Computers & Education 60 (2013) 368–384

Contents lists available at SciVerse ScienceDirect

Computers & Education

journal homepage: www.elsevier.com/locate/compedu

The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism

Jan Moons*, Carlos De Backer

University of Antwerp, Prinsstraat 13, 2000 Antwerp, Belgium

ARTICLE INFO

Article history:
Received 12 April 2012
Received in revised form
20 August 2012
Accepted 21 August 2012

Keywords:
Architectures for educational technology system
Interactive learning environments
Programming and programming languages

ABSTRACT

This article presents the architecture and evaluation of a novel environment for programming education. The design of this programming environment, and the way it is used in class, is based on the findings of constructivist and cognitivist learning paradigms. The environment is evaluated based on qualitative student and teacher evaluations and experiments performed over a three year timespan. As the findings show, the students and teachers see the environment and the way it is used as an invaluable part of their education, and the experiments show that the environment can help with understanding programming concepts that most students consider very difficult.

1. Introduction

This article describes the design and evaluation of a novel programming education environment. To position this environment in this introductory paragraph, we need to take a look at the reasons for developing such an environment and provide a clear overview of the alternatives already available in the field. To this end, we first describe the difficulty associated with the subject matter of introductory programming. Next, we present a concise overview of alternative approaches. Finally we present our rationale for developing a new environment and we introduce the guidelines that have influenced our design.

1.1. Issues with introductory programming courses

Introductory programming courses are considered to be very difficult by many students, often resulting in low retention rates (Baldwin & Kuljis, 2000). This level of difficulty has been acknowledged in literature since many decades (Mayer, 1981). More recently, two large scale research efforts report on these difficulties. The first report is the result of a broad international study under the lead of Michael McCracken entitled "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students" (McCracken et al., 2001). In this report, the authors test the programming competency of students after they have completed their first one or two programming courses. In total, 217 students from four universities took the test. The solutions provided by the students were scored on several criteria, which were aggregated in a general evaluation score. The average general evaluation score for all students, for all exercises, across all schools was 22.9 out of 110.

The second report concerns a multi-national study under the lead of Raymond Lister, entitled "A multi-national study of reading and tracing skills in novice programmers" (Lister et al., 2004). This study was created to check the hypothesis that beginning students lack the important routine programming skill of tracing (or "desk checking") through code. Over 600 students from seven countries were tested on twelve multiple choice questions of two types. The first type required the students to predict the outcome of a piece of code. The second type required students to choose the correct completion of a piece of code from a small set of possibilities. The results were unexpectedly poor – over a quarter of the students failed the most basic tests.

* Corresponding author.
E-mail address: jan.moons@ua.ac.be (J. Moons).

0360-1315/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compedu.2012.08.009

This problem is caused in part by the inherent difficulty of the programming task. Students have to learn how to interpret and work with many new, abstract and interdependent concepts, that have a static as well as a dynamic component. Programming is called a "wicked problem" (Degrace & Stahl, 1998), that is often too big, too ill-defined, and too complex for easy comprehension and solution (Petre & Quincey, 2006). This level of inherent complexity is widely recognized in literature (Jeffries, Turner, Polson, & Atwood, 1981; Kim & Lerch, 1997). Programming demands complex cognitive skills such as procedural and conditional reasoning, planning, and analogical reasoning (Kurland, Pea, Clement, & Mawby, 1986).

Of course, besides the inherent difficulty of the subject, the problem could also be caused in part by an incorrect way of teaching this subject. During the past decades researchers have been finding ways to improve the performance of students of a first programming course (often called CS1 course) by making changes to the way they teach the subject. The next paragraph provides a concise overview of these proposed solutions.

1.2. Proposed solutions

Instructors have approached this problem in four ways. The first is through the use of a certain programming education methodology. This manifests itself mostly in the use of a certain order for introducing various subjects. Examples are the objects-first and the procedures-first approach. The choice between these methodologies is the topic of one of the more heated debates in computer science education (Astrachan, Bruce, Koffman, Kölling, & Reges, 2005; Bailie, Courtney, Murray, Schiaffino, & Tuohy, 2003; Bergin, Eckstein, Wallingford, & Manns, 2001).

The second is by employing various constructivist-inspired active learning techniques, such as role-play, active story-telling and workshops. A comprehensive overview of these techniques can be found in Bergin, Eckstein, Manns et al. (2001), Bergin, Eckstein, Wallingford et al. (2001). As an example, in Jiménez-Diáz, Gómez-Albarán, Gómez-Martín, and González-Calero (2005), the authors present a tool that enables role-play. The goal of the tool is to enable students to understand the message passing mechanism. This is important because of the difficulties students face in understanding the dynamic aspects of software (Ragonis & Ben-Ari, 2005). In their environment, users get to play an object at runtime, and can react to messages being passed around. Method call, method return and object creation are represented by throwing balls around between objects containing the parameters. These active learning techniques can also be applied without the help of software tools, as is demonstrated in Andrianoff and Levine (2002), where they use classroom role-playing to introduce message passing.

The third approach is to use a software language that is tailored to novice programmers. Currently, the most popular languages in industry are Java, C and C++. However, as stated by Pears et al. (2007), there is much debate as to their suitability for a CS1 course. A better alternative might be to use a language specifically designed for education. A famous example is Pascal, whose history is presented in Wirth (1996). For a full description of the virtues of different languages for education, one can consult (Mannila & Raadt, 2006). In their conclusions of a comparison of eleven languages used in CS1 courses, the authors suggest that the most suitable languages for teaching, Python and Eiffel, are languages that have been designed with teaching in mind. For a history of programming languages used in CS1 courses, and the rationale for choosing these languages, the reader is referred to Giangrande (2007).

The fourth approach consists of the use of software environments, of which there are three types. The first type consists of microworlds. Microworlds are software applications that take the form of graphical environments in which the programmer can move an object, such as a robot or a kangaroo, on a canvas using a predetermined set of commands. The concept of a microworld was developed by Seymour Papert, who pioneered the genre with the LOGO language and the LOGO turtles (Papert, 1985). The LOGO environment has given rise to an entire family of microworlds. Well-known examples are Karel the Robot (Pattis, 1995), Jeroo (Sanders & Dorn, 2003a,b) and Alice (Cooper, Dann, & Pausch, 2003). Also, real-world environments such as the Lego Mindstorms Kit (Holmboe, Borge, & Granerud, 2004) could be considered microworlds, as they enable the manipulation of objects through a simplified programming language.

The second type of software environment consists of algorithm visualization tools. The goal of these tools is to visualize what happens during the runtime of algorithms, such as searching or sorting algorithms – of varying complexity. Notable examples are Tango, Polka and Samba (Stasko, 1990), Animal (Rössling, Schüer, & Freisleben, 2000), Jawaa (Rodger, 2002), Jhavé (Naps, Eagan, & Norton, 2000), MatrixPro and Trakla (Korhonen, 2003; Korhonen, Malmi, & Saikkonen, 2001), and Alvis Live! (Hundhausen & Brown, 2008).

The third type of software environment consists of program visualization tools, often integrated into a custom educational IDE. The goal of these tools is to present the compile-time or runtime structure of a program. In the case of object-oriented programs, this means the presentation of the class diagram or of the object diagram. Some tools put more emphasis on the IDE part, such as DrJava (Allen, Cartwright, & Stoler, 2002), BlueJ (Kölling, Quig, Patterson, & Rosenberg, 2003), ProfessorJ (Gray & Flatt, 2003) and JGrasp (Cross & Hendrix, 2006). These software applications provide a complete working environment for the novice programmer, often adding features to simplify programming, such as automatic method generation. They also sometimes provide visualization features to improve students' understanding of the programming concepts.

Other program visualization tools emphasize the visualization over the IDE. Notable examples of program visualization systems are JIVE (Gestwicki & Jayaraman, 2004), JEliot 3 (Moreno & Joy, 2007) and Ville (Rajala, Laakso, Kaila, & Salakoski, 2008). These tools all show a visual presentation of the basic constructs of the Java programming language. The HDPV system of Sundararaman and Back (2008) is more difficult to classify. It possesses characteristics of program visualization as well as algorithm visualization.

1.3. Contribution

It is in this last category that this paper makes a contribution. In our search for a tool to help our teachers and students during the introductory programming course we have evaluated many of the available educational environments mentioned above. Although many of them provided valuable assistance, none covered the entire set of concepts introduced in the introductory programming course, nor did these systems seem to take into account the valuable lessons that can be learnt from educational theory (constructivism, cognitive science, …) when designing educational environments. For instance, very few tools actually employ sound visual design principles. Color is seldom