18

Principles of Programming Education

Michael E. Caspersen

Chapter outline

Chapter synopsis

The defining characteristics of the computer is its programmability, and programming is the essence of computing/informatics. Indeed, computing is much more than programming, but programming – the process of expressing one's ideas and understanding of the concepts and processes of a domain in a form that allows for execution on a computing device without human interpretation – is essential to computing.

Teaching and learning programming is not easy; in fact, it is considered one of the grand challenges of computing education. In this chapter, we describe the nature of the challenge, and we provide a dozen teaching principles to help overcome the challenge.

18.1 Introduction

Writing a chapter about the principles of teaching of programming is an intriguing task but for many reasons also challenging – an entire book could be written on the subject.

This decision has advantages and disadvantages. An advantage is that the chapter is applicable regardless of which language technology you as a teacher intend to use. A disadvantage is the lack of concrete examples, expressed in a specific programming technology that you can apply directly in your teaching.

Section 18.2 describes the challenge of teaching programming. In section 18.3 – the heart of the chapter – we present a dozen principles that can help overcome some of the challenges of teaching and learning programming.

18.2 Teaching and learning programming is a grand challenge

In some ways, programming education has changed dramatically over the past more than fifty years. We have experienced a rich and successful development in programming language technologies and an accompanying development of teaching practices.

However, in other ways, things have not changed that much, and it is still the case that typical introductory programming textbooks devote most of their content to presenting knowledge about a particular language (Robins, Rountree and Rountree, 2003).

Exposing students to the process of programming is merely implied but not explicitly addressed in texts on programming, which appear to deal with 'program' as a noun rather than as a verb.

But teaching programming is much more than teaching a programming language. Knowledge about a programming language is a necessary but far-from-sufficient condition for learning the practice of programming. Students also need knowledge about *the programming process*, that is, how to *develop* programs, and they need to extend that knowledge into *programming skills*.

David Gries (1974: 82) pointed this out already in 1974, when he wrote the following:

> Let me make an analogy to make my point clear. Suppose you attend a course in cabinet making. The instructor briefly shows you a saw, a plane, a hammer, and a few other tools, letting you use each one for a few minutes. He next shows you a beautifully-finished cabinet. Finally, he tells you to design and build your own cabinet and bring him the finished product in a few weeks. You would think he was crazy!

Clearly, cabinet making cannot be taught simply by teaching the tools of the trade and demonstrating finished products, but neither can programming.

Nevertheless, judging by the majority of past as well as contemporary textbooks, this is what is being attempted. In Kölling (2003), a survey of thirty-nine major selling textbooks on introductory programming was presented. The overall conclusion of the survey was that all books are structured according to the language constructs of the programming language; the process of program development is often merely implied rather than explicitly addressed.

A typical structure of a section on a specific language construct (e.g. the while loop) is the presentation of a problem followed by a presentation of a program to solve that problem and a discussion of the program's elements. From the viewpoint of a student, the program was developed in a single step, starting from a problem specification and resulting in a working solution.

This pattern of introducing material creates – unintentionally – the illusion that programming is trivial and straightforward. The fact that we all, when we start addressing a problem, start with incomplete and incorrect programs, which we then gradually modify by extending, refining and restructuring our implementation until we arrive at an acceptable solution, seems to be swept under the carpet as if it was an embarrassing secret that must not be mentioned. While the ultimate solution to the problem is explained in detail, the process – how we go about developing the solution – is almost entirely neglected in textbooks and beginners' courses (Caspersen and Kölling, 2009). Essentially, programming is one of the best-kept secrets of programming education!

In a time where computing/informatics education is becoming general education for all and students don't choose to learn programming out of personal interest, the challenge not only persists but also is reinforced.

According to du Boulay (1989) the difficulties of novices learning programming can be separated into five partially overlapping areas:

- Orientation: finding out what programming is for
- The notional machine: understanding the general properties of the machine that one is learning to control
- Notation: problems associated with the various formal languages that have to be learned, mastering both syntax and semantics
- Structure: the difficulties of acquiring standard patterns or schemas that can be used to achieve small-scale goals such as computing the sum using a loop or implementing a 0..* association between two classes
- Pragmatics: the skill of how to specify, develop, test and debug programs using whatever tools are available

The good news is that there are some relatively simple and effective didactical principles that help alleviate the challenge; we organize the principles in four categories:

1. Progression
   - **1.1.** Be application oriented
   - **1.2.** Let students progress from consumer to producer (use–modify–create)
   - **1.3.** Organize the progression in terms of complexity of tasks, not complexity of language constructs
2. Examples
   - **2.1.** Provide exemplary examples
   - **2.2.** Provide worked examples
   - **2.3.** Establish motivation through passion, play, peers and meaningful projects
3. Process
   - **3.1.** Reveal process and pragmatics
   - **3.2.** Provide scaffolding through stepwise self-explanations
   - **3.3.** Apply and teach incremental development through stepwise improvement (i.e. extend, refine, restructure)

4. Abstraction
   - **4.1.** Reinforce specifications
   - **4.2.** Reinforce patterns
   - **4.3.** Reinforce models and conceptual frameworks (program *into* a language)

These teaching principles are described in detail in the next section. However, while the principles help overcome many aspects of the challenge of teaching and learning programming, it still persists (Gries, 2002: 5):

> Programming is a skill, and teaching such a skill is much harder than teaching physics, calculus or chemistry. People expect a student coming out of a programming course to be able to program any problem. No such expectations exist for a calculus or chemistry student. Perhaps our expectations are too high.
>
> Compare programming to writing. In high school, one learns about writing in several courses. In addition, every college freshman takes a writing course. Yet, after all these courses, faculty members still complain that students cannot organise their thoughts and write well! In many ways, programming is harder than writing, so why should a single programming course produce students who can organise their programming thoughts and program well.

Is writing hard? Is teaching writing hard? Well, clearly it depends a great deal upon what you are trying to (teach your students to) write. Writing a novel, a textbook or a dissertation is very hard, and it is hard to teach how to do so. Writing an article, a feature or a report is also fairly hard and hard to teach. Writing a birthday greeting, a to-do list or a text message is much easier and requires very little instruction apart, of course, from learning the basics of (reading and) spelling.

Learning programming is not very different from learning writing. Most of the time, programming is creative and fun. However, like writing, programming is not trivial, and we teachers must embrace the challenge with enthusiasm as well as with a humble attitude and reasonable expectations. The most important aspect of learning programming as well as writing is to program/write things that matter to us.

18.3 Principles for teaching programming

In this section, we describe twelve teaching principles, or didactical principles, organized in four categories. The principles, which are all backed by research and experience, can help overcome some of the challenges of teaching and learning programming.

Progression

Principle 1.1: Be application oriented

Traditionally, introductory programming courses apply a bottom-up approach, in the sense that students are introduced to basic and foundational concepts and expected to master these before more advanced concepts and principles are introduced. Hence, in a traditional programming

course, students are often trained in constructing a trivial program as the very first activity, and then later on they are trained in adding more layers of complexity to a system in terms of user interfaces, databases, and so on. For the technically inclined students, this may be a feasible approach, but for a more general audience, this could pose severe motivational problems, as we are dealing with a wider range of students with much more diverse interests and backgrounds.

There is an even more important reason why a traditional bottom-up approach is fallible. For a general audience, we are not aiming at developing detailed and concrete competences in a specific programming technology; instead, we are aiming at developing interest, critical thinking, creativity and broader skills in programming and computational thinking and practice. Therefore, we recommend an application-oriented top-down approach. This implies to start various teaching activities by introducing well-known or familiar applications, which is then split apart for conceptual and/or technical examination, evaluation and modification.

For motivational reasons, we recommend applications based on the criteria that they must matter to students in the relevant age range, applications which they find interesting to use and hopefully to examine and improve. Examples could include pedagogical lightweight versions of Facebook, iTunes/Spotify, YouTube, Twitter, Blogs, Photoshop, Instagram and similar applications. Or it could be something embedded in a physical context based on Internet of Things, for example wearables and smart clothes. But these are just examples; in general, the choice of application types depends on the specific context and target group (Caspersen and Nowack, 2013).

For a specific technology like Scratch, there are a great number of approaches and domains that have inspired educators and researchers to develop teaching materials, for example a data-driven approach (Dasgupta and Resnick, 2014), a creativity- and maker-driven approach (Brennan, Balch and Chung, 2014) and a computing concept-driven approach (Armoni and Ben-Ari, 2010; Meerbaum-Salant, Armoni and Ben-Ari, 2013).

Principle 1.2: Let students progress from consumer to producer (use–modify–create)

Pattis (1990) introduces the *call before write* approach to teaching introductory programming, arguing that it 'allows students to write more interesting programs early in the course and it familiarizes them with the process of writing programs that call subprograms; so it is more natural for them to continue writing well-structured programs after they learn how to write their own subprograms'.

Meyer (1993) introduces the notion of the *inverted curriculum* as follows: 'This proposal suggests a redesign of the teaching of programming and other software topics in universities on the basis of object-oriented principles. It argues that the new "inverted curriculum" should give a central place to libraries, and take students from the reuse consumer's role to the role of producer through a process of "progressive opening of black boxes." '

) briefly mentions the notion of *consuming before producing* by providing three specific examples. One example is as follows: 'BlueJ allows beginning with an object "system" with just one class where students just interactively use instances of this class (they *consume* the notion of interacting with an object via its interface). *Producing* the possibility of interacting with an object, on the other

hand, requires more knowledge about class internals and should thus be done after the principle of interaction with objects is well understood.'

The *consume before produce* principle is applicable to a wide number of topics, for example code, specifications, class libraries and frameworks/event-driven programming.

Code: The principle may be applied with respect to the way students write code at three levels of abstractions: method level, class (or module) level and modelling level as follows: (1) *Use methods* (as indicated above, BlueJ allows interactive method invocation on objects without writing any code). At this early stage, students can perform experiments with objects in order to investigate the behaviour and determine the actual specification of a method. (2) *Modify methods* by altering statements or expressions in existing methods. (3) *Extend methods* by writing additional code in existing methods. (4) *Create methods* by adding new methods to an existing class. This may also be characterized as *extend class*. (5) *Create class/module* by adding new classes/modules to an existing model. This may also be characteriszd as *extend model*. (6) *Create model* by building a new model for a system to be implemented.

Specifications: Specifications and assertions can be expressed in many ways, for example as Javadoc, test cases, general assertions in code, loop invariants, class invariants and system invariants (constraints in the class model, for instance, a specific multiplicity on a relation between two classes). In all cases, students are gradually exposed to reading and comprehending specifications prior to producing specifications themselves.

Class libraries: Not many years ago, the standard syllabus for introductory programming courses encompassed implementation of standard algorithms for searching and sorting as well as implementation of standard data structures such as stacks, queues, linked lists, trees and binary search trees.

These days, standard algorithms and data structures are provided in class libraries, ready to be used by programmers. By using class libraries that provide advanced functionality, students can do much more interesting things more quickly. Also, experience as consumer presumably motivates learning more about the principles and theory behind advanced data structures and packages for distributed programming, and so on.

Consequently, algorithms and data structures are one of the areas where we can sacrifice material in order to find room for all the new things that make up a modern introductory programming course.

Frameworks/event-driven programming: Sometimes even using a piece of software can be a daunting task. Frameworks are examples of such complex pieces of software.

Frameworks may constitute a part of an introductory programming course, but in order to ease comprehension of a complex frameworks with call-back methods (inversion of control), it helps to provide a stepping stone in the form of a small and simple framework, which students consume by making a few simple instantiations. After the road has been paved, you may provide a more general taxonomy for frameworks/event-based programming, which is now more easily understood and grasped in the context of the simple toy framework. With the concrete experiences and the taxonomy in the bag, students are prepared to embark on using more complex frameworks.

Christensen and Caspersen (2002) provide a more thorough discussion of an approach to teaching frameworks and event-based programming in introductory programming courses.

Use–Modify–Create: Lee et al. (2011) describe how computational thinking takes shape for middle and high school youth in the United States. They propose using a three-stage progression for

engaging youth in computational thinking. This progression, called use–modify–create, describes a pattern of engagement that was seen to support and deepen youths' acquisition of computational thinking (see also Chapter 3, 'Perspectives on Computing Curricula').

Caspersen and Bennedsen (2007) describe a similar three-stage progression for working with programs, called use–extend–create. Caspersen and Nowack (2013) also describe use–modify–create as a specialization of consumer-to-producer. A more elaborate or fine-grained version is use–alter–test–modify–assess–refine–evaluate–create.

conclude that the use–modify–create progression is a useful framework for educators and researchers looking at how computational thinking develops and how that development can be supported.

Principle 1.3: Organize the progression in terms of complexity of tasks, not complexity of language constructs

Typically, progression in introductory programming courses is dictated by a bottom-up treatment of the language constructs of the programming language being used, and this is the way most textbooks are structured. We hold as a general principle that the progression in the course is defined in terms of the complexity of the worked examples presented to the students and the corresponding exercises and assignments.

Example: Progression

A concrete example of progression in terms of complexity of task is provided by the classical Turtle Graphics, developed by Seymour Papert. Students can start by making utterly simple programs/drawings and then gradually, based on the student's ambitions and skills, progress to make more advanced and ultimately quite complicated programs/drawings. Thus, Turtle Graphics exhibit what Papert called 'low floor, high ceiling'; that is, it allows for easy entrance without restricting the power of expression.

Another example is to let the progression be defined in terms of complexity of program structure or architecture (e.g. classes and their relationship) by starting with very simple programs with simple functionality and only few components with very simple relationships and then progress to more complex programs with increasingly complex structure and richer functionality. This could be starting with a program with just one component (representing, say, a die, a date, a person, an image, a song, etc.); then working with programs with two components (representing, say, a die and a die cup, a date and a clock, a person and a party, a song and a playlist, etc.) and then go on to working with programs with many components and more complex structures (e.g. a game, music player, an image processing app, etc.). See Bennedsen and Caspersen (2004b) for further details.

It is of course possible to turn things around and start out using a more complex app, then zoom in and work on a smaller part, perhaps just one component, while modifying this.

Examples

Principle 2.1: Provide exemplary examples

Examples are important teaching tools. Research in cognitive science confirms that 'examples appear to play a central role in the early phases of cognitive skill acquisition' (VanLehn, 1996). An alternation of worked examples and problems increases the learning outcome compared with just solving more problems (Sweller and Cooper, 1985; Trafton and Reiser, 1993).

Students generalize from examples and use them as templates for their own work. Examples must therefore be easy to generalize and consistent with current learning goals.

By perpetually exposing students to 'exemplary' examples, desirable properties are reinforced many times. Students will eventually recognize patterns of 'good' design and gain experience in telling desirable from undesirable properties.

With carefully developed examples, teachers can minimize the risk of misinterpretations and erroneous conclusions, which otherwise can lead to misconceptions. Once established, misconceptions can be difficult to resolve and hinder students in their further learning (Clancy, 2004; Ragonis and Ben-Ari, 2005). See also Chapter 20, in which Juha Sorva details many different misconceptions.

A good example must be understandable by students. Without 'understanding', knowledge retrieval works only on an example's surface properties, instead of on its underlying structural and conceptual properties (Trafton and Reiser, 1993; VanLehn, 1996).

A good example must furthermore effectively communicate the concept(s) to be taught. There should be no doubt about what exactly is exemplified. The structural form of information affects the form of the knowledge encoded in human memory. Conceptual knowledge is improved by best examples and by expository examples, where the best example represents an average, central or prototypical form of a concept. To minimize cognitive load, an example should furthermore exemplify only one new concept (or very few) at a time.

The two example properties, (1) understandable by students and (2) effectively communicating the concept(s) to be taught, might seem obvious. However, the recurring discussions about the harmfulness or not of certain common examples show that there is quite some disagreement in the teaching community about the meaning of these properties. For further details, including many more references, see Börstler, Caspersen and Nordström (2007, 2016) and Börstler, Nordström and Paterson (2011).

Principle 2.2: Provide worked examples

Studies of students in a variety of instructional situations have shown that students prefer learning from examples rather than learning from other forms of instruction. Students learn more from studying examples than from solving the same problems themselves.

A worked example (WE), consisting of (1) a problem statement and (2) a procedure for solving the problem, is an instructional device that provides a problem a solution for a learner to study (Atkinson et al., 2000; Chi et al., 1989; LeFevre and Dixon, 1986). WEs are meant to illustrate how similar problems might be solved, and WEs are effective instructional tools in many domains, including programming. WEs combined with faded guidance are particularly effective (Caspersen and Bennedsen, 2007).

Atkinson et al. (2000) emphasize three major categories that influence learning from worked examples; Caspersen and Bennedsen (2007) present the categories as how-to principles of constructing and applying examples in education: (1) How to construct examples, (2) how to design lessons that include examples and (3) how to foster students' thinking process when studying examples. We return to the latter when we discuss Principle 3.2: Provide scaffolding through stepwise self-explanations.

Bennedsen and Caspersen (2004a) illustrate implicitly how WEs can be used to teach programming using a systematic, model-based programming process. Caspersen and Bennedsen (2007) present an instructional design for an introductory programming course based on a thorough use of WE, and Caspersen (2007) provides an overview of WE literature related to programming education as well as a survey of the related cognitive load theory (CLT).

Principle 2.3: Establish motivation through passion, play, peers and meaningful projects

Mitch Resnick and his research group at the Lifelong Kindergarten at the MIT Media Lab have been developing new technologies, activities and strategies to engage young people in creative learning experiences (Resnick, 2014). Their approach is based on four core elements, sometimes called the Four P's of Creative Learning:

- *Projects*. People learn best when they are actively working on meaningful projects – generating new ideas, designing prototypes, refining iteratively.
- *Peers*. Learning flourishes as a social activity, with people sharing ideas, collaborating on projects and building on one another's work.
- *Passion*. When people work on projects they care about, they work longer and harder, persist in the face of challenges and learn more in the process.
- *Play*. Learning involves playful experimentation – trying new things, tinkering with materials, testing boundaries, taking risks, iterating again and again.

In this example, WE and faded guidance is applied in five stages as follows:

1. In a video, present development of a player with two components, Playlist and Track. (A complete WE)
2. In a lecture, present (a partial) development of a similar example, that is, with the same structure but different cover stories. This could be, say, a simple banking system with components Account and Transaction. (A partial WE)
3. In a lab session, students use, modify and extend both examples.
4. In an exercise, students extend the player by adding an Image component that allows several images to be displayed when a Track is played.
5. In an assignment, students implement a similar system (again, similar structure, but different cover story), say, a notebook app with Notes, Keywords, Contacts, and so on.

Process

As mentioned in the first section, the process of programming is one of the best-kept secrets of programming education.

Thus, students are left on their own to find their process of programming. Instead of leaving the students on their own, we as educators must help students to develop a systematic process to learning programming; and we must provide guided tours to proper program development.

This section deals with the process from three perspectives: how to reveal the programming process, how to facilitate process-based self-explanations and how to conceptualize the programming process as consisting of three independent (but of course related) activities: extension, refinement and restructuring – stepwise improvement.

Principle 3.1: Reveal process and pragmatics

Revealing the programming process to beginner students is important, but traditional static teaching materials such as textbooks, lecture notes, blackboards, slide presentations, and so on are insufficient for that purpose. They are useful for the presentation of a product (e.g. a finished program), but not for the presentation of the dynamic process used to create that product.

In addition, the use of traditional materials has another drawback. Typically, they are used for the presentation of an ideal solution that is the result of a non-linear development process. Like others (Soloway, 1986; Spohrer and Soloway, 1986), we consider this to be problematic, because it will inevitably leave the students with the false impression that there is a linear and direct 'royal road' from problem to solution.

This is very far from the truth, but the problem for novices is that when they see their teacher present clean and simple solutions, they think they themselves should be able to develop solutions in a similar way. When they realize they cannot do so, they blame themselves and feel incompetent. Consequently, they will lose self-confidence and, in the worst case, their motivation for learning to program.

Therefore, we must also teach about the programming process. This can include the task of using tools and techniques to develop the solution in a systematic, incremental and typically non-linear way. An important part of this is to expound and to demonstrate that

- many small steps are better than a few large ones
- the result of every little step should be tested
- prior decisions may need to be undone and code refactored
- making errors is common also for experienced programmers
- compiler errors can be misleading/erroneous
- online documentation for class libraries provides valuable information and
- there is a systematic, however non-linear, way of developing a solution for the problem at hand

We cannot rely on the students to learn all of this by themselves, but by using an apprenticeship approach, we can show them how to do it. WEs are highly suitable for this purpose (see principle 2.2), and they can be effectively communicated via videos. For many more details on this, see Bennedsen and Caspersen (2005).

Principle 3.2: Provide scaffolding through stepwise self-explanations

Self-explanations provide guidance regarding the way that students can study and understand instructional material. Clark, Nguyen and Sweller (2005) define SE as 'a mental dialog that learners have when studying a worked example that helps them understand the example and build a schema from it'. According to Chiu and Chi (2014), the activity of self-explaining promotes learning through the elaboration of information being studied, associating this new information with learners' prior knowledge, making inferences and connecting two or more pieces of the given information.

The benefits of self-explanations were first shown by Chi et al. (1989). They found that good students' explanations provided justifications for steps in the examples and related those steps to the concepts presented in the instructional material. Those students also monitored their understanding while studying the examples.

More information on self-explanation and stepwise self-explanation (a specialization of self-explanation related to stepwise improvement) can be found in Caspersen (2007) and Aureliano, Tedesco and Caspersen (2016).

Principle 3.3: Apply and teach incremental development through stepwise improvement (i.e. extend, refine, restructure)

In traditional stepwise refinement (Dijkstra, 1969; Wirth, 1971; Back, 1978, 1988; Morgan, 1990), programming is regarded as the one-dimensional activity of refining abstract programs (i.e. programs containing non-executable specifications) to concrete programs (i.e. executable code) through a series of behaviour-preserving program transformations. The fundamental assumption of traditional stepwise refinement is that the complete specification, or the requirements, is known and addressed from the outset. Typically, stepwise refinement is described as a strict top-down process of programming.

Programming by stepwise improvement (Caspersen 2007), on the other hand, is characterized as an explorative activity of discovery and invention that takes place in the three-dimensional space of *extension*, *refinement* and *restructuring*. Extension is the activity of extending the specification to cover more (use) cases; refinement is the activity of refining abstract code to executable code to meet the current specification, and restructure is the activity of improving non-functional aspects of a solution without altering its observable behaviour, such as design improvements through refactoring, efficiency optimizations or portability improvements.

A very simple example of stepwise improvement is the development of an app that can show a date and advance to the next/previous date by pushing dedicated buttons in the user interface. Such an app can be developed according to stepwise improvement in the following extension steps:

1. Construct (part of) the user interface, no functionality
2. Make the app work except for the last day of a month (assume thirty days in every month and ignore leap years)
3. Make the app work for a variable number of days per month
4. Make the app work for leap years (except centuries)

5. Make the app work for centuries (except four-centuries)
6. Make the app work for four-centuries

By breaking the problem down like this and developing the program in a number of increments/iterations where the specification is gradually extended, the programming task becomes much more manageable. Also, it allows for a success/celebration every time a new version is finished. There are many pedagogical advantages of organizing students' programming process according to stepwise improvement.

From the stepwise improvement framework, Caspersen and Kölling have designed a novice's process of (object-oriented) programming called STREAM (Caspersen and Kölling, 2009).

Abstraction

Principle 4.1: Reinforce specifications

In programming, it is essential, at many levels of abstraction, to be able to distinguish and separate *what* a (part of a) program does from *how* it does it. A description of what a program part does is called a specification; the implementation, that is, the actual code of the program part, is the ultimate description of how it does it. Typically, one specification has many implementations; that is, there may be many concrete ways to obtain a desired outcome (to meet a specification).

A specification may be expressed as a name of a function, as a comment in natural text or in a more formal way. The concrete syntactic expression of a specification is not essential; it is the notion of specification itself, and the ability to separate specification from implementation, that is essential. As Pattis (1990) points out: 'the linguistic ability to cleanly separate a subprogram's specification from its implementation' is required in order to practice the 'call before write' approach.

We therefore hold as principle that the notion of specification is treated as a first-class citizen in introductory programming courses. See Caspersen (2007) for more on this.

Principle 4.2: Reinforce patterns

A pattern captures and describes (the essence of) a recurring structure or process in a given domain. In music, there are patterns of chords that, with minor or major variations of the melody, are used again and again (search the web for 'three chords' or 'four chords' and see for yourself). This is also the case in programming, where programming patterns are used again and again to obtain variations of essentially the same structure or process.

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

We reinforce patterns at different levels of abstraction including elementary patterns, algorithm patterns and design patterns, but equally importantly, we provide a conceptual framework for object orientation that qualifies modelling and programming and increases transfer. Furthermore, we

stress coding patterns for standard relations between classes (Knudsen and Madsen, 1988; Madsen et al., 1993; East et al., 1996; Muller, 2005; Caspersen, 2007; Caspersen and Bennedsen, 2007).

Principle 4.3: Reinforce models and conceptual frameworks (program into a language)

The so-called Sapir-Whorf hypothesis from linguistics states that *language defines the boundaries of thought*. Programming languages are artificial and simple languages, and if a programmer's thoughts are nurtured only via the constructs of a specific programming language, these thoughts will be severely constrained and have very limiting boundaries.

All programming languages have limitations, and we overcome these limitations by thinking and designing in terms of richer and more appropriate concepts and structures, which we simulate in the technology at hand.

In the early days of assembly programming, programmers used jump and compare instructions to *simulate* selection and iteration. This was also the case for the earliest high-level languages with go-to statements as the only control structure. Then, control structures for selection and iteration were developed.

Similarly, in the early days of programming, there was no programming language support for arrays (lists) and records (tuples). Consequently, these had to be simulated through careful and minute programming activities.

As long as there was no support for subprograms but the notion was conceived, programmers had to simulate call and return (and, in the general case, maintain not one but a stack of return addresses).

When there was no support for classes, but the notion of abstract data type was conceived, programmers had to simulate this, again through structured and minute programming activities.

The historic development of programming languages can be viewed as a constant interplay between programming language constructs and architectural abstractions. Limitations in programming languages generate new concepts, architectural abstractions, which can be simulated in existing programming languages. Gradually, these architectural abstractions find their way into mainstream languages but only to generate more advanced needs and foster new architectural abstractions.

In object-oriented programming, UML and similar modelling languages provide a richer conceptual framework than most object-oriented programming languages support. For example, the notion of association (a special relation between program components) is not directly supported but has to be simulated through structured and minute programming activities. In principle, this is exactly the same situation as simulation of a while loop with go-to statements.

When teaching programming, it is important to provide a conceptual framework, which is richer than the concrete programming language being used. Programming should not be done *in* a language, but *into* a language.

Conceptual modelling. As a general foundation for informatics/computing, we recommend a general introduction to conceptual modelling. Unfortunately, there are no introductory texts on the subject, but Kristensen and Østerbye (1994: 83–86) provide a nice discussion on the subject;

the authors present the topic in the context of object-oriented programming languages, but it has much wider applications and implications.

The exposition in Kristensen and Østerby, which originates from Madsen, Møller-Perdersen and Nygaard (1993), presents a model of abstraction consisting of three abstraction processes (and their inverses): classification (exemplification), aggregation (decomposition) and generalization (specialization). The model may not be complete, but it has shown its applicability in many cases, including data modelling and object-oriented modelling, and has the potential to become a much more general framework for informatics and computational thinking. The current focus on computational thinking emphasizes abstraction and decomposition as major aspects (); however, the above-mentioned framework for conceptual modelling provides a richer and more general approach.

Sowa (1984) provides a broader approach to the topic but with a different perspective.

There are many learning-theoretic arguments for adopting a conceptual framework approach, for example a model-driven approach, to programming. We provide two (for more details, see Caspersen, 2007):

1. Because of their generic nature, the abstract models directly support schema creation and transfer:

> Well-designed learning environments for novices provide *metacognitive managerial guidance* to focus the students' attention and *schema substitutes* by optimizing the limited capacity of working memory in ways that free working memory for learning. Good instruction will segment and sequence the content in ways that reduce the amount of new information novices must process at one time and, as much as possible, reinforce domain patterns to support schema acquisition and improve learning.

2. Variation of form (e.g. cover story) can help novices realize that there is a many-to-one relationship between form and problem type: when students see a variety of cover stories used for identical or similar structures (of class models), they are more likely to notice that surface features are insufficient to distinguish among problem types and that problem categorization according to structural similarities (patterns) is imperative to enable reuse of solution schemas (Quilici and Mayer, 1996).

Models provide an excellent overview and generic approach to introductory programming. If pedagogical development tools more completely supported integration of code and UML-like models, we conjecture that the effect would be even better.

Example: Elementary patterns

Elementary patterns, that is, patterns for elementary program structure, exist in a number of variations, for example roles of variables (Sajaniemi, 2008), selection patterns (Bergin, 1999) and loop patterns (Astrachan and Wallingford, 1998).

Roles of variables represent programming knowledge that can be explicitly taught to students and which are easy to adopt in teaching. A number of roles for variables was identified and described by Sajaniemi (2008), for example fixed value, organizer, stepper, most-recent holder, one-way flag, most–wanted holder and so on.

Bergin (1999) divides selection patterns in three categories: basic selection patterns, strategy patterns and auxiliary patterns. Examples of basic selection patterns are Whether or Not and Alternative Action, examples of strategy patterns are Short Case First and Default Case First and examples of auxiliary patterns are Positive Condition and Function for Positive Condition.

Astrachan and Wallingford (1998) present a number of loop patterns related to sweeping over a (linear) collection of data. Examples of elementary loop patterns are Linear Search, Guarded Linear Search, Process All Items and Loop and a Half.

Key points

- Teaching and learning programming is a grand challenge
- In a time where computing/informatics education is becoming general education for all and students don't choose to learn programming out of personal interest, the challenge not only persists but also is reinforced
- Exposing students to the process of programming is merely implied but not explicitly addressed in texts on programming, which appear to deal with 'program' as a noun rather than as a verb
- Some relatively simple and effective teaching principles can help alleviate and overcome the challenge:
  - Progression: Be application-oriented; let students progress from consumer to producer (use-modify-create); organize progression in terms of complexity of task, not complexity of language constructs
  - Examples: Provide exemplary examples; provide worked examples; establish motivation through passion, play, peers and meaningful projects
  - Process: Reveal process and pragmatics; provide scaffolding through stepwise self-explanations; apply and teach incremental development through stepwise improvement (i.e. extend, refine, restructure).
  - Abstraction: Reinforce specifications; reinforce patterns; reinforce models and conceptual frameworks (program *into* a language)

For further reflection

- The use of examples is important in helping students to improve their programming skills. Consider how worked examples could be used in your classroom to support a particular programming topic.
- Considering the principles presented in this chapter, how do you think these could be incorporated at different points with young, beginner programmers to enable *progression* in programming.

References

Armoni, M., and Ben-Ari, M. (2010), Computer Science Concepts in Scratch, Licensed under Creative Commons. https://stwww1.weizmann.ac.il/scratch/scratch_en/.

Astrachan, O., and Wallingford, E. (1998), Loop Patterns. https://users.cs.duke.edu/~ola/patterns/plopd/loops.html.

Atkinson, R. K., Derry, S. J., Renkl, A., and Wortham, D. (2000), 'Learning from Examples: Instructional Principles from the Worked Examples Research', *Review of Educational Research*, 70: 181–214.

Aureliano, V. C. O., Tedesco, P. C. de A.R., and Caspersen, M. E. (2016), 'Learning Programming through Stepwise Self-Explanations', in *Proceedings of the 11th Conferencia Iberica de*, Sistemas y Technologias de Information.

Back, R. J. (1978), *On the Correctness of Refinement Steps in Program Development*, Helsinki, Finland: Department of Computer Science, University of Helsinki.

Back, R. J. (1998), *Refinement Calculus: A Systematic Introduction*, Berlin: Springer-Verlag.

Bennedsen, J., and Caspersen, M. E. (2004a). 'Programming in Context—A Model-First Approach to CS1', in *Proceedings of the Thirty-Fifth SIGCSE Technical Symposium on Computer Science Education, SIGCSE,* 477–81.

Bennedsen, J., and Caspersen, M. E. (2004b), 'Teaching Object-Oriented Programming—Towards Teaching a Systematic Programming Process', *in Proceedings of the Eighth Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts*, 18th European Conference on Object-Oriented Programming.

Bennedsen, J., and Caspersen, M. E. (2005), 'Revealing the Programming Process', in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education, SIGCSE*, 186–90.

Bergin, J. (1999), Patterns for Selection. http://www.cs.uni.edu/~wallingf/patterns/elementary/papers/selection.pdf.

Brennan, K., Balch, C., and Chung, M. (2014), 'Creative Computing', Harvard Graduate School of Education. http://scratched.gse.harvard.edu/guide/.

Börstler, J., Caspersen, M. E., and Nordström, M. (2007), 'Beauty and the Beast—Toward a Measurement Framework for Example Program Quality'. Technical Report UMINF-07.23, Umeå, Sweden: Department of Computing Science, Umeå University.

Börstler, J., Caspersen, M. E., and Nordström, M. (2016), 'Beauty and the Beast: On the Readability of Object-Oriented Example Programs', *Software Quality Journal*, 24 (2), 231–46.

Börstler, J., Nordström, M., and Paterson, J. H. (2011), 'On the Quality of Examples in Introductory Java Textbooks', *Transactions on Computing Education*, 11: 1–21.

Caspersen, M. E. (2007), Educating Novices in the Skills of Programming, DAIMI PhD Dissertation PD-07-4, Denmark: Department of Computer Science, Aarhus University. http://www.cs.au.dk/~mec/dissertation/Dissertation.pdf.

Caspersen, M. E., and Bennedsen, J. (2007), 'Instructional Design of a Programming Course—A Learning Theoretic Approach', in *Proceedings of the 2007 International Workshop on Computing Education Research*, 111–22, Atlanta.

Caspersen, M. E., and Kölling, M. (2009), 'STREAM: A First Programming Process', *ACM Transactions on Computing Education*, 9(1): 1–29.

-1
0
+1

Caspersen, M. E., and Nowack, P. (2013), 'Computational Thinking and Practice—A Generic Approach to Computing in Danish High Schools', in *Proceedings of the 15th Australasian Computer Education Conference*, 137–43, Adelaide, South Australia.

Chi, M. T. H., Bassok, M., Lewis, M., Reimann, M., and Glaser, R. (1989), 'Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems', *Cognitive Science*, 13: 145–82.

Chiu, J. L., and Chi, M. T. H. (2014), 'Supporting Self-Explanation in the Classroom', in A. Benassi, C. E. Overson and C. M. Hakala (eds), Applying Science of Learning in Education: Infusing Psychological Science into the Curriculum. http://teachpsych.org/ebooks/asle2014index.php.

Clark, R. C., Nguyen, F., and Sweller, J. (2005), *Efficiency in Learning: Evidence-Based Guidelines to Manage Cognitive Load*, Chichester: Wiley.

Christensen, H. B., and Caspersen, M. E. (2002), 'Frameworks in CS1—A Different Way of Introducing Event-Driven Programming', in *Proceedings of the Seventh Annual Conference on Innovation and Technology in Computer Science Education*.

Clancy, M. (2004), 'Misconceptions and Attitudes That Interfere with Learning to Program', in S. Fincher and M. Petre (eds), *Computer Science Education Research*, 85–100, Abingdon: Taylor & Francis.

Dasgupta, S., and Resnick, M. (2014), 'Engaging Novices in Programming, Experimenting, and Learning with Data', *ACM Inroads*, 5 (4): 72–5.

Dijkstra, E. W. (1969), *Notes on Structured Programming*, TH Report 70. Technical University Eindhoven, Netherlands. https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

du Boulay, B. (1989), 'Some Difficulties of Learning to Program', *Studying the Novice Programmer*, 57–73, Hillsdale, NJ: Lawrence Erlbaum.

East, J. P., Thomas, S. R., Wallingford, E., Beck, W., and Drake, J. (1996), *Pattern-Based Programming Instruction*, in *Proceedings of the ASEE Annual Conference and Exposition*, Washington, DC.

Gries, D. (1974), 'What Should We Teach in an Introductory Programming Course?' *in Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education*, 81–9.

Gries, D. (2002). 'Where Is Programming Methodology These Days?' *ACM SIGCSE Bulletin*, 34 (4), 5–7.

Grover, S., and Pea, R. (2013), 'Computational Thinking in K-12: A Review of the State of the Field', *Educational Researcher*, 42 (1): 38–43.

Knudsen, J. L., and Madsen, O. L. (1988), 'Teaching Object-Oriented Programming Is More Than Teaching Object-Oriented Programming Languages', *ECOOP '88 European Conference on Object-Oriented Programming*, 21–40. Berlin: Springer.

Kölling, M. (2003), The Curse of Hello World', Workshop on Learning and Teaching Object-Orientation—Scandinavian Perspectives, Oslo.

Kristensen, B. B., and Østerbye, K. (1994), 'Conceptual Modeling and Programming Languages', *ACM SIGPLAN Notices*, 29(9): 81–90.

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., and Werner, L. (2011), 'Computational Thinking for Youth in Practice', *ACM Inroads*, 2(1): 32–7.

LeFevre, J., and Dixon, P. (1986), 'Do Written Instruction Need Examples?' *Cognition and Instruction*, 3: 1–30.

Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. (1993), *Object-Oriented Programming in the BETA Programming Language,* Wokingham: Addison-Wesley.

Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. (2013), 'Learning Computer Science Concepts with Scratch', *Computer Science Education*, 23 (3): 239–64.

-1
0
+1

Meyer, B. (1993), 'Towards an Object-Oriented Curriculum', *Journal of Object-Oriented Programming*, 6 (2): 76–81.
Morgan, C. (1990), *Programming from Specifications*, Upper Saddle River, NJ: Prentice Hall.
Muller, O. (2005), 'Pattern Oriented Instruction and the Enhancement of Analogical Reasoning'. in *Proceedings of the 2005 International Workshop on Computing Education Research*, 57–67.
Pattis, R. E. (1990), 'A Philosophy and Example of CS-1 Programming Projects', in *Proceedings of the Twenty-First SIGCSE Technical Symposium on Computer Science Education*, 34–39.
Quilici, J. L., and Mayer, R. E. (1996), 'Role of Examples in How Students Learn to Categorize Statistics Word Problems', *Journal of Educational Psychology*, 88: 144–61.
Ragonis, N., and Ben-Ari, M. (2005), 'A Long-Term Investigation of the Comprehension of OOP Concepts by Novices', *Computer Science Education*, 15 (3): 203–21.
Resnick, M. (2014), 'Give P's a Chance', in *Constructionism and Creativity Conference*, Opening Keynote, Vienna.
Robins, A., Rountree, J., and Rountree, N. (2003), 'Learning and Teaching Programming: A Review and Discussion', *Journal of Computer Science Education*, 13 (2): 137–72.
Sajaniemi, J. (2008), 'Roles of Variables'. http://www.cs.joensuu.fi/~saja/var_roles/.
Schmolitzky, A. (2005), 'Towards Complexity Levels of Object Systems Used in Software Engineering Education', in *Proceedings of the Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts, 19th European Conference on Object-Oriented Programming*. https://tinyurl.com/yd9vv5w6.
Soloway, E. (1986), 'Learning to Program—Learning to Construct Mechanisms and Explanations', *Communications of the ACM*, 29 (9): 850–8.
Spohrer, J. C., and Soloway, E. (1986), 'Novice Mistakes: Are the Folk Wisdoms Correct?', *Communications of the ACM*, 29 (7): 624–32.
Sweller, J., and Cooper, G. (1985), 'The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra', *Cognition and Instruction*, 2: 59–89.
Sowa, J. F. (1984). *Conceptual Structures: Information Processing in Mind and Machine,* Boston, MA: Addison-Wesley.
Trafton, J. G., and Reiser, B. J. (1993), 'Studying Examples and Solving Problems: Contributions to Skill Acquisition', Washington, DC: Technical report, Naval HCI Research Lab.
VanLehn, K. (1996), 'Cognitive Skill Acquisition', *Annual Review of Psychology*, 47: 513–39.
Wirth, N. (1971), 'Program Development by Stepwise Refinement', *Communications of the ACM*, 14 (4): 221–7.