructivism is typically concerned with one (or sometimes both) of two things: 1) sociological issues: the construction of bodies of knowledge by societies at large (e.g., bodies of scientific knowledge), or 2) psychological issues: the construction of individuals' knowledge. Constructivist views on sociology have been seminally influenced by Kuhn (1962) and taken to an extreme by the so-called Edinburgh strong

[2]It may not be as simple as that. This is amusingly illustrated by Derry's (1996, p. 172) anecdote, in which constructivist journal editors pick on an eminent Immanuel Kant Professor of Psychology for his use of words such as "say", which supposedly suggest a knowledge-transmission view of learning, despite the fact that the professor and his academic ancestors "were constructivist when constructivism was not cool".

programme led by David Bloor (see, e.g., Phillips, 2000). They are of primary interest to sociologists and philosophers of science but less pertinent for my present purposes. I will largely focus on *psychological constructivism*, which applies more directly to education.

Psychological constructivisms can be divided into *personal constructivisms* and *social constructivisms*[3]. Personal constructivists emphasize the idiosyncratic construction of individual knowledge. Social constructivists instead emphasize the importance of the social and cultural nature of individuals' knowledge construction. Radical constructivism, the influential extremist version of personal constructivism fathered and fronted by von Glasersfeld (e.g., 1982, 1995, 1998), is worthy of separate mention as it is a staple of constructivist literature that has been claimed to represent "the state of the art in epistemological theories for mathematics and science education" (Ernest, 1995, p. 475). *Situated learning*, which also appears in Figure 6.1, is a specific form of social constructivism; I will say more about it in Section 6.6.

6.2 Knowledge is an (inter-)subjective construction

The subsections below briefly outline constructivist positions on learning, epistemology, and educational goals.

6.2.1 We learn by combining prior knowledge with new experience

Two ideas are central to most constructivist views of learning (Howe and Berv, 2000).

1. The learning of something new builds on the learner's existing knowledge and interests that learners bring into the learning context.
2. Learning is the construction of new understandings through the interaction of the existing knowledge and new experience.

These ideas are very generic. Precisely what kinds of processes take place as prior knowledge interacts with new experience, and what kinds of conceptual structures are constructed as a result, are not something that constructivists agree on. Some constructivists do not even look for a more specific description, sometimes because they hold the view that not only is everyone's knowledge different, but also the way in which construction takes place and the nature of the resulting conceptual structures are idiosyncratic and cannot be universally described. So-called cognitive constructivists seek to reconcile constructivist ideas with the information-processing views of cognitive science (e.g., Derry, 1996) and use, e.g., schema theory and mental model theory to explain construction. Conceptual change theories (Section 6.5 below) are also espoused by some constructivists.

While the above principles of learning are readily accepted by many who do not call themselves constructivists, constructivist views on epistemology are not all equally uncontroversial.

6.2.2 Do we all have our own truths?

The traditional, commonsense view of ontology is that there exists a real world that is independent of our understandings of it. Traditional, commonsense epistemology suggests that knowledge reflects an existing reality, and the truth value of a proposition depends on its correspondence with said reality.

Constructivism questions the very notions of objective knowledge and truth. For the full-blooded constructivist, an objective reality is unknowable, if there is one at all. Von Glasersfeld makes a distinction between ontological reality and each individual's lived, experiential reality. Only the latter is relevant:

> *for the constructivist, 'existence' must not be interpreted ontologically but epistemologically. That is to say, it refers to the realm of cognitive operating and structuring, and not to the realm of 'being' in the traditional sense.* (von Glasersfeld, 1982)

[3]The label "social constructivism" is variously attached to one or both of what I call "social constructivism" and what I call "sociological constructivism" (and some authors do write simultaneously on both sociological and psychological issues). In addition, the term "social constructionism" is used in a number of ways. I will steer clear of the latter term and use "social constructivism" to mean those constructivisms that have an interest in psychology and emphasize social aspects of individuals' knowledge construction.

Social constructivist reasoning takes a different path, but the conclusion is the same; for instance, Gergen describes his (extreme) social constructivism as ontologically "mute" (Prior McCarty and Schwandt, 2000).

No objective knowledge

According to radical constructivists, knowledge is non-foundational and non-representational. It is not a copy, a reflection, or a representation of an external, observer-independent ontological reality. Instead, all that we call knowledge is derived – constructed – from the lived, experiential world of the individual mind (e.g., von Glasersfeld, 1998). Social constructivists disagree with radical constructivists' emphasis on the individual, but share with them a key epistemological claim. According to extreme proponents of both branches of constructivism, knowledge refers only to the individual's (personal constructivism) or the social group's (social constructivism) constructions.

For the radical constructivist, knowledge is idiosyncratic and "no grounds exist for believing the conceptual structures that constitute meanings or knowledge are held in common among different individuals [...] and one can never say whether or not two people have produced the same construct" (Howe and Berv, 2000). For the social constructivist, on the other hand, knowledge *is* shared; sharing is in fact what creates knowledge, which is defined as "temporal locations in dialogic space – samples of discourse that are accorded status as "knowledgeable tellings" on given occasions" (Gergen, 1995, p. 30). According to this interpretation, whether something counts as knowledge does not depend on an external reality but on whether the participants in a 'language game' jointly grant that something the status of knowledge. Any knowledge so constructed by those participants is particular to that game and incommensurable with others' knowledge.

No truth, only viability

Since individuals' or groups' knowledge can be compared neither to that of other individuals or groups, nor to an ontological reality, constructivism in its more extreme forms abandons the notions of correct and incorrect, true and false. All knowledge is individually or socially constructed, imperfect, and fallible.

Many constructivists think of knowledge less as something that is possessed than as the ability to accomplish something in a particular context. Von Glasersfeld (1995) exhorts: "Give up the requirement that knowledge represents an independent world, and admit instead that knowledge represents something that is far more important to us, namely what we can *do* in our *experiential world*" (pp. 6–7, original emphasis). At the social constructivist extreme, being knowledgeable is conceived as a status given by participants in a language game to one of their number: "For the purposes of the conversation, 'I know' when I speak in ways that enable you to treat me as if I know, and vice versa. We successfully generate dialogue because we are mutually accorded the status of knowledgeables across time." (Gergen, quoted by Prior McCarty and Schwandt, 2000, p. 58)

This procedural view of knowledge affords the notion of *viability*, the radical constructivist's stand-in for truth. Viability refers to the pragmatic adequacy and task-relevance of knowledge, as determined by the knower.

> *Simply put, the notion of viability means that an action, operation, conceptual structure, or even a theory, is considered "viable" as long as it is useful in accomplishing a task or in achieving a goal that one has set for oneself. Thus, instead of claiming that knowledge is capable of representing a world outside of our experience, we would say* [...] *that knowledge is a tool within the realm of experience.* (von Glasersfeld, 1998)

For instance, knowledge of a programming language that allows a particular bug-free program to be written is viable for that task. The same knowledge may not be viable for the task of writing a different program. Different programmers have possibly vastly different conceptual structures that may or may not match the defined standard of the programming language and its technical implementation. However, for the radical constructivist, such judgments are beside the point as long as one's knowledge is viable for one's intended purposes. Only when one's knowledge is inadequate is it necessary to construct a new understanding.

Context-dependence of knowledge

The constructivist claim goes beyond stating that the viability of knowledge – that is, the usefulness of the conceptual structures we have constructed – is context-bound. In the constructivist view, the content and the very nature of our conceptual structures depend on the contexts in which we created them. Knowledge, then, is not only specific to each individual or group, but to learning contexts as well.

Many constructivists emphasize the difficulty of transferring knowledge from one context to another. Achieving viability in one context is barely any guarantee of viability in another kind of context. An example is given by Greening (1999, p. 50), who discusses the knowledge students construct in traditional lectures. According to Greening, such knowledge may only be useful (viable) in the "fairly artificial and constrained" educational system itself – in an exam, for instance – rather than in real-life situations that call for it.

These views on epistemology have led constructivists to oppose traditional education.

6.2.3 Is it impossible to standardize any educational goals?

In traditional views of education, teachers – as possessors of more true knowledge than learners – are in a position to decide what kind of knowledge learners should acquire. The job of learners is to receive true knowledge from their teachers and from books. Whether they succeed can be summatively measured. Constructivists – personal and social alike – challenge this view, questioning the primacy of the teacher, the setting of educational goals that are the same for each student, and the meaningfulness of standardized assessment.

From the radical constructivist perspective, the knowledge of a teacher or a curriculum-writer speaks merely of their own experiential reality; social constructivists like Gergen likewise reject claims of teachers' authority and consider teachers and learners to be equal participants in the social constitution of knowledge (see, e.g., Gergen, 1995; Prior McCarty and Schwandt, 2000). Educators are not in a privileged position that would justify externally setting educational goals for others; instead, the goals of education must emerge from the learners themselves, their interests, and their interactions with new experiences and with other people. To the fully committed constructivist, knowledge is not content that exists independently of the learner; correspondingly, constructivist education is less about covering content than it is about developing a range of views on interesting matters, with everyone involved serving as both learner and teacher.

Reassessing assessment

The way educational goals elude standardization naturally carries over to the constructivist view of assessment. Many constructivists maintain that since knowledge and viability are context-dependent, assessment should be based on fitness for purpose rather than correspondence with 'fact'. Furthermore, since the goals of education are largely learner-set, and viability learner-determined, it is the learners themselves who are primarily responsible for judging whether they have reached their goals (Greening, 1999). In the social perspective, viability is assessed via social negotiation that compares one's knowledge with that of others, with the learning community serving as a test bench.

This view of assessment is in sharp contrast with the traditional objectivist view. It is clearly not accepted by all constructivists. Larochelle and Bednarz (1998), who adopt a strongly constructivist stance, are not satisfied with those self-proclaimed constructivist educators who nominally accept the fact that learners construct their own knowledge, but fail to take constructivist thinking to its conclusion; it is only accepted that many roads lead to Rome, but they all still lead to Rome. To extend Larochelle and Bednarz's metaphor, the radical constructivist view is that the Eternal City might not even exist; rather, learners construct their own experiential Romes. If the learner is satisfied that they have got a satisfactory peek at the Coliseum, then they have reached a viable destination (for now).

Figure 6.2: A radical destructivist (de Lipman, 1897).

6.3 But a wishy-washy constructivist also acknowledges reality

'Wishy-washy' constructivists generally accept some form of the realist proposition that a world exists, and search for a reasonable epistemological middle ground that avoids completely jettisoning the concepts of truth and reality while at the same time allowing for subjectivity. According to Phillips (1995, p. 12), "any defensible epistemology must recognize [...] the fact that nature exerts considerable constraint over our knowledge-constructing activities, and allows us to detect (and eject) our errors about it". One possible basis for such an epistemology was provided by Popper (1972), who suggested an ontology of three interacting 'worlds'. World$_1$ is a world of physical objects. World$_2$ consists of psychological, subjective experiences constructed by people of the other two worlds. World$_3$, which is dependent on the first two worlds, contains the products of the human mind, such as art, mathematics, values, and science. The latter exists as a result of human activity and is maintained by it, rather than existing as an ideal world that transcends World$_1$ and World$_2$ as Plato would have it. Popper's is essentially a moderate constructivist view (as discussed, e.g., by Harlow et al., 2006; Niiniluoto, 1980).

Regarding goals and assessment, a wishy-washy constructivist might say that while everyone has their own Rome and their own Appian Way, those constructions and their usefulness are affected by the location of the actual city. Ideally, experiential Romes should be reasonably close to the real one, with all roads leading at least to its near vicinity.

6.4 For better learning, constructivisms tend to encourage collaboration in authentic contexts

Where the traditional teacher-centered direct instruction is the norm in teaching, there exists a paradox: "The more you teach, the less they learn." [...] *the time the teacher spends on direct instruction takes away from the dialogue, negotiation, debate and assessment that could take place between students.* [...] *Learning in its constructivist sense therefore requires us*

> *to reduce teaching that is based on direct instruction and to emphasize the social interaction between learners and personal reflection.* (Sahlberg, 1996, p. 79, my translation)

Constructivist pedagogy encourages teachers to engage the learner in order to to help them construct their own knowledge and to take individual differences and prior knowledge into account. Constructivism is generally associated with a move away from traditional pedagogies in which the learners' role is relatively passive. Some constructivists vehemently oppose lectures in particular. Decontextualized teacher-given practice tasks devoted to narrow topics (such as those found in many textbooks) are also criticized as 'drill and kill'. For many constructivist teachers, engagement implies hands-on tasks.

Even bearing in mind that there are different varieties of constructivism, a general trend can be observed in that constructivists tend towards learning environments that are situated in (or emulate) complex, realistic contexts, and that require students to solve problems. Social constructivists in particular (but personal constructivists as well) emphasize social interaction in learning environments: students collaborate, negotiate, and compare their ideas with those of the other students and the teacher. Some constructivists advocate using ill-structured problems that students need to make sense of on their own or in groups, with limited guidance from teachers. This form of constructivist pedagogy can be considered as a form of *discovery learning* (e.g., Bruner, 1979; Papert and Harel, 1991), as it leaves students to discover solutions on their own – this is a hotly debated issue that I will return to in Section 6.8 below.

Problem-based learning (PBL) is a currently popular pedagogy that exemplifies many constructivist principles (see, e.g., Savery and Duffy, 1995; Norman and Schmidt, 1992). PBL is driven by substantial, realistic problems that groups of learners try to solve with limited help (and no ready-made solutions) from teachers and tutors:

> *The principal idea behind problem-based learning is that the starting point for learning should be a problem, a query or a puzzle that the learner wishes to solve. . . Problem-based courses use stimulus material to engage students in considering a problem which, as far as possible, is presented in the same context as they would find it in 'real life;' this often means that it crosses traditional disciplinary boundaries. Information on how to tackle the problem is not given, although resources are available to assist the students to clarify what the 'problem' consists of and how they might deal with it. Students work cooperatively in a group or team with access to a tutor who is often not an expert in the field of the particular problem presented, but someone who can facilitate the learning process.* (Boud and Feletti, quoted by Kay et al., 2000)

PBL seeks to activate students' prior knowledge in the search for solutions to meaningful problems, to encourage the sharing of cognitions amongst learners and (self-)explanation of solutions, and to foster the development of self-directed learning skills (Norman and Schmidt, 1992).

Inquiry-based learning (IL) is another popular constructivist approach to teaching that is closely related to PBL – indeed, it is often almost indistinguishable from it (Hmelo-Silver et al., 2007). Whereas PBL arises from the diagnosis problems of medical education, IL draws on the scientific method: students form questions, then collect and analyze data to answer them.

Despite the emphasis on hands-on activity and methods such as PBL and IL, many constructivists are careful to point out that using a particular method has no intrinsic value – a lecture can be constructivist if it succeeds in engaging learners to construct viable knowledge, just as richly contextualized learner-driven groupwork fails when inappropriately used.

6.5 Conceptual change theories deal with the dynamics of knowledge construction

This section briefly introduces a group of learning theories related to constructivism. Depending on interpretation, these *conceptual change theories* can be said to be either versions of constructivism or separate theoretical frameworks that have considerable commonalities with forms of constructivism. They certainly fall under the broad usage of the term “constructivism” (see Section 6.1).[4]

Theories of conceptual change characterize the kinds of conceptual structures that people have, and when and how those structures change as we learn. Of particular interest to conceptual change theorists are people’s intuitive, naïve (as opposed to scientific) conceptions of phenomena. Problematically for the educator, as well as the learner, the naïve knowledge that arises from everyday experience is notoriously resilient to change.

Conceptual change theories generally fall into one of two ‘families’, termed by Özdemir and Clark the *knowledge-as-theory* and *knowledge-as-elements* perspectives (Özdemir and Clark, 2007; diSessa, 2006).[5]

Knowledge-as-theory

One well-established family was seminally influenced by the work of Posner, Strike, and McCloskey (and their colleagues) which in turn draws on Kuhn’s paradigm shifts and the Piagetian notion of accommodation (see, e.g., Posner et al., 1982; Strike and Posner, 1985).

Knowledge-as-theory perspectives liken individuals’ knowledge – even naïve knowledge – to scientific theories. What this means is not that individuals are necessarily aware of their knowledge in the same way as scientists are of their theories, or that they seek to verify it as scientists do. The similarity to scientific theory is merely that even naïve knowledge is considered to be organized and coherent, to allow consistent application across contexts, and to be replaceable by a new ‘theory’ when it is found lacking.

When a learner experiences something new, they relate it to their existing ideas and judge its consistency with them. As existing concepts form coherent structures, they are not independent of each other; changes to one concept require changes in others. The difficulty of getting rid of misconceptions is explained by this interdependency: making broad, complex changes to one’s existing overall ‘theory’ – misconceived or otherwise – is a daunting task. However, when sufficient conditions are met, a revolutionary change akin to a scientific paradigm shift (Kuhn, 1962) occurs in the conceptual ecology as one’s old ‘theory’ is replaced by a new one. A revolution is not necessarily abrupt, but may occur gradually (Posner et al., 1982). Such a revolution does not result in anarchy; on the contrary, it restores the conceptual ecology to a state where one’s new ‘theory’ accommodates the disruptive new ideas that previously failed to fit in.

From a practical point of view, the meat of knowledge-as-theory perspectives is in what they say about the conditions under which conceptual revolutions occur. Posner et al. (1982) detail four primary conditions for conceptual change: 1) dissatisfaction with existing conceptions; 2) a sufficient minimal understanding of the new conception; 3) the apparent plausibility of the new conception (it must seem on the surface as if it could solve problems that one’s existing conception does not), and 4) an apparent possibility that the new conception will be fruitful in leading to new insights beyond present needs. Many learning events start with anomalies that lead to dissatisfaction and increase the plausibility and apparent usefulness of new conceptions. Anomalies “provide the sort of *cognitive conflict* (like a Kuhnian state of "crisis") that prepares the student’s conceptual ecology for an accommodation” (Posner et al., 1982, p. 224, my italics). From a knowledge-as-theory perspective, teachers should foster cognitive conflict that leads learners to confront and supplant their misconceptions.

Knowledge-as-elements

Another family of conceptual change theories, championed by Andrea diSessa, lies on the other side of the “fault line between coherence and fragmentation” (diSessa, 2006). According to these theories, naïve knowledge is not coherent and consistent but consists of a collection of quasi-independent, often tightly context-bound elements that form a loosely connected structure. In diSessa’s influential knowledge-as-elements theory, these low-level elements of naïve knowledge are called *p-prims* (phenomenological primitives). A p-prim is a minimal abstraction of common phenomena that is – for whoever experiences

[4]I have placed this discussion of conceptual change theories in this chapter on constructivism. This body of work also intersects significantly with cognitive psychology, as indeed does constructivism in general (assuming a broad definition of the latter term).

[5]In the literature, it is not rare for the term “conceptual change theories” to be used to refer exclusively to what I call knowledge-as-theory perspectives.

it – intuitive and self-evident. An example of a p-prim in naïve physics is the notion of 'bouncing': as a smaller object comes into impingement with a large or otherwise immobile other object, the smaller object will recoil (diSessa, 1993). 'Bouncing' is a common-sense equivalent of a physical law: it helps explain other phenomena but does not itself, as a primitive, require explanation.[6]

Knowledge-as-elements perspectives emphasize context-dependence. There is no overarching theory-like structure that allows for generalization. As the learner forms a new idea, it becomes an element in the learner's conceptual ecology, associated with a particular context. It is possible for a learner to hold several contradictory understandings of a phenomenon, each particular to a different context. Instead of having a single consistent theory for the many physical phenomena governed by $F = ma$, for instance, one may have entirely distinct p-prims for bouncing and throwing, and even for throwing objects of a certain shape.

Marton (1993) provides a nice summary of diSessa's view of learning:

> *This view of characterizing what it takes to learn physics contradicts other characterizations in which the transition between naive and scientific physics has been seen in Kuhnian terms as the replacement of one world view with another.* [...] *The picture presented by diSessa is more evolutionary than revolutionary: Scientific physics evolves from naive physics more by organization than by reorganization, more by structuring than restructuring.* (p. 228)

According to knowledge-as-elements perspectives, it is context-specificity that makes misconceptions resilient to change; instruction intended to correct a misconception may simply add a parallel understanding rather than replace the old. However, some proponents of knowledge-as-elements maintain (from an explicitly constructivist position) that rather than being a problem, misconceptions are a useful or even necessary basis for further learning (e.g., Smith et al., 1994). Although Smith et al. agree with knowledge-as-theory perspectives that states of cognitive conflict are "certainly desirable and conducive to conceptual change" (p. 22), they oppose the idea of *confronting* misconceptions and trying to replace them with correct understandings. The challenge of learning from such a knowledge-as-elements perspective is not to replace the learner's naïve misconception-ridden understanding with another – there is no well-organized, general structure to replace! – or even to replace individual misconceptions. Instead, the goal is to help the learner to organize their knowledge elements, to find borders for the rules they have constructed so as not to over- or under-generalize, to select the most productive ideas and refine them, and to observe similarities across contexts. The elements of naïve knowledge – misconceptions included – are the raw materials for such processes, which, if learning is successful, evolve into a theory-like scientific understanding.

Recent developments have seen knowledge-as-theory and knowledge-as-elements move somewhat closer to each other, with researchers suggesting that both evolutionary and revolutionary changes are important during learning and that both families of conceptual change theory can inspire improvements in pedagogy (see, e.g., Özdemir and Clark, 2007; Hammer, 1996).

6.6 Situated learning theory sees learning as communal participation

This section introduces another theory that is a relative or form of constructivism, depending on interpretation. The theory of *situated learning* conceives of learning as an intrinsically contextualized process of social participation in a community (Lave and Wenger, 1991). Although it is not always phrased in such terms, situated learning theory is based on a social constructivist epistemology in which knowledge exists within a community rather than within individuals.

Ben-Ari (2004, pp. 86–87) primes us on the lexicon of situated learning, as defined by Lave and Wenger:

> *The learner is considered to be a participant within a* community of practice *(CoP). Learning occurs by a process of apprenticeship called* legitimate peripheral participation *(LPP): (a) the learner* participates *in a community of practice, (b) the learner's presence is* legitimate

[6]Although they were concerned with societal phenomena rather than physical ones, and their approach was less theoretical, Run-D.M.C. (1984) might as well have been rapping about p-prims as they cautioned would-be inquirers: "Don't ask me, because I don't know why, but it's like that, and that's the way it is."

in the eyes of the members of the community, and (c) initially, the learner's participation is peripheral*, gradually expanding in scope until the learner achieves full-fledged membership in the community.*

Situated knowledge

The place of knowledge is within a community of practice. (Lave and Wenger, 1991, p. 100)

Situated learning theorists, like constructivists in general, emphasize the contextualization of knowledge and difficulties in transfer. Knowledge is seen as located – situated – in the lived world of concrete practice in which it is put to use, and tied to that cultural, social, and physical context. All knowledge is situated in some way or another, as "a community of practice is an intrinsic condition for the existence of knowledge" (ibid., p. 98). Conversely, the importance of individuals, even the recognized 'masters' or teachers within a community, as bearers of knowledge is of lesser importance: "mastery resides not in the master but in the organization of the community of practice of which the master is part" (ibid., p. 94). This is a quintessentially social-constructivist view.

Learning is participation

According to Lave and Wenger, learning occurs as the learner participates in the practice of a community in some useful – although initially minor, even menial – capacity.

Participation in the cultural practice in which any knowledge exists is an epistemological principle of learning. The social structure of this practice, its power relations, and its conditions for legitimacy define possibilities for learning (i.e., for legitimate peripheral participation). (Lave and Wenger, 1991, p. 98)

Participation does not mean mere observation and imitation of what the senior members of the community do, but taking an active, social, useful, collaborative role within the community that enables the learner to take part in the community's everyday activities, becoming familiar with all its aspects, including "who is involved; what they do; what other learners are doing; [...] how masters talk, walk, work, and generally conduct their lives; [...] how, when, and about what old-timers collaborate, collude, and collide, and what they enjoy, dislike, respect, and admire" (ibid., p. 95). The other members of a community are exemplars of social roles and professions that learners can aspire to. The learner's gaining of expertise is embodied in changes that take place within the community: the learner gradually progresses from the periphery to ever more central roles, eventually reaching mastery.

The problem of decontextualized schooling

Since learning and its goals are characteristics of a community of practice, Lave and Wenger (1991, p. 97) advise against analyzing curricula "apart from the social relations that shape legitimate peripheral participation". According to them, engaging in the practice of a community is not merely the future goal of a learning process (as it is traditionally seen in the context of schooling that prepares students for later participation in other, professional communities) but is an intrinsic aspect of learning. In their seminal work, Lave and Wenger (1991) deliberately skirt around questions of pedagogy, but, as a whole, the situated learning movement is severely critical of the notion of teaching knowledge that is abstracted from authentic practice, as "the organization of schooling as an educational form is predicated on claims that knowledge can be decontextualized" (ibid., p. 40).

In the situated view, all learning is situated, and useful results are only likely to be obtained by situating learning in the community of practice in which the skills that are learned are genuinely needed. Participating in a traditional school community, for instance, will only (or mostly) foster skills that are needed in that inauthentic setting; "there are vast differences between the ways high-school physics students participate in and give meaning to their activity and the way professional physicists do" (ibid., p. 99) – for the purpose of becoming physicists, they are participating in the wrong community of practice.

The challenge of the educational enterprise, from the situated learning point of view, is to find ways for learners to become members of meaningful communities of practice and to engage in legitimate, useful participation right from the start. The key to this is access for newcomers to all that membership of the community entails: ongoing activity, old-timers and other members, information, resources, and opportunities for participation. Lave and Wenger point out that although this is essential, it is always problematic all the same.

6.7 Constructivisms are increasingly influential in computing education

We feel that constructivist principles are exerting strong influences on professional practice in computer science education. However, constructivism – as a body of theory – maintains a relatively low visibility within our discipline. [. . .] [Constructivism] *has spawned a host of principles for good practice that have propagated independently of theoretical roots.* (Greening and Kay, 2001)

According to Greening (1999), "a constructivist future for computer science education is almost assured". Be that as it may, the influence of constructivisms has indeed become more explicit in both computing education and CER since the late 1990s.

Compared to its impact on mathematics and science education, the visible influence of constructivisms in computing education has been minor. Certainly, there are famous initiatives in programming education that are based on a kinstheory of constructivism that goes by the name of *constructionism* (Papert and Harel, 1991). The 'toolkits-for-learning-by-creating-for-sharing' spirit of constructionist education is embodied in the Logo and Smalltalk programming languages, and, more recently, Scratch (MIT Media Lab, n.d.), Squeak (Ingalls et al., 1997), Lego Mindstorms (Lego Group, n.d.), and Media Computation (Media Computation teachers, n.d.). Equally certainly, many computing education practices are compatible with many forms of constructivism. For instance, project-based teamwork on open-ended problems that are intended to be fairly authentic is relatively common in programming education.

However, it is only in recent years that constructivism has become a clearly visible force in CER. The use of "constructivism" (as a buzzword, or as a genuine theoretical framework) is increasingly common and constructivism in some form is now used to justify and inform pedagogical interventions in programming education (Hadjerrouit, 1998; Gray et al., 1998; Van Gorp and Grissom, 2001; Parker and Becker, 2003; Lui et al., 2004; Gonzalez, 2004; Wulf, 2005; Thota and Whitfield, 2010; Yadin, 2011, and many more), to motivate research questions or approaches (e.g., Madison and Gifford, 1997; Rajlich, 2002; Knobelsdorf, 2008; Ma et al., 2011), or as an analytical tool that retroactively justifies existing pedagogy (Pullen, 2001). Some recent work in CER has explored pedagogy that builds on the notion of legitimate peripheral participation (e.g., Hundhausen, 2002; Guzdial and Elliott Tew, 2006) or otherwise used situated learning as a research framework (e.g., Booth, 2001b; Knobelsdorf and Schulte, 2007). The related constructivist approach of cognitive apprenticeship, which seeks to make thinking processes visible (Collins et al., 1989) has also found some recent applications in computing education (e.g., Caspersen and Bennedsen, 2007; Bareiss and Radley, 2010), as has problem-based learning (e.g., Kay et al., 2000; Kinnunen and Malmi, 2005; Nuutila et al., 2005, and see Section 10.1).

The vast majority of CER papers that mention "constructivism" are concerned directly with pedagogy. Only a few authors have considered the applicability of a constructivism as a theory to computing education from a wide perspective. The discussion in two such publications, by Greening (1999) and Ben-Ari (2001a), forms the spine of Subsections 6.7.1 and 6.7.2 below. In Subsection 6.7.3, I review Ben-Ari's commentary on the applicability of situated learning theory to computing education.

6.7.1 Programmers' jobs feature ill-structured problems in an ill-structured world

Greening (1999) argues that, from the constructivist point of view, the main challenge of learning programming 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. To cope with such contexts, students need to learn the skills of proper design, coding style, requirements elicitation, software development processes, teamwork, and communication. Greening advocates PBL as a pedagogical technique suitable for learning these skills.

Greening further claims that constructivism is much needed in computer science education to address the modern trends of information explosion, rapid technological change, globalization, and the resulting need to recognize the validity of multiple viewpoints on knowledge. "Education will not be able to assume that a singular world view will provide an adequate working model; it will need to deal instead with multiple world views in flux" (p. 67). This is something that constructivism, with its subjective view of knowledge, is more comfortable with than the traditional objectivist paradigm is.

When it comes to various human aspects of computing, such as software engineering practices, good design, usability, etc., 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 (2001a). The computer does not negotiate...

6.7.2 The novice needs to construct viable knowledge of the computer

> *Even if no effort is made to present a view of what is going on 'inside' the learners will form their own.* (du Boulay, 1986)

Ben-Ari (2001a) discusses the application of constructivism – primarily personal constructivism of a cognitivist bent – to computing education, drawing a number of conclusions that "seem to follow directly from constructivist principles". Ben-Ari's conclusions rest on three claims. The first is that learners necessarily construct knowledge about the phenomena they encounter, for better or worse. The other two are characteristics of computing as a discipline:

> *I claim that the application of constructivism to CSE must take into account two characteristics that do not appear in natural sciences:*
>
> - *A (beginning) computer science student has no effective model of a computer.*
> - *The computer forms an accessible ontological reality.*
>
> (Ben-Ari, 2001a, p. 56)

Let us look at each of the three claims in more detail, starting with students' initial models.

Lack of an effective initial model

Constructivism emphasizes the importance of prior knowledge for learning, and knowledge of the computer is, Ben-Ari (2001a) argues, a prerequisite for understanding computing as we know it. However, beginning students of computing lack knowledge that "the student can use to make viable constructions of knowledge based upon sensory experiences such as reading, listening to lectures and working with a computer". That is, students lack knowledge that is viable for the purpose of learning about programming and other aspects of computing.

Some students come to computing studies with self-taught, viable knowledge of the computer and of programming concepts. However, prior knowledge may also be a hindrance: "Autodidactic programming experience is not necessarily correlated with success in academic computer science studies. These students, like most physics students, come with firmly held mental models that are not viable for academic studies" (ibid., p. 58). Depending on which variant of constructivism one subscribes to, dealing with such non-viable prior knowledge is either a matter of correcting it or of refining it to create coherent, viable knowledge (cf. Section 6.5).

Inevitable construction

Ben-Ari (2001a; Ben-Ari and Yeshno, 2006) argues that, according to 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. For instance, a person encountering a WYSIWYG word processor for the first time may construct a model based on an analogy with paper and ink, which is viable to begin with but may soon become non-viable once one starts tinkering with fonts (which are implemented as invisible markup within the visible document). A technical description of how a word processor actually works may contribute towards the construction of a different model.

Ben-Ari (2001a) marshals evidence from the literature – much of which is familiar to readers of this thesis from Section 3.4 and Chapter 5 – to support his argument that when faced with abstractions of the computer, students will necessarily construct their own, often non-viable mental models of what lies beneath. Ben-Ari points out that even though these difficulties with are sometimes attributed to features of particular languages, the phenomenon of constructing non-viable models of the computer does not seem to be constrained to any single language or programming paradigm.

Accessible ontological reality

> *By* accessible ontological reality*, I mean that a "correct" answer is easily accessible, and moreover, successful performance requires that a normative model of this reality must be constructed.* (Ben-Ari, 2001a, p. 56)

In essence, what Ben-Ari is saying is 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 non-viable 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 understanding fairly closely, leaving little room for disagreement. As Ben-Ari (ibid., p. 58) 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!

Implication: underlying models early

If Ben-Ari's claims are accepted, introductory programming students need to learn not just what program code does, but also what goes on beneath, within the computer. Leaving the computer as an abstract 'black box' in teaching means that people will rely on the ineffective, intuitive knowledge that they will nevertheless construct. The students' knowledge of the computer need not be 'complete' but must be viable for the purpose of understanding programs. Ben-Ari goes even further in his pedagogical recommendations, suggesting that students need to confront underlying models (of the computer and software artifacts like word processors) *before* learning about the abstractions above. Encountering abstractions first will lead to the ill-fated construction of intuitive models.

Greening (1999) is skeptical about Ben-Ari's claim[7] that a model of the computer is needed for learning computing, and emphasizes that constructivism is less about prescribing what prerequisite knowledge is needed than it is about acknowledging that prior knowledge plays a part in knowledge construction. Greening 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, p. 73)

I think Greening's statement (as I interpret it) has an unhorrifying point, but may at the same time miss the point that Ben-Ari was (I believe) trying to make.

[7]Greening did not have a time machine. I have referred here to Ben-Ari (2001a), a longer version of a 1998 paper that Greening (1999) commented on.

In any particular course you will be teaching a specific level of abstraction; you must explicitly present a viable model one level beneath the one you are teaching. (Ben-Ari, 2001a, p. 68)

Ben-Ari is not suggesting that introductory programming students (who primarily target an understanding of programs at program code level) should understand in detail how computer hardware or the operating system works. What he is saying is that given a targeted level of abstraction, a lower-level understanding – slightly lower but still as high as possible – is needed that is viable for the purpose of explaining phenomena at the targeted level. To put this claim differently, and into context: in order to learn about programming at the code level, students need to learn about a notional machine (see Section 5.4) that concretizes the semantics of code constructs.

Greening appears to be talking about the importance of understanding the actual machine at a fairly low level. I agree with Greening that levels of abstraction in programming appear still to be rising. That 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 that I cannot foresee, a viable understanding of the computer "one level beneath" the targeted level will still be needed. The level of abstraction of the notional machine that is needed may increase in the future, along with the level of abstraction of programming itself, but a notional machine will nevertheless be required by would-be computer programmers.[8]

6.7.3 Situated learning theory is 'partially applicable' to computing education

Ben-Ari (2004, 2005) discusses the applicability of apprenticeship-driven pedagogy based on situated learning (Section 6.6) to education in computing and other high-tech fields. He identifies open-source software development as an area of computing where legitimate peripheral participation is vividly demonstrated. However, his overall conclusion is that legitimate peripheral participation cannot be accepted as a general model of computing education as "it simply ignores the enormous gap between the world of education and the world of high-tech CoPs [communities of practice]" (Ben-Ari, 2004, p. 98). Ben-Ari questions, among other things, the improbable notion that corporations (genuine communities of practice) would hire and educate newcomers lacking skills that *can* be taught in a school context. He also understandably bristles at the suggestion that learners should choose, or be chosen, professions and future communities of practice early in life, as is the case in the low-tech communities studied by Lave and Wenger (1991), and notes that situated learning may help perpetuate social class structures.

To consistently uphold a policy of real situations [...] *A class of rich kids could be asked to compute the most efficient route to sail a yacht from Nice to Barcelona, while a class of poor kids could be asked to compute the salary at which it is advantageous to give up welfare and take a job.* (Ben-Ari, 2005, p. 372)

Despite these strong criticisms, Ben-Ari does not entirely dismiss the potential of situated learning theory for high-tech education. Taken "metaphorically" rather than at face value, he argues, situated learning can serve as a source of inspiration for computing teachers and computing education researchers. In particular: we should take it seriously that learning computing – even at school or university – is a step on the way towards students' initiation into authentic communities of practice within industry and academia. Learning to be a programmer, for instance, requires the learner to learn about communities in need of programmers and the authentic practices in which they engage, and to become motivated to join such a community. Ben-Ari further notes the domain-tied nature of most genuine computing communities of practice, and argues that computing education should introduce one or more non-computing domains to learners, even if this does not take place in a truly authentic setting but a simulated one.

[8]Ben-Ari's claim raises the suspicion of an infinite regress. To learn at one level, you first need to learn at the level below. To learn at that lower level, do you first have to learn at one below that, and so forth? Does this lead to the conclusion that learning to program requires one to start at the level of subatomic particles and work upward in abstraction from there? The regress is avoided through the observation that the understanding at "one level beneath" does not need to be as comprehensive as the understanding at the target level. At each successively lower level, the understanding needed may be increasingly vague, serving only the purpose of being viable (that is, 'good enough') for learning about the next level up. At a sufficiently low level, which is perhaps not very low, even a trivially teachable or intuitively formed vague understanding is enough.

6.8 Constructivisms have drawn some heavy criticism

The constructivist landscape is not a tranquil place, but a gory battlefield on which some wage civil war while others defend against foreign attacks and lead crusades against non-believers. Much of the critical literature is highly polemical, with constructivists having been denounced variously as inquisitors, Stalinists, drunkards, necromancers, and – worst of all – behaviorists.

I cannot hope to do justice to the complex dialogue in the literature, but will try to present some of the main issues. I will largely ignore the in-fighting between branches of constructivism, and focus on general criticisms of constructivist theories.

I have organized this section around nine contentious points, or 'skirmishes'.

Skirmish 1: relativism

> *The common constructivist move is from uncontroversial, almost self-evident premises stating that knowledge is a human construction, that it is historically and culturally bound, and that it is not absolute, to the conclusion that knowledge claims are either unfounded or relativist.* (Matthews, 1994, p. 143)

Perhaps the most common criticism of constructivist epistemologies (see, e.g., various articles in Phillips, 2000; Steffe and Gale, 1995) is that they easily lead to a denial of reality and to moral relativism: in a world of one's own construction, it is meaningless to care about others. There is widespread agreement that moral relativism is bad, which is why it is a popular beating stick for critics to wield. Constructivist theorists are keen to escape accusations of relativism. On the radical constructivist side, such claims are called "misunderstandings of constructivism" (von Glasersfeld, 1995), and sometimes dodged by labeling one's own version of constructivism non-dualistic, that is, by asserting that the world and humans' constructions of it are inseparable (see, e.g., Howe and Berv, 2000, and cf. the next skirmish). In a rather similar vein, Gergen claims that extreme social constructivism transcends such issues since it does not commence with the external world as its fundamental concern nor with the internal mental world (which would lead to solipsism) but with language (Gergen, 1995); this defense, however, simply leads to solipsism of a different, social kind, according to its critics (Bickhard, 1995).

It has been argued that escaping the trap of utter relativism requires a notion of shared meanings between individuals. Towards this end, von Glasersfeld claims that even though for the radical constructivist, meanings are not genuinely shared, they may be "taken as shared" by individuals, as beliefs that have more or less viability. Howe and Berv are critical:

> *Is the meaning of "taken as shared" merely "taken as shared"?* [...] *How can radical constructivists talk to each other? Worse, how can they talk to themselves? Are von Glasersfeld's meanings on Monday the same as his meanings on Friday, or does he merely take them to be the same? How could he know?* (Howe and Berv, 2000, p. 33)

Social constructivists are keen on shared meanings but also fail to escape accusations of relativism. Taken to the extreme, social constructivism argues that the very notion of knowledge is merely a matter of social agreement, an epistemologically relativist view which readily leads to ontological relativism as well (see, e.g., Prior McCarty and Schwandt, 2000).

Many words have been said in the relativism debate, but a resolution does not seem near.

Skirmish 2: logical flaws: foundationalism, dualism, and the element of surprise

Perceived flaws in constructivist claims about epistemology include the failure to explain the notions of construction and viability, as well as constructivists' claims of non-dualism.

Extreme constructivists consider themselves antifoundational, with no reality to found knowledge on. Prior McCarty and Schwandt (2000) contest this view on several accounts, contending that "it is patently absurd, even hilarious, to consider describing either von Glasersfeld's views or those of Gergen as "antifoundational"". They argue, for instance, that the very idea of knowledge construction depends on an ontology since for construction to happen there need to be some people and materials whose existence precedes construction. Therefore, constructivism is actually a foundationalist theory in which every idea is founded on the metaphysics of construction (pp. 71–72).

Prior McCarty and Schwandt (2000, p. 70) point out another problem with constructivism: its failure to explain surprise. For instance, it is possible for a seemingly healthy person to be surprised when a doctor tells them that they have only two months left, which seems strange if our experiences are independent of ontological reality. According to Prior McCarty and Schwandt, denial of surprise follows from the ontologically agnostic positions of the radical forms of personal and social constructivism. An extension of this argument is that since it is possible for our understandings unexpectedly to turn out to be non-viable, then there must be an ontological reality 'out there' that leads to such developments. The notion of viability therefore appears to lead to an acceptance of ontology (see also Marton and Booth, 1997, pp. 6–8). Arguments of this kind have prompted the suggestion that radical constructivism is a dualist theory, unlike its proponents claim. Dualism comes with baggage of its own; see, e.g., Controversy 6 in Section 5.7 above.

Skirmish 3: the content of teaching and educational norms

Many constructivisms question the idea of educational norms (Section 6.2.3 above). However, critics in science education take issue with the idea that students "construct their own sciences" during education as per the Piagetian maxim "to understand is to invent". Critics point out that even if existing theories are not always 'correct', education needs to familiarize learners with what the broader intellectual community regards as the products of science, and that it is patently impossible for students to themselves construct ideas that scientists have spent centuries researching (e.g., Phillips, 2000, pp. 14, 32, 179).

A related criticism concerns the constructivist dismissal of standardized assessment:

> *In the hands of the most radical constructivists,* [the dismissal of standard evaluations] *implies that it is impossible to evaluate any educational hypothesis empirically because any such test necessarily requires a commitment to some arbitrary, culturally-determined, set of values.* [. . .] *If the "student as judge" attitude were to dominate education, it would no longer be clear when instruction had failed and when it had succeeded, when it was moving forward and when backward.* [. . .] *To argue for radical constructivism seems to us to engender deep contradictions. Radical constructivists cannot argue for any particular agenda if they deny a consensus as to values. The very act of arguing for a position is to engage in a value-loaded instructional behavior.* (Anderson et al., 2000b)

Skirmish 4: Piaget's legacy and the brand of cognitive science

Radical constructivists and cognitive scientists – and others – both draw heavily on the work of Jean Piaget and hold him in great esteem. This has led to arguments between the groups over whose interpretation of Piaget is correct (or more viable). Some cognitivists advise radical constructivists to construct "a more careful understanding of Piaget" that also acknowledges the critical role of the relatively passive process of assimilating knowledge into existing structures (Anderson et al., 2000b). Radical constructivists consider non-radical understandings of Piaget to be naïve (e.g., von Glasersfeld, 1982, 1995, 1998), and suggest that critics who claim that Piaget failed to give sufficient consideration for social issues ought to apply "the necessary attention" to his original work.

Adding perceived insult to injury is the fact that various constructivist authors state that their work is in line with findings from cognitive science, a claim that Anderson et al. (2000b) – who are eminent cognitive scientists – vehemently deny. Anderson et al. do not cite any examples from computing education, but one suspects that they might consider the book chapter by Greening (1999) (discussed above in Section 6.7) to be a case in point: as Greening pushes a constructivist agenda, he suggests that constructivism contributes to educators' awareness of how important it is to avoid cognitive load, although the critical cognitive scientists are suggesting that it is precisely constructivist teaching practices that are *causing* cognitive overload (see Skirmishes 5 and 6 below). Both views may be right at the same time, in their own ways: even if some constructivist teaching practices tend to cognitively overload students, good constructivist teachers are more likely to use substantial scaffolding that reduces cognitive load (although perhaps still not as much as critics might like; see Skirmish 7).

Skirmish 5: ignoring evidence

Various authors have complained about the vagueness of constructivist ideas. For instance, Prior McCarty and Schwandt (2000, p. 42) describe knowledge construction as "a range of hazily imagined mental activities", while Anderson et al. (2000b) claim that constructivists "refuse to focus on details and precise specifications". Indeed, many constructivists seem to be satisfied with a highly abstract notion of construction, although conceptual change theorists (Section 6.5) and cognitive constructivists who rely on schema theory and mental models are exceptions.[9] The vagueness of instructional frameworks and a lack of rigorous studies and testable hypotheses have been acknowledged as weaknesses also by some scholars otherwise sympathetic to constructivism (Tobias and Duffy, 2009).

A related criticism is that constructivisms are said to be oblivious to much that is known through research. Mayer (2004) and Kirschner et al. (2006) accuse constructivists of ignoring half a century of empirical evidence from cognitive psychology regarding the human cognitive apparatus and its effects on learning. Anderson et al. (2000b) pursue a similar line:

> *This* [constructivist] *criticism of practice (called "drill and kill," as if this phrase constituted empirical evaluation) is prominent in constructivist writings. Nothing flies more in the face of the last 20 years of research than the assertion that practice is bad. All evidence, from the laboratory and from extensive case studies of professionals, indicates that real competence only comes with extensive practice.*

Critics have called for radical constructivists and supporters of constructivist pedagogies to produce testable predictions and empirical data that supports their claims. At least part of the challenge may go unheeded, since some extreme forms of constructivism seem even to deny the possibility or relevance of empirical data to educational decisions (Prior McCarty and Schwandt, 2000; Anderson et al., 2000b).

Skirmish 6: complex, authentic contexts

An issue of disagreement between (some) cognitive psychologists and (some) constructivists is the constructivist claim that knowledge is context-dependent and effective learning requires rich contexts, authentic settings, and ill-structured problems which learners make sense of largely on their own. Mayer (2004), Anderson et al. (2000b) and Kirschner et al. (2006) cite numerous empirical studies whose results indicate that knowledge can be transferred between contexts (under the right conditions), and that ill-structured problems, complex settings and other constructivist pedagogies tend to cognitively overload learners and are commonly ineffective.

The psychologists that I have cited do not claim that authentic contexts are never useful or that no knowledge is context-dependent or best learned in a complex context. Anderson et al., for instance, do agree that some skills are best practiced in a complex setting, for motivational reasons or to learn special skills that are unique to the complex situation. However, they object to the sweeping statements on this topic that many constructivists make.

Mayer (2004) warns against "the constructivist fallacy" of equating behavioral activity (hands-on tasks) with fruitful cognitive activity. According to Anderson et al. (2000b), situated learning and constructivism are movements which claim to oppose behaviorism, but which, through their neglect of cognition, have ironically ended up reiterating Burrhus F. Skinner's utopian vision of behaviorist education from his novel *Walden Two*, in which people learn best from being subjected to the control of the community in which they participate.

Situated learning theory places a particularly heavy emphasis on how effective learning is dependent on participation within a complex, authentic context. This makes situated learning a prime target for the critical cognitive scientists. One of the assertions of Kirschner et al. (2006) is a succinct antithesis of the situated theory of learning: "The practice of a profession is not the same as learning to practice the profession."

[9]A colleague of mine once half-seriously commented that "constructivism can be used to justify any pedagogical innovation" and – on a separate occasion – that "constructivism can't be used to justify anything". This apparent contradiction is explained by the fact that constructivism, when taken in the broad sense, is so abstract when it comes to the processes of learning that it says everything and nothing at the same time.

Skirmish 7: minimal guidance pedagogy

Problem-based learning, inquiry learning, and other constructivist teaching approaches are sometimes labeled *minimal guidance* approaches, as in the title of the provocative paper by Kirschner et al. (2006): *Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching*. Kirschner et al. review evidence that points to the supremacy of direct-guidance methods (such as worked-out examples) in teaching. Influential threads within educational psychology, such as cognitive load theory (Section 4.5), emphasize the need to give significant guidance to novice learners, even to the extent of providing ready-made solutions to problems for the learner to study. This contrasts with the fairly common interpretation of constructivist learning theory according to which learners should be left to discover meanings, patterns, and solutions on their own, with little guidance from a teacher. Such a combination of complex contexts with limited guidance has come in for particularly harsh criticism from the cognitivist camp (see, e.g., Mayer, 2004; Kirschner et al., 2006). Mayer laments that in the face of overwhelming evidence to the contrary, pure discovery pedagogies seem to reappear every decade in a different guise – presently constructivism – "like some zombie that keeps returning from its grave". Sweller has argued that constructivist pedagogy has failed to account for the different kinds of instruction needed to learn *biologically primary knowledge* that we have evolved to acquire easily and automatically – such as language, which can be taught by immersing the learner into complexity – and *biologically secondary knowledge*, such as politics or computer programming, which call for direct, explicit instruction (Sweller, 2010a; Sweller et al., 2007)

The paper by Kirschner et al. (2006) has given rise to a rich exchange of views between academics from the two schools of thought (Schmidt et al., 2007; Hmelo-Silver et al., 2007; Kuhn, 2007; Sweller et al., 2007; Tobias and Duffy, 2009). The typical constructivist defense against the attack of Kirschner et al. has been to emphasize how constructivism – or one's particular interpretation of it, at least – is not a pure discovery or minimal guidance approach. Proponents of various constructivist pedagogies such as PBL have been quick to point out that although some implementations of constructivism might involve little direct guidance, at least the particular variant that is being promoted features significant scaffolding. These arguments have not been entirely to the satisfaction of direct-instruction advocates: disagreement persists concerning the amount of guidance needed, and in particular on whether providing the solution to a problem is an inappropriate form of strong scaffolding or a very useful one. Many constructivists emphasize that methods such as PBL improve 'softer' skills that the attacking psychologists do not test for, such as teamwork and self-directed learning skills. Some (such as Kuhn, 2007) even question the goal of teaching knowledge, preferring to emphasize the development of generic thinking and learning skills (which is a goal that goes against the schema theory view of expertise as domain-knowledge dependent; Chapter 4).

Both sides of the argument have sought to present empirical evidence for their respective views and criticized the studies cited by the other side. The book edited by Tobias and Duffy (2009) features some increasingly fruitful dialogue between the two camps. Future research may help us gain better understanding of under what circumstances and for which goals the different pedagogies work best, and how they can complement each other.

Skirmish 8: nothing new

Nearly all critics of constructivism, it seems, have cast doubt upon the originality of constructivist ideas and the pedagogical practices they result in. Phillips (2000), for instance, asks if constructivism has really changed anything or if it is just the newest form of Kant's philosophy, Dewey's progressivism, and discovery learning. Matthews (2000) suggests that constructivism has mostly succeeded in introducing new jargon that – instead of simplifying complex matters, as terminology should – complicates simple matters. Matthews even provides a tongue-in-cheek dictionary that maps "constructivist new speak" to "orthodox old speak". Elsewhere, Matthews (1992) calls constructivism old wine in new bottles, while Levitt (quoted by Patton, 2002, p. 101) goes one better as he describes constructivism as a manifestation of postmodernism that is essentially “a particular technique for getting drunk on one’s own words”.[10]

The less extreme and more wishy-washy one’s constructivism is, the more susceptible one is to accusations of unoriginality and weak potential for genuine change. Greening (1999) warns us that constructivism cannot be merely approached, it must be accepted in its entirety, lest we end up merely diluting our curricula with an assortment of constructivist principles that mean little in isolation from each other. According to Larochelle and Bednarz (1998), “softer” non-epistemological versions of constructivism have “scarcely modified the usual teaching modus vivendi at any level of instruction one chooses to examine”. Some critics of constructivism agree with this latter claim: “A less radical constructivism may contain no contradictions and may bear some truth. However, [. . .] such a moderate constructivism contains little that is new and ignores a lot that is already known” (Anderson et al., 2000b).

Skirmish 9: a disconnect between pedagogy and theory

A corollary of the claim that constructivist theory is vague (see Skirmish 5 above) is that there is a disconnect between constructivist theory and the pedagogies that are commonly advocated by constructivists. That is, it “seems possible for a person who accepts constructivism as a philosophy to adopt any variety of pedagogical practices (or for a teacher who uses constructivist classroom practices to justify doing so in a variety of ways, some of which might not philosophically be constructivist at all)” (Phillips, 2000, p. 18). Many constructivists and critics have commented on this issue; some see it as a problem, others do not.

Ben-Ari (2001a) asks the delicate question of whether “being a constructivist requires an epistemological commitment to empiricism and idealism (or social idealism), as opposed to rationalism and realism that seem to come more naturally to scientists”. Citing the work of Matthews, Ben-Ari further notes how “pedagogical constructivists” avoid the problem by concentrating solely on improving pedagogy and ignoring epistemological details as not worth disputing. The question then becomes whether we need constructivist theory at all. Prior McCarty and Schwandt (2000) certainly do not think so, arguing that even though some constructivists may advocate good pedagogical practices, those practices hardly require, or benefit from, justification in the form of the “garage sale of outdated philosophical falsisms” that is constructivist theory. An example of an explicitly constructivist epistemological position which is nevertheless based on “traditional” pedagogy and experimental findings from cognitive psychology is that of Renkl (2009).

I give the last word to Greening (1999), who points out that one may care less “whether constructivism is *required* to bring about the beneficial changes in learning that have been attributed to it, and more about whether or not it *has*” (p. 50, emphases in original). Even a skeptic of constructivist theories accepts that the constructivist movement is influentially advocating certain pedagogies. Because of this, if for no other reason, it is important to consider how present-day pedagogical innovations relate to constructivism.

[10]Adding to this curious theme linking constructivism and alcohol, Richards’s (1995) description of the problems of constructivism is zymological: social and personal constructivism are analogous with two non-communicating descriptions of the process of beer-brewing.

Chapter 7

Phenomenographers Say: Learning Changes Our Ways of Perceiving Particular Content

Phenomenography is a primarily qualitative tradition of empirical research, whose main contributions lie within education and its hybrid disciplines such as CER. Since the 1970s, phenomenographers have sought to explore the educationally critical variation in how people experience, perceive, or understand various phenomena.[1] 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. Learning, from the phenomenographic point of view, involves becoming able to experience phenomena in significantly different new ways.

In this chapter, I introduce the phenomenographic research tradition and some of its epistemological and learning-theoretical foundations. This enables us to then review phenomenographic work on programming education. My treatment of phenomenography in this chapter will be extended by Chapter 17 in connection with a phenomenographic study on visual program simulation.

Section 7.1 outlines the constitutionalist belief system which provides an epistemological backdrop to phenomenographic research. In Section 7.2 we get a glimpse of the kinds of results that phenomenographic research produces. The implications of phenomenographic theory and research for learning are the topic of Section 7.3. Pedagogical issues follow in Section 7.4, and bring us back to the focus of Part II – what it takes to learn programming – which I discuss from a phenomenographic perspective in Section 7.5. The chapter concludes with the consideration of some criticisms of phenomenography in Section 7.6.

7.1 Phenomenography is based on a constitutionalist worldview

> *Our world is a world which is always understood in one way or in another, it can not be defined without someone defining it. On the other hand, we can not be without our world.* (Marton, 2000, p. 115)

Phenomenographic research is associated with a worldview that is sometimes termed *constitutionalism* (Prosser and Trigwell, 1999). Constitutionalism, which draws on the phenomeno*logy* of Edmund Husserl[2], has been presented as a basis for phenomenographic research by Marton (2000; Marton and Booth 1997). From the constitutionalist point of view, the only world that humans can meaningfully think about is their 'lifeworld' of lived experience. Understanding exists as an internal, two-way relationship between human experiencers and the phenomenon that is experienced. It is inseparable from the experiencer and phenomenon alike, and is a reflection of both.

[1]The words "experience", "perceive", and "understand" are often used interchangeably in the phenomenographic literature. I do the same in this chapter. Phenomenographers use these words in a particular sense. The intended meaning should become clearer over the course of the chapter, and further in Chapter 17.

[2]For linkages between phenomenography and (Husserl's version of) phenomenology, see, e.g., Marton and Booth (1997) and Uljens (1996).

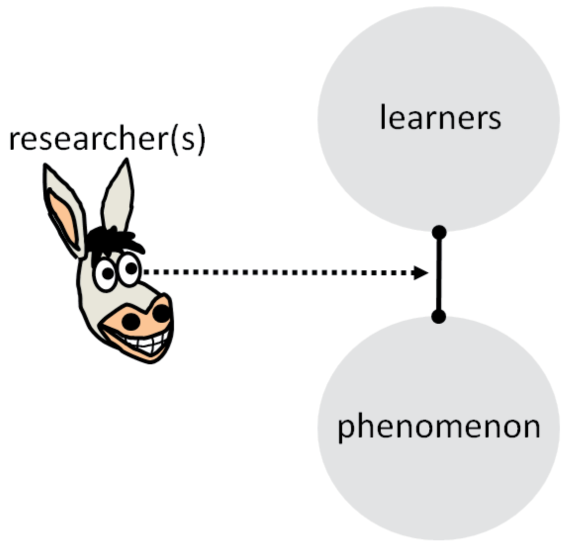


Figure 7.1: The phenomenographic research perspective. The researcher investigates the relationship between a population and a phenomenon; the results are a reflection of both. I have followed a fledgling tradition sparked by Berglund (2002) in my asinine portrayal of the phenomenographer in this figure.

The constitutionalist view is in sharp contrast to cognitive psychology (Chapters 4 and 5) and many forms of constructivism (Chapter 6), as it discards the notion of mental representation, viz., the notion that there exist representations within minds that are distinct from the phenomena that they represent. Marton (2000), for instance, rejects the idea that experiences are structures in a separate mental world, accusing cognitive psychologists of an untenable mind–matter dualism (see Section 5.7). By adopting a view of knowledge as ways of experiencing the world – as internal relations between knower and known – we achieve, Marton argues, a paradox-free non-dualist epistemology. Uljens (1998) remarks that Marton's "non-dualism" is essentially a type of monism. There is only one world, within which people, phenomena, and their relationships exist alike.

Philosophical considerations aside, two practical points arise from the constitutionalist stance. The first concerns research: a phenomenographer investigates neither people nor phenomena exclusively, but their relationship, which reflects both at once. The second concerns learning: learning must be considered in terms of both the learner and the specific content (phenomenon) being learned about. Let us first look at the nature of phenomenographic research and the type of results it produces, and return to learning in Section 7.3.

Investigating experiences of phenomena

> *We should explore* [. . .] *what the world we experience is like, on the one hand, and what our way of experiencing the world is like, on the other hand. And of course: these are not two things. They are one.* (Marton, 2000, p. 115)

Phenomenographers seek to find out how people experience phenomena. But what exactly is a phenomenon and what does it mean to experience one? The constitutionalist answer is that these two questions are inextricably linked. When he investigates experience, the phenomenographic researcher studies the relationships between people and phenomena (Figure 7.1).

A phenomenon is "something in the world – concrete or abstract – which can be delimited from the world, by the researcher and by others, according to their knowledge of the world" (Booth, 1992, p. 53). Marton (2000) defines a phenomenon as a complex of all the different ways in which something can be experienced. A way of experiencing something is one of the various facets that together constitute the phenomenon. An individual's experience of something is shaped by both the experiencer and the phenomenon.

An example. Marton and Booth (1997) summarize a phenomenographic study by Marton, Watkins, and Tang, which explored Open University students' experiences of what learning is. Learning was experienced in one of six ways: increasing one's knowledge, memorizing and reproducing, applying knowledge, understanding and gaining insights about the world, opening one's mind to see things in a different way, and changing oneself as a person. These ways of experiencing learning play a part in constituting what the phenomenon of learning is (along with other ways of experiencing learning that the particular study did not discover). The research results speak not only of the students, or of what learning is, but of both.

7.2 There is a small number of qualitatively different ways of experiencing a phenomenon

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.

> *The constitutionalist view differs significantly from constructivism in that learners are seen to experience what they are learning in a small, identifiable range of different ways (usually between three and seven). An identifiable range of variation is thus assumed to be present in any given group (as compared with the idiosyncratic construction of every individual).* (Bruce and McMahon, 2002, p. 12)

This is not to deny that everyone's conception of a phenomenon is unique. However, it is posited that 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.

Critical aspects

A phenomenon, viewed from a particular perspective, is characterized by certain *critical aspects* (also known as *critical features*; I use these terms interchangeably). These aspects are critical in that they define how ways of perceiving the phenomenon are qualitatively different:

> *Every phenomenon that we encounter is infinitely complex, but for every phenomenon there is a limited number of critical features that distinguish the phenomenon from other phenomena. What critical features the learner discerns and focuses on simultaneously characterises a specific way of experiencing that phenomenon.* (Pang, 2003, p. 148)

Let us take an example from CER. Eckerdal (2006) investigated how CS1 students understand what an object is. Her data – typically for a phenomenographic study – consisted of in-depth interviews with learners. Eckerdal describes three qualitatively different ways of understanding the phenomenon of object, which I have summarized in Table 7.1. There is a logical structure to the three categories listed in the table. Category A describes a very simple way of understanding what an object is: a piece of code. Only the critical aspect of program text is in focus. In the understanding described by Category B, program code is also recognized as a defining aspect of objects, but 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. The understanding in Category C includes those from the two first categories, and further attributes a world-modeling aspect to objects.

Outcome spaces and categories of description

The results of a phenomenographic study usually take the form of an *outcome space* that consists of a small number of interrelated *categories of description*. Table 7.1 is an example of an outcome space

Table 7.1: Qualitatively different ways of understanding the programming concept of object (adapted from Eckerdal, 2006).

Code	Description
A	An object is experienced as a piece of code.
B	An object is experienced as a piece of code, and as something that is active in the program.
C	An object is experienced as a piece of code, as something that is active in the program, and as a model of some real-world phenomenon.

with three categories of description. In each category, the researcher has crystallized a description of a particular way of understanding that is different from the others in an educationally critical sense.

A category does not represent a particular person's understanding or mental representation of a phenomenon, although it is possible for a person, at a particular time, to conceive of the phenomenon in a way that is consistent with a category. The categories are not defined in isolation; an outcome space is defined by the way the conceptions contrast with each other. These differences between categories arise out of the way certain critical features – or relationships between features – are discerned in one category but not in another.

This hierarchicality of Eckerdal's outcome space is typical of phenomenographic results, as is the way some types of understanding can be considered richer while others are more limited. Compared to a richer understanding of a phenomenon (e.g., one that matches Category C), a limited understanding of a phenomenon (e.g., one that matches Category A or B) either focuses on a subset of the critical aspects of a phenomenon or fails to discern relationships between particular critical aspects.

We now have an inkling of what phenomenography is and how phenomenographers view human experience. What does all this mean for learning?

7.3 Learning involves qualitative changes in perception

Knowing the phenomenon can be seen as having a multi-faceted view of the [phenomenal] *object, while learning about it is gaining access to views of further faces and developing an intuitive relationship with the object so that an appropriate face or set of faces is seen in appropriate circumstances* (Booth, 1992, p. 261)

7.3.1 Discerning new critical features leads to learning

Most phenomenographic work focuses on a certain kind of learning that takes place in educational settings: learning to experience particular phenomena in richer, significantly different ways than before, in keeping with learning goals set by teachers or the learners themselves. The discourse on learning within the phenomenographic tradition may sound limited – after all, learners should learn to *do* things, they should acquire new skills, not just new understandings. However, phenomenographers are in fact very concerned with concrete abilities, which are seen as an outcome of learning to experience phenomena in richer ways. "If we are able to handle a situation in a more powerful way, we must first *see* it in a powerful way, that is discern its critical features and then take those aspects into account by integrating them together into our thinking simultaneously, thus seeing them holistically" (Marton, 2007, p. 20). By discerning what is critical about that which they encounter – and what is not critical, but a mere detail – learners become prepared to deal with future situations, which they can make sense of in terms of their critical aspects

(Bowden and Marton, 2004). A way of experiencing is the key to a new way of doing.[3]

From the phenomenographic perspective, learning is not a process that happens solely within a learner's mind, nor something external. To learn to experience the world in a significantly different way implies a change in the two-way relationship between the learner and a phenomenon within the world that they both inhabit. Such changes bring about new ways of discerning the meaning of the phenomenon, the part–whole relationships within the phenomenon, and the relationships of the phenomenon to its surroundings (Marton and Booth, 1997).

The relational view of learning emphasizes that learning is not generic, but involves the development of new perspectives on particular phenomena. As learning always involves both a learner and a phenomenon, it is not sufficient to consider merely cognitive processes that enable learning to take place, but also the content of learning, the particular features of the phenomenon that characterize the significantly different ways of understanding it.

7.3.2 The discernment of critical features requires variation

Pang (2003) distinguishes between two "faces of variation" that are relevant to phenomenographic research. The first type of variation exists in the differences between qualitatively different ways of understanding a phenomenon. This is the kind of variation that I have discussed so far in this chapter. Another kind of variation pertains to a particular critical feature of a phenomenon. This variation is key to discerning a critical feature in a particular way and, consequently, to learning to experience a phenomenon in a significantly different way. The second kind of variation is the domain of variation theory.

Variation theory (Marton and Tsui, 2004; Pang, 2003) is a theory of learning closely associated with, and inspired by, constitutionalist epistemology and the phenomenographic research movement. Its chief tenet is that in order to learn, the learner must discern variation in educationally critical features of the *object of learning*, that is, what one learns about.

As discussed above, to learn to experience a phenomenon in a new way requires new critical features, or relationships between critical features, to be discerned. According to variation theory, each critical feature is associated with a *dimension of variation*. To be able to discern a feature, one must experience variation along the dimension of variation corresponding to that feature.

The jargon of variation theory can get complicated, but the basic idea is very simple, indeed quite commonsensical. To experience something as red, we must experience other colors as well. To experience the height of a person, we must experience people of different heights. Different colors and different heights are values along the dimensions of variation in color and height, respectively. A strategy for solving a programming problem is a value along a dimension of variation; other strategies for solving similar problems are values along the same dimension. To compare strategies, you have to have experienced more than one.

To experience variation in a dimension requires simultaneous awareness of different values along that dimension, either through being exposed to the values at the same time or by juxtaposing one's current experience with prior experiences. In addition, the learner must be able to experience each of the critical features as features of a single phenomenon, joined together in his experience. That is, the learner needs to be focally aware of each critical feature simultaneously so that the critical features are present at the same time as features of the phenomenon, with relationships to other critical features and to the phenomenon as a whole (Marton et al., 2004; Pang, 2003).

7.4 Phenomenographers emphasize the role of content in instructional design

To a teacher, a phenomenographic outcome space may be useful as a tool for analyzing their own teaching, the content they teach about, and the ways in which their students understand (or might understand) the

[3]This is not to say that this kind of learning is the only one there is. Phenomenographers acknowledge that there is learning that involves (merely) further developing or fine-tuning a perspective – a way of experiencing – which one already has access to. However, it is the presumably more profound kind of learning brought about by qualitative shifts in experience that phenomenography focuses on.

content. "The assumption is that if we want to make the student think in a certain way about something, it should be useful to know what other ways there are to think about it" (Johansson et al., 1985, p. 255). Marton and Booth (1997, p. 81) suggest that phenomenographic outcome spaces serve to identify "a notional path of developmental foci for instruction". Some phenomenographers use variation theory as an additional tool to analyze and report differences in ways of experiencing, and to offer pedagogical recommendations.

7.4.1 Teachers' job is to aid students in discerning critical features

In many cases, it is not enough for a teacher to just say to students what the critical features of a phenomenon are. In order for students to actually experience these features, the teacher needs to help learners discern variation in the corresponding dimensions, wherever those dimensions are not already familiar to the learners.

Marton et al. (2004) discuss variation-based pedagogy in terms of creating a *space of learning*, which is "the pattern of variation inherent in a situation" and "comprises any number of dimensions of variation and denotes the aspects of a situation, or the phenomena embedded in that situation, that can be discerned due to the variation present in the situation". A space of learning "delimits what can be possibly learned (in sense of discerning) in that particular situation" as it is "a necessary condition for the learner's experience of that pattern of variation unless the learner can experience that pattern due to what she has encountered in the past" (p. 21).

Marton et al. are at pains to emphasize that their work is about 'making learning possible' rather than guaranteeing learning results, as "no conditions of learning ever *cause* learning" (p. 22). That is not to say that they place the burden of learning entirely on the shoulders of the learner; on the contrary, the teacher's expertise plays a great role in constructing an effective space of learning. The job of the teacher is to facilitate discernment by creating situations in which students get to experience the critical variation. Learners must have the opportunity to observe values along the dimensions of variation that correspond to the critical features of the object of learning. For instance, when learning about object-oriented programming, students should be presented with opportunities to observe objects as code, as interacting agents within programs, and as parts of domain models. Without such a space of learning, Marton et al. argue, a qualitative shift in perception is not possible. Eventually, the critical features of the phenomenon should be discerned not only in isolation but as interconnected aspects of the object of learning; such discernment, too, is affected by how the phenomenon is present in the space of learning.[4]

It is not only what varies that is relevant, but also the static context against which the variation appears and against which it can be discerned. Marton et al. (2004) encourage teachers to think about and exploit 'patterns of variation' through texts, speech, and learning materials to enable learners to discern new variation. These patterns include contrast (comparing values along a dimension of variation), generalization (showcasing an aspect by demonstrating different instances in which it features), separation (varying an aspect while others remain invariant), and – often the most challenging – fusion (varying multiple features at the same time to appreciate their relationships and give a holistic 'feel' for the phenomenon).

7.4.2 For critical features to be addressed, they should be identified

Content-based pedagogy

Constructivists (Chapter 6) promote active and collaborative learning methods such as problem-based learning and groupwork. Cognitive load theory (Section 4.5) emphasizes the usefulness of managed assignments such as worked-out examples. What teaching methods do phenomenography and variation theory recommend?

[4]Ko and Marton (2004, p. 62) describe good mathematics teaching as "planned, choreographed, and well thought-out lessons" which nevertheless offer "plenty of space for the students' own independent and spontaneous ideas". In relation to the present-day buzzwords "teacher-centered" and "student-centered", Ko and Marton note, good teaching can be equally both, if "teacher-centered" is taken to mean that the teacher has the key role of making sure that a space of learning is created that matches the intended learning outcomes, and "student-centered" is taken to mean that students take ownership of the space of learning and participate in creating it.

The answer is none in particular, which is to say, none in general (see, e.g., Marton and Booth, 1997; Marton and Tsui, 2004; Bowden and Marton, 2004). Phenomenographers have contended that many teachers do not evaluate pedagogical approaches in terms of how well they are suited to dealing with particular content, and that many scholars likewise underplay the role of content, chasing instead the chimeric "art of teaching all things to all men". Marton and colleagues oppose the notion of a general pedagogical aid, be that project-based learning, example-based direct instruction, peer teaching, or IT-aided learning. Although these often-proffered techniques may be very useful for some content and in some circumstances,

> *no general approach to instruction can ever ensure that the specific conditions necessary for the learning of specific objects of learning are brought about. In order to do this, we must take the specific objects of learning as our point of departure.* (Marton and Tsui, 2004, p. 229)

What phenomenographers recommend is to use whichever method works for highlighting the particular variation needed in order to learn about the particular content that is being taught (when that content is considered from the point of view of what the learners are intended to learn about it). What works must be discovered for each object of learning separately and is tied to the educationally critical features of the object.

Teachers or researchers?

Pedagogic recommendations based on phenomenography and variation theory blur the line between the teacher and the educational researcher.

If pedagogic solutions need to be discovered for each object of learning separately, if perhaps even generic capabilities such as thinking strategies and communicative skills are domain-specific (as suggested by Marton and Tsui, 2004, p. 229), then the good teacher must be very aware of the pedagogically sensitive aspects of the content they teach. Variation theorists call for teachers to engage "in finding out what the specific conditions [of learning] are in every specific case, and how they can be brought about" (ibid., p. 231). This is a non-trivial task even for the content expert, Bowden and Marton (2004) contend, as the more fundamental and important one's way of seeing things is, the harder it is to even notice it.

In the light of variation theory, teachers should define objects of learning in terms of ways of experiencing particular content, learn about the ways in which learners may experience that content, and determine what variation is critical for experiencing the content in the intended kind of way. Such explorations enable the teacher to enact learning situations which afford the educationally critical variation, and which have a motivating *relevance structure* (Marton and Booth, 1997, Chapter 8) that draws students to experience the critical variation.

From a phenomenographic point of view, the goals of the teacher, then, have much in common with those of the phenomenographic researcher who studies the relationships between people and particular phenomena. Teachers should build on research findings that are particular to the object of learning they wish to teach about.

With that in mind, let us look at what phenomenographic research says about introductory programming.

7.5 Learning to program involves qualitative changes in experience

Phenomenography has gained popularity in the past decades, a development which has impacted on CER as well as other fields. There is a small but growing body of phenomenographic work that investigates the challenges of learning to program. This work can be roughly sorted into two threads. The first thread is concerned with very general questions: in what ways do learners experience what programming is and what it means to learn to program? The second thread is more varied and investigates the ways in which particular programming concepts are understood. Both of these threads were influenced by the seminal doctoral thesis by Booth (1992), which is a good place for us to start looking.

Table 7.2: Different ways of experiencing learning to program (adapted from Booth, 1992).

Code	Description
A	Learning to program is experienced as learning a programming language (or several).
B	Learning to program is experienced as above, and as learning to write programs using various techniques and language features.
C	Learning to program is experienced as above, and as learning to solve problems in the form of programs.
D	Learning to program is experienced as above, and as becoming participant within a programming community.

7.5.1 Learners experience programming differently

Booth (1992) conducted a pioneering phenomenographic study of learning introductory programming, based on interviews with novice programmers during a semester-long CS1 course. She found that her interviewees experienced *what it means to learn to program* in four qualitatively different ways. This outcome space, which I have paraphrased in Table 7.2, features a hierarchical progression from a rudimentary way of understanding ('just learning a language') in Category A to the rich way of understanding in Category D ('learning to solve problems as part of a community'). Booth also found that her interviewees had three qualitatively different *ways of experiencing what programming is*: A) something directed towards the computer; B) something directed towards a problem that one intends to solve, and C) something directed towards a product that is the outcome of the programming activity.[5] *Programming languages*, Booth reports, were experienced in four different ways: A) as utility programs that belong in computer system; B) as sets of code elements of which programs are built; C) as a means of communicating between program components, the programmer, and/or a program's users, and D) as a medium of expression that allows the programmer to express solutions to problems.

Later studies have developed the themes opened up by Booth (1992), lending support to and complementing her findings.

Booth (2001b) herself takes her earlier work – in which the richest ways of understanding are characterized by a communal aspect (e.g., Table 7.2) – and relates it to situated learning theory (Section 6.6) in order to explore the relationships between her categories and three computing subcultures: the academic, the professional, and the informal culture.

Bruce and her colleagues also studied learners' experiences of learning to program (see, e.g., Bruce et al., 2004; Stoodley et al., 2004). Bruce et al. (2004) report that their interviewees experienced *learning to program* as A) following the structure of a programming course to keep up with set assignments; B) learning to code using the right syntax; C) learning to write programs through understanding and integrating the concepts one encounters; D) learning to do what it takes to solve problems, and E) learning what it takes to be a part of the programming community and to think like a programmer. The findings of Bruce et al. partially match and partially extend Booth's earlier work.

Eckerdal et al. (2005) observed that students talk about learning to program in terms of developing a special "way of thinking", which is different from everyday thinking and from the thinking in other subjects they had learned about. Eckerdal et al. report an interview-based phenomenographic study in

[5] It is perhaps useful at this point to clarify that the phenomenographer seeks to look beyond language. Booth's primary focus, for example, was not on the different meanings that students assigned to the *word* "programming" but on the different ways in which students related to the subject they studied. The meaning(s) of the word "programming" are a separate (even though related) issue that is also an important consideration for the programming teacher (cf. Walker, 2011) and indeed for the phenomenographic analyst who tries to get at the underlying conceptualizations.

Table 7.3: Different ways of experiencing programming (adapted from Thuné and Eckerdal, 2010)

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.

which they found that students experience *learning to program* in five different, hierarchically connected ways: A) as learning to understand and use a language; B) as learning a hard-to-define way of thinking related to a programming language; C) as learning to understand computer programs in real life; D) as learning a 'method' of thinking which enables problems to be solved, and E) as learning a skill that can be used outside the programming course. Thuné and Eckerdal (2009, 2010) analyzed students' experiences of *what programming is*. They describe a hierarchical outcome space (Table 7.3), whose categories largely mirror those from Eckerdal et al.'s results – as one might expect, since conceptions of programming and learning to program are interdependent.

7.5.2 A limited way of experiencing programming means limited learning opportunities

Some of the above ways of understanding programming and learning to program are very simple and do not provide a fruitful basis for learning programming concepts. For instance, a learner 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 his overarching view of programming changes in a significant way.

The outcome spaces produced by phenomenographic studies within CER suggest that learning to program may require multiple large-scale qualitative shifts in learners' perceptions of programming. A static perspective limited to program text needs to be developed into a dynamic one that takes execution-time behavior into account. Ultimately, one should learn to view programming as an empowering skill applicable to problems in one's community.

Pedagogical implications

In line with variation theory, phenomenographers in CER emphasize the importance of highlighting variation in the critical aspects of phenomena, rather than championing any specific teaching methods as blanket solutions. When teaching methods such as groupwork and visual tools are advocated, it is with the caveat that these pedagogies will only be useful if they help learners discern new critical aspects and relationships between them. Unsupervised groupwork, for example, may, but also may not, lead to discernment of critical variation. Often, the role of the teacher as a facilitator and guide rather than a transmitter of information, is emphasized. Here is the view of Booth (1992), for instance:

Teaching should encourage well-founded conceptions through example. Teaching should not merely try to bring about expert behaviour in students by offering expert views on the content, but give students the range of challenges which enable them to come to the expert understanding via experience. Teachers should above all be aware of the range of conceptions held by their students, and that poor conceptions are not necessarily caught in lab exercises and examinations.

To learn to view programming in terms of execution-time behavior, for example, learners need to be placed in environments that enable and motivate them to become focally aware of the execution-time behavior of programs and its relationship with program code. Thuné and Eckerdal (2009) apply variation theory to their outcome space concerning conceptions of programming (Table 7.3). They recommend that instruction bring students to focus not only on various pieces of program code, but also: different program actions during execution, different applications of programming to everyday life, different problems for which there are programming solutions, and different contexts within which programming skills can be used. They further warn against varying too many aspects at once and provide examples of teaching techniques that, it is hypothesized, highlight the critical aspects. For instance, making students aware of how a tiny change in program code can lead to dramatic changes in program behavior may be effective in helping students pay attention to the runtime aspect of programs. After this aspect is discerned, software tools may be used to illustrate the relationship between program code and the resulting behavior.

As it is important for a teacher to be aware of the qualitatively different ways in which learners may understand the subject matter, the pedagogical expertise of teachers and teaching assistants is highly important. Booth (2001a) reports a study in which it turned out that the classroom tutors employed did not have a sophisticated way of understanding the goals of the course and in particular failed to see how the content of the course was intended to relate to a broader context. Matters improved greatly with tutor training: "An important insight the course team has gained is that the insights gained by the students can hardly be expected to surpass those of the tutor team. The relevance structure for the students can only be brought about through [the] tutor's experienced relevance." (Booth, 2001a, p. 185)

7.5.3 Learners experience programming concepts in different ways

Some phenomenographic projects have studied how programming students experience particular programming topics that are relevant to at least some CS1 courses.

Booth (1992) found several different ways of experiencing the concepts of recursion and function; the concept of *recursion*, for instance, was experienced in three different ways: A) as a construct in a programming language B) as a means of bringing about repetition, and C) conceptually as self-reference that enables a function to make use of itself.

I have already used as an example Eckerdal and Thuné's (2005) outcome space of different ways of understanding what an object is (Section 7.1). The same paper by Eckerdal and Thuné also reports an analogous outcome space for the concept of class. Later work has suggested that students may initially develop either a static "text conception" or a dynamic "action conception" of an object which they then need to relate to the other conception (Eckerdal et al., 2011). I myself have conducted phenomenographic studies of students' ways of understanding what it means to store objects in memory (Sorva, 2007), and of different kinds of variables (Sorva, 2008).

Stamouli and Huggard (2006) studied students' perceptions of program correctness (see also Booth, 1992). Boustedt explored students' understandings of the concepts of interface and plugin (Boustedt, 2009) and UML class diagrams (Boustedt, 2012). Lönnberg (2012) focused on students' perceptions of bugs in concurrent programs. Thompson (2008, 2010) studied practitioner perceptions of object-oriented programming with a view to improving education on the topic.

These studies contribute to the large body of evidence that shows that novices understand many fundamental programming constructs in limited ways that may hinder further learning and are non-viable for many common programming tasks. The lists of 'misconceptions' in Appendix A include many of the specific limited understandings that these phenomenographic studies have uncovered.

7.6 Phenomenography has not escaped criticism, either

In this section, I paraphrase a few main criticisms of phenomenography from the literature and phenomenographers' responses to those criticisms.

Criticism 1: language and the nature of outcome spaces

Critics from both within the phenomenographic movement (e.g., Säljö, 1994) and outside (e.g., Webb, 1997) have criticized the perceived disregard of interview-based phenomenography for the role of language, communication, and social construction. Scullion (2002, pp. 101–103) reviews this debate. According to him, phenomenographers in Marton's tradition see language as a relatively unproblematic device that the phenomenographer must get past to learn about the interviewee's conceptions. Critics such as Säljö call for a more careful and explicit treatment of language:

> *To me, phenomenography has a weak spot in its lack of a theory of language and communication, and in its almost dogmatic disregard for paying attention to why people talk the way they do. The assumption seems to be that what is meant by what is said can be construed as representing a conception of the phenomenon which one – according to the interviewer – is talking about.* (Säljö, 1994)

A related question is what phenomenographic outcome spaces really tell us. Some critics have argued that it is impossible to genuinely explore and describe what other people really experience. For instance, according to Richardson (1999), phenomenographers fail to deliver on their promise to describe the world as people experience it: "they have to depend on other people's discursive accounts of their experience [and] are merely describing the world as people describe it."

Some phenomenographers are happy to agree that outcome spaces are no more than descriptions of how people describe things in a particular kind of situation. This does not mean they are not useful in practice:

> *I don't wish to assert that I 'know' an individual's conception of a phenomenon. What I do want to be able to say is that, following a given interview context, analysis of the transcripts enables me to differentiate between a number of different ways of seeing the phenomenon that are apparent in that kind of conversation.* [...] *Also, it is not possible for the researcher to 'be' that person; the researcher interprets the communication with the person.* [...] *I am satisfied that phenomenographic research produces descriptions which owe their content both to the relation between the individuals and the phenomenon (that is, their conceptions) and also to the nature of the conversation between the researcher and each individual and its context (which includes the relation between the researcher and the phenomenon).* (Bowden, 2000, pp. 16–17)

Bowden's pragmatic position seems a reasonable way to address criticism of this kind.

Criticism 2: value-loaded norms

Webb (1997), who writes from a post-modernist perspective, is critical of the way phenomenography posits that some ways of understanding are qualitatively better or more correct than others. Webb calls phenomenographic research hopelessly value-loaded and contaminated by the researcher's own conceptual apparatus. He accuses phenomenographers of making normative, even dogmatic judgments, and of forbidding learners to develop alternative views.

Phenomenography does indeed posit that some ways of understanding are richer and better than others in the sense that they enable learners to perform more effectively. Marton and Booth (1997, p. 2), for instance, explicitly take a stand:

> *We are living in an age of relativism, but a fundamental principle we are assuming in this book is that education has norms – norms of what those undergoing education should be learning, and what the outcomes of that learning should be.*

If one dismisses educational norms and embraces relativism full on, phenomenography may have little or nothing to contribute; the usefulness of educational norms is probably axiomatic to most phenomenographers.

In introductory programming education perhaps even more than in some other fields, educational norms are needed in order to succeed, as the computer represents an artificial, non-negotiable ontological reality (see Section 6.7.2). Even where acceptable alternative views can be developed, it is useful for university students to learn to 'see' phenomena in the ways that experts presently 'see' them.

Criticism 3: the quality of qualitative research

Phenomenography is, by and large, a qualitative research approach, and many general criticisms of qualitative research also apply to phenomenography. These include doubts about the reliability, objectivity, generalizability, and overall trustworthiness of interpretive research. I will return to these issues in Chapter 16 when discussing the quality of some of my own research. For now, it must suffice to point out that the rigor of qualitative research is typically evaluated differently than that of traditional quantitative research. Many qualitative researchers do not even attempt to fulfill all the goals of traditional quantitative research, such as replicability.

Criticism 4: unenlightening results

Another of Webb's (1997) criticisms concerns the predictability of phenomenographic results, suggesting they tend to tell us little that is new.

> [Phenomenographic activities are] *informed by theory and prejudice. It seems likely then that phenomenographic research will tend to report the history of a particular discipline as it is understood by the researchers and as they reconstruct it through the people they interview. Phenomenographic explanation is prone to reproduction of the discourses it studies.* [. . .] *A phenomenographer considers himself to have constructed* [the different ways of understanding he reports] *from raw data. It is only of passing interest that these conceptions have a close similarity with the history of the discipline. A critic might ask how it could be otherwise.* (Webb, 1997, pp. 196–197)

Phenomenographers argue back that it is not assumed that the researcher's own understanding does not affect the results, but that rigorous steps are taken to 'bracket' the researcher's understanding of the phenomenon and be as open-minded as possible. Merely reproducing known wisdom is a risk in phenomenography (as well as in qualitative research in general and, indeed, research in general). However, there are clear examples of phenomenography informing education by challenging the existing consensus of the mainstream (an example is Neuman's study of how children experience numbers, described in Marton and Booth, 1997, pp. 57–67).

Criticism 5: doomed to repeat SOLO?

Many phenomenographic outcome spaces bear a resemblance to the SOLO taxonomy (Section 2.2): the lowest-ranking categories describe unistructural ways of understanding that focus on a single aspect of the phenomenon, and richer categories describe increasingly complex multistructural ways of understanding that lead to a coherent relational way of understanding. The outcome space is sometimes capped off by an extended abstract understanding in which the experiencer is capable of taking the phenomenon into novel contexts. Table 7.3 is an example of an outcome space of this kind. This begs the question: are phenomenographic studies doomed to do little more than replicate the SOLO taxonomy in different contexts?

The counterargument is that even when phenomenographic outcome spaces do resemble SOLO, they still describe evidence-supported, concrete instantiations of SOLO for particular phenomena, and detail what particular dimensions of variation pertain to the unistructural, multistructural and relational ways of understanding those phenomena. Prosser and Trigwell (1999, p. 120) point out that SOLO – as its name implies – is concerned merely with the general structure of learning outcomes, whereas phenomenographic outcome spaces describe both the structure of learners' understandings and the meanings learners assign to particular phenomena (see Section 7.1). This concrete information is helpful for teachers as an analytical tool and helps provide "a notional path of developmental foci for instruction" (Marton and Booth, 1997, p. 81). Taking the SOLO taxonomy to different contexts in a rigorous, empirically founded way is not an unworthy undertaking. In this sense, phenomenographic outcome spaces are considerably more than what you get by 'simply' applying SOLO to a phenomenon.

Chapter 8

Interlude: Can We All Just Get Along?

The learning theories reviewed in the preceding chapters come from different research traditions, which use different kinds of research methods and base themselves on different ontological and epistemological assumptions. The final sections of Chapters 5, 6, and 7 show that the traditions and theories are in conflict with each other, with some heavy criticism flying in all directions. It makes sense to pause for a moment to consider whether it is sensible, or indeed possible, to make use of all these frameworks simultaneously.

Section 8.1 below briefly compares the theories and traditions that I introduced in the previous chapters. In Section 8.2, I endorse combining theories to inform educational research from several perspectives.

8.1 There are commonalities and tensions between the learning theories

Figures 8.1 through 8.4 list some claims about learning originating from cognitive psychology[1] (Chapters 4 and 5), constructivism (Chapter 6), and phenomenography (Chapter 7). The reader should note that each of the three headings within these tables corresponds to a broad area, and that these broad areas overlap with each other – there are many proponents of cognitive constructivism, for instance. My intention is not to make sweeping statements and claim, for example, that all constructivists believe every one of the claims I have attributed to constructivism. I merely seek to highlight a selection of significant claims that originate from, and have relatively wide support within, each tradition.

The three traditions are in agreement on some points, in disagreement on others. Sometimes their claims are orthogonal with each other.

Ontological and epistemological issues (Figure 8.1) are at the heart of many of the arguments between proponents of the various theories and research traditions; these concern the nature of learning as well as the trustworthiness of scientific knowledge and the research methods suitable for rigorous research.

When it comes to how learning takes place, each tradition has its own points of emphasis (Figure 8.2). The cognitivist theories that I have reviewed tend to be the most detailed and specific in this respect. Phenomenographers tend to focus on the preconditions for learning which are realized in the variation present in the learning situation (as opposed to the generic processes that take place when the learner learns something). Many constructivist writers are very vague about knowledge representation and the processes of knowledge construction; some consider these to be idiosyncratic. Some constructivists – often with one foot in the cognitivist camp – make use of schema theory or conceptual change theory (Section 6.5).

On the pedagogical side, there are clear tensions between the three camps, especially between strong forms of constructivism and others (Figure 8.3). Where constructivists tend to encourage authentic, complex learning activities that are carried out in collaboration with others and chosen by learners themselves to correspond to their own goals, various cognitive psychologists (Section 6.8) are careful to remind us of the importance of 'managing' learning so that cognitive load does not become excessive and is fruitfully employed through practice to generalize from instances to increasingly automated schemas. This does not always need to involve an authentic, complex context.

[1]In Figures 8.1 through 8.4, "cognitive psychology" refers to schema theory, mental model theory, and cognitive load theory in particular.

Figure 8.1: Views from three traditions: the foundations of learning and research

Cognitive psychology
Knowledge about the world is represented mentally.
The mind is (metaphorically) an information-processing machine.
Working memory is very limited, which constrains simultaneous manipulation of knowledge.
Reliable scientific knowledge is gained primarily through hypothesis testing, experimental setups, and quantitative analysis.

Constructivism(s)
Knowledge is a subjective (or social) construction.
Different individuals/societies have different knowledge and different 'truths'.
Knowledge may not reflect any external world.
Qualitative, interpretive inquiry is (also) needed to gain scientific insight.

Phenomenography
Knowledge resides in experience, which is an inextricable two-way relationship between person and world.
Any phenomenon is understood only in a small number of qualitatively different ways.
Humans have a very limited capability for simultaneous focal awareness of phenomena and their aspects.
Qualitative, interpretive inquiry can illuminate human experience on a collective level.

Figure 8.2: Views from three traditions: mechanisms and processes of learning

Cognitive psychology
Excessive cognitive load on working memory prevents learning.
Domain-specific schemas (mental representations of concepts and patterns) are a key ingredient of expertise.
Abstraction to schemas facilitates chunking: ever larger elements can be dealt with as a single chunk.
Lengthy practice leads to ever more automated schemas which no longer strain working memory.
People are guided by their mental models of particular objects or systems; these models are often simplistic and incorrect.
To successfully simulate a system's behavior, a robust mental model of the system is needed.
With increasing experience, initial mental models of specific systems generalize to abstract, transferable schemas.

Constructivism(s)
Specifics of knowledge construction are often left vague; however, new knowledge is seen as crucially building on prior knowledge.
Some constructivists make use of cognitive theories such as schema theory. Some advocate conceptual change theories. Some emphasize the plurality of perspectives to learning.
Social constructivists emphasize the interpersonal nature of learning and its situatedness within the authentic practice of communities.

Phenomenography
Each phenomenon has certain educationally critical aspects; each way of experiencing a phenomenon can be described in terms of these aspects.
Each such critical aspect is characterized by a dimension of variation.
Discerning variation along a dimension leads to discerning the critical aspect and to new ways of experiencing the phenomenon.
The most significant form of learning is learning to experience phenomena in new, more powerful ways that permit more powerful ways of acting.
Learning proceeds from understanding vague wholes to understanding ever more detailed parts within the wholes.

Figure 8.3: Views from three traditions: pedagogy

Cognitive psychology
Pedagogy should involve practice that helps the formation and automation of schemas.
Decontextualized, deliberate practice is a valuable tool in teaching.
Emphasizing patterns and similarities between cases can aid in schema formation.
Cognitive load should be managed through the careful sequencing of examples and problems.
Complex, authentic problems often lead to cognitive overload.
Active engagement with learning content helps learners form better mental models.
Teachers should seek to find out students' misconceptions about important content and to correct them.

Constructivism(s)
Learners should have a driving role in goal-setting; teacher- or institution-given norms for education are questionable.
Students should be heavily involved in the design of learning activities and their assessment.
Learning works best in rich and complex problem-driven contexts.
Learning tasks should reflect authentic practice by experts.
Learning should involve collaboration between students and their peers, and between students and experts.
Misconceptions or alternative frameworks are inevitable and can be used as a basis for further learning.
Teachers should explore students' prior knowledge and leverage it in teaching.

Phenomenography
Some ways of experiencing phenomena are better than others; education should have norms against which understandings can be judged.
Suitable pedagogies must be discovered separately for particular disciplines and particular subject matter.
A learning situation provides a space of variation in which certain variation can be discerned but other variation cannot.
The teacher's task is to find the best way to make it possible and likely for students to experience the critical variation.
Learning situations should engage the students' interest to explore the variation that is present.
Teachers should explore students' understandings of specific phenomena to discover the critical dimensions of variation.

Figure 8.4: Views from three traditions: programming education

Cognitive psychology
Both novices and experts may use top-down and bottom-up strategies for program comprehension, depending on schema availability.
In the absence of high-level schemas, novices have to work at a lower level more often than experts, and need to mentally trace programs at a lower level of abstraction.
Students have many misunderstandings of very fundamental programming concepts, such as variables and references.
Many of the problems students have involve the dynamics of program execution and the role of the computer (a notional machine) in the process.
Many novices do not know how to mentally trace program execution, nor are they much inclined to do so.
A robust mental model of a notional machine is important for many debugging tasks.
A conceptual model of a notional machine is useful as a teaching aid.

Constructivism(s)
Programming education should involve active work on complex projects that are as realistic as possible.
Novices often have no effective prior knowledge of how the computer works.
Students construct idiosyncratic knowledge of how the computer works at lower levels of abstraction than the code level.
Teaching in CS1 should address a lower level of abstraction than the code level the learners normally operate at.
Program dynamics in particular are a source of many misconceptions, which are troublesome because the computer is an accessible ontological reality that reacts to novice misconceptions in an unforgiving manner.

Phenomenography
Sometimes, novice programmers see programs merely as pieces of text, and programming merely as writing instructions.
One early challenge for the novice programmer is to go beyond this static perspective and to understand programming from other perspectives.
In particular, learning to program involves learning a new way of thinking in terms of what the computer does at execution time.
Other challenges include seeing the relationships between code and the real-world problems that programs solve.
Novices also understand various programming concepts (e.g., object, class, variable) in limited ways that restrict programming ability and further learning.

There is substantial agreement between the traditions, too. There is widespread agreement, for instance, on the usefulness of engaging learning activities that require the learner to take an active role in manipulating the content that is to be learned. Further agreement can be found where the theories have been applied to learning computer programming (Figure 8.4). Cognitivists, constructivists, and phenomenographers alike have reached the conclusion that a, if not the, crucial challenge for many novices at the CS1 level is to come to understand the relationship between program code and the computer, which is realized in the run-time dynamic behavior of programs.

Finally, I should mention that within each of the three traditions I have discussed, there exists a movement that emphasizes the social aspects of learning. This is perhaps most prominent within constructivism, where social constructivism and situated learning theory are highly influential. I admit the value of socio-cultural theories of learning as an additional perspective, but my focus in this thesis is not on the social aspects of learning. I will briefly discuss – mostly criticize – my work from the perspective of situated learning theory in Section 14.5.

8.2 Multiple theories give complementary perspectives

The ontological and epistemological assumptions of some schools of thought are difficult or impossible to reconcile, except by accepting them as alternative frameworks for thought, which can be compared and contrasted to gain new insights. The mental representations of a cognitive psychologist and the monist mind–world relationships of a phenomenographer, for instance, are two explicitly different ways of looking at the nature of knowledge, each with its own strengths and issues. Nevertheless, each of these views is fundamental to a research tradition that has expanded our understanding of what it takes to learn in general, and what it takes to learn computer programming in particular.[2]

The strengths of schema theory, cognitive load theory, and much of educational psychology lie in their detailed and rigorous consideration of the general processes and mechanisms of thought and learning. The strengths of phenomenography lie in the careful exploration of the ways in which particular contents of learning are crucial to the success or failure of the educational enterprise. As for the various flavors of constructivist theory, their main contribution to present-day education is arguably as torch-bearers for certain pedagogies which have previously been underappreciated, such as active and collaborative learning and learner empowerment.

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.

Multiple learning theories can also suggest, for projects such as my present one, different kinds of research approaches that complement each other; Thota et al. (2012) argue that drawing from different paradigms is useful, even necessary, for the present-day computing education researcher. According to the pragmatist 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, pp. 5–6, from the 1868 original). In this spirit, I take an eclectic and pluralistic view of learning theory. This view is consistent with the pragmatic perspective on mixed-methods research advocated by Johnson and Onwuegbuzie (2004), who draw on classical pragmatists such as Peirce. I am inspired by multiple learning-theoretical perspectives, and will make use of multiple research approaches in my own empirical work in Part V of this thesis.[3]

End of interlude. Now, one more theoretical framework of learning.

[2]There are of course still other perspectives on learning, beyond what I can cover here, such as those provided by biochemistry and activity theory, for instance (see, e.g., Jonassen, 2009). The traditions I have covered in my review are arguably the ones that have had the greatest influence on research on introductory programming education.

[3]Am I sitting on fences? I prefer to think of it as a small contribution to fence-lowering.

Chapter 9

Certain Concepts Represent Thresholds to Further Learning

An introductory programming course has a lot of content to cover in a limited time. What makes matters worse is that often a student gets stuck. There is something in the curriculum that he just cannot seem to get past. He cannot cope with the content that follows because he is 'thinking about it the wrong way'; he feels frustrated, insecure, and angry. Too often, the teacher forges ahead to new topics, leaving the student trailing.

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.

Section 9.1 is a brief overview of threshold concept theory. In Section 9.2, I comment on the challenges of identifying threshold concepts. Section 9.3 reviews the work within CER that has sought to identify programming-related threshold concepts. Finally, Section 9.4 considers some pedagogical implications.[1]

9.1 Mastering a threshold concept irreversibly transforms the learner's view of other content

A threshold concept is not a mere 'core concept'. To qualify as a TC, a concept must meet a stricter definition, albeit one which is still being debated. Meyer and Land (2006) list five characteristics that a threshold concept is likely to have.

- A threshold concept significantly *transforms* how the student perceives a subject or part thereof, and perhaps even occasions a shift of personal identity;
- it is probably *irreversible* so that the transformation is unlikely to be forgotten or undone;
- it *integrates* other content by exposing its interrelatedness;

[1]This chapter has been adapted from an earlier publication. Large swaths of text have been reproduced from: Juha Sorva: "Reflections on Threshold Concepts in Computer Programming and Beyond", in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pp. 21–30 ©2010 Association for Computing Machinery, Inc. `http://doi.acm.org/10.1145/1930464.1930467` Reprinted by permission.

- it may *mark boundaries* in 'conceptual space' between disciplines or schools of thought; and
- it is potentially very *troublesome* to students for any of a variety of reasons including conceptual complexity, tacitness in expert practice, apparent meaninglessness, and counter-intuitivity.

The literature attributes some additional characteristics to threshold concepts. I will briefly discuss three: 1) crossing a threshold involves a state of liminality; 2) the resulting transformations lead to new ways of thinking and practicing, and 3) threshold concepts are often associated with generic 'everyday' ideas.

Liminality

A *liminal space* is the transitional period between beginning to learn a threshold concept and fully mastering it (McCartney et al., 2009; Meyer and Land, 2006). While some learners cross some thresholds quickly (as a single a-ha! moment) and sometimes effortlessly, liminality often lasts for a considerable length of time. The liminal space is a fluctuating place of transformation: students in liminal spaces tend to oscillate between old and new states and experience strong, often negative emotions, and may attempt to mimic the behavior of others whom they perceive as having crossed the threshold already. Meyer and Land argue that *pre-liminal variation* – variation in students' perceptions of a threshold concept as the concept first 'comes into view' – plays a key role in how and why some students negotiate liminal spaces productively while others struggle and may give up altogether.

Ways of thinking and practicing

A *way of thinking and practicing* within a discipline or subject area (or community of practice; cf. Section 6.6 above) involves concepts, forms of discourse, values or indeed "anything that students learn which helps them to develop a sense of what it might mean to be part of a particular disciplinary community" (McCune and Hounsell, 2005). Many authors have commented on the apparent relationship between threshold concepts and ways of thinking and practicing: mastering a TC provides a new 'lens' through which to view the subject, opening up new avenues for thought and action (Meyer and Land, 2006; Land and Meyer, 2008; Moström et al., 2009; Zander et al., 2009). Conversely, failing to internalize a TC and make it a part of how one thinks and acts leaves the student stuck. Without crossing the threshold, the student cannot perceive other concepts, new problems, potential solutions, and indeed the very subject they are studying, like a full-fledged member of the disciplinary community would.

Transforming ideas from everyday life

Another framework for curricular analysis that may be used alongside threshold concepts was formulated by Schwill (1994) from the earlier work of Bruner (1960). *Fundamental ideas* are lasting, general ideas of broad, perhaps universal significance. They connect topics within and beyond the borders of an academic discipline. The distinguishing feature of a fundamental idea is its wide applicability – across disciplines, across levels of education, across time, and across the divide between academic pursuits and everyday life. Algorithmization, language, abstraction, and state, among others, have been proposed as fundamental ideas of computer science – they characterize the discipline and yet are applicable beyond its confines (Schwill, 1994; Sorva, 2010).

Fundamental ideas have an 'everyday' character. Building from an observation by Zander et al. (2008), I have suggested that threshold concepts often involve the transformation of one or more fundamental ideas into discipline-specific forms (Sorva, 2010). For instance, the (possible) threshold concept of information hiding is a form of the universal notion of abstraction that is central to how computer scientists think (cf. Colburn and Shute, 2007). The everydayness of the fundamental idea(s) involved in a TC means that students have intuitive pre-liminal understandings of aspects of a TC. Everyday understandings are often 'obvious' and tacit, and may be difficult to change. Shinners-Kennedy (2008) provides a good discussion of how the notion of state – quite common and unproblematic in everyday life – becomes troublesome once the student has to make it central to their conscious thinking and relate it to programming.

9.2 Identifying threshold concepts is not trivial

Meyer and Land (2006) use very cautious language as they describe the characteristics of threshold concepts. A TC "is likely" (p. 7) to have the five main features listed in the previous section; it opens up a way of thinking that "may represent" (p. 3) how people think within a particular discipline or about particular phenomena; it is "often more" (p. 101) than just a core concept, and so forth. Additionally, Meyer and Land do not clearly define what they mean by "concept".

Meyer and Land's caution, although it has been criticized (Rowbottom, 2007), is understandable, since the threshold concept theory is still a young one. Moreover, constructivist learning theory (Chapter 6) suggests that people have different kinds of knowledge structures, and what is a threshold for someone may not be one for someone else – a notion that has also been pointed to as a weakness of the threshold concepts framework (Rowbottom, 2007). However, as nature restricts our knowledge-constructing activities (Section 6.3), certain concepts may be particularly troublesome and transformative (etc.) to most people.

Thought-provoking though it is, Meyer and Land's fairly loose definition of threshold concepts does not make the job of identifying TCs very easy. A variety of interpretations exists concerning what counts as a TC. As I see it, there are two main issues: the use of the word "concept", and different opinions as to how well a candidate concept has to fulfill each of Meyer and Land's five main criteria.

A threshold what?

Various scholars operating within the threshold concepts framework have found it useful to consider educational thresholds that are not necessarily individual concepts with commonly accepted names, but something broader. Lucas and Mladenovic (2006) suggest that learning accounting involves a "threshold conception" – a transformation of world-view not associated with a single concept. Savin-Baden (2006) argues that problem-based learning may become a "threshold philosophy". Some authors simply avoid the word "concept", and speak merely of thresholds. One recent trend in threshold concepts research, suggested by Davies and Mangan (2008), is to characterize disciplinary ways of thinking not so much via individual concepts, but as "webs" of interrelated TCs.

A related issue is that one does not simply 'get' or 'not get' a concept – one gets it in a particular way (cf. Chapter 7). Crossing a threshold means understanding something – which may previously have been understood differently – in a way that opens up a powerful new perspective.

My use of the word "concept" in "threshold concept" may be read as shorthand for "way of understanding certain curricular content".

How TC does it need to be?

Clearly, the concept of pointer does not integrate as much content as the concept of information hiding. Does that make it less of a threshold concept? How widely does a threshold concept have to integrate? How radical does the resulting transformation have to be? Should we keep the bar high, with fewer TCs per subject area, or low, with a smaller difference between a core concept and a TC? Some suggested threshold concepts (in computing and in other disciplines) are claimed to be responsible for large-scale transformations of students' views of an entire discipline, while other 'local thresholds' only integrate a handful of neighboring concepts. Rountree and Rountree (2009) have noted how difficult it is to agree on the granularity of threshold concepts, and to expose possible hierarchies of threshold concepts at different levels of granularity.

For me, much of the promise of the threshold concepts framework lies in how a threshold concept is curricular content that is not just another bit of content, but provides a way of dealing with a lot of other concepts. Of greatest interest are those concepts whose effect is the greatest. As Perkins (2006) puts it, TCs include concepts that are more than particularly tough conceptual nuts – some threshold concepts shape students' sense of entire disciplines or large subject areas, giving them access to new ways of reasoning, gaining knowledge and problem solving. Below, I set the bar high, as I discuss candidates for 'big' TCs that change how students think about and practice computer programming, and that provide them with new tools for tackling many kinds of programming problems.

Telling TCs apart from other content

I have argued (Sorva, 2010) that if we are to learn about the nature of threshold concepts and identify them, we need to learn to discern contrasts between TCs and other forms of curricular content. One such form is fundamental ideas, introduced above. The gently developing, extremely broad nature of fundamental ideas contrasts with the drastically transformative, troublesome, irreversible, discipline-specific nature of TCs. Identifying fundamental ideas may also help the search for threshold concepts in that transformative thresholds may be found where fundamental ideas from everyday thought 'meet' a discipline.

Another non-TC form of content is what I have called a *transliminal concept* (Sorva, 2010), that is, a concept that 'lies across the threshold'. A transliminal concept is not in itself transformative or crucial for developing a new way of thinking and practicing, although it may certainly be a core concept which is well worth understanding. However, mastering a transliminal concept is predicated on first acquiring a new way of thinking and practicing by crossing the threshold. Learners may struggle with transliminal concepts, sometimes not because the concept itself is so troublesome, but because their progress is blocked by the barrier of the prerequisite threshold concept. Many transliminal concepts can be associated with a single threshold concept; this is indeed likely to be the case, as a threshold concept has wide applicability within a discipline.

9.3 The search is on for threshold concepts in computing

Learning to program is not just the accumulation of commands, templates, and strategies, but the development of a new way of thinking (Section 7.5). In the words of one of the students quoted by Moström et al. (2009), one needs to become more than "just someone typing in code". The threshold concepts framework suggests that coming to grips with certain content is decisive in transforming a 'code typist' into a programmer. But which content? In Section 9.3.1 below, I argue that program dynamics represents a 'big' transformative threshold that beginners must cross. Section 9.3.2 reviews other candidate threshold concepts within programming.

9.3.1 Program dynamics is a strong candidate for an introductory programming threshold

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 existence of the latter aspect of a program, which I will call *program dynamics*, is much less so.

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; see Section 5.4). The ability to view programs as dynamic is required to genuinely understand a legion of transliminal concepts and distinctions: variables and values; function declarations vs. function calls; classes, objects, and instantiation; expressions and evaluation; static type declarations vs. execution-time types; scope vs. lifetime; etc. The dynamic use of memory to keep track of program state is central to much of this integrative power (cf. Vagianou, 2006; Shinners-Kennedy, 2008).

Transformation

Learning to see a program in dynamic terms transforms a learner's understanding of programming concepts. A phenomenographic study by Eckerdal and Thuné (2005) showed that there is an educationally significant qualitative difference between seeing an object as a piece of code and seeing it as something active (dynamic) in a program. Even more pertinently, the same authors found a more generic difference between how students see computer programming as writing code or as a way of thinking that relates program code to what happens during execution (Thuné and Eckerdal, 2009, and see Section 7.5).

Program dynamics further takes the everyday notion – fundamental idea – of state and transforms it into something central in 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 possible the key skill of program tracing: stepwise reasoning about runtime behavior in terms of what the notional machine does as it executes a program (Section 5.5).

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. We have seen in the preceding chapters plentiful evidence of novices failing to learn what happens when a program is run. From a threshold concepts perspective, Vagianou (2006) comments on the use of memory in programming, noting that novice programmers may not "realize *how* such use takes place and their active role (through the program) in this process". In light of the evidence, it is not too surprising that novice programmers often do not systematically trace the execution of their programs, sometimes because they do not know how, sometimes because they fail to perceive that as a useful pursuit (Section 5.5).

Part of the troublesomeness of program dynamics lies in its tacitness. As I noted in Section 5.5, 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, or 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 counter-intuitive to novices (Shinners-Kennedy, 2008).

Boundaries

Vagianou (2006) points to how end users and programmers have different stances towards 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 non-programmers. 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 vs. al., 1989, and see Section 14.5.3).

Irreversibility

I have little to offer in the way of research-based evidence of irreversibility. However, I have never heard of a programmer forgetting how to see programs as dynamic, traceable entities once they have made that concept their own, nor do I 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 novice who has not yet crossed this early 'obvious' threshold.

9.3.2 Other programming thresholds have also been suggested

Much of the work on threshold concepts within CER has been done by various lineups drawn from the multinational "Sweden Group" of researchers. In an early paper, Eckerdal et al. (2006a) discussed the relationship of threshold concepts to other theoretical frameworks that are influential within the context of computing. Based on the literature, they suggested abstraction and object-orientation as two possible threshold computers in computing. In their later work, they found evidence in student interviews to support the claim that object-oriented programming and pointers could be threshold concepts (Boustedt et al., 2007). The latter was later rephrased as "memory/pointers" (Zander et al., 2008); I have suggested (Sorva, 2010) that the pointer may be a transliminal concept for the TC of addressable memory.

Moström et al. (2008, see also Thomas et al., 2010) returned to the candidate TC of abstraction and argued on the basis of their data that abstraction *in general* does not appear to be a threshold concept even though they found evidence of transformative, integrative and troublesome experiences that students have had with specific forms of abstraction in software design and implementation. Building on their work, I suggested information hiding as a threshold concept that transforms the fundamental idea of abstraction to a discipline-specific form (Sorva, 2010).

The "Sweden Group" have also found evidence supporting the existence of liminal spaces in computing education (Eckerdal et al., 2007; McCartney et al., 2009) and described the transformations of identity and ways of thinking and practicing that take place as computing students learn threshold concepts (Moström et al., 2009; Zander et al., 2009).

Vagianou (2006) proposed program–memory interaction as a threshold concept in introductory programming; my discussion of program dynamics extends her argument. Reynolds and Goda (2007) suggested that the pervasive themes given in ACM curricula (e.g., abstraction and professionalism) fulfill the criteria for threshold concepts. As discussed above, such topics might be better interpreted as fundamental ideas; the same goes for Shinners-Kennedy's (2008) suggestion of state. Flanagan and Smith (2008) contended that in introductory programming, the nature of programming languages is an overall threshold that students need to overcome before being able to tackle smaller "local thresholds" (transliminal concepts?) such as the Java interface construct.

Rountree and Rountree (2009) critically review the literature on threshold concepts and conclude by emphasizing how expert computer scientists, rather than students, should be focal in the ongoing effort to identify threshold concepts. They mention generics and recursion as possible TCs within computing. Holloway et al. (2010) considered recursion as well as object-orientation to be threshold concepts as they sought to develop a quantitative instrument for assessing whether a concept is a TC or not. I have suggested that the idea of object interaction – objects collaborating through message passing – is a TC that opens up to the object-oriented paradigm (Sorva, 2010; see also Sien and Chong, 2011).

9.4 Pedagogy should center around threshold concepts

The programming curriculum is a conceptual space inhabited by entities of various kinds. Teachers need to identify which areas of knowledge constitute transformative thresholds, which threads enter the curriculum as fundamental ideas, and what roles other important concepts may play in students' attempts to navigate the conceptual space. Different pedagogical solutions may be required for teaching TCs than for other content.[2] Teachers may influence the paths that students take, affecting when a TC is first seen, the angle from which it is initially viewed, and the transliminal concepts that are seen behind it. It is not likely that any sequencing of content is universally better than all the others, but the teacher who bears in mind the characteristics of different kinds of conceptual content is better positioned to make good decisions.

I will use the proposed TC of program dynamics as a running example as I consider some pedagogical implications of threshold concepts.

9.4.1 Threshold concepts demand a focus, even at the expense of other content

From a pedagogical point of view, threshold concepts are supremely important, as lack of mastery of a threshold concept renders further teaching inefficient if not entirely fruitless. In Cousin's (2006) words, the threshold concepts framework suggests "a less is more approach" to teaching: the teacher should spend time and effort on the selected concepts that transform the student's view of the discipline and make 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 instantiation 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

[2]For instance, it might be that fundamental ideas are initially best approached by fostering learners' everyday intuitions (as suggested by Bruner, 1960), but that TCs require actively challenging established everyday thinking in pedagogy.

execution step by step. Without the TC of program dynamics, understanding 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" (Meyer and Land, 2006).

Perkins (2008) notes that there is a cost to be paid for teaching TCs – just as they are hard to learn, they are hard to teach – but that it is a cost that is generally worth paying. TCs demand an emphasis in teaching and assessment, even at the expense of other concepts. 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.

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. Each threshold concept within a curriculum should be worked on enough to make sure that students genuinely work them into their ways of thinking and practicing. This requires time: the student who is rushed may experience "psychological vertigo" (to borrow an expression from Booth, 2006), trip on the threshold, and fall flat on their face.

9.4.2 Teachers should make tacit thresholds explicit and manipulable

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 them deal with uncertainty and the emotional issues involved. 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. Students need to be engaged in actively and consciously manipulating each threshold concept in order to internalize it.

Consider again the example of program dynamics. CS1 teachers should try to help their students become aware of the importance of this concept, its possible troublesomeness, and how the students themselves think about programs and reason about what programs do. One way to accomplish this may be through tools, visualizations, and metaphors that concretize the dynamic aspects of programs. By making program dynamics visible, the teacher may not only help the student to think about programs dynamically but make the student more aware of what they are doing. A student who is aware of thinking about program execution as a dynamic step-by-step process – perhaps in terms of a visualization – may find it easier to grasp the general principles embodied in the TC and relate them to multiple contexts.

Meyer and Land (2006) call for a pedagogy of threshold concepts that engages the student in manipulating the conceptual material. This can be a challenging task, since threshold concepts tend to be abstract and tacit. Program dynamics is no exception: as noted, many students appear to be not only unable but disinclined to trace the dynamics of their programs. Students should be placed in situations where they feel they *need* a new way of thinking – a threshold concept – in order to explain something they wonder about or to solve a pertinent problem. Marton and Booth (1997) speak 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". There are many ways to build a relevance structure for program dynamics, for instance. Having students confront the reasons for bugs in their own programs is one option; another is requiring students to predict program behavior and face the fact that their predictions often fail. To build relevance structures, teachers may employ specific transliminal concepts such as pass-by-reference and instantiation. These concepts puzzle students and cannot be fully understood without a shift of perspective. Transliminal concepts may serve as 'educational macguffins' that draw students into and through the 'main plot' of learning a threshold concept. For students to eventually appreciate the power of the generic TC, a combination of multiple transliminal concepts may be needed.

Part III will present many specific examples of making program dynamics visible.