Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 178 (2007) 153–160
www.elsevier.com/locate/entcs

Observer Architecture of Program Visualization

Amruth Kumar[1] and Stefan Kasabov

Ramapo College of New Jersey
505, Ramapo Valley Road
Mahwah, NJ, USA

Abstract

We propose Observer architecture for program visualization. The principles of Observer architecture are modular, model-driven visualization with one-directional coupling, hierarchical delegation, message-passing and archival by visualizers. The architecture is scalable. The resulting visualization can be distributed and modified independent of the model. The Observer architecture has been implemented in online tutors for programming called problets.

Keywords: Program visualization, Observer architecture

1 Introduction

Researchers have found that on problem-solving transfer tasks [11], animation with narration outperforms animation only, narration only, or narration before animation Similarly, on recall tasks, narration with visual presentation outperforms narration before visual presentation [1]. These results support a dual-coding hypothesis [15] that affirms two types of connections among stimuli and representations: representational connections between stimuli and the corresponding representations (verbal and visual), and referential connections between verbal and visual representations.

Computer Science researchers have similarly concluded that visualization must be accompanied by textual explanation in order to be pedagogically effective [12][17]. Explanations help learners understand what they see [3]. Complementing visualization with explanation is one of the 11 recommendations made for improving the effectiveness of visualization in a recent study [13]. In fact, one of the researchers states: "perhaps, our focus should change from algorithm visualization being supplemented by textual materials to textual materials being augmented and motivated

[1] Email: amruth@ramapo.edu

doi:10.1016/j.entcs.2007.01.045

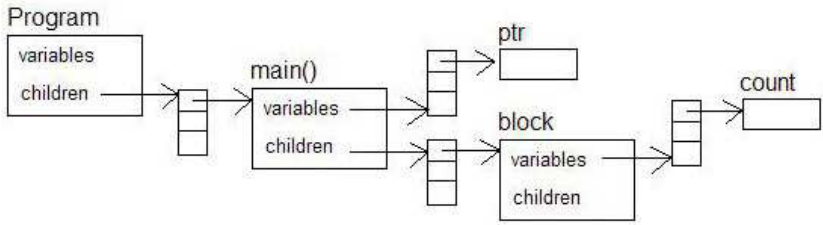

Fig. 1. Domain model of a C++ program involving pointers

by algorithm visualization" [12]. This is in fact, what we have undertaken: our programming tutors already provide textual explanation of the step-by-step execution of programs [9]. We have now supplemented the text explanations with program visualization. We had proposed a model-based scheme of visualization at a previous visualization workshop [7]. In this paper, we will describe the resulting architecture, called Observer architecture.

In the Observer architecture, visualization of a program is driven by observation of its execution. Observer architecture relies on the availability of an observable model of the executing program. We will present the details of such a model in section 2. We will present the principles of Observer architecture of visualization in section 3 and provide an example of its implementation in online tutors that we have developed for programming, called problets (www.problets.org). We will enumerate the advantages of Observer visualization in section 4 and end with discussion in section 5.

2 The Domain Model

Our programming tutors, called problets, automatically build a model of the program presented in each problem [8]. The model consists of the structure and behavior of the program. The structure describes the programming objects and constructs, and how they are interconnected within a program. The behavior describes the execution semantics of the constructs used in the program. Consider the following C++ pointer code with a nested block:

```
 void main(){
      int * ptr;
      {
           int count = 5;
           ptr = &count;
      }
      cout << *ptr;
}
```

The structure of the domain model for this program is shown in Figure 1. In the figure, variables (such as global variables) and functions such as `main()` are components of the program. The pointer variable `ptr` and a block are components inside the function `main()`. The variable `count` is a component of the block. The

behavior of a variable is modeled as a state diagram, with *declared*, *assigned* and *referenced* as nodes/states and possible operations (such as assignment) as arcs. Problets build such a model for each program and simulate it to generate 1) the output of the program; 2) semantic and run-time errors in the program, if any; 3) narration of the step-by-step execution of the program [9].

3 Principles of Observer Architecture of Program Visualization

In the Observer architecture of program visualization, each component in the program model is an observable object. Each of these objects is coupled with an Observer object whose responsibilities are to track and visualize the changes in the state of the observable object. The overall structure of the visualization objects is isomorphic to the structure of the model. This architecture is inspired by the Model-View-Controller pattern used for graphical user interface construction [5].

The principles behind observer visualization are:

- **Model-driven:** Each visualization object is driven by a domain object, called model, that can be simulated to derive the model's behavior. The visualization and model are separate objects with mutually exclusive responsibilities.
- **Modular:** Each visualization object is responsible for visualizing only the model with which it is coupled.
- **One-directional Coupling:** A visualization object renders itself by first consulting the model with which it is coupled, detecting any change in the state of the model, and reflecting this change in its rendering. In other words, the visualization object follows the corresponding model; the visualization object does not effect changes in the model; the model does not actively drive the visualization, and may not even be aware of the existence of the visualization object.
- **Hierarchical delegation:** Each visualization object delegates its components to render themselves before rendering itself, e.g., the function visualizer for main() delegates the pointer visualizer and the block visualizer to render themselves before rendering itself.
- **Message-passing for coordination:** Since each visualization object is modular, it is often necessary to coordinate among two or more visualization objects, e.g., when animating the assignment of the value of one variable to another, or depicting the assignment of a pointer to point to a variable or heap object. Message-passing is used for such coordination. In message-passing, a visualization object that needs to coordinate with another visualization object passes a message up and down the visualization hierarchy. A visualization object that receives the message acts on it if it is the intended target of the message, and passes it onwards if it is not.
- **Archival:** The visualization object maintains a history of the state changes of its model for future inspection by the learner, e.g., a variable visualizer maintains a

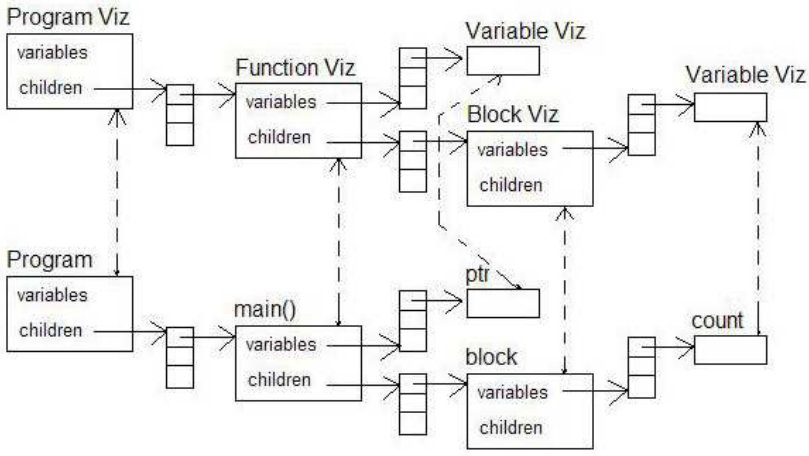

Fig. 2. Visualization of the Program model shown in Figure 1. "Viz" objects are observers.

history of all the values previously assigned to the variable. This design simplifies the model, allowing it to focus solely on the semantics of program execution.

Some of these principles, viz., separation of model and view, and one-directional coupling between the model and view have been proposed in the Matrix framework for building algorithm visualization [6]. Our work may be seen as an extension of the Matrix architecture in that we have proposed hierarchical delegation, message-passing for coordination and archival functions. Whereas a static repository is used in the Matrix architecture to keep track of the visualizations, we use hierarchical delegation.

The visualization objects created by problets for the components in the program model shown in Figure 1 are shown in Figure 2. Dashed arrows connect observable objects in the model to their corresponding observers in the visualization hierarchy. The direction of the dashed-arrows denotes the direction in which data flows from the observable to the observer. The visualization of the program is obtained by rendering the observer objects.

Figure 3 shows the snapshot of the visualization generated by problets using Observer architecture for a C++ program involving multiple levels of nested blocks and a pointer. In the snapshot, the data space of the executing program is shown as consisting of a global space, stack and heap. The activation record of the function `main()` is shown in the stack, and the variables of nested blocks are shown nested in this activation record. Message-passing was used by the visualizer to draw the arrow connecting `referencePointer` to the variable `weight`. Note that left and right scroll controls are provided for variable visualizers so that the learner can examine the archived values of each variable.

As is typical of visualization environments, the following four learner controls are provided at the bottom: 1) one step forward; 2) one step backward; 3) fast forward to the end; and 4) rewind to the start. Since interaction is one of the keys to effective visualization, we did not provide auto-play facility. The visualization of step-by-step execution of the program code in the left panel (the currently executing line of

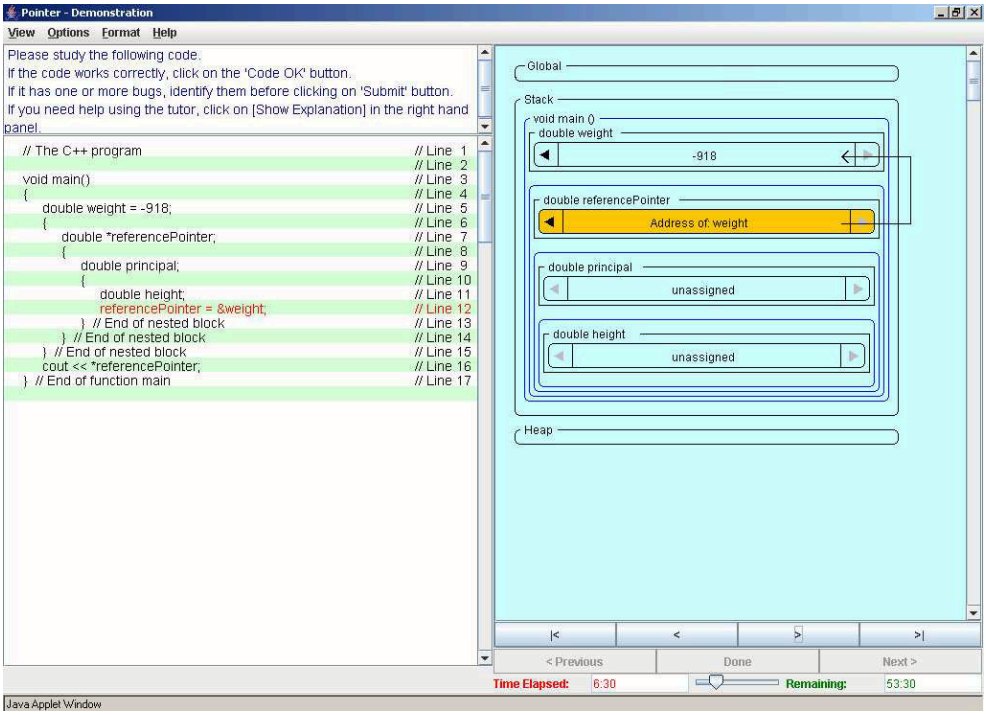

Fig. 3. Snapshot of the visualization of a program with multiple levels of nested blocks

code highlighted in red) is coordinated with the visualization of the data space in the right panel (the variables affected by the currently executing line highlighted in orange).

4 Advantages of Observer Architecture

There are several advantages to the Observer architecture of visualization:

- Observer-architecture is model-driven, i.e., visualization is driven by a model of the domain that can be simulated. Model-driven visualizations are capable of capturing more of the semantics of the domain being modeled [14]. They support custom input data sets, one of the ways to improve the effectiveness of visualization [13] - in the case of program visualization, this includes visualizing learner-written programs. They reduce instructor overhead, one of the impediments to adoption and use of visualization [13], because, instead of specifying the visualization, the instructor can specify the program for which the visualization is desired, and let the model-driven observer architecture automatically create the visualization for the program.
- The architecture is scalable - when new constructs are added to the domain, visualization can be extended by adding corresponding observer objects in the visualization hierarchy and composing them as prescribed by the model. This facilitates incremental development of the visualization system, which is a plus when dealing with a large domain such as programming.

- A recent study suggests that eliminating the tight coupling between the visualizer and the program model might facilitate "easier and more widespread development of effective visualization systems" [14]. In this vein, we have been working with the University of Pittsburgh to develop a distributed client-server architecture for code visualization [10]. Since the coupling between the program model and visualizers in the Observer architecture is one-directional, this architecture is well-suited for a distributed client-server deployment.
- The visualization presented in Figure 3 was designed for students in the junior/senior level Programming Languages course, hence the mention of stack, heap and global space, all components of the data space during program execution. Since the architecture defines a clear separation between observable and observer objects, we can develop alternative visualizations of the same program by simply developing alternative observer objects, e.g., a visualization for Computer Science I, wherein, the variables are shown as independent objects in the visualization space rather than as components embedded in scope objects. In other words, visualization can be modified independent of the model - Observer architecture enjoys the benefits of the Model-View-Controller pattern [5].
- Adapting to the knowledge level of the user is one of the recommendations made in a recent study for improving the effectiveness of visualization [13]. Adaptive visualization "helps focus student attention on least understood concepts" [4]. Our premise is that adaptation should prescribe, not proscribe. The implications are: 1) Unnecessary details should be hidden, not deleted; 2) Learner should always have the option of viewing any and all hidden details. We propose an object/event ontology for adaptation - domain objects and events are used as the basis for determining whether a visualization detail is displayed or hidden. Indexing the visualization details by domain object(s)/event(s) permits continual adaptation of visualization. Since observer architecture is model-driven, and objects/events can be easily identified in the domain model, observer architecture is ideally suited for prescriptive and continual adaptation of visualization.

5 Discussion

We have proposed Observer architecture for visualization. The architecture is general and domain-independent - it can be used for algorithm visualization just as well as program visualization, as long as an appropriate model is available. This may be seen as a disadvantage of Observer architecture for program visualization: the model needed for program visualization is a language interpreter, constructing which can be a daunting undertaking.

The best known model-driven program visualization system is Jeliot 3 for Java [2]. It uses a modified version of the source code interpreter DynamicJava (www.koala.ilog.fr/djava) as its model. It uses an intermediate code and an interpreter for the intermediate code to coordinate the model with the visualizer. The objects in the intermediate code interpreter, such as Value and Variable maintain a reference to their corresponding visualization objects, called actors, such as Value-

Actor and VariableActor. Therefore, Jeliot 3 "pushes" data from the model to the visualizer rather than have the visualizer passively observe the model, as proposed in the Observer architecture.

DynamicJava maintains a centralized model not isomorphic to the actual data space of an executing program, e.g., it maintains separate stacks for the variables, operators, etc. In contrast, the model used in problets is faithful to the actual run-time environment of a program and the visualization is choreographed with the model one-to-one. Therefore, the denotational match between the visualization and the expert's mental model is high, which, according to Epistemic Fidelity theory [16] promotes efficient transfer of the mental model to the learner, especially in the Programming Languages course. Finally, in Jeliot 3, the Director centrally coordinates all the animation. In problets, the visualization is hierarchically delegated to the various components.

Problets are driven by parameterized templates. The user can execute and visualize a new program by simply entering its template into a problet. By defining a clear separation between observable and observer objects, Observer architecture makes it easy for the developer to provide multiple visualizations of a programming construct. When the developer provides alternative visualizations, the user can select among them as easily as selecting from a menu. The user will be able to customize the visualization to the extent that the individual visualization objects allow them, e.g., visualization of a variable may permit the user to change its appearance or animation scheme. Since the visualization objects are composed to build the overall visualization, the user can customize the visualization of each type of program object independent of the visualization of the other objects.

We plan to evaluate the visualization in problets in fall 2006. We would like to evaluate the effectiveness of the visualization with and without text explanation for programming problems. In addition to the visualization of data space reported in this paper, currently, we are also developing visualization of control flow, wherein, the control constructs in a program are depicted as elements in a flowchart, and the flowchart is animated to visualize the flow of control in the program. Our objective is to provide three coordinated visualizations of the same program in addition to text explanation: 1) code visualization - highlighting the line of code being executed - see the left panel in Figure 3; 2) data visualization as presented in the right panel in Figure 3; and 3) control visualization in terms of a flowchart. These multiple views are expected to improve the effectiveness of visualization [13]. Finally, we plan to adapt the visualization to the needs of the learner. We have the necessary object/event infrastructure in place in the model. We need to extend our observer objects to take adaptation into account.

Acknowledgement

Partial support for this work was provided by the National Science Foundation's Educational Innovation Program under grant CNS-0426021 and by the Ramapo College Foundation.

References

[1] Baggett, P., *Role of temporal overlap of visual and auditory material in forming dual media associations*, Journal of Educational Psychology **76** (1984), pp. 408–417.

[2] Ben-Ari, M., N. Myller, E. Sutinen and J. Tarhio, *Perspectives on program animation with jeliot: In diehl, s. (ed.)*, Software Visualization. LNCS **2269** (2002), pp. 31–45.

[3] Brusilovsky, P., *Explanatory visualization in an educational programming environment: connecting examples with general knowledge*, in: *Proceedings of the 4th International Conference on Human-Computer Interaction, EWHCI'94, Berlin: Springer-Verlag*, 1994, pp. 202–212.

[4] Brusilovsky, P. and H. Su, *Adaptive visualization component of a distributed web-based adaptive educational system*, in: *Proceedings of the 6th international conference on Intelligent Tutoring Systems,*, 2002, pp. 229–238.

[5] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns, Elements of Reusable Object-Oriented Software," Addison Wesley, 1995.

[6] Korhonen, A. and L. Malmi, *Design pattern for algorithm animation and simulation*, in: *Proceedings of the First Program Visualization Workshop*, 2001, pp. 89–100.

[7] Kumar, A., *Model-based animation for program visualization*, in: *Proceedings of the Second Program Visualization Workshop*, 2002, pp. 37–44.

[8] Kumar, A., *Model-based reasoning for domain modeling in a web-based intelligent tutoring system to help students learn to debug c++ programs*, in: *Proceedings of Intelligent Tutoring Systems (ITS 2002)*, 2002, pp. 792–801.

[9] Kumar, A., *Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors*, Technology, Instruction, Cognition and Learning (TICL) Journal (to appear).

[10] Loboda, T., A. Frengov, A. Kumar and P. Brusilovsky, *Distributed framework for adaptive explanatory visualization*, in: *Proceedings of The Fourth Program Visualization Workshop*, 2006, pp. 11–16.

[11] Mayer, E. and R. Anderson, *Animations need narrations: An experimental test of a dual-coding hypothesis*, Journal of Educational Psychology **83** (1991), pp. 484–490.

[12] Naps, T., J. Eagan and L. Norton, *Jhave - an environment to actively engage students in web-based algorithm visualizations*, in: *Proceedings of the 31st SIGCSE Technical Symposium, Austin, TX*, 2000, pp. 109–113.

[13] Naps, T., R. Fleischer, M. McNally, G. Rossling, C. Hundhausen, S. Rodger, V. Almstrum, A. Korhonen, J. Velazquez-Iturbide, W. Dann and L. Malmi, *Exploring the role of visualization and engagement in computer science education.*, SIGCSE Bulletin **35** (2003), pp. 131–152.

[14] Naps, T., G. Rossling, P. Brusilovsky, J. English, D. Jarc, V. Karavirta, C. Leska, M. McNally, A. Moreno, R. Ross and J. Urquiza-Fuentes, *Development of xml-based tools to support user interaction with algorithm visualization*, SIGCSE Bulletin **37** (2005), pp. 123–138.

[15] Paivio, A., *Mental representations: A dual coding approach*, New York: Oxford University Press (1990).

[16] Roschelle, J., *Designing for cognitive communication: Epistemic fidelity or mediating collaborating inquiry*, Computers, Communication and Mental Models. D.L. Day and D.K. Kovacs (Eds.), Taylor and Francis, London (1996), pp. 13–25.

[17] Stasko, J., A. Badre and C. Lewis, *Do algorithm animations assist learning? an empirical study and analysis.*, in: *Proceedings of the INTERCHI 93 Conference on Human Factors in Computing Systems, Amsterdam*, 1993, pp. 61–66.