Preface

 Everybody should learn how to program a computer because it teaches you how to think. - Steve Jobs

Computer Science is the study of computers and algorithmic processes, including their principles, their hardware and software designs, their applications, and their impact on society (Tucker, 2006).

In the education context, Computer Science encompasses several topics such as algorithmic problem-solving, computing and data analysis, human-computer interaction, programming, security, Web design, robotics, and many others. In Computer Science courses, students develop computational and critical thinking skills and how to create, not simply use, new technologies. This fundamental knowledge is crucial to prepare students for the 21st century, eager for people with this training, regardless of their ultimate field of study or occupation, such as:

- **Arts:** Designing, developing, and composing digital music and special effects for movies;
- **Finances:** Designing and overseeing automated trading services;
- **Healthcare:** Designing and developing security and privacy for medical records or new remote monitoring systems for patients;
- **Information Technology:** Designing and developing software and hardware systems for mobile communication devices, networks, applications, and games;
- **Manufacturing:** Designing, developing, and using simulations to improve products;
- **Retail:** Analyzing data to predict trends and improve inventory management;
- **Weather Forecasting:** Designing, developing, and interpreting models to predict the behavior of the weather (for instance, hurricanes).

In fact, the United States Bureau of Labor Statistics predicts that one in every two Science, Technology, Engineering, and Math (STEM) jobs will be in computing in 2020, with more than 150k job openings annually, making it one of the fastest growing occupations. The other important point is that these jobs will pay 75 percent more than the national median annual salary.

These predictions and the breadth of new ways in which computing knowledge prepares people for multiple careers demands even more the inclusion of foundational computer science courses in all levels of education, such as K–12. These courses aims to teach the basic and core concepts of computing, such as abstraction, creativity, algorithms, programming, and Internet, that are fundamental to computer science and also suitable to many others disciplines.

In this context, several organizations are bringing Computer Science classes to every K-12 school. One that stands out is Code.org. This non-profit organization, led by brothers Hadi and Ali Partovi, aims to motivate people, especially school students in the United States, to learn to code. Their website includes
free coding lessons and encourages them to include more computer science classes in the curriculum. On December 2013, Code.org launched the Hour of Code 2013 challenge nationwide to promote computer science. Some of the most important tech companies and their founders, including Bill Gates and Mark Zuckerberg, have put up about $10 million for Code.org.

THE CHALLENGES

Despite the need to learn Computer Science by every student in every school, there are many issues that must be addressed.

Introductory programming courses are characterized by an extensive curricula and a high enrollment of students. This poses a great workload for faculty and teaching assistants responsible for the creation, delivery, and assessment of student exercises. These courses are also regarded as difficult and often have high failure and dropout rates. Researchers have pointed out several causes for these rates from teaching methods (lectures on programming language syntaxes), from the lack of feedback to the subject complexity. In fact, learning how to program means integrating knowledge of a wide variety of conceptual domains, such as computer science and mathematics, while developing expertise in problem understanding, problem solving, unit testing, and other skills. Additionally, students peter out when they need to understand and apply abstract programming concepts like control structures or when they need to create algorithms that solve concrete problems. All these issues pose huge challenges in the computer programming teaching-learning process.

Some key points for the success of this process are practice and feedback. For someone to acquire, improve, or even maintain a complex skill, it is necessary to practice it on a regular basis (Gross & Powers, 2005; Eckerdal, 2009). The amount of practice required depends on the nature of the activity and on each individual. How well an individual improves with practice is directly related with her or his inherent aptitudes, previous know-how, and on the feedback. If feedback is either non-existent or inappropriate, then the practice tends to be ineffective or even detrimental to learning.

There are several complex skills that require constant practice, where exercise solving is a key component, such as management, health sciences, and electronics. Playing business games in management courses, or simulating a human patient in life sciences courses, or simulating an electronic circuit in electronics courses are examples of learning processes that require the use of special authoring, rendering, and assessment tools. These tools should be integrated into instructional environments in order to provide a better learning experience. However, these tools would be too specific to incorporate in an e-learning platform. Even if they could be provided as pluggable components, the burden of maintaining them would be prohibitive to institutions with few courses in those domains.

This book reflects yet another domain with complex evaluation: computer programming. Introductory programming courses are generally regarded as difficult and often have high failure and dropout rates (Ala-Mutka, 2005; O’Kelly & Gibson, 2006; Robins, Rountree, & Rountree, 2003). Researchers pointed out several causes for these rates (Esteves, Fonseca, Morgado, & Martins, 2010). The most consensual are:

- **Teaching Methods**: Lectures and programming language syntaxes (Lahtinen, Ala-Mutka, & Jarvinen, 2005; Schulte, Carsten, & Bennedsen, 2006);
- **Subject Complexity**: Learning how to program means integrating knowledge of a wide variety of conceptual domains such as computer science and mathematics while developing expertise in
problem understanding, problem solving, unit testing, and others. Additionally, students petered out when they needed to understand and apply abstract programming concepts like control structures or to create algorithms that solve concrete problems (Esteves et al., 2010).

- **Student Motivation:** The public image of a “programmer” as a socially inadequate “nerd” (Jenkins, 2002) and the reputation of programming courses as being extremely difficult negatively affects the motivation of the students (Gomes & Mendes, 2007).

Many educators claim that “learning through practice” is by far the best way to learn computer programming and to engage novice students (Gross & Powers, 2005; Eckerdal, 2009). Practice in this area boils down to solving programming exercises. Nevertheless, solving exercises is only effective if students receive an assessment on their work. An exercise solved wrong will consolidate a false belief, and without feedback, many students will not be able to overcome their difficulties.

Assessment plays a vital role in learning (Ala-Mutka, 2005). However, automatic assessment of exercises other than multiple choice can be a rather complex task. This kind of evaluation differs significantly from evaluations supported by most LMSs, encoded in the IMS Question and Test Interoperability (IMS QTI) specification. The data model of QTI was designed for questions with a set of pre-defined answers and cannot handle evaluation domains with specialized requirements such as the computer programming. For instance, the assessment of programming exercises requires tests cases, program solutions, compilation lines, and other data that cannot be encoded in QTI. Besides the lack of a formal description for programming exercises, the interaction of assessment tools with other systems is not mature enough since there is no communication specifications as stated in several surveys (Leal & Queirós, 2010; Queirós & Leal, 2011).

Automatic assessment in computer programming domains can be applied in two distinct learning contexts: curricular and competitive learning.

Introductory programming courses are part of the curricula of many engineering and sciences programs. These courses rely on programming exercises, assignments, and practical examinations to consolidate knowledge and evaluate students. The enrolment in these courses is usually very high, resulting in a great workload for the faculty and teaching assistants responsible for assessing student programs.

While the concept of “winners and losers” can hinder the motivation of students (Vansteenkiste & Deci, 2003), competitive learning is a learning paradigm that relies on the competitiveness of students to increase their programming skills (Burguillo, 2010; Siddiqui, Khan, & Akhtar, 2008). This is the common goal of several programming contests where students at different levels compete, such as the International Olympiad in Informatics (IOI) for secondary school students, the ACM International Collegiate Programming Contest (ICPC) for university students, and the IEEExtreme for IEEE student members. In this context, several tools are used to allow students to train or participate in programming contests. These tools, such as Programming Contests Management Systems (PCMS) and Online Judges (OJ), rely also on the assessment of programming exercises.

In both scenarios, the manual assessment of programming assignments poses significant demands on the time of teachers (Douce, Livingstone, & Orwell, 2005). Apart from being time-consuming, manual assessment hinders the consistency and accuracy of assessment results as well as allowing unintended biases and a diverse standard of marking schemes (Romli, Sulaiman, & Zamli, 2010). This demand stimulated the development of automated learning and assessment systems in many universities (Ala-Mutka, 2005) as a means for grading the programming exercises of students as well as giving feedback on the quality of their solutions (Tremblay, Guérin, Pons, & Salah, 2008; Spacco, Hovemeyer, & Pugh,
2006). This feedback support is crucial for the computer programming learning (Wang & Wong, 2008; Mory, 2007), especially for first-year students who need to be adequately engaged in order to learn programming (Jena, 2002). Furthermore, immediate feedback motivates students to continue practising (Daly, 1999; Truong, 2007).

Beyond the automatic assessment, another relevant topic in this domain is the availability of programming exercises. It is important that an e-learning system provides a collection of exercises covering a course syllabus and with different levels of difficulty. It has been shown that this can improve the performance of students and their satisfaction levels (Wang & Wong, 2008). Students with lower computer skills can begin by solving easier problems in order to learn progressively and to stay motivated to solve the harder problems later. At the same time, this gives them experience, which is one of the factors that has a large influence on student success in learning programming. In recent years, a large number of programming exercises have been developed and published mostly for use in programming contests. These exercises are generally stored in proprietary systems (e.g. Online Judges) for their own use. Despite some efforts (Queirós & Leal, 2012) to define a common format to describe programming exercises, each of these systems has its own exercise format, making it difficult to share among instructors and students. This poses several issues on the interoperability of the assessment systems with other e-learning systems.

A number of learning tools and environments have been built to assist both teachers and students in introductory programming courses. Rongas, Kaarna, and Kalviainen (2004) established a classification for these tools dividing them into four categories:

1. Integrated development interfaces,
2. Visualization tools,
3. Virtual learning environments, and
4. Systems for submitting, managing, and testing exercises.

To the best of the editor’s knowledge, no e-learning environment described in the literature integrates all these facets (Verdú, Regueras, Verdú, Leal, Castro, & Queirós, 2011; Gomes & Mendes, 2007; Esteves et al., 2010).

Several systems (Jena, 2008; Verdú et al., 2011; Xavier & Coelho, 2011; Guerreiro & Georgouli, 2008) try to address this issue allowing the integration of automatic assessment tools with course management systems, but these approaches rely on ad hoc solutions or proprietary plug-ins rather on widely accepted international specifications for content description and communication among systems.

**DESCRIPTION AND ORGANIZATION OF THE BOOK**

This book presents a comprehensive and recent view of the issues of learning computer programming and shows different pedagogical and technological strategies to address these issues. At the same time, it identifies new trends on this topic from pedagogical strategies to technological approaches.

The book is organized into 12 chapters. A brief description of each of the chapters follows:

Chapter 1 presents and summarizes a variety of research areas that directly or indirectly have influenced Computer Science Education Research, particularly associated to the teaching and learning of programming. The chapter also includes a description of work done at the University of Coimbra, trying to define more adequate pedagogical strategies for introductory programming courses.
Chapter 2 discusses new trends on Computer Science Education, such as the use of open source social networking in education. A comparison with different social learning platforms and new e-learning standards and specifications is made to clarify the benefits of their adoption.

Chapter 3 discusses the effectiveness of one of the various small-group learning methods in maximizing students’ academic achievement in undergraduate computer science classrooms. The findings imply that the present evidence-based research supports the effectiveness of active small-group learning methods in promoting students’ achievement in computer science classrooms.

Chapter 4 contains a comprehensive survey of the teaching and learning through practice of optimization algorithms. In particular, three important issues are reviewed: educational methods, educational software systems, and students’ difficulties. The survey intends to consolidate current knowledge about the education of this class of algorithms for both computer science teachers and computer science education researchers.

Chapter 5 analyzes the grading and evaluation in Massive Open Online Courses (MOOC), especially in science and engineering courses, within the context of education and grading methodologies, and discusses possible perspectives to pursue grading quality in massive e-learning courses.

Chapter 6 shows how a customized simulation game can be used to bring a domain typical for utilization of formal methods, the railway domain, to students and thus motivate them to learn these sophisticated ways of software development. By means of two examples, it demonstrates that such a tool, despite its limited scope, can be used to teach a variety of concepts related to formal methods. It also discusses related approaches to teaching formal methods, describes the customized game and its application in teaching, and evaluates experiences with the application.

Chapter 7 identifies the difficulties that lead students of Language Processing (LP) courses to fail: the level of abstraction associated with some of the basic concepts in the area and the technical capacities required to implement efficient processors, which negatively affect the students’ motivation to learn the main topics. It also discusses the characteristics that a language should have to be a motivating case study.

Chapter 8 presents a sports science approach to address the difficulties of learning computer programming closely related with aspects of personality: self-confidence, resiliency, creativity, and autonomy. The sports science approach emerged naturally as all the authors were involved in high performance training for several years, as coaches, athletes, or psychologists of an elite Women’s Match Racing Team. The chapter presents a case study that took place at the Polytechnic Institute of Setubal with 28 students with different backgrounds and a workload of 8 hours per day.

Chapter 9 presents an e-learning framework defined as a conceptual tool to organize and facilitate technical interoperability among systems and services in domains that use complex evaluation. These domains need a diversity of tools, from the environments where exercises are solved to automatic evaluators providing feedback on the attempts of students, not forgetting the authoring, management, and sequencing of exercises. The Ensemble framework is used on a specific domain, computer programming, and its acceptability is validated on a classroom experiment with encouraging results.

Chapter 10 presents a tool, integrated in the Moodle learning management system, that allows students to train the process of designing relational databases. The tool also allows them to practice with SQL queries that are executed over relational databases previously designed. This chapter also describes the result of a qualitative analysis of the tool’s use in an engineering course offered at the University of Valladolid and focused on the teaching of the Web applications development. The results of the refereed study reveal that the tool was found useful by both students and teachers to support the teaching and learning process of relational databases.
Chapter 11 describes the process of programming a framework with the purpose of achieving real time human gesture recognition. This is already a challenge, but the ultimate goal is to enable new ways of Human-Computer Interaction through expressive gestures and to allow a performer the possibility of controlling (with his gestures), in real time, artistic events.

Chapter 12 provides a conceptual design model for a computer programming learning environment. This environment uses the portal interface design model gathering information from a network of services such as repositories and program evaluators. The design model includes integration with learning management systems, a central piece in the MOOC realm, endowing the model with characteristics such as scalability, collaboration, and interoperability.

CONCLUSION

This book presents innovative teaching strategies and new learning paradigms in the computer programming domain. The main contribution of the book is that it can be summarized as a valuable resource for practitioners and as a reference for research scholars and computer science teachers and students pursuing computer science-related subjects.

This book clearly impacts the field and contributes with new trends to foster computer science education. In a world where technological advances appear at a dizzying pace, all computer science educators must keep up with these changes and adapt teaching methodologies in order to enhance the study of computer programming and motivate students to achieve this complex but exciting and emerging area.

Ricardo Queirós
Polytechnic Institute of Porto, Portugal

REFERENCES


