Systemic Issues

12

Novice Programmers and Introductory Programming

Anthony V. Robins

12.1 Introduction

One of the central topics in computing education research (CEdR) is the exploration of how a person learns their first programming language, also described in terms such as understanding "novice programmers," introductory programming, teaching and learning in "CS1" (a first course in computer science), and so on. This chapter explores key issues and surveys some of the important research in this domain.

Programming languages are complex artificial constructs. Like the grammatical rules of natural language, they consist of a relatively small number of elements that can be combined in infinitely many productive ways. The question of how people come to understand and practically apply such a body of knowledge is inherently interesting in the context of disciplines such as psychology and education. There is also a significant practical interest from computing educators. Decades of experience have shown that learning to program is a difficult process for many people. Introductory programming courses typically have high rates of student dropout and failure. This creates significant challenges for educators, who naturally want their students to progress successfully, and sometimes face significant institutional pressures if they do not.

Issues around programming have also been a focus of attention for major IT companies such as IBM and Google and an important focus for the technology sector's main professional and scientific society, the Association for Computing Machinery (ACM). The ACM acknowledges both the challenges facing teachers and the lack of consensus on many topics, and therefore offers little advice on teaching and a range of suggestions on curriculum structure, which is indicative of the wide variety of opinion in the field (ACM/IEEE-CS, 2013). More recently, the question of how best to teach programming has attracted the attention of national teaching organizations and governments, as several countries grapple with how best to deliver technology topics and when to introduce programming in the school curriculum.

In short, the **motivational context** for exploring the teaching and learning of programming includes academic, pedagogical, and practical factors. The field has attracted considerable interest from researchers, teachers, practitioners, industry, and governments alike. There is now a significant body of relevant literature, but many important questions remain open.

The sections of this chapter review the challenge of programming (from various perspectives), literature relating to novice programmers (largely from a cognitive perspective), and recommendations relating to teaching programming (in the context of a typical CS1 course). Although the focus is on CS1, much of this material is relevant to learning programming in any context (e.g., schools).

12.2 The Challenge of Programming

12.2.1 Historical Perspectives

The ACM was founded as a scientific and educational computing society in 1947. Computing developed as a widespread technology and commercial reality during the 1950s and 1960s. Programming rapidly emerged as the central challenge:

> It was generally assumed that coding the computer would be a relatively simple process of translation that could be assigned to low-level clerical personnel. It quickly became apparent that computer programming, as it came to be known, was anything but straightforward and simple. Skilled programmers developed a reputation for creativity and ingenuity, and programming was considered by many to be a uniquely intellectual activity, a black art that relied on individual ability and idiosyncratic style. By the beginning of the 1950s, however, programming had been identified as a key component of any successful computer installation. By the early 1960s, the "problem of programming" had eclipsed all other aspects of commercial computer development.
>
> (Ensmenger, 2010, p. 29)

The early decades of rapid growth were characterized by a continual shortage of programmers, a state of "chronic crisis" (Gibbs, 1994).

Prior to the development of academic qualifications, most programmers were trained "in house" by their employers. The outcomes were very mixed, with high rates of failure. In 1962, the US Army applied its own tests to 190 trainees in their Automatic Data Processing Programming course "in an attempt to reduce the wasted training time and costs associated with the prevailing high attrition rate" (Bauer, Mehrens, & Vinsonhaler, 1968).

The obvious challenges of programming and the desire to reduce the costs of selecting and training successful programmers led to the development of various screening tests. The most significant of these was the IBM Programmer Aptitude Test (PAT), first released in 1955. By the early 1960s, an estimated 80 percent of businesses employing programmers used aptitude tests, around half of them the PAT (Lawson, 1962). In 1967 alone, the PAT was administered to over 700,000 people (McNamara, 1967). Other popular tests included the Computer Programmer Aptitude Battery and the Wolfe Programming Aptitude Tests, as described, for example, by Pea and Kurland (1984).

Despite their widespread use, it was never clear that early aptitude tests were actually effective:

> Ever since the 1950s, when the [PAT] was developed by IBM to help select programmer trainees, consistently modest correlations (at their best from 0.5 to 0.7, hence accounting for only a quarter to a half of the variance), and in many cases much lower, have existed between an individual's score on such a measure and his or her assessed programming skill.
>
> (Pea & Kurland, 1984)

Many studies from the 1960s to the 1980s reported that the predictions of programmer aptitude tests with respect to actual job performance were poor, and that tests of this type should not be administered to university students because the results were unreliable (Robins, 2010). More sophisticated tests were subsequently developed, with the Berger Aptitude for Programming Test (B-APT) emerging as the most popular. Several alternatives for predicting success were also explored, including the use of demographic factors, past high school achievement (SAT scores), mathematical ability, and "general cognitive processes" such as problem-solving strategies. No reliable predictor of programming ability was found, however, and even large-scale analysis of multiple factors results in only limited predictive power (Robins, 2010).

Beyond in-house training, a large number of vocational schools sprang up during the mid-1960s. These were "generally profit-oriented enterprises more interested in quantity than quality" (Ensmenger, 2010, p. 75), with correspondingly poor outcomes. Increasing academic involvement was reflected in the foundation of the ACM Special Interest Group in Computer Personnel Research (SIGCPR) in 1962, and its publication of two major journals, *Computer Personnel* and the yearly *Proceedings of the Nth Annual Computer Personnel Research Conference*. The ACM introduced and popularized the idea of "computer science" as a discipline, and an ACM committee developed the first standardized curriculum. For different reasons, both vocational schools (regarded as too lax) and early academic programs (regarded as too stringent) were seen as problematic, and "Neither was believed to be a reliable short-term solution to the burgeoning labor shortage in programming" (Ensmenger, 2010, p. 80).

Two well-received and influential books appeared in the 1970s: *The Psychology of Computer Programming* (Weinberg, 1971) and *The Mythical Man Month* (Brooks, 1975). Weinberg presented the first empirical study of programming as a complex human activity, setting the stage for much that followed in the field of CEdR. Brooks addressed the practical process of software development and the reasons that so many programming-based projects failed, drawing attention to the need for new ideas about how to manage programmers.

One of the most influential claims to emerge from early research was that of huge variability in the productivity of professional programmers. An early IBM study claimed that a good programmer was at least 25 times more efficient than an average one (Sackman, Erickson, & Grant, 1968), with others claiming an even greater disparity. Sackman et al. opined that "When a programmer is good, he is very, very good. But when he is bad, he is horrid."

The stubborn problems in predicting and training successful programmers, combined with the significant variability in their effectiveness, led to a widespread acceptance of the claim that good programmers are "born, not made" (Dauw, 1967; Webster, 1996). Other persistent and widely held beliefs about programming can also be traced back to these early decades. These include claims that programming is an inherently creative or "artistic" process, that it is not amenable to "scientific" analysis or standard managerial methods, and that programmers are typically male and often socially withdrawn. While many would argue today that these are mostly outmoded myths (see Chapter 16), they have influenced both the nature of CEdR and the public perception of programming. Further perspectives on the history of research into programming can be found in Chapter 1 of this Handbook.

12.2.2 The Nature of Learning Outcomes

The opportunity to learn programming today is available in many forms, including a wide range of academic courses, private and commercial training, online resources such as interactive environments, open course materials, and massive open online courses (MOOCs). This chapter focuses on issues that are relevant to a first academic course in programming, often called CS1 (Computer Science 1). Other modes of teaching and learning are addressed elsewhere in the Handbook.

12.2.2.1 High Dropout and Failure Rates

Consistent with the historical issues surrounding programming, CS1 has typically been regarded as difficult for students, with persistent and widespread reports of high student fail and dropout rates; see, for example, Newman, Gatward, and Poppleton (1970), Garcia (1987), Allan and Kolesar (1997), Sheard and Hagan (1998), Guzdial and Soloway (2002), Beaubouef and Mason (2005), Kinnunen and Malmi (2006), Howles (2009), Guzdial (2010), Corney, Teague, and Thomas (2010), Teague (2011), Mendes et al. (2012), and Watson and Li (2014). This issue has been one of the main practical concerns for computing teachers, and one of the main drivers of research in CEd.

In an attempt to quantify the problem, Bennedsen and Caspersen (2007) surveyed 63 institutions internationally, collecting data on numbers of students who *abort* (drop out), *skip* the final exam, or sit and *fail*. They observed an aggregated failure rate that was on average roughly 33 percent and commonly up to 50 percent or more, but with huge institutional variation (as is to be expected with different countries, institutions, courses, and policy settings). In a follow-up study, Watson and Li (2014) surveyed the literature relating to 51 institutions. The results reported were very similar, with the same average fail rate and huge variation. Fail rates varied by different countries, but did not vary over programming language or improve over time.

Both studies found that small class size was correlated with lower fail rates. Both informally conclude that while fail rates in CS1 courses appear to be high, they are not "alarmingly" so. Unfortunately, both studies suffer from failing to compare CS1 courses with other courses at the same institution, making it difficult to separate the effects of institutional variation from the effects of programming as a subject. (Luxton-Reilly, 2016, compares the 67 percent pass rate found in both studies with an 82 percent pass rate "across all degree-level courses" in New Zealand.) The authors also themselves urge caution in the interpretation of their results. The first study is based on a survey of research-active computing teachers with an institutional response rate of just 12.7 percent. The second is based on an analysis of research literature. In both cases, it is highly likely that the data are drawn from engaged and active sources, possibly presenting the upper end of the spectrum of outcomes. Bennedsen and Caspersen (2007) note that "We hypothesize that if we could see the full picture, things would look very different, but we have no data to support this belief."

12.2.2.2 Fragile Learning

Concerns have also been raised about whether all of those who pass CS1 have learned what they should. An early study of students who had completed a single semester of programming (Soloway et al., 1983) found that when asked to write a loop that calculated an average, only 38 percent were able to complete the task correctly (even when syntax errors were ignored). The averaging task, called the "Rainfall Problem," has been something of a benchmark in the literature ever since. A similar study of students with two years of programming instruction (Kurland et al., 1989) concluded that "many students had only a rudimentary understanding of programming." In an overview, Winslow comments that "One wonders … about teaching sophisticated material to CS1 students when study after study has shown that they do not understand basic loops" (Winslow, 1996).

The most influential work on the limitations of learning in CS1 was the report of a 2001 ITiCSE working group (McCracken et al., 2001). The "McCracken group" consisted of ten authors from eight tertiary institutions in various countries. The group assessed the ability of a combined pool of 216 post-CS1 students using a common set of programming problems selected such that "students in any type of Computer Science programme should be able to solve them" (McCracken et al., 2001). The majority of students performed much worse than their teachers expected, with most failing to finish the problem set. Given the scale and the multinational nature of the collaboration, these results were widely viewed as significant and compelling.

The McCracken study motivated a range of follow-up projects. Utting et al. (2013) revisited the study with improved support for 418 student participants, finding both well- and poorly-performing groups. The Leeds group (Lister et al., 2004) examined the performance of 941 nearly or recently completed CS1 students. The study used multiple-choice questions designed to explore basic programming skills and the ability to trace (follow and reason about) short pieces of code. The results showed that "many students were weak at these tasks," suggesting that "such students have a fragile grasp of skills that are a prerequisite for problem-solving" (Lister et al., 2004).

Unfortunately, such studies suggest that many students are "passing" CS1 courses without a strong grasp of programming basics. This is consistent with ample anecdotal evidence that some students who attempt later programming courses are poorly equipped to do so, contributing to the widespread perception of programming as a difficult topic at all levels.

12.2.2.3 Bimodal Outcomes

Based on the evidence so far reviewed, it is tempting to assume that programming is simply harder than most other topics to learn and to teach. This alone would more than justify the field of CEdR and attempts to find effective methods. A further observation complicates this simple view, however, namely that typical CS1 courses often have an unusually high rate of highly achieving students. It is as if there is a significant subset of students who find programming easy:

> In every introduction to programming course, 20% of the students just get it effortlessly – you could lock them in a dimly lit closet with a reference manual, and they'd still figure out how to program. 20% of the class never seems to get it.
>
> (Guzdial, 2007)

Similar comments from other teachers refer to "two populations: those who can, and those who cannot" Dehnadi (2006), and that there is a "double hump" in grade distributions that "has been observed in programming courses all over the world, largely independent of geographical or social context, and over a long period of time" (Kölling, 2009).

This paradoxical state of affairs has of course been the focus of considerable attention. The term "bimodal" is often used to describe the resulting grade distributions (with higher than usual rates of both failure and of high grades, there are necessarily fewer students in the mid-range). Bimodal distributions are described as characteristic of CS1 in, for example, Hudak and Anderson (1990), Bornat, Dehnadi, and Simon (2008), Corney, Teague, and Thomas (2010), Robins (2010), Yadin (2013), and Elarde (2016). Student outcomes in the influential McCracken study discussed above have also been described as bimodal (Lister & Leaney, 2003) and exhibiting the "two hump effect" (Guzdial, 2010), with a follow-up study (Utting et al., 2013) observing that "there are clearly two distinct populations within the current study's overall cohort" (low- and high-performing groups). Note that a wide variation in student outcomes is consistent with historical observations of the variation in professional programmer performance.

In a widely circulated and commented-on draft (Dehnadi & Bornat, 2006), the authors claimed to have developed a diagnostic test that could accurately predict which students would or would not succeed at programming. Other researchers were unable to replicate the results (Bornat, Dehnadi, & Simon, 2008; Caspersen, Larsen. & Bennedsen, 2007), and the claim of a predictive test was later withdrawn (Bornat, 2014). Patitsas et al. (2016) claim to present "Evidence that computer science grades are not bimodal," but the claim is problematic. The grade distributions analyzed are all from one institution, and they do not include data for students who withdraw (abort). The "psychology experiment" demonstrates that participants can be cued to report bimodality in a noisy artificial dataset (see the issue of subject bias in research design; e.g., Mitchell & Jolley, 2012), but this does not show that the pattern does not exist in real-world data sets. For more on evidence relating to bimodal outcomes, see Robins (2018).

Despite contributing to the practice (Robins, 2010), I now consider the use of the term "bimodal" to be somewhat unfortunate. It has invited a focus on statistical definitions and tests, often without clarity on the underlying definition or counting of abort or skip outcomes (as stressed by Bennedsen & Caspersen, 2007), and often without recognition that binning grades in various ways leads to different results (as demonstrated by Höök, 2015). Given that there are huge variations in institutional outcomes and reporting practice, I suggest that the more useful questions are broader and more contextual. If a given CS1 course has a higher failure rate than comparable courses (at the same institution), then this is a concern. If it also typically has a higher rate of high grades, then this is of interest. If the grade distribution reflects both of these effects, then it might usefully but informally be described as "bimodal." It is probably too late to attempt to change the term, which is in widespread use in the literature, but it should be used with caution. We return to a discussion of potential reasons for "bimodal" outcomes in Section 12.3.7.

12.2.2.4 Summary

It is certainly not the case that all CS1 courses have bimodal outcomes, though reports of this trend are common. Nor is it the case that they all have high dropout and failure rates, though reports of this are even more common (and further concerns have been raised about the performance of passing students). We suggest that both trends will be more typically seen in large courses with open entry to students (compared to small and selective courses) and note that institutional background and policy settings, and the intended scope and nature of each individual course, can significantly impact outcomes. Both trends are consistent with the historical development of programming as a discipline and are observed and commented on frequently enough to be part of the "received wisdom" of the CEd community.

12.2.3 The Task

Given the historical and current educational complexities around learning to program, it is useful to briefly examine what is involved. Programming is often called "coding," but this is a very limited term for the richness and complexity of the task.

12.2.3.1 Requirements

A good overview (du Boulay, 1989) describes five overlapping domains that must be mastered: (1) general orientation, what programs are for and what can be done with them; (2) the notional machine, a general model of the computer as it relates to executing programs; (3) notation, the syntax and semantics of a particular programming language; (4) structures, the use of schemata/plans as ways of organizing knowledge; and (5) pragmatics, the skills of planning, developing, testing, debugging, and so on. While most explicit programming instruction and attention is focused on the third item, in general, of course, a novice programmer will be dealing with many of these issues at once, compounding the difficulties.

In a broad review of the literature relating to novice programmers, Robins, Rountree, and Rountree (2003) summarized the range of topics explored using the dimensions shown in Table 12.1. The columns describe the attributes that are required to write a program, namely *knowledge* of a programming language and tools, the *strategies* for applying this knowledge appropriately, and the capacity to construct and compare mental *models* of program states. The rows describe the stages of creating a program, namely the processes of *design*, *generation* (writing code), and *evaluation*. The cells of this framework should be thought of as fuzzy rather than absolute divisions, and once again, at any given time, an actual programmer will usually be dealing with several of these requirements at once.

Table 12.1 A programming framework. Adapted from Robins, Rountree, and Rountree (2003).

	Knowledge	Strategies	Models
Design	Of planning methods, algorithm design, formal methods	For planning, problem-solving, designing algorithms	Of problem domain, notional machine
Generation	Of language, libraries, environment/tools	For implementing algorithms, coding, accessing knowledge	Of desired programs
Evaluation	Of debugging tools and methods	For testing, debugging, tracking/tracing, repair	Of actual program

Rogalski and Samurçay (1990) summarize the task of programming as involving "a variety of cognitive activities, and mental representations," the "construction of conceptual knowledge, and the structuring of basic operations … into schemas and plans," and the need for flexible strategies. Emphasizing the active and dynamic nature of programming, Green (1990) suggested that programming is best regarded as an exploratory process where programs are created "opportunistically and incrementally." Similarly, Davies (1993) concluded that "emerging models of programming behavior suggest an incremental problem-solving process where strategy is determined by localized problem-solving episodes and frequent problem re-evaluation." Kim and Lerch (1997) describe programming as a process of scientific discovery, with different representations required in multiple "problem spaces."

12.2.3.2 Perceived Difficulties and Errors

Along with the multiple necessary competencies and the complex and interactive nature of the programming process, it is also worth noting that strict constraints apply to the final product of the programming process. In human languages, shared context and "common sense" fill in many of the gaps, while ambiguity and miscommunications abound. In programming languages, the final product must be, at least to a certain functional level, complete, unambiguous, and error free.

Lahtinen, Ala-Mutka, and Järvinen (2005) surveyed 559 novice programming students (and 34 teachers) at six European universities. Respondents perceived the most difficult aspects of programming to be "understanding how to design a program to solve a certain task," "dividing functionality into procedures," and "finding bugs from their own programs." None of these issues relate to knowledge of the specifics of any programming language, or even of general language constructs – they are issues of developing mental models ("understanding") and strategy (Table 12.1).

A related study explored the problems encountered by novice students attempting laboratory tasks for two populations (containing roughly 220 and 250 individuals) over two successive years (Garner, Haden, & Robins, 2005; Robins, Haden & Garner, 2006). The most frequently recorded problems were understanding the task, issues relating to overall program design and structure, and "basic mechanics" (a general category covering typos, trivial syntax errors, missing semicolons, and the like). Of the many specific language-related problems observed, the most frequent related to loops, arrays, and passing data to/from modules. Like the study discussed above, this suggests that developing an overall program design/algorithm is a more difficult task than deploying any particular programming language construct (although in aggregate there are many language constructs to consider).

McCall and Kölling's (2014) review attempts to use the analysis of compiler error messages to classify the mistakes made by novices, pointing out that different conceptual mistakes can generate the same error message, and conversely that the same conceptual mistake can manifest itself in different error messages. The authors hand-analyze 333 error messages and the associated code produced by 240 students and other anonymous users of the BlueJ Java programming environment. The most common errors are “Variable not declared,” “; missing,” “Variable name written incorrectly,” and “Invalid syntax.” This is consistent with the dominance of the “basic mechanics” category in the previous study above.

12.3 Novice Programmers

Many topics relating to novice programming have been explored since the 1960s. During the 1970s through to the 1990s, there was an active and productive focus on the “psychology of programming.” This work drew on concepts from cognitive psychology, such as knowledge representation, problem-solving, working memory, and so on (see Chapter 9).

Several key books marked the development of the field. Weinberg (1971) was influential in identifying programming as an area of psychological interest and stimulating research. The collection of papers in *Studying the Novice Programmer* by Soloway and Spohrer (1989) was a significant contribution; similarly, see Hoc et al. (1990). Sheil (1981) is an early review that discusses a range of methodological issues. Other reviews include Robins, Rountree, and Rountree (2003) and Pears et al. (2007). Drawing on these and other sources, this section notes some of the main trends and topics in this area, with a focus on the cognitive properties of novice programmers.

12.3.1 Properties of Novices

One way of exploring the challenges faced by novices in a field is to compare them to experts. Soloway and Spohrer (1989) outline deficits in novice programmers’ understanding of various specific language constructs (e.g., variables, loops, arrays, and recursion), note shortcomings in their planning and testing of code, explore general issues relating to the use of program plans, show how prior knowledge can be a source of errors, and more. Similarly, Winslow (1996) notes that novice programmers are limited to surface and superficially organized knowledge, lack detailed schemata/scripts/mental models, fail to apply relevant knowledge, and approach programming “line by line” rather than using meaningful program “chunks” or structures. In short, novices are “very local and concrete in their comprehension of programs” (Wiedenbeck et al., 1999). Winslow (1996) states that it takes around ten years for a novice to become an expert programmer.

During the early stages of teaching and learning, the contrast between novice and expert is less important than that between different kinds of novice. Perkins et al. (1989) distinguish between “stoppers,” “movers,” and “tinkerers.” Stoppers are those who stop and appear to “abandon all hope” when confronted with a problem or a lack of a clear direction to proceed. They are likely to be those who are frustrated by or have a negative emotional reaction to errors. Movers are those who keep trying, experimenting with and modifying their code. They can use feedback about errors effectively to solve problems and progress. Tinkerers are extreme movers who are not able to trace their code and may be making changes more or less at random, with little chance of progress.

Perkins et al.’s categories have proved to be very enduring and are still cited. Note, however, that the term “tinkering” is variously applied in the CEdR literature. Berland et al. (2013) review many other uses, concluding that “tinkering” is commonly defined as an “exploratory activity” and is suggested to play a crucial role in successful learning.

Two simple functional categories are suggested by the tendency to polarized/bimodal outcomes in CS1 courses as discussed above (Robins, Rountree, & Rountree, 2003). Effective novices are those who make progress in learning to program, typically leading to successful outcomes. Ineffective novices are those who do not make progress (or require inordinate effort and personal attention), typically leading to unsuccessful outcomes. Both the historical use of aptitude testing and much of the research reviewed below can be seen as attempting to predict whether given individuals will be effective or ineffective novice programmers and/or to understand the properties of these groups.

12.3.2 Knowledge

Learning to program involves acquiring both declarative knowledge (e.g., being able to state how a "for" loop works) and practical strategies for its application (e.g., using a "for" loop appropriately in a program) (Davies, 1993). Of the two, it is knowledge that receives the most explicit attention in typical textbooks and CS1 courses, which usually focus on presenting knowledge about a particular language. Related domains include knowledge of computers, programming tools and resources, and theory and formal methods.

One kind of knowledge representation that has historically been identified as central to both reading/understanding and writing programs is the structured chunk of related content. This has variously been called a schema or frame, or (if action oriented) a script or plan (Ormerod, 1990). For example, most experienced programmers will have a schema for the design of a class with encapsulated data fields and a public interface, or a plan for finding the average of the values stored in a one-dimensional array. There is considerable evidence that the plan is the basic cognitive unit used in program design and understanding, but what specifically is meant by a plan has varied considerably between authors (Rist, 1995).

Soloway and Ehrlich (1984) present a study supporting their claims that expert programmers use two types of programming knowledge: plans ("generic program fragments that represent stereotypic action sequences") and "rules of programming discourse" (the conventions that govern the composition of the plans into programs). Expert programmers are characterized in part by the large number of schemata/plans that they have internalized, and many studies have emphasized the importance of acquiring these organizing structures. Brooks (1990) introduced a special issue of the *International Journal of Man–Machine Studies* devoted to plans and other knowledge representations used by programmers. Soloway (1986) proposed that novices should be explicitly taught about common plans and "stereotypical solutions," as well as how to combine and use them (see also Clancy & Linn, 1999). Rist (2004) describes learning to program as a process of schema creation, application, combination, and evaluation, and explores "how changes in the form and structure of knowledge lead to the different types of behaviour seen at different levels of expertise." A distinct approach to understanding programming based on the explicit use of "design patterns" has emerged; see, for example, texts such as Gamma et al. (1995) and Freeman et al. (2004).

Two related educational theories regarding particularly important forms of knowledge – fundamental ideas (Bruner, 1960) and threshold concepts (Meyer & Land, 2003, 2006) – have both received attention within CEd. Fundamental ideas are those that have "wide as well as powerful applicability" and apply "at any stage of development" (Bruner, 1960). Discussing fundamental ideas in CEd, Schwill (1994, 1997) suggests the following four criteria: *horizontal* (the idea is relevant across many disciplines or sub-disciplines), *vertical* (the idea pervades all levels from elementary through to highly advanced), *time* (the idea is recognized as important and it endures), and *sense* (the idea has meaning in "everyday life"). Schwill argues that candidate fundamental ideas in computing include *algorithmization*, *structured dissection*, and *language*, with more specific ideas definable under each category (e.g., under language are the ideas of syntax and semantics).

Threshold concepts are those that are key challenges in learning a given knowledge domain; if successfully acquired, they enable a qualitatively different understanding. Meyer and Land (2003, 2006) describe them as likely to be *transformative* (creating a new way of viewing, understanding, or describing), *integrative* (allowing new connections and relationships to be perceived), *irreversible* (causing a fundamental change that cannot be "unlearned"), *troublesome* (being problematic to grasp or difficult to integrate into current understanding), and *boundary markers* (helping define the scope of the knowledge domain).

Threshold concepts attracted considerable attention within CEdR. In 2005, a group of researchers from several institutions across Europe and the USA launched an ongoing effort to identify threshold concepts in computer science, resulting in several publications, as described in Shinners-Kennedy and Fincher (2013). An initial informal survey of 36 instructors from 9 countries identified 33 candidate concepts "with most popular being: levels of abstraction; pointers; the distinction between classes, objects, and instances; recursion and induction; procedural abstraction; and polymorphism." Note that "while some concepts came up again and again, there was no universal consensus" (Boustedt et al., 2007).

A lack of consensus has proved to be a general problem within the literature on threshold concepts, with problems of definition, subjectivity, granularity, and the like being identified by several authors (e.g., O'Donnell, 2009; Rowbottom, 2007). Shinners-Kennedy and Fincher (2013) conclude that researchers have reached a "dead end" in the exploration of threshold concepts in CEd, but work continues in other disciplines. Further ideas on the interaction between fundamental ideas and threshold concepts in CEd are explored in both Sorva (2010) and Rountree, Robins, and Rountree (2013).

12.3.3 Strategies

As discussed above, programming knowledge necessarily goes hand in hand with the strategies/skills that are required to apply it. The latest ACM Curriculum Report for computer science (ACM/IEEE-CS, 2013) lists one of its goals as being to "identify the fundamental skills and knowledge that all computer science graduates should possess," and notes that "graduates need to understand how to apply the knowledge they have gained to solve real problems." The distinction between programming knowledge and strategies echoes fundamental distinctions in human memory and cognition between declarative (or semantic) and procedural knowledge, or the philosophical contrast between "knowing that" and "knowing how." The field of mathematics education has a long-standing and important distinction between "conceptual" and "procedural" knowledge; see, for example, Hiebert and Lefevre (1986).

Strategies are relevant at all stages of the programming process, from design to evaluation/debugging (Table 12.1). Design may involve utilizing problem-solving strategies such as divide-and-conquer or means–ends analysis, the use of patterns and analogies, or evaluating task- or language-specific factors that influence the structure of an appropriate program. Program generation involves strategies for implementing the design/algorithm, accessing knowledge as required and applying it appropriately, and using any relevant coding environment or tools. Program evaluation may involve strategies for tracing/tracking, testing, and debugging code.

Within CEdR, several authors have stressed the importance or preeminence of the strategic aspects of programming to successful learning outcomes (Davies, 1993; Perkins et al., 1989; Robins, Rountree, & Rountree 2003; Soloway, 1986), and similar discussion can be found using related terms such as programming skills, practice, or problem-solving. Many of the factors that distinguish expert from novice programmers relate to strategies (Sheil, 1981; Widowski & Eyferth, 1986). Perkins and Martin (1986) show (relating to the fragile learning discussed in Section 12.2.2.2) that both knowledge and strategies can be missing (forgotten), inert (learned but not used), or misplaced (learned but used inappropriately), and note that novices are often observed to be using generic and inefficient problem-solving strategies. Eckerdal (2009) argues that "concepts and practise are equally important parts of the learning goals, and equally difficult for students to learn," and that "there is a mutual dependency and complex relationship between the two." In concluding their discussion of threshold concepts, Shinners-Kennedy and Fincher (2013) noted that "the [Threshold Concept] group altered direction and started to search not for threshold concepts in computing, but instead posited the existence of threshold skills."

Davies (1993) reviews a range of literature on programming strategies and suggests that research should move beyond attempts to simply characterize strategies and instead focus on why they emerge and how they relate to factors such as the problem domain, the specific task, and the programming language and tools. In particular, research should focus on "exploring the relationship between the development of structured representations of programming knowledge and the adoption of specific forms of strategy" (Davies, 1993).

An excellent example of research fulfilling this specification is developed in a sequence of studies of novice and experienced programmers by Rist (1986, 1989, 1995, 2004), reviewed in Sorva (2012). Rist describes top-down, bottom-up, forward-development, backward-development, breadth-first, and depth-first design/programming strategies and mechanisms for schema expansion and combination. Rist suggests that programmers use top-down, forward-developing, breadth-first strategies whenever they have a suitable schema/plan available. In the absence of suitable schemata (e.g., for unfamiliar or particularly difficult problems), programmers revert to bottom-up, backward-developing, depth-first strategies in order to develop new solutions and new schemata/plans for later use. Programmers can use a mixture of these strategies as they work on familiar or unfamiliar sub-problems. Implicit in this theory is that the availability of relevant knowledge is a major driver of strategy, confirming the widely agreed-upon principle that the most significant difference between novices and experts is the richness of their respective experience/libraries of learned schemata.

Also implicit in Rist's framework is that the key factor separating novices (who all lack rich schemata) into effective and ineffective groups is the relative effectiveness of the strategies that they are employing (Robins, Rountree, & Rountree, 2003), and therefore the speed with which problems can be solved and schemata acquired. The knowledge required to support this process is available from a range of sources, with courses and textbooks designed to introduce it in a structured way. Without the strategies for accessing this knowledge and applying it to the practical task of programming, however, successful progress and therefore the acquisition of effective schema cannot take place. Conversely a novice with the right initial strategies can teach themselves to program by drawing on knowledge sources as needed. We suggest that progress in the successful teaching of programming can be made by exploring the following questions: What are the strategies employed by effective novices? How do they relate to their knowledge and their relevant mental models? Can these strategies be taught to ineffective novices?

12.3.4 Mental Models

The concept of a "mental model" has a long history in CEdR. Like "schema," the term is adopted from the cognitive science literature, where it is widely used and variously defined (Gentner, 2002; Gentner & Stevens, 1983; Johnson-Laird, 1983). Mental models are generally held to be internal models of how some aspect of the world works, an iconic representation of selected aspects of external objects and systems. Mental models have predictive power – they can be used to understand the observed behavior of the world and reason about future behavior.

One important model that novices need to acquire is of the "notional machine," an abstraction of the software and hardware of a computer that characterizes its role as the executor of programs, and that therefore provides a context for understanding the behavior of those programs (Cañas, Bajo, & Gonzalvo, 1994; du Boulay, 1986; du Boulay, O'Shea, & Monk, 1989; Hoc & Nguyen-Xuan, 1990; Mayer, 1989). du Boulay (1986) suggests that "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," and likens the task to trying to understand how a car engine works based on a diagram. In the absence of an accurate understanding of a notional machine, novices can develop their own "bizarre theories" about how programs are executed (du Boulay, 1986). Mayer (1989) showed that students supplied with a notional machine model (which he called a "concrete model") were better at solving some kinds of problem than those without the model.

du Boulay, O'Shea, and Monk (1989) suggest that different programming languages afford different features of a notional machine, that they can be used to explain "hidden" actions and side effects of a program's operation, and that they should be should be simple and supported with some kind of concrete tool that allows the machine to be observed (a "glass box" instead of a "black box"). The later requirement is met by Berry and Kölling (2014), who propose an example notional machine and graphical notation for object-oriented programming and introduce an implementation of it within the popular BlueJ programming environment for Java (Kölling et al., 2003). See also the "stepper" in DrScheme for a functional programming example (Findler et al., 2002), and similar examples discussed in Chapter 21.

Sorva (2013) presents an excellent review of the notional machine concept and the misconceptions that can arise from incorrect models, even for topics that most programmers regard as obvious, such as simple assignment. A particular strength of the review is the discussion of notional machines in the context of broader theoretical frameworks such as mental models, constructivism, phenomenography, and threshold concepts. Sorva argues that teachers should "acknowledge the notional machine as an explicit learning objective and address it in teaching," and that in some cases, such as object-oriented languages, teaching "may benefit from using multiple notional machines at different levels of abstraction."

Schulte and Bennedsen (2006) surveyed 457 CS1 teachers on the importance and difficulty of broad categories of programming topics. Among their findings, the authors noted that only 29 percent of respondents explicitly addressed a notional machine in their teaching. While factors relevant to notional machines were rated as important, the topic of notional machines as a whole was rated as relevant but not important. This suggests that the theoretical significance of notional machines has not been successfully communicated to teaching practitioners, or at the very least that there is confusion about the definition. See also Chapters 1, 13, 15, 21, and 27 for other perspectives on the notional machine.

Programming is sometimes described as a particular "way of thinking" (Eckerdal, Thuné, & Berglund, 2005). Beyond the notional machine, writing a program involves holding many details in mind, including the problem domain and target design/algorithm, knowledge of a programming language and tools, the current state of the program, and plans and strategies for proceeding. Many of these requirements have been explored within the framework of mental models.

> Models are crucial to building understanding. Models of control, data structures and data representation, program design and problem domain are all important. If the instructor omits them, the students will make up their own models of dubious quality.
>
> (Winslow, 1996)

Problem domain models have been explored by, for example, Brooks (1977, 1983), Spohrer, Soloway, and Pope (1989), Davies (1993), and Rist (1995), and the interaction between "domain models" and "program models" by Corritore and Wiedenbeck (1991), Wiedenbeck and Ramalingam (1999), Wiedenbeck et al. (1999), and Burkhardt, Détienne, and Wiedenbeck (1997, 2002). The topic of program models is complicated by the distinction between a program as it was intended and the program as it actually is. Designs can be incorrect, unpredicted interactions can occur, bugs happen, and programmers are frequently faced with the need to understand unexpected program behavior. This requires the ability to trace code in order to build a model of the program and its behavior (which Perkins et al., 1989, call "close tracking" and describe as "taking the computer's point of view") and the capacity to compare this model with the intended model/behavior.

In some situations (e.g., major bug fixes), significant alterations to a desired program model may be necessary. Gray and Anderson (1987) call alterations to program code "change episodes" and suggest that they can be rich in information, helping to reveal the programmer's models, goals, and planning activities. Wiedenbeck, Fix, and Scholtz (1993) described expert mental models of programs as grounded in the use of schemata/patterns that are hierarchical and multilayered, with explicit mappings between layers and being well connected internally and well founded in the program text. Novice representations generally lacked these characteristics, but in some cases were working toward them. Soloway (1986) suggested that "learning to program amounts to learning how to construct mechanisms and how to construct explanations," and "language constructs do not pose major stumbling blocks for novices … the real problems novices have lie in 'putting the pieces together,' composing and coordinating components of a program."

One of the earliest studies of mental models and programming is also one of the most comprehensive. Mayer (1985) presented a formal analysis of the models underlying BASIC statements, empirical evidence of the utility of the analysis in explaining the way BASIC is learned, comprehended, and used, and an analysis of common misconceptions. Learning the language is more successful when it is based on rich and relevant conceptual knowledge. Mayer suggests that "specific kinds of mental models can be successfully taught and that such training tends to enhance students' ability to solve programming problems." Other empirical studies that explore the problems arising from misconceptions and illustrate the advantages of rich mental models for learning and transfer include Kurland and Pea (1985), Bhuiyan, Greer, and McCalla (1992), Cañas, Bajo, and Gonzalvo (1994), and Shih and Alessi (1993).

Consistent with the fragile learning discussed above, Ma et al. (2007) explore the viability of mental models at the end of a CS1 Java course. The authors found that "approximately one third of students held non-viable mental models of value assignment and only 17% of students held a viable mental model of reference assignment." Unsurprisingly, students with viable mental models performed significantly better than those with non-viable models. Sorva (2013) presents a useful review of the relevant literature and explores the activity of code tracing as an example of the active/predictive nature of mental models. Sorva concludes that the main challenges to the successful running of a program model are "keeping track of program state in working memory, and the difficulty of forming mental models that are robustly founded on context-free runtime semantics of each construct."

As a final observation, the construction of mental models of programs can clearly be supported by a range of tools such as debuggers, software visualization tools, and features of rich programming environments such as BlueJ; see, for example, Storey, Fracchia, and Müller (1999) and Chapter 21.

12.3.5 Cognitive Load

The concept of "cognitive load" has been a more recent addition to the CEdR literature, again adopted from cognitive science. Cognitive load theory is a broad framework for describing the load placed on (or the "effort" expended by) working memory during the execution of a task (Paas, Renal, & Sweller, 2003; Plass, Moreno, & Brünken, 2010; Sweller, 1988, 1994). As originally proposed, the theory described three kinds of load: *intrinsic* (the difficulty or required effort inherent in a specific task or topic), *extraneous* (effort arising from and varying with the way that information is presented), and *germane* (the effort required to integrate new information into permanent schemata). (Subsequent variations are discussed in Chapter 9.) One of the main determinants of intrinsic load is element interactivity – the extent to which the task involves interacting elements that must be held in working memory simultaneously. Many principles of good pedagogy can be construed as attempts to reduce (particularly extraneous) cognitive load for learners or to promote the useful learning resulting from germane load.

It seems obvious that programming tasks typically involve high element interactivity and therefore high intrinsic load. The most effective way to manage this is to exploit one of the known properties of working memory – our ability to "chunk" elements together into meaningful wholes. The capacity of working memory (number of elements that can be simultaneously "held") is generally taken to be 4 ± 1 (Cowan, 2001). However, what exactly constitutes an "element" is not well defined, and elements can be complex, containing other elements. For example, humans are generally able to recall a list of four single-digit numbers (e.g., 3, 7, 2, 9), but they can also recall four multi-digit numbers (72, 123, 18, 446), which between them contain many more digits. In short, the judicious use of "chunking" is a way to effectively hold many elements in working memory (as parts of more complex elements). This is the reason that structured units of knowledge such as schemata are so important in programming (and cognition generally), where experts are distinguished largely by their learned libraries of useful schemata, and novices experience many difficulties as they work to acquire them.

Cognitive load theory leads to several empirically verified "effects" or "principles" (Plass, Moreno, & Brünken, 2010), some of which are discussed in the context of CEd by Sorva (2012). Whereas learning through problem-solving is a popular technique, the cognitive load for novices (lacking the necessary schemata) is high. The *worked-out-example effect* suggests that extraneous load is reduced by studying worked examples of problems rather than trying to solve the problems from scratch, and similarly the *completion effect* suggests that load is reduced when the learner starts with partial solutions. Other examples include the *guidance-fading effect*, stating that novices need extensive support that can be reduced over time, and the *isolated/interacting elements effect*, stating that tasks with high element interactivity will be learned more successfully if elements are first introduced in isolation before being combined.

Sorva (2012) reviews examples where cognitive load theory has been applied to CS1. van Merriënboer (1990) and van Merriënboer and de Croock (1992) present evidence for the completion effect over two experiments where students who modified and extended existing programs achieved better outcomes than control groups that wrote programs from scratch. Garner (2002) presents a programming environment that facilitates code completion examples. Linn and Clancy (1992) demonstrated advantages for novices who were supplied with expert worked examples compared to novices who designed and wrote their own programs. "These activities emphasize the pedagogical value of reading code, as opposed to merely designing and writing it" (Sorva, 2012).

Evidence supporting the guidance-fading effect is described in Stachel et al. (2013). Student participants in a Visual Basic for Applications programming course were divided into a control group and an experimental group, the latter being provided with an additional scaffolding tool to support laboratory assignments. In the first phase of the study, the experimental group achieved better laboratory scores and reported lower self-rated cognitive load scores. These advantages persisted during the second phase when the scaffolding tool was withdrawn, and the experimental group also achieved a higher average final score. Gray et al. (2007) suggest combining worked example programs and guidance fading to generate "faded worked examples" (i.e., worked example sequences with fewer code steps explicitly provided as the learner progresses through the sequence).

Caspersen and Bennedsen (2007) combine cognitive load theory with related ideas (cognitive apprenticeship, skill acquisition, and worked examples) to describe a CS1 design that utilizes "worked examples, scaffolding, faded guidance, cognitive apprenticeship, and emphasis of patterns to aid schema creation and improve learning." Similarly, Mead et al. (2006) combine cognitive load theory, fundamental ideas, threshold concepts and standard curriculum designs to propose the ideas of "anchor concepts" and "anchor graphs" as tools for curriculum planning in the CS1/CS2 sequence. Alexandron et al. (2014) suggest that "scenario-based programming" using visual "live sequence charts" encourages abstract thinking and ordering tasks by level of increasing complexity and is an effective way of reducing cognitive load. Morrison, Dorn, and Guzdial (2014) adapted an existing "Cognitive Load Component Survey" to the domain of introductory programming and observed the correlations between different components of load over two lectures, concluding that the results replicated earlier studies in the domain of statistics and that the revised survey would be a useful tool for comparing pedagogical interventions.

While the previous section concluded that learning outcomes are improved by tools that aid program visualization and the construction of accurate mental models, research on cognitive load suggests a competing design imperative – the need to keep novice programming tools as simple as possible so as to reduce extraneous load. In a study of novices using a block-based programming environment, Mason and Cooper (2013) conclude that "having extra options available in the environment – even if they are not used or referenced – hinders learning," crucially that it also "causes the students to perceive programming in both that environment and subsequent environments as more difficult," and overall that "novice students benefited from a simplified first-programming environment." A review is beyond the scope of this chapter, but the desire to reduce complexity underlies the design of novice programming environments such as DrScheme (Findler et al., 2002), BlueJ (Kölling et al., 2003), and Greenfoot (Kölling, 2010), of "teaching languages" such as Pascal and ABC (precursor to the currently popular Python), and of block-based programming languages.

12.3.6 Taxonomies and Measures

Cognitive scientists have developed many tools for classifying and understanding people and the learning process. These include personality inventories, developmental models, measures of attitude and motivation, IQ tests, and more. Many of these have been applied to novice programmers, often in an attempt to explain or predict the patterns of success or failure discussed above.

The most influential of these tools is Bloom's taxonomy of learning objectives, which has been widely used in educational research and practice. The original taxonomy (Bloom et al., 1956) described six levels of increasingly sophisticated objectives for learning within the "cognitive domain," namely: *Remembering*, *Comprehending*, *Applying*, *Analyzing*, *Synthesizing*, and *Evaluating*. (Similar frameworks were set out for "affective" and "psychomotor" domains.) The Revised Bloom's Taxonomy (RBT) (Anderson et al., 2001; Krathwohl, 2002) proposes a two-dimensional model: the Knowledge Dimension, defined over the categories of Factual, Conceptual, Procedural, and Metacognitive knowledge (the latter two are effectively "strategies" in the language of this chapter), and the Cognitive Process Dimension, defined over the levels of *Remember*, *Understand*, *Apply*, *Analyze*, *Evaluate*, and *Create*.

Versions of Bloom's taxonomy have been widely applied in computing education and in studies of novice programming. The ACM curriculum guidelines discussion of learning outcomes states that "In defining different levels we drew from other curriculum approaches, especially Bloom's Taxonomy, which has been well explored within computer science" (ACM/IEEE-CS, 2013). The taxonomy has influenced many other curriculum documents, including the CSTA K–12 Computer Science Standards (CSTA, 2017) from the Computer Science Teachers Association. In a useful review, Sorva (2012) notes that there is general agreement that programming involves performance at high (and therefore difficult) levels of the taxonomy. Oliver et al. (2004) used a weighted average of the Bloom levels of assessment items to calculate a "Bloom rating" for a range of computing courses. Their analysis suggested that typical CS1 courses have high Bloom ratings compared to other computing topics.

While there is general agreement that the ability to create a program to solve an unfamiliar problem can be classified at the (original) Synthesizing or (revised) Create level, there is less agreement about more specific tasks. Thompson et al. (2008) note that a task may be classified as Apply if the student has relevant knowledge/experience, but as Create otherwise. Sorva (2012) states that "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." Some of the disagreement may relate to varying assumptions about the backgrounds and current levels of varying students. Bloom's original group stressed the role of prior knowledge in determining relevant levels, and this is consistent with the importance of known schemata and cognitive load, reviewed above. Further CEdR studies that employ Bloom's taxonomy are too numerous to review individually; they include Buck and Stucki (2000), Lister (2000), Lister and Leaney (2003a, 2003b), Scott (2003), Johnson and Fuller (2006), Whalley et al. (2006), Thompson et al. (2008), Starr, Manaris, and Stalvey (2008), Lopez et al. (2008), Alaoutinen and Smolander (2010), Meerbaum-Salant, Armoni, and Ben-Ari (2010), Gluga et al. (2012), Sarawagi (2014), and Ginat and Menashe (2015).

Another very influential educational tool is the developmental stage theory of Jean Piaget (Piaget, 1964, 1971a, 1971b). Piaget defined the following four stages of children's development: *sensorimotor* (from birth to the acquisition of language at around 2 years old), *pre-operational* (to 7 years), *concrete operational* (to 11 years), and *formal operational* (to adulthood at between 15 and 20 years). The later stages in particular are defined largely in terms of the acquisition of logical capacities such as transitive inference. Within CEd, it has been suggested (Barker & Unger, 1983) that developmental stages, particularly the transition to the formal operational stage, may have an important impact on computational thinking. In a brief review, White and Sivitanides (2002) conclude that reaching the formal operational stage "is a required cognitive characteristic of people for learning procedural programming," and they claim that "the majority of adults and many college students fail to develop to full formal operational thinking skills." In contrast, Bennedsen and Caspersen (2006) found no correlation between stage of development (particularly "abstraction ability") and final grade in an introductory object-oriented programming course. Further ambiguous results are reviewed by Lister (2011).

Piaget's theory has been criticized, particularly in terms of the observed variation between the capacities of individuals of different ages and the lack of explanation for the "miraculous" transition between stages (Feldman, 2004). Attempts to address such problems have led to various "neo-Piagetian" theories. These typically distinguish stages of development based on "features of the child's information processing system" (such as speed of processing and working memory capacity) rather than logical competence (Morra et al., 2007), and include "domain specificity" (that an individual may display different levels of performance across different domains; for some task types, performance is highly dependent on relevant knowledge).

Lister (2011) summarizes a strong neo-Piagetian position, "that people, regardless of their age, are thought to progress through increasingly abstract forms of reasoning as they gain expertise in a specific problem domain." Lister goes on to explore aspects of programming within neo-Piagetian interpretations of functionally defined (but still classically named) stages, noting that many students do not progress beyond the pre-operational stage, while much instruction is delivered at the formal operational level (see also Corney et al., 2012). Teague (2015) conducted think-aloud studies with novice programmers and found evidence consistent with the neo-Piagetian model. Falkner, Vivian, and Falkner (2013) present an analysis of students' reflections on their software development processes, characterizing the stages in terms of "representative mental models" based on observed behaviors and strategies.

Inspired by Piaget and influential within CEd, the SOLO taxonomy (Biggs & Collis, 1982) is a general educational framework for describing the "Structure of the Observed Learning Outcome" in terms of levels of increasing complexity, from *pre-structural* (displaying no understanding), to *uni-structural*, *multi-structural*, *relational*, and *extended abstract* (understanding is abstracted to a high level and may be generalized to other tasks or topics). Brabrand and Dahl (2009) analyzed a range of courses at a Danish university (that consistently uses the SOLO taxonomy to specify course goals), finding that programming-related competencies were typically relational, and that computing courses in general had significantly higher levels than mathematics or other science courses.

A range of studies have found that the taxonomy can be fairly consistently applied to evaluating novice programmers (Clear et al., 2008; Lister et al., 2006; Sheard et al., 2008; Whalley et al., 2006). The BRACElet project (an ITiCSE working group) conducted a multiyear, multinational study of novice programmers that analyzed examination answers for both code reading and writing tasks, and refined and extend earlier SOLO level definitions as applied to programming (Lister et al., 2010). Other applications to programming include Jimoyiannis (2013), Ginat and Menashe (2015), Izu, Weerasinghe, and Pope (2016), and Castro and Fisler (2017).

The Bloom and SOLO taxonomies are complementary, and they are sometimes discussed together in the context of CEdR (e.g., Whalley et al., 2006). Creating a program for an unfamiliar task requires performance at the SOLO relational level/Bloom synthesizing or create level. As noted above, programming-related courses have been rated as more challenging than other courses using both the SOLO (Oliver et al., 2004) and Bloom (Brabrand & Dahl, 2009) taxonomies. Difficulties have been observed for both taxonomies in categorizing performance or tasks reliably and in accounting for the way that prior experience/variation in existing knowledge or skills affects classification level. Fuller et al. (2007) discuss these and other problems for applying generic taxonomies to computing. The authors propose a version of Bloom's revised taxonomy adapted to the requirements of computing tasks, conceived of as a matrix through which different paths are possible. Similarly, Bower (2008) proposed a hierarchy of task types based specifically on programming.

In a fascinating study, Margulieux, Catrambone, and Schaeffer (2018) present a methodology that appears to rank learning domains by complexity for the learner. Students solving problems in one of three domains (programming, chemistry, or statistics) were supplied with relevant subgoal-labeled worked examples and/or subgoal-labeled explanatory text. A different pattern of results was observed in the three domains: "While the subgoal labeled worked example consistently improved performance, the subgoal labeled expository text, which interacted with subgoal labeled worked examples in programming, had an additive effect with subgoal labeled worked examples in chemistry and no effect in statistics" (Margulieux, Catrambone, & Schaeffer, 2018). The results suggest that programming is the most difficult domain (both forms of support material interacted in improving performance), followed by chemistry, then statistics. The authors note that "Differences in patterns of results are believed to be due to complexity of the content to be learned."

A wide range of other tests and instruments have been used to explore novice programmers and the factors that influence their performance. The review presented in Robins (2010) covers the following topics: *demographic factors* – following on from early aptitude tests, a range of subsequent studies have explored the significance of factors such as age, gender, ethnicity, marital status, grade point average, mathematics background, science background, ACT/SAT math scores, ACT composite score, SAT verbal scores, high school rank, previous computing experience, and more. *Cognitive capacity* – aptitude tests and related research have used a range of tasks including letter series, figure analogies, number series, verbal meaning, and tests of accuracy, mathematical reasoning, algorithmic execution, alphanumeric translation, deductive and logical ability, the ability to reason with symbols, the detection of patterns, and reading comprehension (many of these are components of common IQ tests as discussed in the next section). *Cognitive style* – Is there a particular learning style or personality type that contributes to success? Tests that have been used to explore this possibility include the Myers–Briggs Type Indicator and the Kolb Learning Style Inventory. *Attitude and motivation* – Similarly, is attitude/motivation critical? Studies have explored the Biggs revised two-factor Study Process Questionnaire (R-SPQ-2F), students' self-reports, measures of self-efficacy, and factors such as perfectionism, self-esteem, coping tactics, affective states, and optimism. General conclusions are discussed in the next section.

12.3.7 Predicting or Accounting for Novice Outcomes

Given the wealth of research on novice programmers, what factors are most significant in influencing success at learning to program and what, if anything, explains the pattern of polarized/bimodal outcomes that is often observed in CS1? As noted above, frustration at the failure of early attempts to predict aptitude led to the widely held and subsequently enduring belief that programmers are "born and not made" (Dauw, 1967; Webster, 1996) or (tongue in cheek) that there exists a "geek gene" for programming (Lister, 2010): either you have it or you don't. If so, we would expect to have found some evidence, component, or correlate of this innate ability by now. In this section, the strongest potential factors identified in the above review are briefly evaluated and a different kind of possible explanation is discussed.

The early attempts to develop cognitive aptitude tests met with limited success. Building on this experience, a number of subsequent tests and studies of demographic and cognitive factors likewise did not reach strong conclusions (Robins, 2010). The range of studies of multiple factors that find conflicting results or at best modest statistical correlation with programming success (typically as measured by final course grade) include Mayer and Stalnaker (1968), Bateman (1973), Newstead (1975), Wileman, Konvalina, and Stephens (1981), Wileman, Konvalina, and Stephens (1981), Pea and Kurland (1984), Curtis (1984), Werth (1986), Evans and Simkin (1989), Cronan, Embry, and White (1989), Subramanian and Joshi (1996), Wilson and Shrock (2001), Rountree et al. (2004), Woszczynski, Haddad, and Zgambo (2005), Bennedsen and Caspersen (2005), Ventura (2005), Bergin and Reilly (2006), Simon et al. (2006), and Lau and Yuen (2011). The use of behavioral measures to augment such "traditional" factors in predictive models is discussed by Carter, Hundhausen, and Adesope (2017). In short, no factor or combination of factors that clearly predict success in learning a first programming language has been found.

The most widely studied and intuitively appealing potential cognitive factor is mathematical ability. Most (though not all) studies that explore it find that it is one of the better predictors. However, as was noted more than 30 years ago:

> To our knowledge, there is no evidence that any relationship exists between general math ability and computer programming skill, once general ability has been factored out. For example, in some of our own work we found that better Logo programmers were also high math achievers. However, these children also had generally high scores in English, social studies, and their other academic subjects as well. Thus, attributing their high performance in computer programming to their math ability ignores the relationship between math ability and general intelligence.
>
> (Pea & Kurland, 1984)

As we might expect, a high IQ is also moderately associated with success in programming (Pea & Kurland, 1984), and most of the individual mathematical, verbal, spatial, and logical factors noted above are employed in a range of IQ tests. One of the most pervasive and general results about IQ, however, is that performance on various standard psychometric measures is highly correlated, a phenomenon known as "the positive manifold." This is sometimes used as an argument for the existence of a single general factor of intelligence called "g." Thus, the various cognitive factors that are weak to moderate predictors of success in programming may have no explanatory power that is independent of IQ. Furthermore, given that intelligence is at least roughly normally distributed in the population, it is not at all obvious how variations in IQ can simply account for any bimodal distribution of outcomes.

Affective factors such as motivation, constructive attitudes to learning, positive expectations, and high self-efficacy or effort are also usually found to be correlated with success in programming (see Chapter 28). However, the same proviso applies. These factors are moderate predictors of success in many domains, and thus are not likely to account for any particular properties or pattern of outcomes in programming.

Developmental factors appear to offer a strong potential explanation for programming outcomes. For example, White and Sivitanides (2002) suggest that in cases of bimodal grade distributions, "The low mode may indicate Piaget's concrete operation stage" and "The high mode may indicate Piaget's formal operation stage." While this is intuitively appealing for students in the critical age range (15–20 years), it does not work for younger or older learners, who are just as likely to exhibit polarized outcomes. This is an example of the kind of problem with Piaget's original theory that motivated the neo-Piagetian and SOLO frameworks, replacing the reliance on chronological age with levels that are complex, contextual, and dependent on individual factors such as prior knowledge. Note that these frameworks are therefore descriptive rather than predictive: the range of studies reviewed above describe observed behaviors (or artifacts) as exhibiting performance at different levels. But to then use that level as an "explanation" of success at programming seems rather circular – learners who are observed to perform well at programming are observed to perform well at programming.

The failure of more than 40 years of research to find a factor or factors that strongly and reliably (let alone uniquely) predict success or failure suggests that we may be looking in the wrong place for an explanation of programming outcomes. Maybe there is no "geek gene"/innate capacity or combination of cognitive or other factors that predicts or accounts for success or failure at programming, any more or less than for other domains of learning. Robins (2010) proposed a different kind of possible explanation for programming outcomes: the Learning Edge Momentum (LEM) hypothesis.

The theoretical foundation for LEM is the principle that we learn "at the edges" of what we already know by adding to existing knowledge. The more that new information is given a meaningful interpretation (i.e., the richer and more elaborate the links between new and old knowledge), the more effective learning appears to be (e.g., see the topics of educational scaffolding, transfer in learning, analogy, and the zone of proximal development as discussed in Chapter 9). The hypothesis is simply that, given some target domain of concepts to be learned, successful learning makes it somewhat easier to acquire related concepts, and unsuccessful learning makes it somewhat harder. In other words, the early acquisition (or otherwise) of concepts in a new domain becomes self-reinforcing, creating momentum toward successful or unsuccessful outcomes.

This LEM effect will vary in strength depending on the extent to which the concepts in the target domain are either independent or interdependent. When the domain consists of tightly integrated concepts (strong and well-defined edges), the momentum effect will be strong. Robins (2010) further proposed that a typical programming language is a domain of concepts that are unusually tightly integrated (at one end of the spectrum when compared to other domains). Factors that appear to support this proposal include the formal precision of programming languages (syntax and semantics), the ratings that programming content has received compared to other subjects on both the SOLO and Bloom taxonomies (Brabrand & Dahl, 2009; Oliver et al., 2004), preliminary evidence that programming is more complex than other learning domains (Margulieux, Catrambone, & Schaeffer, 2018), and the lack of agreement among computing educators on the correct order in which to teach programming language concepts.

If we accept these assumptions, then a plausible explanation for polarized/bimodal distributions of outcomes in CS1 emerges. They occur not because CS1 students are somehow different from others, but because the subject matter is different. The tightly integrated nature of language concepts results in a strong LEM effect. It is not the case that programming is simultaneously both hard and easy to learn for two different populations; rather, programming effectively becomes both harder and easier to learn for two different emerging groups. In short, an inherent systemic bias arising from the interaction between the learner and the learned acts to drive different subsets of the student population toward extreme outcomes. In this context, the historical failure to distinguish special kinds of programming students is entirely understandable. Programming students are much like any others, and they succeed or fail for reasons that are idiosyncratic and complex (although this review has stressed the importance of effective strategies, and as in other subjects, factors such as IQ, attitude, and prior experience are significant).

Studies that have supported predictions of the LEM account of programming outcomes, particularly with respect to the importance of the first one to three weeks of a CS1 course, include Porter and Zingaro (2014), Porter, Zingaro, and Lister (2014), Hola and Andreae (2014), and McCane et al. (2017).

12.3.8 Further Topics

The material in this section has explored the psychology of novice programmers, focusing on their knowledge, strategies, mental models, cognitive load, classification in various learning-related taxonomies, and the nature of possible explanations for the polarized outcomes often observed in typical CS1 courses. Many other topics relevant to novice programming have been explored, from narrow topics such as the relative ease of learning different kinds of programming language, to very broad ones such as issues of equity and diversity in the makeup of CS1 courses. Some of these are major topics explored elsewhere in this Handbook. A very brief pointer to some relevant topics and literature is noted below.

Researchers have debated the merits of teaching different kinds of programming language, such as procedural vs. object-oriented languages (Burkhardt, Détienne, & Wiedenbeck, 2002; Kunkle & Allen, 2016; Rist, 1995; Schulte & Bennedsen, 2006; Wiedenbeck & Ramalingam, 1999; Wiedenbeck et al., 1999) and the recent use of block-based languages as an alternative or supplement to textual languages (Bau et al., 2017; Maloney et al., 2010; Price & Barnes, 2015; Weintrop & Wilensky, 2015; Weintrop, Killen & Franke, 2018). The relationship between the separate but related skills of program code generation/writing and comprehension/reading (also called tracing or tracking) have been widely studied (Brooks 1977, 1983; Busjahn & Schulte, 2013; Corritore & Wiedenbeck, 1991; Davies, 1993; Lister et al., 2004; Lopez et al., 2008; Rist, 1995; Venables, Tan, & Lister, 2009; Whalley et al., 2006; Wiedenbeck et al., 1999). Various methods have been used to explore the specific difficulties experienced by novice programmers (Altadmri & Brown, 2015; Ebrahimi, 1994; Garner, Haden, & Robins, 2005; Jadud, 2006; Lahtinen, Ala-Mutka, & Järvinen, 2005; McCall & Kölling, 2014; Pea, 1986; Soloway & Spohrer, 1989; Spohrer & Soloway, 1989; Winslow, 1996). For further issues relating to programming paradigms, see Chapter 13 of this Handbook. Pedagogical tools such as programming environments are explored in Chapter 21, issues relating to prior knowledge and misconceptions in Chapter 27, attitude and motivation in Chapter 28, the teaching of programming in schools in Chapter 18, and pervasive issues of equity and diversity in Chapter 16.

12.4 Teaching and Learning in CS1

The basics of effective teaching and learning are the same in most subjects. For teachers, they include the provision of clear and relevant course materials, clear learning objectives, assessment that is well aligned with objectives, rich and timely feedback to students, fostering student engagement, pastoral care, competent classroom skills, and more. Educators are increasingly exploring the use of new tools and methods such as online resources and social media, feedback mechanisms, peer assessment, blended learning, flipped classrooms, and the use of naturally occurring performance measures ("learning analytics"). For students, learning outcomes will be influenced by motivation and attitude, forms of engagement, IQ, attitudes to learning, time management skills, personal circumstances, sociocultural factors, and more. The teaching and learning process as a whole is interpreted in the context of underlying educational philosophies or styles, such as cognitivism or constructivism. In general, the goal is to foster deep learning of principles and skills and to create independent, reflective, lifelong learners.

Most of these topics are explored elsewhere in the Handbook. In this section, we focus on issues that are specific to introductory programming as a subject or are particularly significant in this context. Pears et al. (2007) provide an excellent review of relevant literature; see also Robins, Rountree, and Rountree (2003).

12.4.1 A Lack of Agreement

There is unfortunately no agreement on the practical details of the best way to teach programming, or even on fundamentals such as which topics should be taught and what order they should be taught in. The most influential source of curriculum advice, the Joint Task Force on Computing Curricula, begins its chapter on introductory courses as follows:

> Computer science, unlike many technical disciplines, does not have a well-described list of topics that appear in virtually all introductory courses. In considering the changing landscape of introductory courses, we look at the evolution of such courses from CC2001 to CS2013 … we believe that advances in the field have led to an even more diverse set of approaches in introductory courses than the models set out in CC2001. Moreover, the approaches employed in introductory courses are in a greater state of flux.
>
> (ACM/IEEE-CS, 2013)

For perspectives from teachers, see the paper by Bruce (2004) titled "Controversy on how to teach CS 1: A discussion on the SIGCSE-members mailing list." It was suggested above that this lack of agreement arises in part because of the densely connected and interdependent nature of the concepts in the domain of a programming language. There is no one right path through the maze. Despite disagreements on practical specifics, however, there is consensus on many theoretical issues and guidelines arising from the experience of practitioners and from CEd research.

12.4.2 CS1 Design and Pedagogy

Various iterations of the ACM Computing Curricula have presented a range of course options and exemplars that serve as useful reference points for the field. Beyond such guidelines, however, there are many challenges involved in learning to program, as reviewed in Section 12.3. In this context, the design and delivery of a CS1 course should be realistic in its expectations and systematic in its development. In a significant survey of the literature on teaching programming, Pears et al. (2007) discuss three general approaches based on "the primary emphasis of the instructional setting," namely: "problem solving, learning a particular programming language, and code/system production."

Schneider (1978) presents ten principles that he suggests capture the "essential objectives of an initial programming course," all of which remain relevant for consideration today. The principles (abstracted from the explanatory text) are:

> 1) Students should immediately be taught that a clear, concise problem statement is always the first step in programming. 2) The single most important concept in a programming course is the concept of an algorithm. 3) It is important to introduce the duality of data structures and algorithms in the programming process. 4) Choose a programming language that enhances the learning process. 5) The presentation of a computer language should concentrate on semantics and program characteristics not syntax. 6) The presentation of a computer language must include concerns for programming style from the very beginning. 7) The subject of debugging should be formally presented. 8) The subject of program testing and verification should be formally presented. 9) The subject of documentation should be formally presented. 10) A student should be introduced to realistic programming applications and realistic programming environments.
>
> (Schneider, 1978)

Linn and Dalbey (1989) set out an ideal "chain of cognitive accomplishments" for teaching and learning programming. The links of the chain are: (1) features of the language being taught; (2) design skills, including knowledge templates/schemata/plans and the procedural skills of planning, testing, and reformulating code; and (3) problem-solving skills, including knowledge, strategies and procedural skills abstracted from the specific language that can be applied to new languages and situations.

Recognizing the importance of problem-solving, many of the ACM course exemplars since 2001 address it before or along with language features (ACM/IEEE-CS, 2013). Some (but not all) studies show improved outcomes for this approach (Davies, 2008; Hill, 2016; Koulouri, Lauria, & Macredie, 2014). Rist (1995) and Winslow (1996), however, suggest that problem-solving is necessary but not sufficient for programming. Winslow notes, for example, that most undergraduates can average a list of numbers, but fewer than half of them can write a loop to do the same operation. A discussion of the issues involved in problem-based learning, a description of various examples, and a three-year longitudinal follow-up of students is described in Kay et al. (2000). The authors observe "a substantial improvement in basic programming competence" (although possible confounding factors are acknowledged). The relationship between problem-solving and programming skills is extensively reviewed by Palumbo (1990).

Fincher (1999) asks, "What are we doing when we teach programming?" and compares the following four conceptual frameworks: "syntax-free," "problem-solving," "literacy," and "computation-as-interaction." Felleisen et al. (2001) argue that programming is for everyone, and that it is best learned by focusing on the design process. The authors provide "a set of explicit design guidelines" (such as data- and test-driven program design and writing examples before code) for developing computational solutions in a step-by-step manner. Other resources produced by the Program by Design group (http://programbydesign.org) include a specialized programming environment for beginners. Fisler (2014) provides evidence of the success of this approach in a study of five CS1 classes at four institutions (using program by design methods and a functional language). Students in this cohort significantly outperformed other reported study results on the classic Rainfall Problem. For a very different perspective that argues in favor of teaching programming based on formal methods such as predicate calculus and proofs of correctness, see Edsger Dijkstra's much-debated "On the cruelty of really teaching computer science" (Dijkstra, 1989).

Underlying much of the debate about the strengths and weaknesses of various approaches to teaching is the issue of transfer in learning. Facts and skills that are learned in one context (e.g., problem-solving) do not necessarily transfer to other contexts (e.g., writing code). Mayer (1992) notes that, in practice, CS1 courses have typically focused on language features, with varying opportunities to learn in ways that promote transfer. Transfer in learning programming is further discussed in Chapter 3, and transfer as a general phenomenon is learning is explored in Chapter 9.

The known difficulties of teaching programming have motivated various special languages and tools, an early example being the Logo language and "turtle graphics" introduced in 1967. du Boulay, O'Shea, and Monk (1989) made a case for the use of simple, specially designed teaching languages. Examples of such languages include Logo (released in 1967), Pascal (in 1970), Eiffel (in 1986), Python (in 1989), and Alice (in 1994), as well as the student languages of Racket (in 2001) and Scratch (in 2003). Pears et al. (2007) discuss factors that influence the choice of language in current courses and trade-offs such as richness vs. complexity. Commercially popular languages such as Java, C, and C++ are dominant for practical reasons (e.g., student demand), but their suitability for teaching has been much debated. The obvious and widespread intuition that syntactic complexity hampers learning has been supported by many studies (e.g., Koulouri, Lauria, & Macredie, 2014; Mannila, Peltomäki, & Salakoski, 2006; Yadin, 2011). One popular approach to exercises for teaching and assessing programming is Parsons' problems (Parsons & Haden, 2006), which present unordered statements that can be correctly ordered into working code. This approach is generally held to reduce cognitive load by providing syntactically correct building blocks.

Soloway and Spohrer (1989) summarize several suggestions relating to the design of programming environments/tools for novices, including the following: the use of "graphical languages" to make control flow explicit; a simple underlying machine model; short, simple, and consistent naming conventions; graphical animation of program states (with no "hidden" actions or states); design principles based on spatial metaphors; and the gradual withdrawal of initial supports and restrictions. Kelleher and Pausch (2005) present a taxonomy and review of languages and environments "designed to make programming more accessible to novice programmers of all ages." Some recent research on visualization tools, programming environments, and block-based programming languages is noted above; see Pears et al. (2007) and Chapter 21 for much broader reviews.

Given that much is now known about novice programmers, it is possible for teachers to anticipate and attempt to support different kinds of learners. Within any large class, there is likely to be a group who are making excellent progress (effective novices), a large group who are struggling (ineffective), and others in between. It is not possible for a typical course to perfectly suit both the effective and ineffective groups – CS1 will almost certainly move too quickly for many students and too slowly for some. This can be partially addressed by trying to set the course at the level of the "average" student and providing both extension work for the high-achieving group and targeted support to those who are struggling. If the strategies of effective novices can be identified, it may be possible to promote effective strategies to all groups. In other words, rather than focusing exclusively on the end product of programming knowledge, teachers could focus at least in part on the enabling step of functioning as an effective novice. Considerations of motivation and self-efficacy (Chapter 28) and gender and diversity (Chapter

16) are also important in this context. Ideally, course design and delivery would motivate all students, engage them in the process, and support them with the tools and strategies needed to become effective learners of programming.

12.4.3 Dimensions of Learning and Practical Examples

Considering CS1 in terms of the three dimensions of knowledge, strategies, and mental models provides a useful framework for course design and delivery. Making these explicit to students may also aid their understanding of the learning process in which they are engaged. Successful learning on all three dimensions is crucially dependent on broad experience of practical programming tasks.

The typical CS1 course focuses on knowledge of the elements of a programming language and practice in their application. As noted in Section 12.2.3.2, however, surveys and studies of novice programmers have shown that the main problems that they experience relate not to individual language constructs, but to overall program design and structure. This is consistent with frequent recommendations in the literature that instruction should emphasize the combination and use of language features and the underlying issue of program design. Spohrer and Soloway suggested "focusing explicitly on specific strategies for carrying out the coordination and integration of the goals and plans that underlie program code," that "students should be made aware of such concepts as goals and plans, and such composition statements as abutment and merging," and that "students be given a whole new vocabulary for learning how to construct programs" (Spohrer & Soloway, 1989). Mayer (1989) suggested that "explicit naming and teaching of basic schemata … may become part of computer programming curricula." Bearing out this prediction:

> A minor but remarkable collection of programming education research from the past ten to fifteen years concerns a pattern-based approach to instruction which utilize a shift from emphasis on learning the syntactic details of a specific programming language to the development of general problem-solving and program-design skills.
>
> (Caspersen & Bennedsen, 2007)

(Here, the authors use "pattern" as more or less synonymous with "schema".) The use of full-fledged design patterns (Freeman et al., 2004; Gamma et al., 1995), which is particularly common in an object-oriented context, is widely regarded as too complex for CS1 unless simplified and customized (Hundley, 2008; Lewis et al., 2004; Wick, 2005).

Several authors have noted that the teaching of knowledge structures must be anchored in, and learning may most effectively emerge from, practical experience and examples.

> The acquisition of schemata, such as a general design schema or programming plans, requires mindful abstraction, presupposes the confrontation with a well-chosen range of problems and their solutions (i.e., worked examples), and provides analogies that may guide subsequent behavior in solving unfamiliar aspects of new programming problems.
>
> (van Merriënboer & Paas, 1990)

Caspersen and Bennedsen (2007) also stress the importance of worked examples, along with "scaffolding, faded guidance, cognitive apprenticeship, and emphasis of patterns to aid schema creation and improve learning."

As noted in Section 12.3.3, many authors have suggested that strategies are the most important factor in determining the success or failure of learning to program. We have similarly proposed (Robins, Rountree, & Rountree, 2003) that it is differences in strategy that most significantly separate ineffective and effective novices. Despite their importance, strategies typically receive less explicit attention in CS1 than knowledge. Furthermore, as Brooks (1990) points out, strategies themselves cannot (in most cases) be deduced from the final "static" form of a program, even though they may have had a strong impact in the design and coding process and thus on the final form. As pedagogical examples, finished programs are rich and accessible sources of information about the language, but the strategies that created those programs are much harder to make explicit.

These factors highlight the importance of actively developing programs and explicitly addressing the strategies involved as part of CS1 design and delivery. While this can be demonstrated in lectures and teaching resources, hands-on experience is obviously the most effective form of learning for students. This highlights again the need for well-designed example tasks and the practical opportunities for students to engage with them. The design, delivery, and assessment (see below) of laboratory/practical sessions may be the most important element of CS1.

The mental models that novices must develop, both with respect to an underlying notional machine and as important aspects of planning, understanding, and debugging programs, are also very important to successful learning outcomes. As noted above, a broad survey (Schulte & Bennedsen, 2006) suggests that teachers regard an explicit notional machine as relevant, but only 29 percent of respondents explicitly addressed it in their teaching. Mental models of specific programs are of course internal, personal, and relative to particular examples, and therefore impossible to "teach," but many aspects of typical CS1 design and pedagogy can implicitly support learners in the process of their acquisition. Example methods include the use of modeling languages such as UML, demonstrating and encouraging the practice of program tracing/tracking, graphical representations of program states and animations of their operations, and the use of tools such as debuggers or programming environments that facilitate the observation of program states.

In summary, programming knowledge, skills, and mental models cannot be effectively acquired in the abstract – they must be anchored in rich practical experience. This highlights the importance of laboratory/practical sessions in the design and delivery of CS1. Programming tasks also have many pedagogically useful features. Each one can form a "case-based" problem-solving session where students can work and learn at their own pace. The feedback supplied by compilers and other tools is immediate, consistent, and (ideally) informative. For teachers, change episodes (where an alteration is made to code) may be rich in information about the students' models, plans, and goals (Gray & Anderson, 1987). For students, the reinforcement and motivation derived from creating a working program can be very powerful (programs with graphical and animated output can be particularly popular). One of the strongest results to emerge from the study of practical programming tasks is the success of collaborative work and "peer learning" (see Chapters 29 and 30).

12.4.4 Effective and Ideal Interventions

One of the most important ways of supporting learning in educational contexts in general is the effective use of formative assessment: "Assessment that is explicitly designed to promote learning is the single most powerful tool we have for raising standards and empowering life-long learning" (Assessment Reform Group, 1999). While this finding is not specific to CEd, there are a number of discipline-specific factors to consider in the assessment of programming, as discussed in Chapter 14. It may well be that understanding and improving our assessment practices is the single most effective intervention we can make. Hattie (2009, 2012) presents a large-scale meta-analysis of a range of factors influencing educational outcomes in general.

The literature reviewed above includes many research-based examples of course design decisions or interventions that are specific to teaching programming and have been shown to have a beneficial effect on learning outcomes. These include: the use of syntactically simple languages and rich (but not overly complex) programming environments and tools; providing and encouraging the use of a notional machine that underlies the language; attention to both reading (tracing/tracking) and writing code; a focus on problem-solving, combining language elements, and program design; explicit attention to programming strategies; facilitating the acquisition of appropriate mental models; the extensive use of well-designed example programs and practical tasks; and (Chapters 29 and 30) the use in this context of pair programming and peer learning methods.

In a very useful review, Vihavainen, Airaksinen, and Watson (2014) attempt to quantify the improvements in learning outcomes that can be attributed to various kinds of intervention. From a broad initial sample, the authors identified 32 articles describing interventions in CS1 courses that included pre- and post-intervention pass rates, sometimes over multiple instances (e.g., semesters), for a total of 60 interventions. Within this sample, the authors collaboratively coded the intervention types. The ten most commonly observed interventions were *collaboration* (encouraging student collaboration), *content change* (updates to teaching material), *contextualization* (alignment toward a specific context; e.g., games or media), *CS0* (the creation of a preliminary course), *game theme* (introduction of a game-themed component), *grading schema* (e.g., increasing the weighting for practical tasks), *group work* (e.g., team-based learning or cooperative learning), *media computation* (programming in the context of digital media), *peer support* (pairs, groups, or peer tutors), and *support* (an umbrella term for increased teacher hours or additional support channels, for example). The most frequent intervention was content change (followed by peer support and collaboration); the least was CS0.

As an overall summary, the average course pass rate prior to an intervention was a mean of 61.4 percent (standard deviation [SD] 1.15 percent) and after an intervention was a mean of 74.4 percent (SD 11.7 percent). The average "realized improvement" (the extent to which the post-intervention pass rate closed the gap between the pre-intervention pass rate and 100 percent) for each of the interventions ranged from media computation (mean 48 percent, SD 16 percent) to game theme (mean 18 percent, SD 23 percent). The authors note that there was considerable variation over individual interventions and combinations of interventions, including 5 (of the 60) cases of pass rates decreasing. They conclude that "the interventions reported in the literature increase introductory programming course pass rates by one third on average," and that "no statistically significant differences between the effectiveness of the teaching interventions were found." However, given that "the results suggest that almost any planned intervention improves the existing state," the source articles need to be evaluated for the extent to which they did (or did not) account for common threats to validity such as experimenter bias or the Hawthorne effect (Mitchell & Jolley, 2012).

Returning to the LEM hypothesis (Section 12.3.7), recall that it suggests that either early success or failure in learning programming become, over time, compounding effects. Once negative momentum is established, it is very hard to overcome; ideally, positive momentum should be established right from the start. This implies that the very early stages of CS1 are critical to outcomes. Everything possible should be done to ensure that the initial student experiences are successful and to facilitate learning during this critical time. Particular attention should be paid to the careful introduction of concepts and the systematic development of the connections between them. Any extra resources or support (e.g., increased access to tutors) should be focused on the early stages. Students showing signs of disengagement (missing labs or tutorials, failing to submit work) should be followed up immediately and actively, as early as in the first week. Students could also be told why the early period of learning is critical, as this "meta-knowledge" may increase engagement and motivation. They should absolutely be encouraged to seek immediate help at the first sign of difficulty – keeping up is vital. Studies that have supported the predictions of the LEM hypothesis, particularly with respect to the importance of the early weeks of a CS1 course, were noted in Section 12.3.7.

Most CS1 courses are constrained by practical considerations of resources, time, and large student populations. In an ideal world, we would like to provide individual and personally designed tuition and support to every student. It may be that breakthroughs in intelligent tutoring systems will one day achieve this ideal. In the meantime, however, the closer we can get to this goal within the practical constraints that we have, the more successful learner outcomes will be.

One suggestion in this context is to address the usual constraint of a single fixed flow (rate and path of progression) through the curriculum. The clear message from the literature is that there is no point in expecting a student to acquire a new layer of complex concepts if the foundation of prerequisite concepts does not exist. This could potentially be addressed by introducing some flexibility into the delivery of the course, so that students are more able to work and learn at their own pace and in ways that allow them to make sustainable progress. For example, CS1 could be offered in multiple streams that progress at different rates or vary in the amount of material covered. Students could select streams, or move between them, possibly as guided by an early diagnostic test. “Recovery streams” could be offered so that students have the option to backtrack and revise. For maximum flexibility, streams (through which students progress at the same rate) could be replaced by self-paced learning, where courses consist of just resource materials and a sequence of exercises to be completed at any time. Mastery models of learning and apprenticeships are common in some teaching contexts and may be interesting to explore as alternative models for the teaching of programming.

12.5 Discussion

There are many reasons why so much of CEdR is directed at the topics of novice programmers and CS1 (this **motivational context** was described in the introduction). From this research, many **implications for practice** have emerged. We know that both the teaching and the learning of programming can be improved, resulting in better learning outcomes. We know a fair bit about specific methods that improve teaching and/or learning, although perhaps we don’t know much of this for sure (see below). It is therefore incumbent on us as teachers to familiarize ourselves with this literature, to adopt the methods that will work in our context, and to be the best teachers that we can be. Specific methods that appear to be effective in practice for programming are summarized in Section 12.4.4. These should be seen in the context of the broader issues of pedagogic methods, motivation and affect, tools, assessment, issues of equity and diversity, and other matters that are addressed elsewhere in this Handbook. As always, the most general and important lesson is that by understanding the learner and designing our courses to support the learning process, we can achieve better outcomes as teachers. Much is known about novice programmers as learners, giving teachers much to work with.

There remain many **open questions**. Further investigation and replication within most CEdR topics would be welcome (in a sense, all questions are open). In particular, we can’t be sure that we know all of the reasons why programming is so difficult to learn for so many, and we still don’t know much about why it appears to be quite easy for some. Most CEdR has been inward looking. Although we think programming has some unique challenges, we still don’t know much about how it is similar to or different from other topics for teachers and for learners. We absolutely don’t know the best language(s) for learners or the best way to teach any given language, and we have yet to find the most effective ways of supporting learners.

Of the dimensions of knowledge, strategies, and mental models, we know least about the mental models constructed by learners. How are they acquired? How do they vary? How can we facilitate useful ones? How do we intervene and correct false ones? How do we best reduce the cognitive load of programming or learning to program? While we know a fair bit about the differences between experts and novices, we know less about the differences between effective and ineffective novices. What are effective novices doing that works so well? Can it be identified, and can it be used to support currently ineffective novices? Have we found out as much as we can about why some novice programmers succeed and not others? Are the crucial differences cognitive, attitudinal, or behavioral, or impossible to separate? What is best teaching practice as it relates to each of these dimensions? We haven’t found one yet, but is there a configuration of circumstances or a diagnostic test that can accurately predict who will or will not be successful at programming? Can we yet rule out the meme of the programmer gene and do away with the idea that programmers are born and not made? Many countries are currently embarking on large-scale educational experiments, but will the move toward teaching computational thinking or elements of programming in schools be successful? How best can it be supported? What are the implications for teaching and learning programming at later levels?

As argued in the introduction to this Handbook, this is an important and exciting time to be engaged in CEdR. A focus on novice programmers and the topics of teaching and learning programming is likely to remain of central importance to the field for the foreseeable future.

References

ACM/IEEE–CS Joint Task Force on Computing Curricula (2013). *Computer Science Curricula 2013*. New York: ACM Press and IEEE Computer Society Press.

Alaoutinen, S., & Smolander, K. (2010). Student self-assessment in a programming course using Bloom's Revised Taxonomy. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10)* (pp. 155–159). New York: ACM.

Alexandron, G., Armoni, M., Gordon, M., & Harel, D. (2014). Scenario-based programming: Reducing the cognitive load, fostering abstract thinking. In *Companion Proceedings of the 36th International Conference on Software Engineering* (pp. 311–320). New York: ACM.

Allan, V. H., & Kolesar, M. V. (1997). Teaching computer science: A problem solving approach that works. *ACM SIGCUE Outlook*, **25**(1–2), 2–10.

Altadmri, A., & Brown, N. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 522–527). New York: ACM.

Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., Raths, J., & Wittrock, M. C. (Eds.) (2001). *A Taxonomy for Learning and Teaching and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. New York: Addison Wesley Longman.

Assessment Reform Group (1999). *Assessment for Learning: Beyond the Black Box*. Cambridge, UK: Cambridge University Press.

Barker, R. J., & Unger, E. A. (1983). A predictor for success in an introductory programming class based upon abstract reasoning development. *ACM SIGCSE Bulletin*, **15**(1), 154–158.

Bateman, C. R. (1973). Predicting performance in a basic computer course. In *Proceedings of the Fifth Annual Meeting of the American Institute for Decision Sciences* (pp. 130–133). Atlanta, GO: AIDS Press.

Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable programming: Blocks and beyond. *Communications of the ACM*, **60**(6), 72–80.

Bauer, R., Mehrens, W. A., & Vinsonhaler, J. F. (1968). Predicting performance in a computer programming course. *Educational and Psychological Measurement*, **28**, 1159–1164.

Beaubouef, T. B., & Mason, J. (2005). Why the high attrition rate for computer science students: Some thoughts and observations. *Inroads – The SIGCSE Bulletin*, **37**(2), 103–106.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, **39**(2), 32–36.

Bennedsen, J., & Caspersen, M. E. (2005). An investigation of potential success factors for an introductory model-driven programming course. In *Proceedings of the First International Workshop on Computing Education Research (ICER '05)* (pp. 155–163). New York: ACM.

Bennedsen, J., & Caspersen, M. E. (2006). Abstraction ability as an indicator of success for learning object-oriented programming? *SIGCSE Bulletin*, **38**(2), 39–43.

Bergin, S., & Reilly, R. (2006). Predicting introductory programming performance: A multi-institutional multivariate study. *Computer Science Education*, **16**(4), 303–323.

Berland, M., Martin, T., Benton, T., Petrick Smith, C., & Davis, D. (2013). Using learning analytics to understand the learning pathways of novice programmers.

Journal of the Learning Sciences, **22**(4), 564–599.

Berry, M., & Kölling, M. (2014). The state of play: A notional machine for learning programming. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education* (pp. 21–26). New York: ACM.

Bhuiyan, S., Greer, J. E., & McCalla, G. I. (1992). Learning recursion through the use of a mental model-based programming environment. In *International Conference on Intelligent Tutoring Systems* (pp. 50–57). Berlin, Germany: Springer.

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. New York: Academic Press.

Bloom, B., Englehart, M. D., Furst, E. J., Hill, W. H., & Krathwohl, D. (1956). *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*. New York: Longmans.

Bornat, R. (2014). Camels and humps: A retraction. Retrieved from: http://eis.sla.mdx.ac.uk/staffpages/r_bornat/papers/camel_hump_retraction.pdf

Bornat, R., Dehnadi, S., & Simon (2008). Mental models, consistency and programming aptitude. In *Proceedings of the Tenth Australasian Computing Education Conference (ACE 2008)* (pp. 53–62). Darlinghurst, Australia: Australian Computer Society.

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., & Zander, C. (2007). Threshold concepts in computer science: Do they exist and are they useful? *ACM SIGCSE Bulletin*, **39**(1), 504–508.

Bower, M. (2008). A taxonomy of task types in computing. *ACM SIGCSE Bulletin*, **40**(3), 281–285.

Brabrand, C., & Dahl, B. (2009). Using the SOLO taxonomy to analyze competence progression of university science curricula. *Higher Education*, **58**(4), 531–549.

Brooks, F. P. (1975). *The Mythical Man-Month: Essays on Software Engineering*. New York: Addison-Wesley.

Brooks, R. E. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man–Machine Studies*, **9**, 737–751.

Brooks, R. E. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man–Machine Studies*, **18**, 543–554.

Brooks, R. E. (1990). Categories of programming knowledge and their application. *International Journal of Man–Machine Studies*, **33**(3), 241–246.

Bruce, K. B. (2004). Controversy on how to teach CS 1: A discussion on the SIGCSE-members mailing list. *ACM SIGCSE Bulletin*, **36**(4), 29–34.

Bruner, J. S. (1960). *The Process of Education*. Cambridge, MA: Harvard University Press.

Buck, D., & Stucki, D. J. (2000). Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development. *ACM SIGCSE Bulletin*, **32**(1), 75–79.

Burkhardt, J. M., Détienne, F., & Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, **7**(2), 115–156.

Burkhardt, J. M., Détienne, F., & Wiedenbeck, S. (1997). Mental representations constructed by experts and novices in object-oriented program comprehension. In *Proceedings of the IFIP TC13 International Conference on Human–Computer Interaction (INTERACT '97)* (pp. 339–346). London, UK: Chapman & Hall.

Busjahn, T., & Schulte, C. (2013). The use of code reading in teaching programming. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research* (pp. 3–11). New York: ACM.

Cañas, J. J., Bajo, M. T., & Gonzalvo, P. (1994). Mental models and computer programming. *International Journal of Human–Computer Studies*, **40**(5), 795–811.

Carter, A. S., Hundhausen, C. D., & Adesope, O. (2017). Blending measures of programming and social behavior into predictive models of student achievement in early computing courses. *ACM Transactions on Computing Education (TOCE)*, **17**(3), 12.

Caspersen, M. E., & Bennedsen, J. (2007). Instructional design of a programming course: a learning theoretic approach. In *Proceedings of the Third International Workshop on Computing Education Research* (pp. 111–122). New York: ACM.

Caspersen, M. E., Larsen, K. D., & Bennedsen, J. (2007). Mental models and programming aptitude. *ACM SIGCSE Bulletin*, **39**(3), 206–210.

Castro, F. E. V., & Fisler, K. (2017). Designing a multi-faceted SOLO taxonomy to track program design skills through an entire course. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17)* (pp. 10–19). New York: ACM.

Clancy, M. J., & Linn, M. C. (1999). Patterns and pedagogy. *ACM SIGCSE Bulletin*, **31**(1), 37–42.

Clear, T., Whalley, J., Lister, R. F., Carbone, A., Hu, M., Sheard, J., Simon, B., & Thompson, E. (2008). Reliably classifying novice programmer exam responses using the SOLO taxonomy. In *21st Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2008)* (pp. 23–30). Auckland, New Zealand: National Advisory Committee on Computing Qualifications.

Corney, M., Teague, D., & Thomas, R. N. (2010). Engaging students in programming. In *Proceedings of the Twelfth Australasian Conference on Computing Education Volume 103* (pp. 63–72). Darlinghurst, Australia: Australian Computer Society,

Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. In *Proceedings of the Fourteenth Australasian Computing Education Conference Volume 123* (pp. 77–86). Darlinghurst, Australia: Australian Computer Society.

Corritore, C. L., & Wiedenbeck, S. (1991). What do novices learn during program comprehension? *International Journal of Human–Computer Interaction*, **3**, 199–222.

Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, **24**, 87–185.

Cronan, T. P., Embry, P. R., & White, S. D. (1989). Identifying factors that influence performance of non-computing majors in the business computer information systems course. *Journal of Research on Computing in Education*, **21** (4), 431–441.

CSTA (2017). About the CSTA K–12 Computer Science Standards. Retrieved from www.csteachers.org/page/standards

Curtis, B. (1984). Fifteen years of psychology in software engineering: Individual differences and cognitive science. In *Proceedings of the 7th International Conference on Software Engineering* (pp. 97–106). New York: IEEE.

Dauw, D. (1967). Vocational interests of highly creative computer personnel. *Personnel Journal*, **46**(10), 653–659.

Davies S. P. (1993). Models and theories of programming strategy. *International Journal of Man–Machine Studies*, **39**, 237–267.

Davies, S. P. (2008). The effects of emphasizing computational thinking in an introductory programming course. In *Frontiers in Education Conference (FIE 2008)* (p. T2C-3). New York: IEEE.

Dehnadi, S. (2006). Abstract for Dehnadi & Bornat (2006). Retrieved from www.eis.mdx.ac.uk/research/PhDArea/saeed

Dehnadi, S., & Bornat, R. (2006). The camel has two humps (working title). Retrieved from www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf

Dijkstra, E. W. (1989). On the cruelty of really teaching computer science. *Communications of the ACM*, **32**(12), 1398–1404.

du Boulay, B., O'Shea, T., & Monk, J. (1989). The black box inside the glass box: Presenting computing concepts to novices. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 431–446). Hillsdale, NJ: Lawrence Erlbaum.

du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, **2**(1), 57–73.

du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway & J.

C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 283–299). Hillsdale, NJ: Lawrence Erlbaum.

Ebrahimi, A. (1994). Novice programmer errors: Language constructs and plan composition. *International Journal of Human–Computer Studies*, **41**(4), 457–480.

Eckerdal, A. (2009). *Novice Programming Students' Learning of Concepts and Practice* (doctoral dissertation). Acta Universitatis Upsaliensis.

Eckerdal, A., Thuné, M., & Berglund, A. (2005). What does it take to learn "programming thinking"? In *Proceedings of the First International Workshop on Computing Education Research* (pp. 135–142). New York: ACM.

Elarde, J. (2016). Toward improving introductory programming student course success rates: experiences with a modified cohort model to student success sessions. *Journal of Computing Sciences in Colleges*, **32**(2), 113–119.

Ensmenger, N. L. (2010). *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. Cambridge, MA: MIT Press.

Evans, G. E., & Simkin, M. G. (1989). What best predicts computer proficiency? *Communications of the ACM*, **32**(11), 1322–1327.

Falkner, K., Vivian, R., & Falkner, N. J. (2013). Neo-Piagetian forms of reasoning in software development process construction. In *Learning and Teaching in Computing and Engineering (LaTiCE)* (pp. 31–38). New York: IEEE.

Feldman, D. H. (2004). Piaget's stages: The unfinished symphony of cognitive development. *New Ideas in Psychology*, **22**, 175–231.

Felleisen, M., Findler, R. B., Flatt, M., & Krishnamurthi, S. (2001). *How to Design Programs: An Introduction to Programming and Computing*. Cambridge, MA: MIT Press.

Fincher, S. (1999). What are we doing when we teach programming? In *Frontiers in Education Conference (FIE'99) Volume 1* (pp. 12A4-1–12A4-5). New York: IEEE.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., & Felleisen, M. (2002). DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, **12**(2), 159–182.

Fisler, K. (2014). The recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (pp. 35–42). New York: ACM.

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns: A Brain-Friendly Guide*. Sebastopol, CA: O'Reilly Media.

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T. L., Thompson, D. M., Riedesel, C., & Thompson, E. (2007). Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bulletin*, **39**(4), 152–170.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. New York: Addison-Wesley.

Garcia, R. A. (1987). *Identifying the Academic Factors that Predict the Success of Entering Freshmen in a Beginning Computer Science Course* (doctoral dissertation). Texas Tech University.

Garner, S. (2002). Reducing the cognitive load on novice programmers. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications* (pp. 578–583). Chesapeake, VA: AACE.

Garner, S., Haden, P., & Robins, A. (2005). My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. In *Proceedings of the Seventh Australasian Computing Education Conference (ACE2005) CRPIT 42* (pp. 173–180). Darlinghurst, Australia: Australian Computer Society.

Gentner, D. (2002). Mental models, psychology of. In N. Smelser & P. B. Bates (Eds.), *International Encyclopedia of the Social and Behavioral Sciences* (pp. 9683–9687). Amsterdam, The Netherlands: Elsevier Science.

Gentner, D., & Stevens, A. L. (Eds.) (1983). *Mental Models*. Hillsdale, NJ: Erlbaum.

Gibbs, W. W. (1994). Software's chronic crisis. *Scientific American*, **271**(3), 86–95.

Ginat, D., & Menashe, E. (2015). SOLO taxonomy for assessing novices' algorithmic design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 452–457). New York: ACM.

Gluga, R., Kay, J., Lister, R., Kleitman, S., & Lever, T. (2012). Coming to terms with Bloom: an online tutorial for teachers of programming fundamentals. In *Proceedings of the Fourteenth Australasian Computing Education Conference Volume 123* (pp. 147–156). Darlinghurst, Australia: Australian Computer Society.

Gray W. D., & Anderson J. R. (1987). Change-episodes in coding: When and how do programmers change their code? In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical Studies of Programmers: Second Workshop* (pp. 185–197). Norwood, NJ: Ablex.

Gray, S., St Clair, C., James, R., & Mead, J. (2007). Suggestions for graduated exposure to programming concepts using fading worked examples. In *Proceedings of the Third International Workshop on Computing Education Research* (pp. 99–110). New York: ACM.

Green T. R. G. (1990). Programming languages as information structures. In J. M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gillmore (Eds.), *Psychology of Programming* (pp. 117–137). London, UK: Academic Press.

Guzdial, M. (2007) What makes programming so hard? Retrieved from http://home.cc.gatech.edu/csl/uploads/6/Guzdial-blog-pieces-on-what-is-CSEd.pdf

Guzdial, M. (2010). Why is it so hard to learn to program? In A. Oram & G. Wilson (Eds.), *Making Software: What Really Works, and Why We Believe It* (pp. 111–124). Sebastopol, CA: O'Reilly Media.

Guzdial, M., & Soloway, E. (2002). Teaching the Nintendo generation to program. *Communications of the ACM*, **45**(4), 17–21.

Hattie, J. A. (2009). *Visible Learning: A Synthesis of 800+ Meta-Analyses on Achievement*. Abingdon, UK: Routledge.

Hattie, J. (2012). *Visible Learning for Teachers: Maximizing Impact on Learning*. Abingdon, UK: Routledge.

Hiebert, J., & Lefevre, P. (1986). Conceptual and procedural knowledge in mathematics: An introductory analysis. In J. Hiebert (Ed.), *Conceptual and Procedural Knowledge: The Case of Mathematics, 2* (pp. 1–27). Hillsdale, NJ: Erlbaum.

Hill, G. J. (2016). Review of a problems-first approach to first year undergraduate programming. In S. Kassel & B. Wu (Eds.), *Software Engineering Education Going Agile* (pp. 73–80). Basel, Switzerland: Springer International Publishing.

Hoc J. M., & Nguyen-Xuan, A. (1990). Language semantics, mental models and analogy. In J. M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gillmore (Eds.), *Psychology of Programming* (pp. 139–156). London, UK: Academic Press.

Hoda, R., & Andreae, P. (2014). It's not them, it's us! Why computer science fails to impress many first years. In *Proceedings of the Sixteenth Australasian Computing Education Conference Volume 148* (pp. 159–162). Darlinghurst, Australia: Australian Computer Society.

Höök, L. J., & Eckerdal, A. (2015). On the bimodality in an introductory programming course: An analysis of student performance factors. In *Learning and Teaching in Computing and Engineering (LaTiCE 2015)* (pp. 79–86). New York: IEEE.

Howles, T. (2009). A study of attrition and the use of student learning communities in the computer science introductory programming sequence. *Computer Science Education*, **19**(1), 1–13.

Hudak, M. A., & Anderson D. E. (1990). Formal operations and learning style predict success in statistics and computer science courses. *Teaching of Psychology*, **17**(4), 231–234.

Hundley, J. (2008). A review of using design patterns in CS1. In *Proceedings of the 46th Annual Southeast Regional Conference (ACM SE'08)* (pp. 30–33). New York: ACM.

Izu, C., Weerasinghe, A., & Pope, C. (2016). A study of code design skills in novice programmers using the SOLO taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 251–259). New York: ACM.

Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour. In *Proceedings of the Second International Workshop on Computing Education Research* (pp. 73–84). New York: ACM.

Jimoyiannis, A. (2013). Using SOLO taxonomy to explore students' mental models of the programming variable and the assignment statement. *Themes in Science and Technology Education*, **4**(2), 53–74.

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

Johnson, C. G., & Fuller, U. (2006). Is Bloom's taxonomy appropriate for computer science? In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling* (pp. 120–123). New York: ACM.

Kay, J., Barg, M., Fekete, A., Greening. T., Hollands, O., Kingston, J., & Crawford, K. (2000). Problem-based learning for foundation computer science courses. *Computer Science Education*, **10**, 109–128.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, **37**(2), 83–137.

Kim, J., & Lerch, F. J. (1997). Why is programming (sometimes) so difficult? Programming as scientific discovery in multiple problem spaces. *Information Systems Research*, **8**(1), 25–50.

Kinnunen, P., & Malmi, L. (2006). Why students drop out CS1 course? In *Proceedings of the Second International Workshop on Computing Education Research* (pp. 97–108). New York: ACM.

Kölling, M. (2009). Quality-oriented teaching of programming. Retrieved from: https://blogs.kcl.ac.uk/proged/2009/09/04/quality-oriented-teaching-of-programming

Kölling, M. (2010). The Greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*, **10**(4), 14.

Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Computer Science Education*, **13**(4), 249–268.

Koulouri, T., Lauria, S., & Macredie, R. D. (2014). Teaching introductory programming: A quantitative evaluation of different approaches. *ACM Transactions on Computing Education (TOCE)*, **14**(4), 26.

Krathwohl, D. R. (2002). A revision of Bloom's taxonomy: An overview. *Theory Into Practice*, **41**(4), 212–218.

Kunkle, W. M., & Allen, R. B. (2016). The impact of different teaching approaches and languages on student learning of introductory programming concepts. *ACM Transactions on Computing Education (TOCE)*, **16**(1), 3.

Kurland D. M., Pea, R. D., Clement, C., & Mawby, R. (1989). A study of the development of programming ability and thinking skills in high school students. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 83–112). Hillsdale, NJ: Lawrence Erlbaum.

Kurland, D. M., & Pea, R. D. (1985). Children's mental models of recursive LOGO programs. *Journal of Educational Computing Research*, **1**(2), 235–243.

Lahtinen, E., Ala-Mutka, K. & Järvinen, H. M. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin*, **37**(3), 14–18.

Lau, W. W., & Yuen, A. H. (2011). Modelling programming performance: Beyond the influence of learner characteristics. *Computers & Education*, **57**(1), 1202–1213.

Lawson, C. (1962). A survey of computer facility management. *Datamation*, **8**(7), 29–32.

Lewis, T. L., Rosson, M. B., & Pérez-Quiñones, M. A. (2004). What do the experts say?: Teaching introductory design from an expert's perspective. *ACM SIGCSE Bulletin*, **36**(1), 296–300.

Linn, M. C., & Dalbey, J. (1989). Cognitive consequences of programming instruction. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 57–81). Hillsdale, NJ: Lawrence Erlbaum.

Linn, M. C., & Clancy, M. J. (1992). The case for case studies of programming problems. *Communications of the ACM*, **35**(3), 121–132.

Lister, R. (2000). On blooming first year programming, and its blooming assessment. In *Proceedings of the Australasian Conference on Computing Education (ACSE '00)* (pp. 158–162). Darlinghurst, Australia: Australian Computer Society.

Lister, R. (2010). Geek genes and bimodal grades. *ACM Inroads*, **1**(3), 16–17.

Lister, R. (2011). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference Volume 114* (pp. 9–18). Darlinghurst, Australia: Australian Computer Society.

Lister, R., & Leaney, J. (2003a). Introductory programming, criterion-referencing, and Bloom. *ACM SIGCSE Bulletin*, **35**(1), 143–147.

Lister, R., & Leaney, J. (2003b). First year programming: Let all the flowers bloom. In *Proceedings of the Fifth Australasian Computing Education Conference (ACE 2003) Volume 20* (pp. 221–230). Darlinghurst, Australia: Australian Computer Society.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., & Simon, B. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, **36**(4), 119–150.

Lister, R., Clear, T., Bouvier, D. J., Carter, P., Eckerdal, A., Jacková, J., Lopez, M., McCartney, R., Robbins, P., Seppälä, O., & Thompson, E. (2010). Naturally occurring data as research instrument: Analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin*, **41**(4), 156–173.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin*, **38**(3), 118–122.

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research* (pp. 101–112). New York: ACM.

Luxton-Reilly, A. (2016). Learning to program is easy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 284–289). New York: ACM.

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2007). Investigating the viability of mental models held by novice programmers. *ACM SIGCSE Bulletin*, **39**(1), 499–503.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, **10**(4), 16.

Mannila, L., Peltomäki, M., & Salakoski, T. (2006). What about a simple language? Analyzing the difficulties in learning to program. *Computer Science Education*, **16**(3), 211–227.

Margulieux, L. E., Catrambone, R., & Schaeffer, L. M. (2018). Varying effects of subgoal labeled expository text in programming, chemistry, and statistics. *Instructional Science*, **46**(5), 707–722.

Mason, R., & Cooper, G. (2013). Distractions in programming environments. In *Proceedings of the Fifteenth Australasian Computing Education Conference Volume 136* (pp. 23–30). Darlinghurst, Australia: Australian Computer Society.

Mayer R. E. (1989). The psychology of how novices learn computer programming. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 129–159). Hillsdale, NJ: Lawrence Erlbaum.

Mayer, D. B., & Stalnaker, A. W. (1968). Selection and evaluation of computer personnel – The research history of SIG/CPR. In *The Proceedings of the 1968 ACM National Conference (23rd ACM National Conference)* (pp. 657–670). New York: ACM.

Mayer, R. E. (1985). Learning in complex domains: A cognitive analysis of computer programming. *Psychology of Learning and Motivation*, **19**, 89–130.

Mayer, R. E. (1992). Teaching for transfer of problem-solving skills to computer programming. In E. De Corte, M. C. Linn, H. Mandl, & L. Verschaffel (Eds.), *Computer-Based Learning Environments and Problem Solving. NATO ASI Series (Series F: Computer and Systems Sciences), Vol. 84* (pp. 193–206). Berlin, Germany: Springer.

McCall, D., & Kölling, M. (2014). Meaningful categorisation of novice programmer errors. In *Frontiers in Education Conference (FIE)* (pp. 1–8). New York: IEEE.

McCane, B., Ott, C., Meek, N., & Robins, A. (2017). Mastery learning in introductory programming. In *Proceedings of the Nineteenth Australasian Computing Education Conference* (pp. 1–10). New York: ACM.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, **33**, 125–180.

McNamara, W. J. (1967). The selection of computer personnel: Past, present, future. In *Proceedings of the Fifth SIGCPR Conference on Computer Personnel Research (SIGCPR '67)* (pp. 52–56). New York: ACM Press.

Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. S., & Thomas, L. (2006). A cognitive approach to identifying measurable milestones for programming skill acquisition. *ACM SIGCSE Bulletin*, **38**(4), 182–194.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2010). Learning computer science concepts with Scratch. In *Proceedings of the Sixth International Workshop on Computing Education Research (ICER '10)* (pp. 69–76). New York: ACM.

Mendes, A. J., Paquete, L., Cardoso, A., & Gomes, A. (2012). Increasing student commitment in introductory programming learning. In *Frontiers in Education Conference (FIE)* (pp. 1–6). New York: IEEE.

Meyer, J. H., & Land, R. (Eds.) (2006). *Overcoming Barriers to Student Understanding: Threshold Concepts and Troublesome Knowledge*. London, UK: Routledge.

Meyer, J. H. F., & Land, R. (2003). *Threshold Concepts and Troublesome Knowledge: Linkages to Ways of Thinking and Practising within the Disciplines (ETL Project: Occasional Report No. 4)*. Edinburgh, UK: University of Edinburgh Press.

Mitchell, M. L., & Jolley, J. M. (2012). *Research Design Explained*, 8th edn. Wadsworth, CA: Cengage Learning.

Morra, S., Gobbo, C., Marini, Z., & Sheese, R. (2007). *Cognitive Development: Neo-Piagetian Perspectives*. New York: Psychology Press.

Morrison, B. B., Dorn, B., & Guzdial, M. (2014). Measuring cognitive load in introductory CS: Adaptation of an instrument. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (pp. 131–138). New York: ACM.

Newman, R., Gatward, R., & Poppleton, M. (1970). Paradigms for teaching computer programming in higher education. *WIT Transactions on Information and Communication Technologies*, **7**, 299–305.

Newstead, P. R. (1975). Grade and ability predictions in an introductory programming course. *ACM SIGCSE Bulletin*, **7**, 87–91.

O'Donnell, R. (2009). Threshold concepts and their relevance to economics. In *14th Annual Australasian Teaching Economics Conference (ATEC 2009)* (pp. 190–200). Brisbane, Australia: School of Economics and Finance, Queensland University of Technology.

Oliver, D., Dobele, T., Greber, M., & Roberts, T. (2004). This course has a Bloom rating of 3.9. In *Proceedings of the Sixth Australasian Conference on Computing Education (ACE '04)* (pp. 227–231). Darlinghurst, Australia: Australian Computer Society.

Ormerod T. (1990). Human cognition and programming. In J. M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gillmore (Eds.), *Psychology of Programming* (pp. 63–82). London, UK: Academic Press.

Paas, F., Renkl, A., & Sweller, J. (2003). Cognitive load theory and instructional design: Recent developments. *Educational Psychologist*, **38**(1), 1–4.

Palumbo, D. (1990). Programming language/problem-solving research: A review of relevant issues. *Review of Educational Research*, **60**(1), 65–89.

Parsons, D., & Haden. P. (2006). Parson's programming puzzles: A fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education (ACE '06) Volume 52* (pp. 157–163). Darlinghurst, Australia: Australian Computer Society.

Patitsas, E., Berlin, J., Craig, M., & Easterbrook, S. (2016). Evidence that computer science grades are not bimodal. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 113–121). New York: ACM.

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, **2**(1), 25–36.

Pea, R. D., & Kurland, D. M. (1984). *On the Cognitive Prerequisites of Learning Computer Programming. Technical Report No. 18*. New York: Bank Street College of Education.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., & Paterson, J. (2007). A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, **39**(4), 204–223.

Perkins, D. N., & Martin, F (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway & S. Iyengar (Eds.), *Empirical Studies of Programmers, First Workshop* (pp. 213–229). Norwood, NJ: Ablex.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1989). Conditions of learning in novice programmers. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 261–279). Hillsdale, NJ: Lawrence Erlbaum.

Piaget, J. (1964). Part I: Cognitive development in children: Piaget development and learning. *Journal of Research in Science Teaching*, **2**(3), 176–186.

Piaget, J. (1971a). The theory of stages in cognitive development. In D. R. Green, M. P. Ford, & G. B. Flamer (Eds.), *Measurement and Piaget* (pp. 1–11). New York: McGraw-Hill.

Piaget, J. (1971b). Developmental stages and developmental processes. In D. R. Green, M. P. Ford, & G. B. Flamer (Eds.), *Measurement and Piaget* (pp. 172–188). New York: McGraw-Hill.

Plass, J. L., Moreno, R., & Brünken, R. (Eds.) (2010). *Cognitive Load Theory*. Cambridge, UK: Cambridge University Press.

Porter, L., & Zingaro, D. (2014). Importance of early performance in CS1: Two conflicting assessment stories. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (pp. 295–300). New York: ACM.

Porter, L., Zingaro, D., & Lister, R. (2014). Predicting student success using fine grain clicker data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (pp. 51–58). New York: ACM.

Price, T. W., & Barnes, T. (2015). Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 91–99). New York: ACM.

Rist, R. S. (1995). Program structure and design. *Cognitive Science*, **19**, 507–562.

Rist, R. S. (1986). Plans in programming: Definition, demonstration, and development. In E. Soloway & S. Iyengar (Eds.), *Empirical Studies of Programmers* (pp. 28–47). Norwood, NJ: Ablex Publishing.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, **13**, 389–414.

Rist, R. S. (2004). Learning to program: Schema creation, application, and evaluation. In S. Fincher & M. Petre (Eds.), *Computer Science Education Research* (pp. 175–195). London, UK: Taylor & Francis.

Robins, A. V. (2010). Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education*, **20**, 37–71.

Robins, A. V. (2018). Outcomes in introductory programming. *Computer Science Technical Report, OUCS-2018-02,* The University of Otago. Retrieved from www.otago.ac.nz/computer-science/otago685184.pdf

Robins, A. V., Haden, P., & Garner, S. (2006). Problem distributions in a CS1 course. In *Proceedings of the Eighth Australasian Computing Education Conference (ACE2006), CRPIT, 52* (pp. 165–173). Darlinghurst, Australia: Australian Computer Society.

Robins, A. V., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, **13**(2), 137–172.

Rogalski J., & Samurçay R. (1990). Acquisition of programming knowledge and skills. In J. M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gillmore (Eds.), *Psychology of Programming* (pp. 157–174). London, UK: Academic Press.

Rountree, J., Robins, A., & Rountree, N. (2013). Elaborating on threshold concepts, *Computer Science Education*, **23**(3), 265–289.

Rountree, N., Rountree, J., Robins, A., & Hannah, R. (2004). Interacting factors that predict success and failure in a CS1 course. *ACM SIGCSE Bulletin*, **36**(4), 101–104.

Rowbottom, D. P. (2007). Demystifying threshold concepts. *Journal of Philosophy of Education*, **41**, 263–270.

Sackman, H., Erickson, W. J., & Grant, E. E. (1968). Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, **11**(1), 3–11.

Sarawagi, N. (2014). A flipped CS0 classroom: Applying Bloom's taxonomy to algorithmic thinking. *Journal of Computing Sciences in Colleges*, **29**(6), 21–28.

Schneider, G. M. (1978). The introductory programming course in computer science: Ten principles. *ACM SIGCSE Bulletin*, **10**(1), 107–114.

Schulte, C., & Bennedsen, J. (2006). What do teachers teach in introductory programming? In *Proceedings of the Second International Workshop on Computing Education Research* (pp. 17–28). New York: ACM.

Schwill, A. (1997). Computer science education based on fundamental ideas. In D. Passey & B. Samways (Eds.), *Information Technology. IFIP Advances in Information and Communication Technology* (pp. 285–291). Boston, MA: Springer.

Schwill, A. (1994). Fundamental ideas of computer science. *EATCS-Bulletin*, **53**, 274–295.

Scott, T. (2003). Bloom's Taxonomy applied to testing in computer science classes. *Journal of Computing in Small Colleges*, **19**(1), 267–274.

Sheard, J., & Hagan, D. (1998). Our failing students: A study of a repeat group. *ACM SIGCSE Bulletin*, **30**(3), 223–227.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., & Whalley, J. L. (2008). Going SOLO to assess novice programmers. *ACM SIGCSE Bulletin*, **40**(3), 209–213.

Sheil, B. A. (1981). The psychological study of programming. *Computing Surveys*, **13**, 101–120.

Shih, Y. F., & Alessi, S. M. (1993). Mental models and transfer of learning in computer programming. *Journal of Research on Computing in Education*, **26**(2), 154–175.

Shinners-Kennedy, D., & Fincher, S. A. (2013). Identifying threshold concepts: From dead end to a new direction. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (pp. 9–18). New York: ACM.

Simon, Fincher, S., Robins, A., Baker, B., Box, I., Cutts, Q., de Raadt, M., Haden, P., Hamer, J., Hamilton, M., Lister, R., Petre, M., Sutton, K., Tolhurst, D., & Tutty, J. (2006). Predictors of success in a first programming course. In *Proceedings of the 8th Australasian Conference on Computing Education Volume 52* (pp. 189–196). Darlinghurst, Australia: Australian Computer Society,

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1983). What do novices know about programming? In B. Shneiderman & A. Badre (Eds.), *Directions in Human–Computer Interactions* (pp. 27–54). Norwood, NJ: Ablex.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, **29**(9), 850–858.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge.

IEEE Transactions on Software Engineering, **5**, 595–609.

Soloway, E., & Spohrer, J. C. (Eds.) (1989). *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum.

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

Sorva, J. (2012). *Visual Program Simulation in Introductory Programming Education* (doctoral dissertation 61/2012). Aalto University.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education (TOCE)*, **13**(2), 8.

Spohrer, J. C., & Soloway, E. (1989). Novice mistakes: Are the folk wisdoms correct? In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 401–416). Hillsdale, NJ: Lawrence Erlbaum.

Spohrer, J. C., Soloway, E., & Pope, E. (1989). A goal/plan analysis of buggy Pascal programs. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 355–399). Hillsdale, NJ: Lawrence Erlbaum.

Stachel, J., Marghitu, D., Brahim, T. B., Sims, R., Reynolds, L., & Czelusniak, V. (2013). Managing cognitive load in introductory programming courses: A cognitive aware scaffolding tool. *Journal of Integrated Design and Process Science*, **17**(1), 37–54.

Starr, C. W., Manaris, B., & Stalvey, R. H. (2008). Bloom's taxonomy revisited: Specifying assessable learning objectives in computer science. *ACM SIGCSE Bulletin*, **40**(1), 261–265.

Storey, M. A., Fracchia, F. D., & Müller, H. A. (1999). Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, **44**(3), 171–185.

Subramanian, A., & Joshi, K. (1996). Computer aptitude tests as predictors of novice computer programmer performance. *Journal of Information Technology Management*, **7**, 31–41.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, **12**(2), 257–285.

Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction*, **4**(4), 295–312.

Teague, D. (2015) *Neo-Piagetian Theory and the Novice Programmer* (doctoral thesis). Queensland University of Technology.

Teague, M. M. (2011). *Pedagogy of Introductory Computer Programming: A People-First Approach* (master's thesis). Queensland University of Technology.

Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., & Robbins, P. (2008). Bloom's taxonomy for CS assessment. In *Proceedings of the Tenth Conference on Australasian Computing Education (ACE '08)* (pp. 155–161). Darlinghurst, Australia: Australian Computer Society.

Utting, I., Tew, A. E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., Paterson, J., Caspersen, M., Kolikant, Y., Sorva, J., & Wilusz, T. (2013). A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITICSE Working Group Reports Conference on Innovation and Technology in Computer Science Education* (pp. 15–32). New York: ACM.

van Merriënboer, J. J. G. (1990). Strategies for programming instruction in high school: Program completion vs. program generation. *Journal of Educational Computing Research*, **6**(3), 265–285.

van Merriënboer, J. J. G., & de Croock, M. B. M. (1992). Strategies for computer-based programming instruction: Program completion vs. program generation. *Journal of Educational Computing Research*, **8**(3), 365–394.

van Merriënboer, J. J., & Paas, F. G. (1990). Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice. *Computers in Human Behavior*, **6**(3), 273–289.

Venables, A., Tan, G., & Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research* (pp. 117–128). New York: ACM.

Ventura, P. (2005). Identifying predictors of success for an objects-first CS1. *Computer Science Education*, **15**(3), 223–243.

Vihavainen, A., Airaksinen, J., & Watson, C. (2014). A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (pp. 19–26). New York: ACM.

Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (pp. 39–44). New York: ACM.

Webster, B. F. (1996). The real software crisis: The shortage of top-notch programmers threatens to become the limiting factor in software development. *Byte Magazine*, **21**, 218.

Weinberg, G. M. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

Weintrop, D., & Wilensky, U. (2015). To block or not to block, that is the question: Students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 199–208). New York: ACM.

Weintrop, D., Killen, H., & Franke, B. (2018). Blocks or Text? How Programming Language Modality Makes a Difference in Assessing Underrepresented Populations. In *Proceedings of the 13th International Conference of the Learning Sciences* (ICLS2018) (pp. 328–335). London, UK: International Society of the Learning Sciences.

Werth, L. H. (1986). Predicting student performance in a beginning computer science class. *ACM SIGCSE Bulletin*, **18**(1), 138–143.

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Ajith Kumar, P. K., & Prasad, C. (2006). An Austalasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. In *Proceedings of the 8th Australian Conference on Computing Education (ACE '06)* (pp. 243–252). Darlinghurst, Australia: Australian Computer Society.

White, G., & Sivitanides, M. (2002). A theory of the relationships between cognitive requirements of computer programming languages and programmers' cognitive characteristics. *Journal of Information Systems Education*, **13**(1), 59–66.

Wick, M. R. (2005). Teaching design patterns in CS1: A closed laboratory sequence based on the game of life. *ACM SIGCSE Bulletin*, **37**(1), 487–491.

Widowski, D., & Eyferth, K. (1986). Comprehending and recalling computer programs of different structural and semantic complexity by experts and novices. In H. P. Willumeit (Ed.), *Human Decision Making and Manual Control* (pp. 267–275). Amsterdam, The Netherlands: North–Holland, Elsevier.

Wiedenbeck, S., Fix, V., & Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: An empirical study. *International Journal of Man–Machine Studies*, **25**, 697–709.

Wiedenbeck, S., & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human–Computer Studies*, **51**(1), 71–87.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, **11**(3), 255–282.

Wileman, S. A., Konvalina, J., & Stephens, L. J. (1981). Factors influencing success in beginning computer science courses. *Journal of Educational Research*, **74**, 223–226.

Wilson, B. C., & Shrock, S. (2001). Contributing to success in an introductory computer science course: A study of twelve factors. *ACM SIGCSE Bulletin*, **33**(1), 184–188.

Winslow, L. E. (1996) Programming pedagogy – A psychological overview. *ACM SIGCSE Bulletin*, **28**(3), 17–22.

Woszczynski, A., Haddad, H., & Zgambo, A. (2005). Towards a model of student success in programming courses. In *Proceedings of the 43rd Annual Southeast Regional Conference – Volume 1 (ACM-SE 43)* (pp. 301–302). New York: ACM.

Yadin, A. (2011). Reducing the dropout rate in an introductory programming course. *ACM Inroads*, **2**(4), 71–76.

Yadin, A. (2013). Using unique assignments for reducing the bimodal grade distribution. *ACM Inroads*, **4**(1), 38–42.

13

Programming Paradigms and Beyond

Shriram Krishnamurthi and **Kathi Fisler**

13.1 Introduction and Scope

Programming is central to computing. It is both the practical tool that actually puts the power of computing to work and a source of intellectual stimulation and beauty. Therefore, programming education must be central to computing education. In the process, instructors need to make concrete choices about which languages and tools to use, and these will depend on their goals. Some emphasize the view of programming as a vocational skill that students must have to participate in the modern digital economy. Some highlight programming as an exciting, creative medium, comparable to classical media such as natural language, paint, and stone. Irrespective of the motivation for teaching programming, from a professional computing perspective, programs are a common medium for representing and communicating computational processes and algorithms.

Different languages have different affordances. As many people have remarked, programming languages are a human–computer interface. At the same time, programming languages are executed by computers, which work with an unyielding logic that is very different from that of most human-to-human communication. This tension between human comprehension and computer interpretation has manifested in many computing education research (CEdR) studies over the years. Indeed, navigating that tension (and