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

used, motion is often overlooked, objects can sometimes not be repositioned, there are often too few ways of interacting with the environment etc.

Thus we have set out to design an environment that fills this gap. Our research project has resulted in a programming visualization environment that provides students with a live view of the code as it executes, emphasizing the dynamic properties of programming concepts, which are arguably the most difficult properties to grasp. It does so by showing a visual representation of concepts such as classes, objects, method frames, variables, etc. on canvas right inside their Integrated Development Environment. This visual representation integrated in the IDE makes for a flexible demonstration tool for the teacher, and provides a smooth and rapid program, view, adjust cycle for the student.

In the following sections we expand on the driving principles behind our design. Specifically, we discuss the following principles:

- The environment should be able to show the relevant programming concepts taught in a typical object-oriented introductory programming course.
- The environment should follow proven principles of constructivist and constructionist learning theories – and allow for the active manipulation by students and teachers.
- The environment should follow the principles of cognitive science, such as the principles described in the cognitive load theory, in order to reduce load on working memory.
- The environment should keep in mind the basic notions of human visual perception in the design of the visualization.

Each item of this list will be explained in further detail in the following paragraphs.

2. Guiding principles for a program visualization environment

2.1. Relevant concepts

First and foremost, it is important that a program visualization tool can actually present most relevant programming concepts taught in an introductory programming course, making sure to include the concepts that are considered difficult by the students. Although every introductory course will be a little different, and it is impossible to find one perfect set satisfying everyone's needs, we have aimed at finding a relevant set of concepts in six ways. First, we have consulted teachers at a Belgian university. Second, we have consulted publicly available on-line programs of other universities to get an overview of the concepts that they currently teach in their introductory programming courses. Third, we have consulted available literature listing the most important constructs according to computer science teachers (Schulte & Bennedsen, 2006). Fourth, we have constructed our own survey asking the students to indicate how difficult they considered each of the concepts and tasks taught in our introductory programming course. They had to score these concepts and tasks on a 7 point Likert scale. This approach has been validated before (Marcus, Cooper, & Sweller, 1996) and has been used e.g. by Carlson, Chandler, and Sweller (2003). Our assumption was that the more difficult concepts should definitely be supported in the program visualization. Fifth, we have consulted other studies and surveys that also investigate the difficulty of programming concepts (Garner, Haden, & Robins, 2005; Lahtinen, Ala-Mutka, & Järvinen, 2005; Milne & Rowe, 2002; Robins, Haden, & Garner, 2006). Finally, we have consulted studies investigating the frequency of errors and error types in introductory programming courses (Ahmadzadeh, Elliman, & Higgins, 2005; Jadud, 2006; Thompson, 2004). Based on these sources, Table 1 presents the list of constructs that are supported by the environment. The list presents the various concepts grouped in larger subject domains. The abbreviations next to the subject domains indicate whether they feature in the consulted introductory programming course programs. The columns refer to the sources containing a ranking of the difficulties of these subjects. The cells contain the ranking for a particular concept in a particular article.

2.2. Constructivism and constructionism

The second principle states that the environment should follow the principles advocated by the constructivism learning paradigm. Constructivism is based on the notion that "the most personal of what [a man] knows is that which he has discovered for himself" (Bruner, 1979). The concept of learning by doing has found its way into computer science and engineering education since many years. For instance, the constructivist learning style of engineering students is described in Felder and Silverman (1988). The authors conclude that "most engineering students are visual, sensing, inductive, and active. [...] Most engineering education is auditory, abstract, deductive, passive, and sequential. These mismatches lead to poor student performance, professorial frustration, and a loss to society of many potentially excellent engineers."

One particular branch of constructivism, constructionism, was pioneered by Seymour Papert, the inventor of the educational LOGO microworld (Papert, 1980, 1985; Rubinstein, 1975). In contrast to constructivism, constructionism puts more emphasis on the creation of an artifact to improve learning.

The influence of constructivism and constructionism is clear when looking at various proposed solutions. Microworlds let novice programmers experience the consequences of their own programming actions. Active learning techniques such as story-telling and role-play are directly advocated by constructivist literature, and algorithm and program visualization environments often allow active manipulation by the students.

For more information on the constructivist paradigm we refer to Bruner (1979), Glaserfeld (1987), Piaget (1970), Vygotsky (1978).

2.3. Cognitive load theory

The third principle states that the environment should follow the principles of cognitive science, such as the principles described in the cognitive load theory (CLT), in order to reduce load on working memory. CLT is a learning theory based on cognitive science, which investigates human cognitive architecture. Most cognitive scientist now accept the basic model of cognitive architecture as

Table 1
A list of concepts to be supported by the program visualization environment.

MIT: Massachusetts Institute of Technology GAT: Georgia Institute of Technology COR: Cornell University CAM: Cambridge University UA: University of Antwerp	Student survey at the University of Antwerp	(Schulte & Bennedsen, 2006)	(Milne & Rowe, 2002)	(Garner et al., 2005)	(Lahtinen et al., 2005)
Variables (MIT, GAT, COR, CAM, UA)					
Variable types	14,35	8	28		
Variable scope	13	4	20	10	10
Assignment of values to variables	32				10
Selection and repetition structures (MIT, GAT, COR, CAM, UA)					
If statement	27	1	36		12
Switch statement	11	1	26		12
While loop	20	1	24	7	11
Do loop	17	1	24	7	11
For loop	25	1	24	7	11
Methods (MIT, GAT, COR, CAM, UA)					
Method structure	22,30	10			
Method frames	22,30	3	13,27	6	9
Message passing	22,30	3	13,27	6	9
Return values	24		31	6	
Recursion	7	23	5		3
Classes and objects (MIT, GAT, COR, CAM, UA)					
Class structure		5,7	15		
Object structure			15		
Object communication	23,28,29		15		
Reference variables and values	8	19	1,13		1
Inheritance		21	14		
Encapsulation	16	14	16		
Static versus instance variables	12	20			
Arrays (MIT, GAT, COR, UA)					
Arrays as objects		2	19	5	8
Array structure		2	19	5	8
Arrays of primitives	18	2	19	5	8
Arrays of references	5	2	19	5	8
Array index	21	2	19	5	8
Algorithms (GAT, UA)					
Search algorithms	9	12			
Sort algorithms	3	12			
Data structures (linked list and binary tree)	1,2	21			
Additional requirements (MIT, GAT, COR, CAM, UA)					
Debugging	6,10,15	11			
Standard libraries		18			4

proposed by Atkinson and Shiffrin (1968). Atkinson divides memory into three stores – the sensory register, the short-term store and the long-term store. Information from the environment enters the mind through the sensory register. The sensory register has an extremely limited decay period, i.e. the time it takes for registered data to disappear from the register. Atkinson mentions a decay period for visual information of several hundred milliseconds. The short-term store is also referred to as "working memory". All conscious processing happens in working memory. When referring to cognitive load, researchers refer to the load on working memory. Working memory also has a limited decay period. Atkinson mentions a decay period for items registered in the audio-verbal-linguistic store of 15–30 s. The final component is long-term memory, which serves as a virtually inexhaustible and permanent store of information.

Most research on human cognitive architecture is focused on working memory, long-term memory and the interaction between them. Working memory is limited in two ways. As we already mentioned, information in working memory is only retained for a few seconds. Working memory is also constrained in capacity. In Miller (1956), George Miller determines that the number of items that can be held simultaneously in working memory is seven, plus or minus two, depending on circumstances. Other evidence suggests an even more limited capacity of four plus or minus one (Cowan, 2001). The cognitive model was later expanded by Baddeley and Hitch in Baddeley and Hitch (1974). In their work, the authors demonstrate the existence of different types of working memory that operate more or less independently. The visuo-spatial sketchpad deals with visual information, the phonological loop deals with verbal information and the central executive is the coordinating processor. This expansion implies an extension of working memory capacity when offering information through multiple modes.

This research stream has resulted in various theories. The dual coding theory (Paivio, 1986) and the multimedia theory (Mayer, 2001) state that learning can be improved by offering information in visual as well as textual modes. The cognitive load theory describes various effects that influence the load on limited working memory. This research has resulted in various guidelines. For instance, the split-attention effect states that working memory load can be reduced by physically integrating visual diagrams and textual statements (Sweller, 2002), or more generally, that working memory load can be reduced if students are not required to split their attention between two physically separated representations of information.

For novice programmers, cognitive load surely is high. The problem is illustrated in Bailie et al. (2003): "from the first line of a Java program you know we are in serious trouble: public static void main(String[ ] args). We have visibility modifiers, return types, method

names, a class, parameters and arrays and we haven't started the program." Many similar languages face the same problem – even for writing very simple working programs, students need to master a comparatively large amount of new concepts and techniques.

The statement is echoed by Xinogalos, Satratzemi, and Dagdilelis (2006), who state that one of the most significant difficulties that beginner students face is the extended instruction set of programming languages. The problem is not just that there are a whole host of concepts to be learned, there is also a high level of element interactivity, which means that the elements depend on each other. For instance, in order to implement a method, one must also know about parameters, return values, access modifiers etc. This means that many elements must be present in working memory at once, increasing intrinsic cognitive load.

Even though these findings are very relevant to programming education, very few educational software projects take these findings into account explicitly.

2.4. Visual perception

There are good reasons for choosing a visual program representation to help novice programmers. According to Ware (2004), "visual displays provide the highest bandwidth channel from the computer to the human. We acquire more information through vision than through all of the other senses combined. The 20 billion or so neurons of the brain devoted to analyzing visual information provide a pattern-finding mechanism that is a fundamental component in much of our cognitive activity". Indeed the visual capacity of our brain is enormous. Even so, many experiments have indicated that certain ways of presenting visual information are better than other ways.

One interesting feature of a human visual system is its capability of preattentive processing (Treisman, 1982), the processing of information before we consciously pay attention to it. Pre-attentive processing determines what features of a visual presentation will catch our attention. Using these pre-attentive features can improve the efficiency of visual searches and thus lower cognitive load. The following features are all known to impact pre-attentive processing (Wolfe, 2000): color (hue, lightness, colorfulness), orientation, curvature, size, motion, depth cues (3D effects), Vernier offsets (relative positioning), luster (shininess) and shape (or at least certain aspects of shape such as closure, intersections and terminators). Using these properties to indicate interesting areas in the computer visualization will lead users to focus more easily on these areas and to become less distracted by other areas, in turn leading to a lower load on working memory. All of these features have been verified empirically. For instance, the use of color in human computer interaction environments was studied in Michalski and Grobelny (2008). The next paragraphs highlight some of the more important perceptual characteristics that can be of use in a programming environment.

2.4.1. The Gestalt principles

The "Gestalt" principles or laws were developed by three German scientists in the beginning of the twentieth century (see e.g. Koffka (1935) for an original text by the group). "Gestalt" means pattern in German, so the "Gestalt" principles deal with the way we perceive patterns. There are eight principal "Gestalt" laws, dealing with proximity, similarity, connectedness, continuity, symmetry, closure, relative size and common fate. More in particular, these laws deal with the way certain features (or patterns of features) lead us to infer relations and connections. The "Gestalt" laws are very useful as input into design directives for efficient visual presentations, because they are based on our human perceptual characteristics. In practice this means that they work automatically and without effort from our working memory.

The Gestalt principles can provide valuable input for the design of program visualization. For instance, the "continuity principle" indicates that continuous lines are to be preferred to sharp-cornered lines in a program visualization diagram. The "proximity principle" indicates that objects of similar types should be presented closely together. The "similarity principle" could for instance be used by giving similar constructs similar shapes and different constructs different shapes. The "connectedness principle" can be used to indicate relations. The qualities of connectedness were not originally described by the "Gestalt" scientists, but were added by Palmer and Rock (1994).

2.4.2. Color

A lot of research in the field of visualization has focused on the properties of colors and their value in different types of visualizations. For instance, it has been discovered that the value of color lies primarily in the recognition of attributes or types of objects, not in the mere recognition of the presence of an object. In Tufte (1990) four different uses for color are described – to label (color as noun), to measure (color as quantity), to represent or imitate reality (color as representation), and to enliven or decorate (color as beauty). So, in the case of program visualization software, the goal of color might be to label different types of objects.

Although the value of color in data visualization is widely recognized, it must be applied with certain care. Tufte mentions negative returns when more than 20 to 30 colors are used. The difficulty with applying color has enticed Tufte to state as the primary guideline for the use of color in information displays: "Above all, do no harm". Healey, Booth, and Enns (1996) has found that no more than five to ten color codes can be easily distinguished at any one time. If we stay below this boundary, the use of color for labeling is beneficial.

The value of color in IT architecture diagrams has been acknowledged in literature before. For instance, in Koning, Dormann, and Vliet (2002), the authors present a list of 163 guidelines that experienced architects apply when designing IT architecture diagrams. Of these 163 guidelines, 33 deal with the use of color in diagrams.

Based on elaborate research, Ware (2004) has presented a list of colors that can safely be used for labeling of information on a computer display (see Table 2). The colors mentioned in the list have widely agreed-upon names and are easy to recognize visually because they are far apart in the color space. This list of colors can be used in the design of the visual presentation of programming concepts.

Table 2
Safe colors to use on a computer display.

Red	Green	Yellow	Blue
Black	White	Pink	Cyan
Gray	Orange	Brown	Purple

2.4.3. Motion

In Peterson and Dugas (1972), the role of motion in visual target detection was investigated. One of the conclusions of the report was that the useful field of view for detecting moving targets was much bigger than the field of view for detecting static targets. In their experiment, if a user is fixating on a point on a display, the target would have to appear in a 6–7° radius around this fixation point in order to be detected. Moving targets on the other hand can be detected in the entire field of view.

The experiments presented in Bartram, Ware, and Calvert (2003) also resulted in several conclusions regarding the use of motion. First, in the experiment, the use of color and shape is inferior in all situations and under all circumstances to the use of motion. Second, the use of motion results in low detection error rates and low mean response times. Finally, there was no significant difference between the detection of near and far icon changes for the motion cues, while there was a significant difference for the color and shape cues (changes in these features were detected significantly later if they were outside the focus point). The results of these studies are useful in the design of program visualization tools. For instance, motion can be used to draw attention to important parts of the program visualization canvas for complicated programs.

3. Design guidelines for the program visualization environment

This section translates the general high-level principles described in Section 2 into design guidelines specifically for program visualization environments.

3.1. Make the design interactive to improve learning

The program visualization environment should allow students to build their own real programs, and allow them to evaluate their work in a quick "create, test, improve" cycle. This interactivity is a requirement based on the constructivism learning paradigm, and has been described as a key feature of well-designed visualization systems by Naps et al. (2002). Interactivity can be applied to program visualization in the following ways:

- First and foremost, program visualization environments should allow the creation of custom programs by the students. Considering the taxonomy by Naps, such a program visualization environment could be attributed the level of "creation", which is the next-to-highest level of interactivity. Every student in fact "creates" its own visualization based on the code he writes. This is in contrast to many algorithm visualization software environments, where either a pre-rendered animation of an algorithm is presented, or the user can change the algorithm only by changing the input values.
- One could improve interactivity by integrating on-demand explanations of concepts. In this way, when the student is not yet familiar with something he or she sees in the visual presentation, or something unexpected occurs, he could interact with the visualization by asking more information about the concept.
- To elaborate on the previous guideline, one should also include pop-up quizzes during program execution (e.g. asking for the value of a variable at certain points in time.). This way, not only can the student see his program evolve visually, but he or she is also encouraged to truly understand the visual presentation.
- An important part of the interaction concerns the controls that can be used to traverse the animation of the running program. Most program visualization tools use some sort of transport controls, akin to the transport controls on most CD or DVD players. In other words, the user can step through the program (next button) or proceed through the program automatically (play button). To improve this basic transport schema, one could also provide a back button, allowing students to undo and re-execute certain program execution lines, leading to a better understanding of the causal relations between the code and the actions caused by the code. One could also include a transport control that advances to the next breakpoint. This way, when the student or the teacher wants to inspect a particular part of the program, all irrelevant parts can be skipped, reducing cognitive load.
- A program visualization tool should allow the user to reposition objects on the scene. A running object-oriented program can become so complex that currently available automated positioning algorithms are likely to achieve sub-optimal results. In addition, some examples, such as a linked list or a binary tree, might benefit from a very specific positioning of the objects to improve comprehensibility.
- The user should be able to use the canvas of the visualization in a flexible manner, just like the canvas of a traditional drawing program. Among others this means that the user should be able to zoom and pan when required. An overview window might help with locating objects in a more elaborate example where it is impossible to fit all objects on the screen at the same time.

3.2. Use the guidelines discovered by cognitive science

The cognitive load theory, based on cognitive science, has discovered many guidelines that are beneficial to the design of instructional material in general. Some of these can also be applied directly to program visualization design:

- A program visualization environment should place related concepts (of similar or different media, such as pictures and explanatory text) closely together. Mayer (2001) has uncovered that students perform better […] when corresponding words and pictures are presented near rather than far from each other and that […] students learn better when corresponding words and pictures are presented simultaneously rather than successively". This guideline can be translated to a program visualization environment. For instance, this guideline implies that the method text and method frame visuals should be placed closely together. In a similar manner, exception objects and exception explanations should be presented visually close together. Finally, most program visualization tools show either the compile-time structure (the class diagram) or the runtime structure. None of the program visualization environments show the two

structures side by side. Because the difference between these two types of structures is difficult for student to comprehend, we should place them visually close to each other.
- The CLT guidelines also indicate that information should be kept concise, and irrelevant information should be hidden. Most program visualization tools use lines and arrows for presenting relations in a visual presentation. However, there are many types of relations in a running program – the relations in an inheritance hierarchy, the relations between reference variables and objects, the relations between subsequent methods on the method stack and the relations between objects and their respective classes. Presenting all these relations at the same time on a canvas can quickly lead to information overload. Object-oriented languages are especially complex in their relations. Therefore, it is important to let the user choose which relations are relevant at any given time, and present only these relations. For the same reason, sometimes it might be beneficial to hide the method code from the method visuals. For instance, when the visual presentation presents a long running sorting algorithm, only the code of the algorithm itself will be relevant. In this case, an option that only presents the code of the active method frame will help reduce irrelevant information.
- The CLT guidelines also indicate that relevant information should be highlighted. During execution of a program, every method frame will have exactly one code line that is being executed. If more than one method frame is present in the program, these "active" code lines represent the "trace" of the program. It is important to clearly highlight this trace to the student, as this trace is essential to comprehend the concept of the method stack.

3.3. Use the characteristics of human perception in the design of the visualization

Section 2.4 introduced the importance of various visual properties of a visual presentation. Based on the multitude of experiments indicating the dramatic effects of features such as location, color and motion, it is surprising how little they are used in current program visualization designs. These are some design guidelines that can be used to present a more comprehensible visualization to students:

- The primary role of dynamic program visualization software is to show the user the constructs that are created during program execution. These constructs can have many relations to each other. The "Gestalt" principles have indicated that node-link diagrams are very good at presenting such information.
- Program visualization tools should make correct use of colors. The colors that were presented in the previous chapters should be the first choice, with no more than 6 colors used for different types of concepts. One could use different colors e.g. for objects, arrays, static structures and compile-time structures, and use small variations in the color used for objects to denote objects of different types.
- Different shapes can be used to distinguish between different types of concepts. For instance, one can use rounded rectangles for presenting objects, and straight-cornered rectangles for presenting classes.
- The previous section argued that motion is a very good type of cue for drawing attention to information. The nature of program visualization software is such that at every execution of a code line many things can happen on the canvas. The addition of new structures, such as an object or a method frame, is generally fairly easy to spot, but much more difficult to spot are changes in the values of variables. Good program visualization software design should therefore always use motion such as "flickering" to indicate when a variable has changed value. Motion can also be used to transport parameter values from calling method frames to called method frames and return values from called method frames to calling method frames to illustrate method passing.

4. Use and design of the environment

The previous sections explained the background, the rationale and the design guidelines on which the novel environment is based. This section will first describe the use of the environment, so the reader can accurately evaluate the usefulness of the design. Next, we present the general architecture of the environment and the visual presentation of the programming concepts.

4.1. Use of the environment

The environment is used in three ways in an introductory programming course focusing on object-oriented programming concepts. First, the environment is used throughout the entire course as a teaching aid for introducing the various subjects. The environment allows the teacher to introduce many concepts that would be impossible to present graphically using a simple whiteboard. A whiteboard for instance has no options for going back and forth through a program execution run. This is a serious drawback since the most difficult aspects of programming concepts are the dynamic aspects. Second, students use this environment to inspect exercises prepared by the teacher as well as their own endeavors in class and at home. This enables the students to immediately assess the changes they have made to their code in a visual presentation. Seeing the concepts presented visually time and time again also helps in instilling the correct mental picture of the concepts. Third, the students are required to prepare a presentation each week using the visualization environment. This way, they are able to present a new solution each week to their peers, often leading to discussions based on the visualization presented on the screen. This presentation exercise makes sure that the students use the visualization during the week, and having to explain the concepts to their peers makes sure that they understand them through and through. These three ways of using the environment are all equally valuable. The first way helps the teacher explain the concepts. The second way helps to create the correct mental pictures of the concepts for the students. And the third way forces the students to use the environment at home and to really grasp the presented concepts in order to present them to their peers.

4.2. Architecture

Since none of the evaluations by students and teachers indicate any difficulty in using the actual source code editor, we saw no reason to reinvent the IDE itself. Therefore, the EVizor (Educational Visualization of the Object Oriented Run-time) environment is implemented as

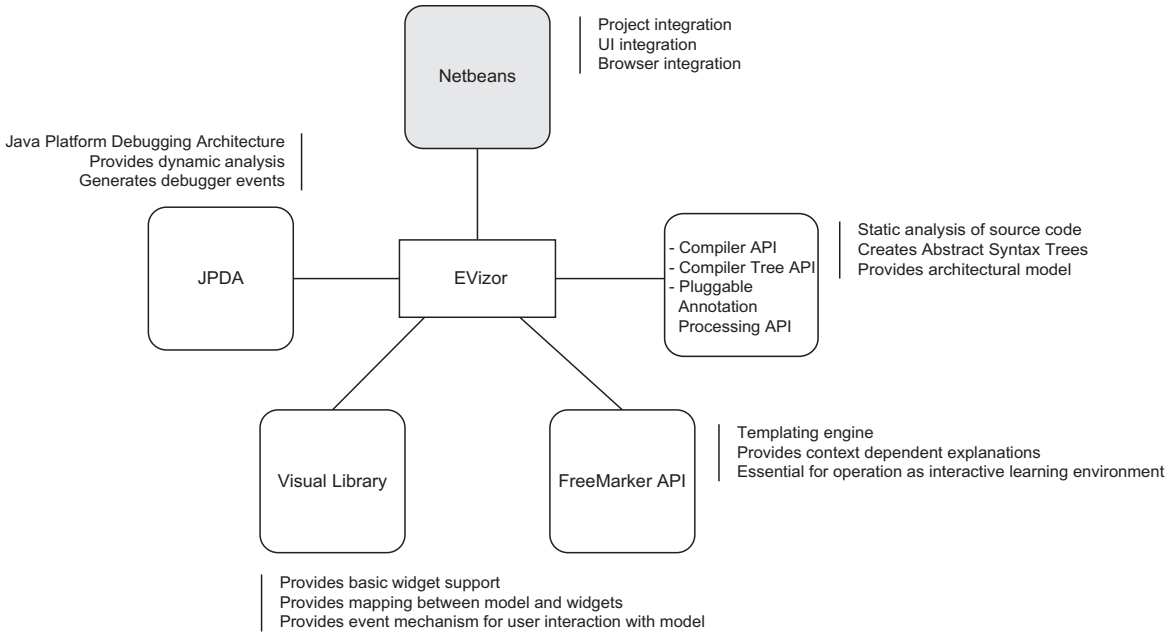

Fig. 1. The EVizor components.

a plugin for an existing IDE (Netbeans). Fig. 1 presents the components used in the environment. When the student or teacher creates a program, EVizor first uses the various compiler APIs to generate a model of the compile-time structure in memory. Next, during program execution, a debugging architecture (JPDA) is used to capture program events and record them in a stack. This stack can be traversed in two ways, so that the user can advance or rewind the program. After each event, the program is stopped, so the student can examine the result of the code lines on the visual display. At each step, the visualization is updated with new values, object, classes, method frames, through the use of the Netbeans Visual Library. The FreeMarker API is used to generate pop-up quizzes and information panes using content from an online database. This way, the teacher can update quizzes and information panes easily, and all students can receive these updates in real time.

4.3. Design of the programming concepts

While it would lead too far to present the design of all of the concepts in detail, this section will present a selection of the design features of various programming concepts.

Fig. 2 presents the animation that occurs when a variable changes value in three subsequent frames (in the visualization software these frames are presented in one fluid motion). Of course, in the actual implementation, this animation is smooth. The animation makes use of motion flicker to attract students' attention to the changes that occur in the animation. In addition, the active method line is highlighted in the method text. This method text can be hidden or displayed on command.

Fig. 3 presents the representation of arrays and reference variables. Arrays are represented in a different color from regular objects, considering their special status as the most basic data structure in most object oriented programming languages. Since arrays are objects, these objects have an object ID, which is represented above the object. The reference variable "ints" holds this ID as its value. Arrows can be presented or hidden to connect these reference variables to their corresponding objects. In addition, reference variables are presented in the same color as the type of object they refer to.

Fig. 4 presents the presentation of compile-time and runtime structures of the program on the same canvas. This presentation indicates to the students the important difference between the structure, and shows for instance clearly that many objects can be generated based on the same class.

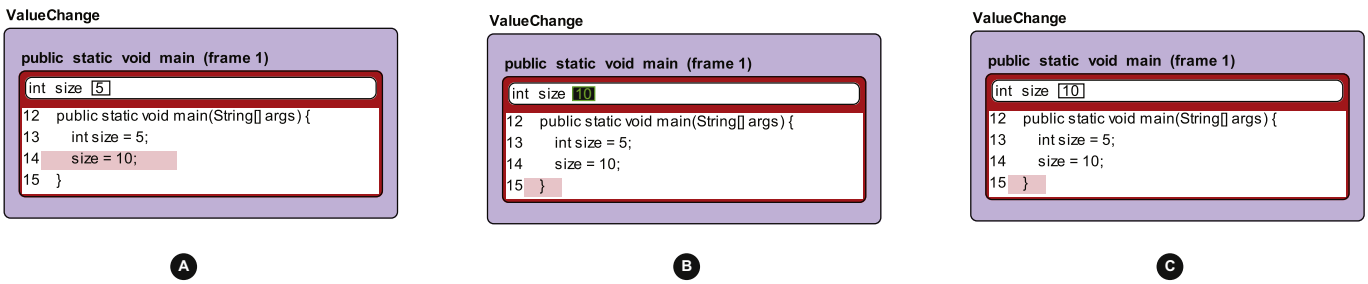

Fig. 2. In this example, the local variable 'size' is declared and initialized with a value of 5. Next, the value 10 is assigned to the variable. The visualization shows which variable is updated using motion (flicker), and indicates that the previous value is overwritten.

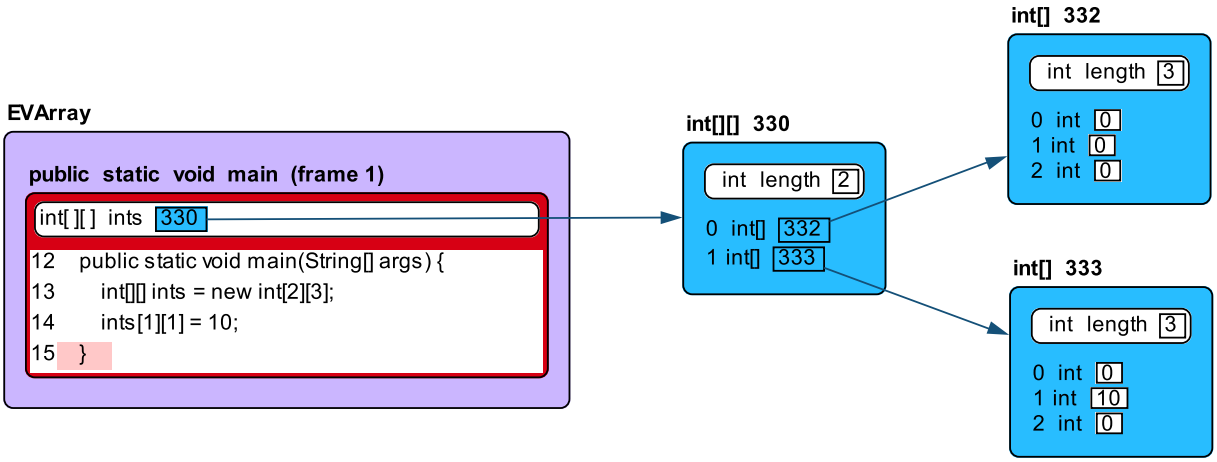

Fig. 3. In this example a two-dimensional array variable is declared and instantiated. The presentation shows how multi-dimensional arrays are instantiated in memory. Next, the value 10 is assigned to one of the slots of the multi-dimensional array.

Fig. 5 also presents the integration of the runtime structure and the compile-time structure. However, in this complex example, the difference between the two types of structures becomes more obvious. In addition, the zoom and pan features are a necessity to allow for the easy presentation of such complex programs by the teacher.

Fig. 6 presents the visualization of a recursive method, one of the more complex concepts for students to grasp. The recursion example is also used as the subject of the experiment presented in Section 6. The students can clearly see the various method frames of the same type in the visualization. In addition, the currently executing method frame is presented in a red color, and the active and waiting code lines of all method frames are highlighted to clearly indicate the stack. As a side note, one must never forget that even clearly presented visualizations benefits from clear explanations by a teacher. Certainly this more complex example of the recursive method call would benefit from clear explanations in a classroom setting.

Finally, Fig. 7 presents the visualization of an exception. These exceptions are linked to the exact line generating the exception, and an explanation is provided by the environment. In addition, the student can visit various on-line sources explaining the event, such as the university website, the Java documentation and Wikipedia. The same mechanism can be used on all object types.

In addition to these examples; the visualization environment has many more features. One the one hand, there are many more concepts that can be visualized, such as inheritance and static versus instance objects. On the other hand, there are interactive features such as pop-up quizzes that are downloaded at run-time from a website containing a database of prepared exercises with pop-up quizzes and information panels.

5. Qualitative pilot evaluation

In order to assess the environment qualitatively, we have recorded the evaluations of students during three consecutive years of using the environment.

The students providing the evaluation ($N = 68$) are in their second bachelor year of the Management Information Systems curriculum at the Faculty of Applied Economics of the University of Antwerp, Belgium. The survey was administered during the sixth lesson of the course "Programming and data structures". The students had been using the visualization since their first lesson, so students had five lessons of experience with the component. In total sixty-eight students filled out the survey. The scale used for the questions was a seven point Likert scale, with labels strongly disagree (1), disagree (2), somewhat disagree (3), neither agree nor disagree (4), somewhat agree (5), agree (6),

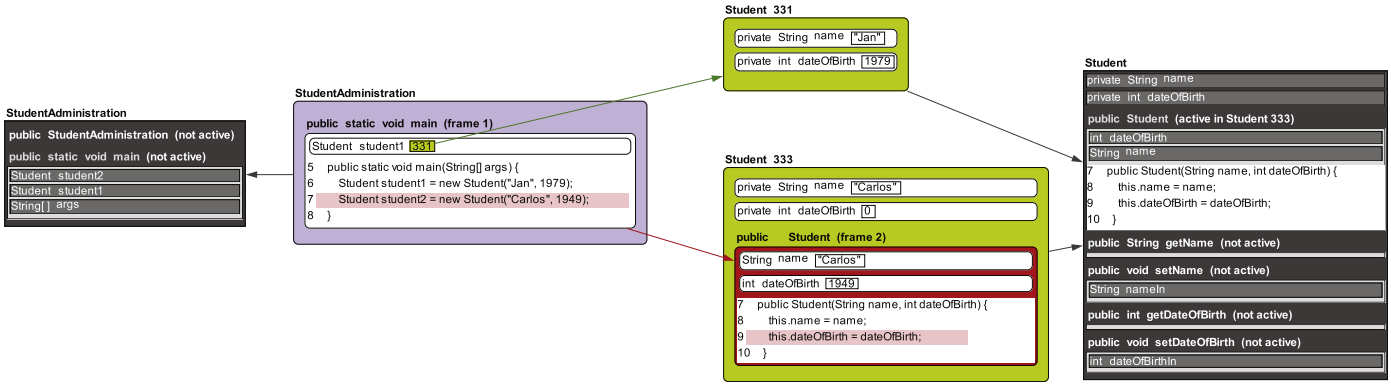

Fig. 4. This example show the compile-time structure together with the run-time structure. Both structures can be hidden and shown on demand. In more complex examples, it is easy to move the relevant compile-time structures next to the relevant run-time structures using mouse-over gestures.

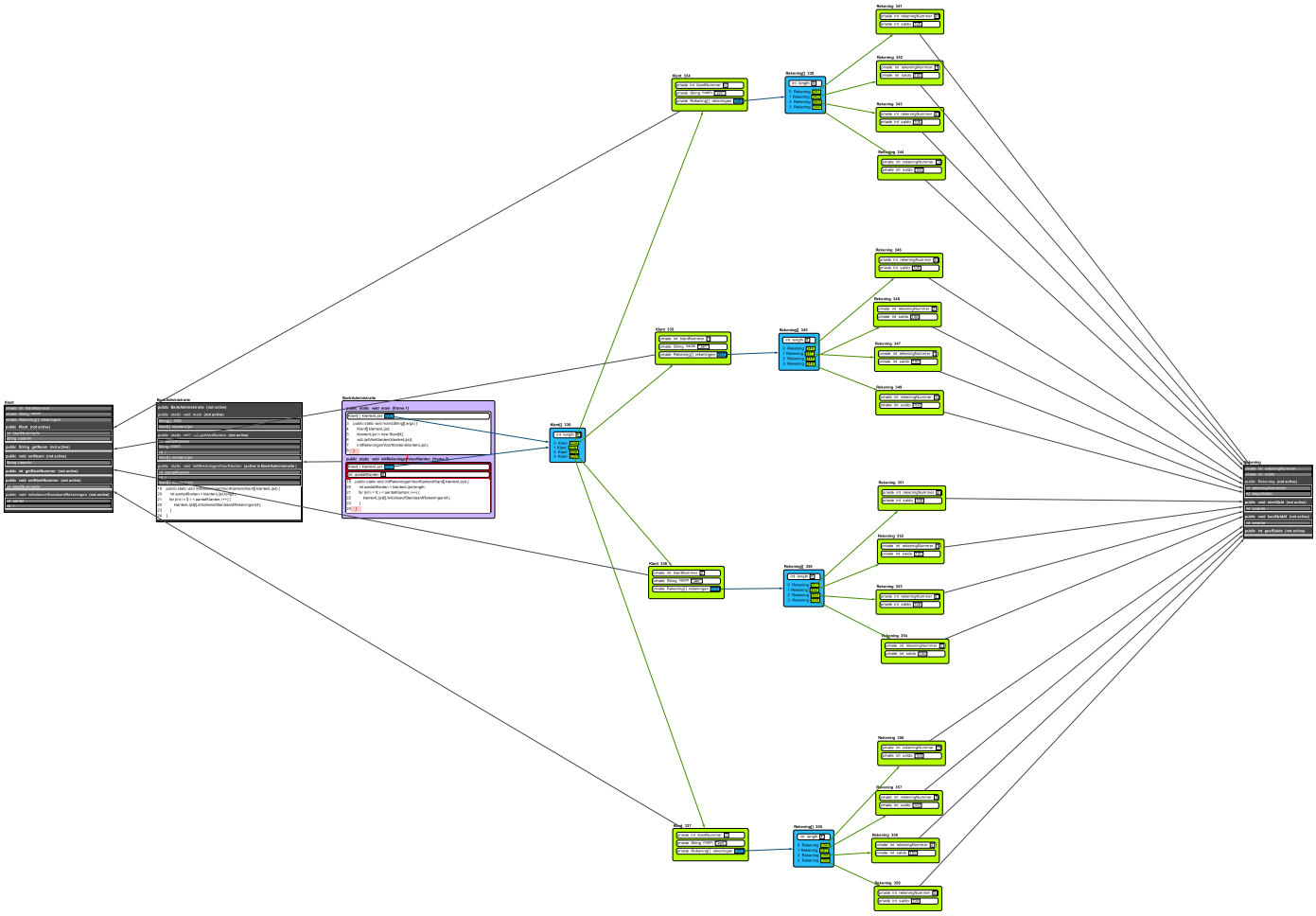

Fig. 5. The environment is also capable of handling more complicated run-time situation using zoom and pan features. In this figure, a running program structure is completely zoomed out. In a typical classroom setting, the student or teacher will adjust the zoom-level based on whether the item of interest is the overview itself, or a specific item within the program.

and strongly agree (7). For reasons of consistency, the questionnaire was always distributed during the sixth lesson of the course, which limits the list of specific concepts that are evaluated. The anonymous questionnaire is distributed and collected by a non-faculty member, and the teacher is not present during the evaluations.

Table 3 presents the results of the survey. These results indicate a favorable attitude of the students toward the environment in general and toward the presentation of the specific concepts. Student were also allowed, but not required, to write down remarks on the environment as they saw fit. In total, forty-two students provided additional remarks on what they liked and disliked. One student wrote “The visualization clearly indicates which steps are being executed and this makes it easier to understand why my design sometimes does not work.” Another mentioned “The visualization helps you follow step by step what exactly goes on in the program and, consequently, helps you better understand the program as a whole.” Another wrote “The visualization helps me understand the exercises, concepts and methods. I think it is a definite added value for this course.”

Some students also suggested improvements, such as “I would like increased support for exception handling”, “I would like additional explanations in the case of an exception” and “I would like a fast forward button, because otherwise the visualization sometimes takes too long”. Most of these requests have been added in later versions of the environment.

6. Experimental pilot evaluation

6.1. Experiment set-up and design

In order to have a more formal evaluation of the program visualization environment, an experiment was designed that would measure students’ performance on a test related to the understanding of the concept of recursion. In our own survey of the difficulty assigned by students to 35 programming concepts, this concept ranked number 7, in Milne and Rowe (2002) it ranked number 5 out of 36, and in Lahtinen et al. (2005) it ranked number 3 of 12. Clearly, this is one of the more difficult concepts to grasp. The goal of the experiment was to see whether the students’ comprehension of recursion could be improved through the use of the environment.

The first two years, the following set-up was used. Every student was presented with two questionnaires which were administered sequentially. In total, they had 10 min to complete each questionnaire. Thirty-four students of the introductory programming course at the

Fig. 6. This example shows four frames of a recursive method call. In the actual environment, students or teachers see all the frames and can move between frames using forward and back buttons.

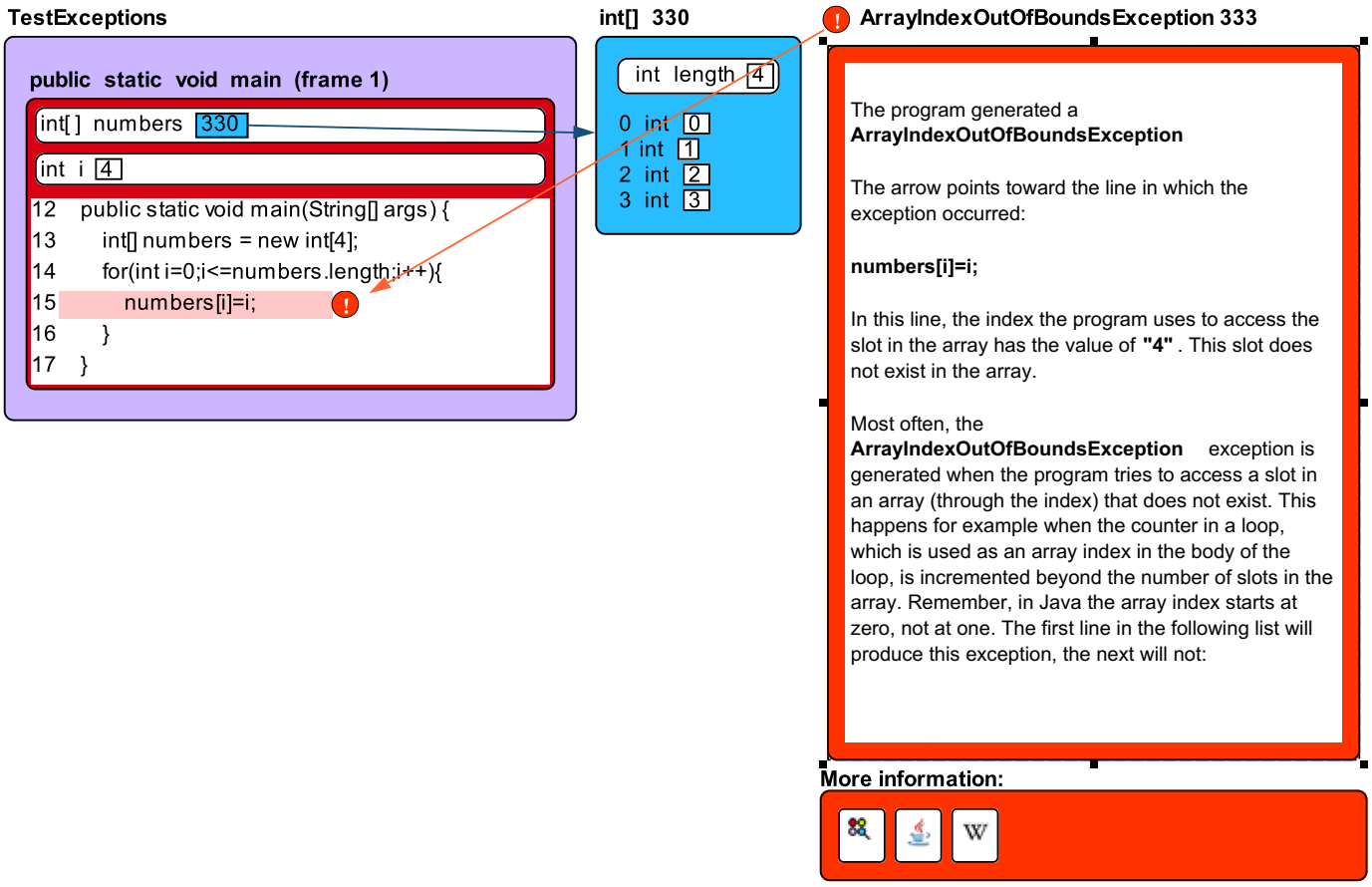

Fig. 7. This example shows the presentation of exception objects in the visualization, and the integration with on-line resources to explain the actual exception. In this example, the student tried to access index '4' in an array which only has indexes '0' to '3'.

University of Antwerp took part in this experiment. The questionnaires contained the textual representation of a program containing a recursive multiplication method. The method names were obfuscated, so the naming could not give an indication of the goal of the program. The students were asked the same set of questions on each questionnaire. The first questionnaire can be considered the pre-test. For this first questionnaire, the students had only the program text at their disposal, without any program visualization features. The treatment consisted of the availability of the program visualization environment to display the runtime execution of the same program. The program code was available digitally during the posttest, so the students did not have to waste time in copying this text in the environment. In all recorded cases, the students had worked with the visualization environment at least during five lessons of four hours each.

Table 3
Evaluation of the environment by the students.

ID	Topic	Average score
General topics		
G1	The visualization helps me understand the programming concepts	6.73
G2	The visualization helps me understand the exercises	6.12
G3	The visualization is easy to understand	6.29
G4	The visualization makes the lessons and the exercises more fun	5.59
G5	I have often used the visualization during the course	5.47
G6	I have often used the visualization at home to understand the exercises	5.59
G7	The visualization is a definite added value to the programming course	6.18
G8	I find it time-consuming to use the visualization during the exercises	3.18
Specific topics		
S1	The visualization helps in understanding variables (values, assignment, scope, …)	6.28
S2	The visualization helps in understanding iteration structures (for, while, do, …)	6.33
S3	The visualization helps in understanding selection structures (if, switch)	6.06
S4	The visualization helps in understanding methods	6.00
S5	The visualization helps in understanding control flow (parameters, return values, method calls)	5.65
S6	The visualization helps in understanding references	5.97
S7	The visualization helps in understanding arrays	5.82
S8	The visualization makes it easier to keep an overview over the values of the variables in the program	6.33

Table 4
SOLO level interpretation guidelines for the experiment.

SOLO level	Category	Description
0	Blank	Question not answered
1	Prestructural	Shows substantial lack of knowledge of programming constructs or is unrelated to the question.
2	Unistructural	A description of one portion of the code
3	Multistructural	A line by line description of all the code. Some summarization of individual statements may be included.
4	Relational	A summary of the purpose of the code.

There are two types of questions in the experiment. One type requires a descriptive answer, in this case the question "describe the general goal of the program". The other type requires a numerical answer, in this case the questions "How many times is method 1 executed over the course of the program?", "At most how many frames of method 1 are present on the method stack at any one time?" and "What is the value of the variable called result at the end of the program?".

The descriptive question is assigned a SOLO level. SOLO, or Structure of Observed Learning Outcomes, is a taxonomy that measures the level of abstraction of student responses (Biggs & Collis, 1982). It has been used in experiments for computer science education before, for instance in Clear et al. (2008), Bennedssen and Caspersen (2008), Lister, Simon, Thompson, Whalley, and Prasad (2006). In the SOLO taxonomy, each subsequent category represents a higher level of understanding. To reduce error, the SOLO category of the answers was evaluated by two independent educators, and the results were averaged. Each educator based its' conclusion of the adaptation of the SOLO levels to programming education, presented in Table 4, based on Clear et al. (2008). Differences of two or more categories were discussed and consensus was reached among the educators. The numerical questions are graded as either right or wrong.

The third year, the same questionnaire was used, but now the group was randomly split in a treatment group, and a control group. The control group was only presented with the standard Netbeans IDE to solve the questions while the treatment group could use the program visualization environment. In total, 20 students were present in the control group, while 19 students were present in the treatment group. Again, each group had 10 min to fill out the questionnaire. Both groups had access to the digital source code.

So, in the first two years the experiment was executed as a one group pre-test post-test pre-experimental design. In the third year, to avoid possible carryover effects and because of the availability of a larger sample, a two-group posttest-only randomized experimental design was used. The results of the two types of designs are presented separately.

6.2. Results and discussion

First we present the results of the one group design experiment ($N = 34$). Fig. 8 presents the percentage of correct answers for each of the numerical questions of the pretest and the posttest. It is clear that the treatment has significantly improved the result of the students' performance on these numerical questions. The mean score on this part of the test was .8 out of 3 for the pretest and 2.5 out of 3 for the posttest, with an average improvement in test performance of 1.6. Using a paired T test (assuming normal distribution of the test scores), we come to a t value of 8.749, showing a statistically significant improvement using the treatment ($p = .000$).

This means that, at the very least, presenting a clear visual display of the concept of recursion improves the concrete understanding of students of what is going on in the recursive program (this is tested in the experiment). While this does not guarantee an improved understanding of the concept in general (as this was not tested), the correct presentation and the resulting comprehension of the program will help to instill a correct schema of the concept of recursion in the minds of the students.

Fig. 9 presents the results of the SOLO evaluation of the first question of the questionnaire, asking for the general goal of the program.

There is a significant difference in the distribution of the SOLO level of the answers to the question "describe the general goal of the program". As an example, one student answered

"The goal is to create the sum of firstnumber and secondnumber in a special way."

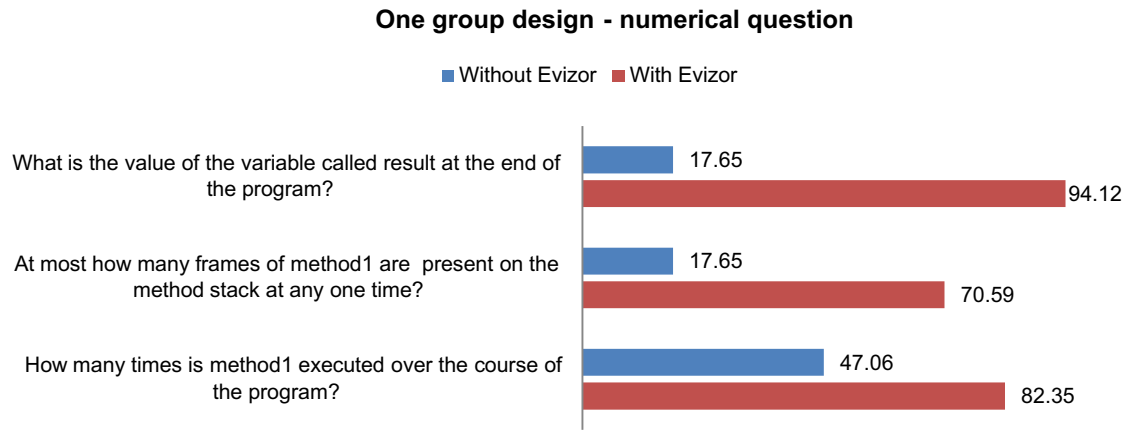

Fig. 8. Percentage of correct answers of the one group design experiment for the numerical questions.

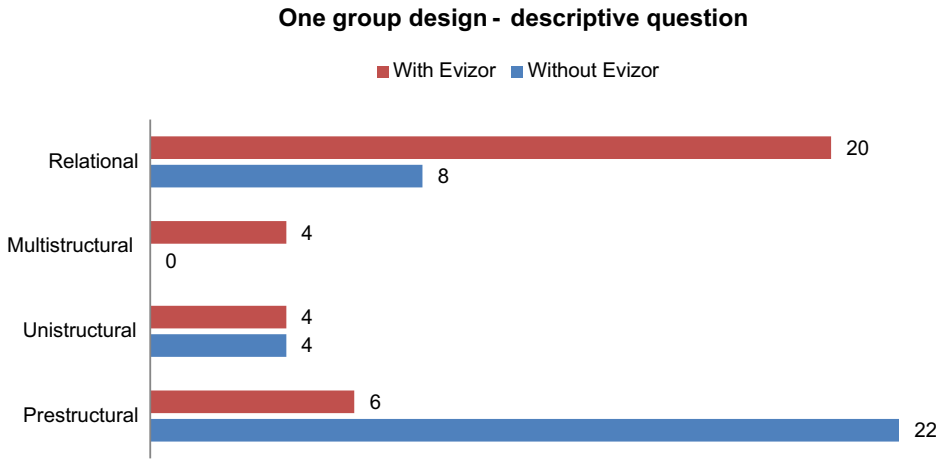

Fig. 9. Results of one group design for the SOLO evaluation of the descriptive question.

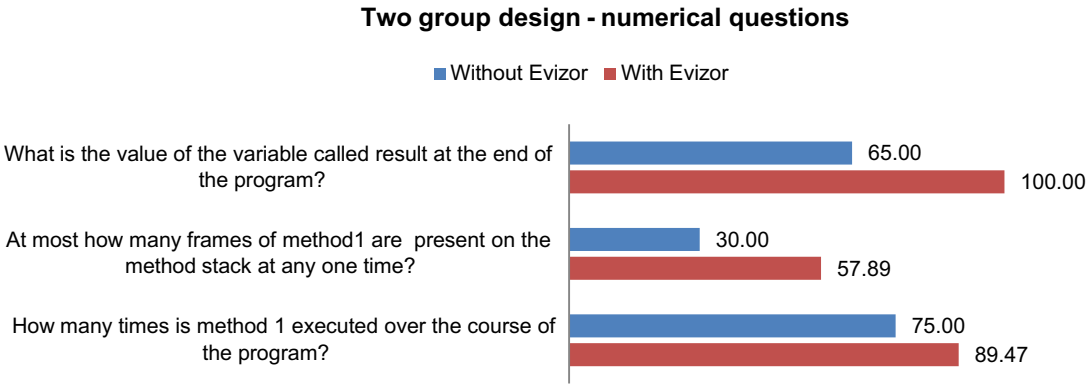

Fig. 10. Percentage of correct answers of the between-subject design experiment for the numerical questions.

This type of answer is considered prestructural, because it fails to show any insight into the way the method actually works, and what the goal of the program is. Another student answered

"Method 1 calls itself and deducts 1 from the second number until it reaches the value of 1 and then returns."

Although still not correct, this answer is graded with a SOLO level of unistructural, because it shows insight in a part of the code, but it fails to grasp the complete workings of the program, and also fails to see the complete picture. An answer that resulted in a SOLO level of relational is the following

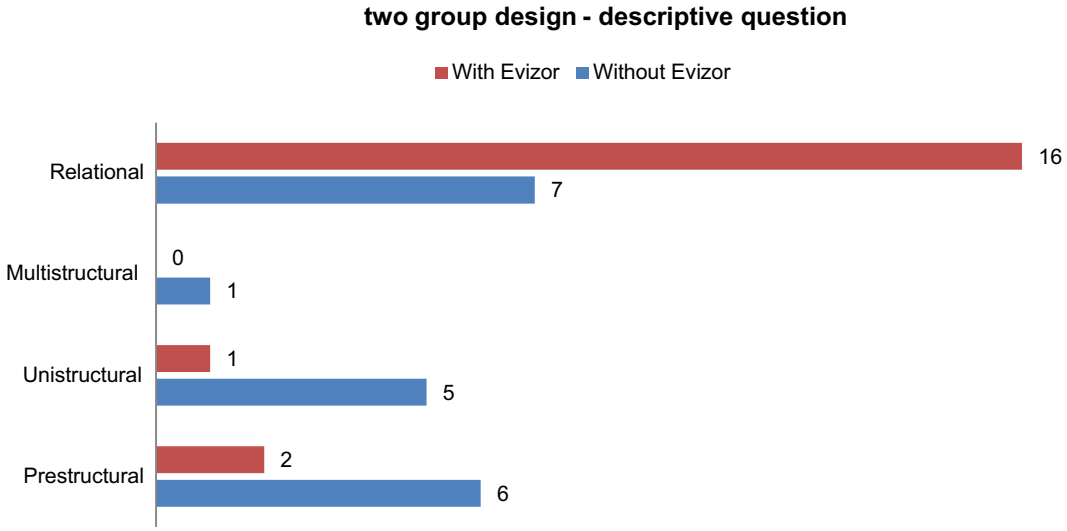

Fig. 11. Results of two group design for the SOLO evaluation of the descriptive question.

"The multiplication of two numbers, number1 and number2, through a method that calls itself."

This answer shows that the student was able to follow the entire structure of the program and deduce the final goal of the program. The average SOLO level of the pre-test (without EVizor) was 1.82. The average SOLO level for the post-test (with EVizor) was 3.18.

Fig. 10 presents the percentage of correct answers in the control group and the treatment group for each of the numerical questions. The control group ($N = 20$, the group that used the regular Netbeans IDE) had a mean score of 1.70 out of 3. The treatment group ($N = 19$, the group that used EVizor) had a mean score of 2.47 out of 3. Using an independent samples *T*-test to compare the groups we get a t value of 2.83 ($p = .007$), meaning we can assume a statistically significant effect here as well. It is logical that the effect here is smaller than in the one group design, because in the two group design, the control group could use the Netbeans IDE, which is likely to positively influence the result versus the use of just the paper version.

Fig. 11 present the result of the SOLO evaluation by the teachers for the two group design. Again, there is a clear difference in the SOLO level of the answers on the descriptive question. The average SOLO level of the control group was 2.35. The average SOLO level of the treatment group was 3.58.

These experiments have indicated clearly that the novel program visualization environment has a positive influence on the comprehension of the students of a program containing a recursive method. The students were able to answer the questions, both the descriptive question and the numerical questions, more correctly. While this does not guarantee an actual better comprehension of the concept of recursion, it does guarantee that a well-designed program visualization environment can present a better or more comprehensible picture of the concept than either a print-out of the code or a traditional IDE.

7. Conclusion and future work

This article has introduced a novel educational programming environment that uses program visualization techniques to help teachers and students of an introductory programming course. The environment is inspired by constructivist and cognitivist learning paradigms and the design of the programming concepts is based on principles of perception. These paradigms and theories were translated in a set of guidelines for program visualization environments. Based on tests with other program visualization environments we have concluded that none covered all guidelines sufficiently. This has lead the authors to design and implement a prototype program visualization environment, which has now been used for three successive years in an introductory programming course.

This environment has been evaluated both qualitatively, using a questionnaire, and quantitatively using an experiment. The qualitative evaluation by the students in three consecutive years has shown their appreciation of the environment and the approach. The quantitative evaluation has shown that the environment can indeed help with the visual presentation of complex programming concepts such as recursion.

While the initial evaluations and experiments have indicated very favorable results, it would be interesting to expand them in several ways. For instance, one could directly compare this environment to similar environments, such as JEliot or JIVE. The problem with such a comparison however is the difficulty to ensure internal validity, considering the wide range of variables not under control of the researcher. One would have to administer the various environments to very similar groups, with very similar backgrounds, making sure that the groups are confident in using the environments. A more useful experiment would be to evaluate what happens when several of the design features, such as the use of color, movement, links, information panes etc. are disabled. Although these features have been validated in other settings, and it seems unlikely that they would not apply here, the environment lends itself to check this experimentally. For instance, one could turn off the motion feature, and see whether it is more difficult for a student to see what is going on during runtime. This way of working would allow the researcher to keep control over all other variables, and only change the independent variable.

The environment itself might also be further expanded. One interesting opportunity is the further expansion and evaluation of the online integration features. While the features are already usable and appreciated by the students, one could imagine the possibility for students to create their own "augmented" exercises using information panes and pop-up quizzes, which could be shared with their peers and used in the weekly demonstrations. It would also be interesting to see how this constructionist approach evaluates against the currently used approach.

References

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. In *ITiCSE '05 proceedings of the 10th annual SIGCSE conference on innovation and technology in computer science education, Vol. 37* (pp. 84–88). New York, NY, USA: ACM.

Allen, E., Cartwright, R., & Stoler, B. (2002). DrJava: a lightweight pedagogic environment for Java. In *SIGCSE '02 proceedings of the 33rd SIGCSE technical symposium on computer science education, Vol. 34* (pp. 137–141). New York, NY, USA: ACM.

Andrianoff, S. K., & Levine, D. B. (2002). Role playing in an object-oriented world. In *SIGCSE '02 proceedings of the 33rd SIGCSE technical symposium on computer science education* (pp. 121–125). New York, NY, USA: ACM.

Astrachan, O., Bruce, K., Koffman, E., Kölling, M., & Reges, S. (2005). Resolved: objects early has failed. In *SIGCSE '05 proceedings of the 36th SIGCSE technical symposium on computer science, Vol. 37* (pp. 451–452). New York, NY, USA: ACM.

Atkinson, R. C., & Shiffrin, R. M. (1968). The psychology of learning and motivation: advances in research and theory. In K. W. Spence, & J. T. Spence (Eds.). *The psychology of learning and motivation: Advances in research and theory, Vol. 2* (pp. 89–195). Academic.

Baddeley, A., & Hitch, G. (1974). In G. A. Bower (Ed.). *Recent advances in learning and motivation, Vol. 8* (pp. 47–90). Academic Press.

Bailie, F., Courtney, M., Murray, K., Schiaffino, R., & Tuohy, S. (2003). Objects first – does it work? *Journal of Computing Sciences in Colleges, 19*(2), 303–305.

Baldwin, L. P., & Kuljis, J. (2000). Visualisation techniques for learning and teaching programming. In *Proceedings of the 22nd international conference on information technology interfaces, 2000. ITI 2000* (pp. 83–90), Pula, Croatia.

Bartram, L., Ware, C., & Calvert, T. (2003). Moticons: detection, distraction and task. *International Journal of Human-Computer Studies – Notification User Interfaces, 58*(5), 515–545.

Bennedssen, J., & Caspersen, M. E. (2008). Abstraction ability as an indicator of success for learning computing science? In *ICER '08 proceeding of the fourth international workshop on computing education research* (pp 15–26). New York, NY, USA: ACM.

Bergin, J., Eckstein, J., Manns, M.-L., Sharp, H., Voelter, M., Wallingford, E., et al. (2001). The pedagogical patterns project. In J. Haungs (Ed.), *OOPSLA '00 Addendum to the 2000 proceedings of the conference on object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM.

Bergin, J., Eckstein, J., Wallingford, E., & Manns, M. L. (2001). Patterns for gaining different perspectives. In *PLoP 2001 8th conference on Pattern Languages of Programs (2001)*, Illinois, USA.

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the quality of learning: The SOLO taxonomy (educational psychology series)*. Academic Press.

Bruner, J. (1979). *On knowing: Essays for the left hand*. Belknap Press.
Carlson, R., Chandler, P., & Sweller, J. (2003). Learning and understanding science instructional material. *Journal of Educational Psychology, 95*(3), 629–640.
Clear, T., Edwards, J., Lister, R., Simon, B., Thompson, E., & Whalley, J. (2008). The teaching of novice computer programmers: bringing the scholarly-research approach to Australia. In *ACE '08 proceedings of the tenth conference on Australasian computing education* (pp. 63–68). Darlinghurst, Australia: Australian Computer Society, Inc.
Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. In *SIGCSE '03 proceedings of the 34th SIGCSE technical symposium on computer science education* (pp. 191–195). New York, NY, USA: ACM.
Cowan, N. (2001). The magical number 4 in short-term memory: a reconsideration of mental storage capacity. *Behavioral and Brain Sciences, 24*(1).
Cross, J. H., & Hendrix, T. D. (2006). jGRASP: a lightweight IDE with dynamic object viewers for CS1 and CS2. In. *ITICSE '06 proceedings of the 11th annual SIGCSE conference on innovation and technology in computer science education, Vol. 38*. New York, NY, USA: ACM, 356–356.
Degrace, P., & Stahl, L. H. (1998). *Wicked problems, righteous solutions: A catalog of modern engineering paradigms*. Prentice Hall.
Felder, R. M., & Silverman, L. K. (1988). Learning and teaching styles in engineering education. *Engineering Education, 78*(7), 674–681.
Garner, S., Haden, P., & Robins, A. (2005). My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *ACE '05 proceedings of the 7th Australasian conference on computing education, Vol. 42* (pp. 173–180). Darlinghurst, Australia: Australian Computer Society, Inc.
Gestwicki, P. V., & Jayaraman, B. (2004). JIVE: java interactive visualization environment. In *OOPSLA '04 companion to the 19th annual ACM SIGPLAN conference on object-oriented programming systems, languages, and applications* (pp. 226–228). New York, NY, USA: ACM.
Giangrande, J. E. (2007). CS1 programming language options. *Journal of Computing Sciences in Colleges, 22*(3), 153–160.
Glaserfeld, E.v (1987). *The construction of knowledge — Contributions to conceptual semantics*. Intersystems Publications.
Gray, K. E., & Flatt, M. (2003). ProfessorJ: a gradual introduction to Java through language levels. In *OOPSLA '03 companion of the 18th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications* (pp. 170–177). New York, NY, USA: ACM.
Healey, C. G., Booth, K. S., & Enns, J. T. (1996). High-speed visual estimation using preattentive processing. *ACM Transactions on Computer-Human Interaction, 3*(2), 107–135.
Holmboe, C., Borge, R., & Granerud, R. (2004). Lego as platform for learning OO thinking in primary and secondary school. In *LNCS: Eighth workshop on pedagogies and tools for the teaching and learning of object oriented concepts*, Oslo, Norway.
Hundhausen, C. D., & Brown, J. L. (2008). Designing, visualizing, and discussing algorithms within a CS 1 studio experience: an empirical study. *Computers & Education, 50*(1), 301–326.
Jadud, M. C. (2006). *An exploration of novice compilation behaviour in BlueJ*. Ph.D. thesis, University of Kent at Canterbury.
Jeffries, R., Turner, A., Polson, P., & Atwood, M. (1981). The processes involved in designing software. In *Cognitive skills and their acquisition* (pp. 225–283). Hillsdale, N.J.: Erlbaum.
Jiménez-Diáz, G., Gómez-Albarán, M., Gómez-Martín, M. A., & González-Calero, P. A. (2005). ViRPlay: playing roles to understand dynamic behavior. In *Workshop on pedagogies and tools for the teaching and learning of object oriented concepts*, Glasgow, UK.
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.
Koffka, K. (1935). *Principles of gestalt psychology*. New York: Harcourt-Brace.
Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Computer Science Education, 13*(4).
Koning, H., Dormann, C., & Vliet, H.v (2002). Practical guidelines for the readability of IT-architecture diagrams. In *SIGDOC '02 proceedings of the 20th annual international conference on computer documentation* (pp. 90–99). New York, NY, USA: ACM.
Korhonen, A. (2003). *Visual algorithm simulation*. Ph.D. Thesis, Helsinki University of Technology.
Korhonen, A., Malmi, L., & Saikkonen, R. (2001). Matrix – concept animation and algorithm simulation system. In *AVI '02 proceedings of the working conference on advanced visual interfaces* (pp. 180). New York, NY, USA: ACM.
Kurland, D., Pea, R., Clement, C., & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research, 2*(4), 429–457.
Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. In *ITiCSE '05 proceedings of the 10th annual SIGCSE conference on innovation and technology in computer science education, Vol. 37* (pp. 14–18). New York, NY, USA: ACM.
Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., et al. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin, 36*.
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. In *ITICSE '06 proceedings of the 11th annual SIGCSE conference on innovation and technology in computer science education, Vol. 38* (pp. 118–122). New York, NY, USA: ACM.
Mannila, L., & Raadt, M.d (2006). An objective comparison of languages for teaching introductory programming. In *Baltic Sea '06 proceedings of the 6th Baltic Sea conference on computing education research: Koli Calling 2006* (pp. 32–37). New York, NY, USA: ACM.
Marcus, N., Cooper, M., & Sweller, J. (1996). Understanding instructions. *Journal of Educational Psychology, 88*(1), 49–63.
Mayer, R. E. (1981). The psychology of how novices learn computer programming. *ACM Computing Surveys, 13*(1), 121–141.
Mayer, R. E. (2001). *Multimedia learning*. Cambridge University Press.
McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *ITiCSE-WGR '01: Working group reports from ITiCSE on innovation and technology in computer science education, Vol. 33* (pp. 125–180). New York, NY, USA: ACM.
Michalski, R., & Grobelny, J. (2008). The role of colour preattentive processing in human-computer interaction task efficiency: a preliminary study. *International Journal of Industrial Ergonomics, 38*(3–4), 321–332.
Miller, G. A. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. *The Psychological Review, 63*, 81–97.
Milne, I., & Rowe, G. (2002). Difficulties in learning and teaching programming – views of students and tutors. *Education and Information Technologies, 7*(1), 55–66.
Moreno, A., & Joy, M. S. (2007). Jeliot 3 in a demanding educational setting. *Electronic Notes in Theoretical Computer Science, 178*, 51–59.
Naps, T. L., Eagan, J. R., & Norton, L. L. (2000). JHAVÉ – an environment to actively engage students in web-based algorithm visualizations. In *SIGCSE '00 proceedings of the thirty-first SIGCSE technical symposium on computer science education* (pp. 109–113). New York, NY, USA: ACM.
Naps, T. L., Rössling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., et al. (2002). Exploring the role of visualization and engagement in computer science education. In *ITiCSE-WGR '02: Working group reports from ITiCSE on innovation and technology in computer science education* (pp. 131–152). New York, NY, USA: ACM.
Paivio, A. (1986). *Mental representations: A dual coding approach*. New York: Oxford University Press.
Palmer, S. E., & Rock, I. (1994). Rethinking perceptual organization: the role of uniform connectedness. *Psychonomic Bulletin and Review, 1*(1), 29–55.
Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY, USA: Basic Books, Inc.
Papert, S. (1985). Different visions of logo. *Computers in the Schools, 2*(2–3), 3–8.
Pattis, R. E. (1995). *Karel the robot: A gentle introduction to the art of programming*. Wiley.
Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., et al. (2007). A survey of literature on the teaching of introductory programming. In *ITiCSE-WGR '07: Working group reports on ITiCSE on innovation and technology in computer science education* (pp. 204–223). New York, NY, USA: ACM.
Peterson, H. E., & Dugas, D. J. (1972). The relative importance of contrast and motion in visual detection. *Human Factors, 14*, 207–216.
Petre, M., & Quincey, E. d. (2006). A gentle overview of software visualisation. In *September 2006 newsletter – Psychology of programming interest group*.
Piaget, J. (1970). *Genetic epistemology*. Columbia University Press.
Ragonis, N., & Ben-Ari, M. (2005). On understanding the statics and dynamics of object-oriented programs. In. *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on computer science education, Vol. 37(1)* (pp. 226–230).
Rajala, T., Laakso, M.-J., Kaila, E., & Salakoski, T. (2008). Effectiveness of program visualization: a case study with the ViLLE tool. *Journal of Information Technology Education: Innovations in Practice, 7*.
Robins, A., Haden, P., & Garner, S. (2006). Problem distributions in a CS1 course. In. *ACE '06: Proceedings of the 8th Australian conference on computing education, Vol. 52* (pp. 165–173). Darlinghurst, Australia: Australian Computer Society, Inc.
Rodger, S. H. (2002). Using hands-on visualizations to teach computer science from beginning courses to advanced courses. In *ITiCSE2002, proceedings of the second program visualization workshop*. Aarhus, Denmark: ACM, New York.
Rössling, G., Schüer, M., & Freisleben, B. (2000). The ANIMAL algorithm animation tool. In *ITiCSE '00: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on innovation and technology in computer science education* (pp. 37–40). New York, NY, USA: ACM.
Rubinstein, R. (1975). Using LOGO in teaching. *SIGCUE Outlook, 9*(SI), 69–75.
Sanders, D., & Dorn, B. (2003a). Classroom experience with Jeroo. *Journal of Computing Sciences in Colleges, 18*(4), 308–316.

Sanders, D., & Dorn, B. (2003b). Jeroo: a tool for introducing object-oriented programming. In *SIGCSE '03 proceedings of the 34th SIGCSE technical symposium on computer science education* (pp. 201–204). New York, NY, USA: ACM.
Schulte, C., & Bennedsen, J. (2006). What do teachers teach in introductory programming? In *ICER '06: Proceedings of the 2006 international workshop on computing education research* (pp 17–28) New York, NY, USA: ACM.
Stasko, J. T. (1990). Tango: a framework and system for algorithm animation. *Computer, 23*(9), 27–39.
Sundararaman, J., & Back, G. (2008). HDPV: interactive, faithful, in-vivo runtime state visualization for C/C++ and Java. In *SoftVis '08: Proceedings of the 4th ACM symposium on software visualization* (pp. 47–56). New York, NY, USA: ACM.
Sweller, J. (2002). Visualization and instructional design. In *Knowledge Media Research Center (KMRC)*, Tübingen, Germany.
Thompson, S. M. (2004). *An exploratory study of novice programming experiences and errors*. Master's Thesis, University of Victoria.
Treisman, A. (1982). Perceptual grouping and attention in visual search for features and for objects. *Journal of Experimental Psychology: Human Perception and Performance, 8*(2), 194–214.
Tufte, E. R. (1990). *Envisioning information*. Graphics Press.
Vygotsky, L. S. (1978). *Mind in society: Development of higher psychological processes*. Harvard University Press.
Ware, C. (2004). *Information visualization, perception for design*. Morgan Kaufmann Publishers.
Wirth, N. (1996). Recollections about the development of Pascal. In *History of programming languages-II* (pp. 97–120). New York, NY, USA: ACM.
Wolfe, J. M. (2000). Visual attention. In *Seeing*. Academic Press.
Xinogalos, S., Satratzemi, M., & Dagdilelis, V. (2006). An introduction to object-oriented programming with a didactic microworld: object Karel. *Computers & Education, 47*(2), 148–171.