Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 224 (2009) 57–65
www.elsevier.com/locate/entcs

Rationale Behind the Design of the EduVisor Software Visualization Component

Jan Moons[1] and Carlos De Backer[2]

Department of Management Information Systems, Faculty of Applied Economics
Universiteit Antwerpen
Antwerpen, Belgium

Abstract

The EduVisor software visualization component is a new pedagogical tool specifically developed to address some wide-spread problems in teaching object-oriented technology to novice programmers. The visualization tool is integrated in a world-class IDE, and shows the students the structure of their own creations at runtime. EduVisor is based on a solid grounding in literature and over 25 years of combined experience in teaching a CS1 course. With this component we have set the goal of helping our students progress faster through the most difficult initial stages of programming.

Keywords: Program visualization, CS1, Java programming

1 Introduction

Over the past decades software design has often been described as a *wicked* or difficult problem [10][11][12][2]. Dalbey and Linn [4] note that the average student does not make much progress in an introductory programming course. More recently, there are many reports corroborating this position. For instance, in the infamous McCracken Report [14] the authors noted that the average score on a programming test was only 22.89 out of 110 points for a sample of 216 students. As difficult as it is for students to acquire programming and software design skills, just as difficult is it for teachers to teach those skills.

This paper is concerned with a novel visualization tool that can be used as a teaching aid in CS1 courses. The tool is called EDUcational VISual Object Runtime or EduVisor, and seeks to incorporate a lot of the acquired knowledge from previous visualization projects. The goal of EduVisor is threefold. First, we want to improve students' comprehension of the concepts introduced during the CS1 course. Second,

[1] Email: jan.moons@ua.ac.be
[2] Email: carlos.debacker@ua.ac.be

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2008.12.049

we want them to be able to debug their programs faster. Third, we want to increase the enthusiasm of students by visualizing (and thus reducing the abstraction level) their own efforts at the push of a button. The design of the tool is based on several decades of combined CS1 teaching experience and on a thorough grounding in relevant literature.

In section 2 we describe the driving forces behind the design of EduVisor. Section 3 describes the most important runtime issues one encounters during a CS1 course, which will be used as input to the design of EduVisor. Section 4 shows a small sample of the graphical representation used in the EduVisor component based on a simple use case. Section 5 provides an overview of the resulting properties of the component. Section 6 discusses the similarities and differences between EduVisor and related work. Finally, in section 7 we present our conclusions and provide an outlook on the future development of the EduVisor component.

2 Rationale of the EduVisor software visualization component

As so many educational institutions, the University of Antwerp has migrated from Pascal to C, later to C++ and finally to Java over the past two decades as the language of choice in our CS1 course. The switch to Java was made seven years ago. During our course we have noticed the same basic errors appear again and again, causing students to loose valuable time and generating frustration and disappointment.

On the highest level, these errors can be divided in compile-time errors and runtime errors. The code editor can help with some of the compile-time errors (although the compiler messages are very cryptic to novice programmers), but does nothing to aid in understanding runtime behavior. Thus, over the past five years, we have designed a visual language to illustrate the runtime behavior of a program. The language is, as we tend to say, as simple as possible and as complicated as necessary. We use this visual language when explaining programs at the whiteboard, and students' comprehension of these specific programs has improved markedly. However, when it is time to start programming their own exercises, the same errors tend happen all over again.

This is caused by several issues. First, the nature of their programming efforts is very much trial and error - which is actually a well known fact [1]. Second, the students do not go through the effort of drawing out their solutions in the way we do at the whiteboard. This is not *that* surprising - creating the visual representations for a running program takes quite some time. Encouraging however is that, when we force them to draw their programs on a sheet of paper, most of the time they are able to pinpoint the problems themselves.

Therefore we concluded that an automated software component based on our language could help students in recognizing and correcting their problems sooner. We did an extensive review of visualization components that address some of these issues, but none were found to be completely satisfactory. Section 6 talks in more

detail about these closest alternatives. EduVisor was thus conceived and designed to our specifications. With this new component we have set three interrelated goals:

(i) **Improve students' comprehension of basic programming constructs**: The abstract nature of programming languages makes understanding the concepts very hard for beginners. We, along with many other researchers ([9][15][17]), believe this difficulty can be reduced to some extent by using engaging visualization techniques.

(ii) **Speed up the debugging process of runtime problems**: debugging runtime errors is difficult even for experienced programmers. The standard debuggers that come with the major IDE's are very powerful, but also very difficult to operate - too difficult for novice programmers.

(iii) **Increase their enthusiasm about object-oriented programming**: The visual representation will provide an important incentive to students. As stated by Ross, *it is a tacitly known fact that programmers like to see their creations in action. All artisans are intrigued by what they create, and they like to observe their work from all angles [...]. [18]*

3 CS1 runtime issues

After describing our reasons for developing EduVisor we take a look at the specific problems we would like to address. Table 1 presents a listing which is loosely based on the list of Garner et al. [5], but restructured and rephrased to fit our purpose in two ways. First, the list is rephrased to present the causes rather than the symptoms of programming difficulties. Second, we only include runtime problems in the list, because this is the focus of our visualization tool. The next section details a use-case based on one of these problems and specifies how EduVisor will address it using visualization.

4 A simple EduVisor GUI use-case

This example details a problem we have witnessed recently with one of our students during our first lesson on objects. The goal was to write a program consisting of two classes, a `Bank` class and an `Account` class. The following code presents the main method located in the `Bank` class, containing the problem. The code in italics was **not** present in the student's solution - i.e. the `getValue` method did not get called after calling the `withdraw` method.

```
public static void main(String[] args){
        int value;
        Account account1 = new Account(100);
        Account account2 = new Account(200);
        value = account1.getValue();
        System.out.println(``value of account1 is ''+value);
        account1.withdraw(50);
```

Error	description
A. Failing to understand program design	(i) Failing to identify the correct classes. (ii) Failing to identify the correct methods. (iii) Failing to construct the correct algorithms. (iv) Failing provide the necessary variables.
B. Failing to understand the nature of objects	(i) Failing to understand that objects are persistent structures in memory holding their own state. (ii) Failing to understand that a method can instantiate multiple objects of the same kind. (iii) Failing to understand the difference between static and non-static structures. (iv) Failing to understand that objects can only be queried for their state through a reference. (v) Failing to understand the nature of references (e.g. returning a reference when the calling method already holds that reference).
C. Failing to understand message passing	(i) Failing to understand that methods have to be actively called. (ii) Failing to understand the parameter passing mechanism. (iii) Failing to understand that return variables have to be caught.
D. Failing to understand variables	(i) Failing to understand variable scoping. (ii) Failing to keep track of the values of variables in a running program. (iii) Failing to understand the necessity and operation of a control variable inside a loop.

Table 1
A list of common causes of runtime errors encountered during a CS1 course, loosely based on Garner et al. [5]

```
        //value = account1.getValue();
        System.out.println(``value of account1 is ''+value);
}
```

The student thought he was using the variable of the Account instance because he was referring to the object just before. This is a typical B4 problem - *Failing to understand that objects can only be queried for their state through a reference.*

EduVisor will help the student trace this error through dynamic visualization of the program runtime. Figure 1 shows a snapshot of the proposed visualization style. The following list details some of the visual features of EduVisor.

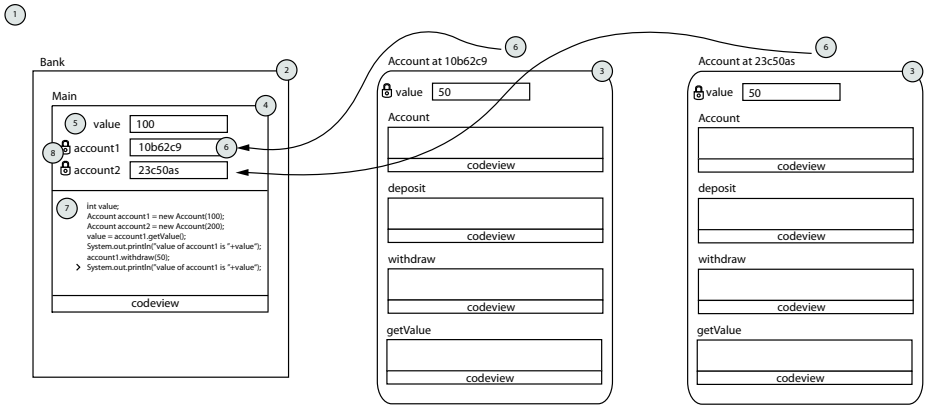

Fig. 1. EduVisor visualization snapshot of this use-case

(i) All information is presented on one single canvas.

(ii) Every class holding static information is represented as a rectangle with the name of the class positioned above the rectangle.

(iii) Every object is represented as a rounded rectangle with the name of the originating class and the hash-code of the object positioned above the rounded rectangle.

(iv) Every method (and every scoped block within a method) has its own area to hold the local variables. The variables are dispensed when the method execution is complete.

(v) Every variable is represented as a named rectangle that can hold a value, either of primitive or of reference type.

(vi) At object instantiation, a new object gets drawn on the canvas including member variables and method areas. The object lives as long as there are references pointing to the object. At instantiation, the memory address is transported to the variable holding the address.

(vii) Every method has a code area which can be uncollapsed. The code area shows the method implementation.

(viii) Every variable and method with reduced visibility relative to an active method (local variables as well as private member variables) is adorned with a lock symbol. The symbols are dynamically updated synchronously with the current active method.

5 EduVisor solutions to CS1 problems

Because we can not elaborate on the architectural details of our solution in this restricted space, we will not detail the libraries and code representations used by

EduVisor. Rather, in this paper we want to describe the ways in which EduVisor will help in solving the categories of CS1 problems we have defined in section 2. The following list contains references to the number of the problem (cf. table 1) that an EduVisor feature addresses. It should be noted that the effectiveness of the software component has not yet been tested with students. The primary reason for this section is to detail how we *expect* EduVisor to help, and any claims presented in this section have yet to be confirmed.

(i) **Failing to understand program design:** the single canvas approach allows the novice programmer to see the entire structure of the program at any time during the execution. All static and dynamic structures as well as all available variables can be seen without having to switch representations. Panning and zooming capabilities help with understanding more complex structures. This unified visual presentation will help the students to see e.g. which classes (A1), methods (A2) and variables (A4) are part of their program and help them understand the deficiencies in their design. The step-wise nature of the visualization will help them understand their algorithms (A3) better.

(ii) **Failing to understand the nature of objects:** every single object is explicitly represented on the canvas using rounded rectangles (B2). Every object contains only non-static member variables, explaining to the students the difference in runtime behavior between static and non-static structures (B3). The values of these variables are *always* visible, which will help the student in understanding the persistent and autonomous nature of an object (B1). Reference variables are represented in a different color than regular variables, and the value of the reference variable is the hash-code of the object. By clicking on the reference variable the corresponding object is highlighted, which will help in understanding the nature of references (B4 and B5).

(iii) **Failing to understand message passing:** Active objects are highlighted on the diagram. This way students see that an object is only active when a method of that object is called (C1). In addition, the values of the variables that are passed as parameters to a method are animated from the calling method to the called method, which helps in understanding the variable passing mechanism (C2). Return variables are also animated. Those return values that are not stored in a variable disappear, explaining the need to store return values (C3).

(iv) **Failing to understand variables:** All variables are always visible on the canvas and presented in their own scope (class, method or block) and adorned with modifier symbols that are dynamically adjusted to reflect the variables visible to an active method. This helps in understanding scoping (D1). In addition to the variables themselves the values of these variables are also visible, helping students keep track of program state (D2) and helping with understanding control variables in loop and selection structures (D3).

6 EduVisor contrasted with related work

Over the past three decades many studies have focused on improving and refining teaching methods for CS1 courses, which has resulted in an extensive pedagogical toolbox that can be used by computer science teachers. Some of the tools teachers have at their disposal are specialized IDE's such as JGrasp [8], BlueJ [12][13] and ProfessorJ [7], programming micro-worlds such as Alice [3] and ObjectKarel [22] and advanced visualization environments such as JEliot3 [16] and JIVE [6]. For reasons of conciseness, in this paper we limit ourselves to only the last category.

The tools we discuss in detail are JEliot3[3] and JIVE[4]. Both have great merit and had considerable influence on the design of EduVisor. JEliot3 is a tool based on over ten years of development, starting with JEliot, later JEliot2000 and finally JEliot3. JIVE has a long history itself, starting as a stand-alone tool and recently reborn as an Eclipse plug-in. We also discuss the program state visualization tool by Seppälä [19], which states similar goals as EduVisor.

JEliot uses several simultaneous representations to present the visualization. The canvas is divided in a memory stack, a constants area and an object heap. In addition, JEliot presents the data as it is processed by the virtual machine, i.e. using a method stack. Our emphasis is on *understanding the program architecture*, i.e. type A problems, not the VM. In addition, due to their particular implementation it is not possible to view all values on the method stack with one look at the canvas. This makes it difficult to *keep track of the values of variables in a running program* (D2). In addition, JEliot's canvas is based directly on the Java AWT classes and proprietary development. This implies certain restrictions, such as the complete absence of select, zoom and pan tools. These features are very important, as described by [20]. His visual mantra of *overview first, zoom and filter, and then detail on demand* is often mentioned as one of the cornerstones of good visualization tools. EduVisor is much more ambitious in this regard, thanks to it's use of an advanced open source visualization library, the Netbeans Visual Library[5].

JIVE has multiple representations of the same runtime behavior. We are presented with an object diagram and with a sequence diagram. However, it is not possible to see the values of variables contained in objects and methods nor the values of the parameters passed to methods and the return values of methods. EduVisor, on the other hand, uses the single canvas approach and shows dynamic behavior directly on this single canvas. This includes all values of reference and primitive variables in the program at any time. JIVE uses the Eclipse Graphical Editing Framework[6] to provide the representation. Jive should thus have zoom and pan features. However, in the most recent version zooming features are available through menu buttons and no easy panning or selection features exist.

The program state visualization tool mentions some of the same goals as Edu-

[3] JEliot3 is available online at http://cs.joensuu.fi/~jeliot/

[4] JIVE is available online at http://www.cse.buffalo.edu/jive/

[5] The Netbeans Visual Library is available online at http://graph.netbeans.org/

[6] GEF is available online at http://www.eclipse.org/gef/

visor. In [19], the authors state that *[their] notation attempts to show most of the runtime state of the program in a single diagram. Essentially, this means displaying all relevant instances, all references to them and some of the contents of the runtime stack together.* However, in their paper the presentation of program state seems to be quite different from the EduVisor presentation. The diagrams do not show the values of the variables, which is crucial in our system. For instance, the diagrams show references between objects as arrows between these objects, but there is no mention of the reference variables holding the objects. In addition, the diagrams do not show objects as environments of execution, i.e. the methods are not represented in the objects. We have found no further mention of this tool in literature.

One of our demands for a visualization tool was easy integration in a widely used IDE. We chose Sun's Netbeans as our platform, an thus EduVisor runs on the same platforms as Netbeans. Jeliot does not provide integration with a widely used IDE. Jive, on the other hand, is integrated with Eclipse - another widely used java IDE. It is not clear whether the program state visualization tool by Seppälä provides IDE integration, but according to the screenshots, it does not.

7 Outlook and conclusion

With EduVisor we have devised a visualization component that can be integrated easily in a world-class IDE such as Netbeans. The code is currently in alpha status but is being further developed as part of the PhD project of the first author. The final intent is to include additional ITS (Intelligent Tutoring System - see Wei et al. [21]) functions such as pop quizzes and course material through XML based code-injection into the intermediate visualization code. Once the code reaches beta in the course of this year, it will be released on a public server. Our first goal now is to further develop this code base, starting with the visualization features and working our way up to the code infusion. Next we will perform experiments to research important features such as the one-canvas philosophy, the animation features and the utility of the additional pedagogical features afforded by the ITS functions.

The main goal of this paper was to present the design philosophy of our EduVisor visualization component. Based on literature and experience we have created a list of common causes of CS1 runtime problems. This list is currently being validated during course sessions and the intermediate results indicate that the list indeed represents the most common issues. The list also serves as input to the design of EduVisor. Finally, we have presented the solutions EduVisor offers to these common problems and contrasted our work with that of similar environments.

References

[1] Ben-Ari, M., *Constructivism in computer science education*, SIGCSE Bulletin **30** (1998), pp. 257–261.

[2] Budgen, D., "Software Design," Addison Wesley, 2003, second edition, rapid.

[3] Cooper, S., W. Dann and R. Pausch, *Teaching objects-first in introductory computer science*, in: *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*

(2003), pp. 191–195.
URL http://portal.acm.org/citation.cfm?id=611966

[4] Dalbey, J. and M. C. Linn, *Cognitive consequences of programming: Augmentations to basic instruction*, Journal of Educational Computing Research **2** (1986), pp. 75–93.

[5] Garner, S., P. Haden and A. Robins, *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* (2005), pp. 173–180.

[6] Gestwicki, P. and B. Jayaraman, *Methodology and architecture of jive*, in: *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization* (2005), pp. 95–104.

[7] Gray, K. E. and M. Flatt, *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* (2003), pp. 170–177.

[8] Hendrix, T. D., I. James H. Cross and L. A. Barowski, *An extensible framework for providing dynamic data structure visualizations in a lightweight ide*, SIGCSE Bulletin **36** (2004), pp. 387–391.

[9] Hundhausen, C. D., S. A. Douglas and J. T. Stasko, *A meta-study of algorithm visualization effectiveness*, Journal of Visual Languages & Computing **13** (2002), pp. 259–290.

[10] Jeffries, R., A. Turner, P. Polson and M. Atwood, "The processes involved in designing software. Cognitive Skills and Their Acquisition." Erlbaum, Hillsdale, N.J., 1981 pp. 225–283.

[11] Kim, J. and F. Lerch, *Why is programming (sometimes) so difficult? programming as scientific discovery in multiple problem spaces.*, Information Systems Research **8** (1997), pp. 25–50.

[12] Kölling, M., *Teaching object orientation with the blue environment*, Journal of Object-Oriented Programming **12** (1999), pp. 14–23.

[13] Kölling, M., B. Quig, A. Patterson and J. Rosenberg, *The bluej system and its pedagogy*, Computer Science Education **13** (2003).

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

[15] Milne, I. and G. Rowe, *Difficulties in learning and teaching programming - views of students and tutors*, Education and Information Technologies **7** (2002), pp. 55–66.

[16] Moreno, A., N. Myller, E. Sutinen and M. Ben-Ari, *Visualizing programs with jeliot 3*, in: *AVI '04: Proceedings of the working conference on Advanced visual interfaces* (2004), pp. 373–376.

[17] Naps, T. L., G. Rössling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger and J. Ángel Velázquez-Iturbide, *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* (2002), pp. 131–152.

[18] Ross, R. J., *Experience with the dynamod program animator*, in: *SIGCSE '91: Proceedings of the twenty-second SIGCSE technical symposium on Computer science education* (1991), pp. 35–42.

[19] Seppälä, O., *Program state visualization tool for teaching cs1*, in: *Program Visualization Workshop*, The University of Warwick, Warwick, UK, 2004, pp. 62–67.

[20] Shneiderman, B., *The eyes have it: A task by data type taxonomy for information visualizations*, in: *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages* (1996), p. 336.

[21] Wei, F., S. H. Moritz, S. M. Parvez and G. D. Blank, *A student model for object-oriented design and programming*, J. Comput. Small Coll. **20** (2005), pp. 260–273.

[22] Xinogalos, S., M. Satratzemi and V. Dagdilelis, *An introduction to object-oriented programming with a didactic microworld: objectkarel*, Comput. Educ. **47** (2006), pp. 148–171.