



ELSEVIER

Available online at www.sciencedirect.com

**Electronic Notes in
Theoretical Computer
Science**

Electronic Notes in Theoretical Computer Science 178 (2007) 101–109

www.elsevier.com/locate/entcs

Providing Data Structure Animations in a Lightweight IDE

Dean Hendrix¹ James H. Cross² Jhilmil Jain³
Larry Barowski⁴

*Department of Computer Science and Software Engineering
Auburn University
Auburn, Alabama 36849 USA*

Abstract

This paper presents the data structure animation tool jGRASP, which can automatically generate multiple synchronized views while the underlying code is being developed. The seamless integration of the the IDE with pedagogically effective software visualizations makes jGRASP an interesting tool for both educators and students.

Keywords: Visualization, data structures, IDE, jGRASP

1 Introduction

Although many visualization techniques have been shown to be pedagogically effective, they are still not widely adopted. The reasons include: lack of suitable methods of automatic-generation of visualizations; lack of integration among visualizations; and lack of integration with basic integrated development environment (IDE) support. To effectively use visualizations when developing code, it is useful to automatically generate multiple synchronized views without leaving the IDE. The jGRASP IDE (<http://jgrasp.org>) provides object viewers that automatically generate dynamic, state-based visualizations of objects and primitive variables in Java. Such seamless integration of a lightweight IDE with a set of pedagogically effective software visualizations should have a positive effect on the usefulness of software

¹ Email: hendrtd@auburn.edu

² Email: crossjh@auburn.edu

³ Email: jainjhi@auburn.edu

⁴ Email: barowla@auburn.edu

visualizations in a classroom environment. Multiple instructors have reported positive anecdotal evidence of their usefulness. We conducted formal, repeatable experiments to investigate the effect of these viewers for singly linked lists on student performance and we found a statistically significant improvement over traditional methods of visual debugging that use break-points. Similar experiments, but which focus on binary search trees, are currently underway.

2 Related Work

The approach we have taken for the state-based viewers in jGRASP to automatically generate the visualization from the user's executing program and then to dynamically update it as the user steps through the source code in either debug or workbench mode. This is somewhat similar to the method used in Jeliot [7]. However, jGRASP differs significantly from Jeliot in its target audience. Whereas Jeliot focuses on beginning concepts such as expression evaluation and assignment of variables, jGRASP includes visualizations for more complex structures such as linked lists and trees. In this respect, jGRASP is similar to DDD [10]. The data structure visualization in DDD shows each object with its fields and shows field pointers and reference edges. In jGRASP, each category of data structure (e.g., linked list vs. binary tree) has its own set of views and subviews which are intended to be similar to those found in textbooks. Although we are planning to add a general linked structure view, we began with the more intuitive "textbook" views to provide the best opportunity for improving the comprehensibility of data structures. We have specifically avoided basing the visualizations in jGRASP on a scripting language, which is a common approach for algorithm visualization systems such as JHAVE [8]. We also decided against modifying the user's source code as is required by systems such as LJV [2]. Our philosophy is that for visualizations to have the most impact on program understanding, they must be generated as needed from the user's actual program during routine development.

3 Motivation

All Computer Science, Software Engineering, and Wireless Engineering majors at Auburn University are required to take the COMP 1210 course (an objects-early CS1 in Java) followed by the COMP 2210 course (a Java-based CS2). Data structures and algorithms are abstract concepts, and the understanding of this topic and the material covered in class can be divided into two levels: a) Conceptual - where students learn concepts of operations such as create, add, delete, sort etc; and b) Coding - where students implement the data structure and its operations using any programming language (Java in our case). Attrition from our computing majors is most noticeable during the CS2 course.

We conducted paper-based surveys and one-on-one interviews in Fall 2004 and Spring 2005 to understand the aspects of the CS2 course that students find most difficult [4]. One result of the surveys was a clear indication that students did not

find fundamental concepts difficult to understand but had much more trouble with the implementation. This survey result was supported by data from the course grades. About 75% of students indicated that they had an appropriate level of expertise in Java to complete the requirements of CS2. The basic problem was that students have difficulty transitioning from static textbook concepts to dynamic programming implementation [9]. Thus, there is a need to bridge the gap from concepts to implementation.

[1] report that between 75-80% of students are visual learners. Most students will retain more information when it is presented with visual elements (pictures, diagrams, flowcharts, etc). In programming, visual learners can benefit from creating diagrams of problem solutions (e.g., flowcharts) before coding. Similarly, visual representations of data structure states should help in data structure understanding. Thus, it would be beneficial to have a tool that enables students to visualize both the conceptual and the implementation aspects of data-structures.

We surveyed over 21 tools that are used for the purpose of data structure visualization [5] and found that most tools (more than 14 in our survey) focused on conceptual understanding. We found that only 7 implementation level tools were available to help students during program comprehension and debugging activities. None of these implementation tools fulfilled all of our goals, viz.,

- serve the dual purpose of classroom demonstration and development environment (i.e. can be used for lab exercises and assignments)
- provide automatic generation of views
- provide multiple and synchronized views
- provide full control over the speed of the visualization
- bridge the gap between abstract learning and code implementation

4 jGRASP Object Viewers

During execution, Java programs will usually create a variety of objects from both user and library classes. Since these objects only exist during execution, being able to visualize them in a meaningful way can be an important element of program comprehension. Although this visualization can be done mentally for simple objects, most programmers can benefit from seeing more tangible representations of complex objects while the program is running.

Beginning with version 1.8, the jGRASP IDE provides a family of dynamic viewers for objects and primitives. These viewers are the most recent addition to the software visualizations provided by jGRASP. The purpose of a viewer is to provide one or more views of a particular class of objects. When a class has more than one view associated with it, the user can open multiple viewers on the same object with a separate view in each viewer. These viewers are tightly integrated with the jGRASP workbench and debugger and can be opened for any item in the Workbench or Debug tabs from the Virtual Desktop (see Figure 1). Since the jGRASP integrated debugger is used to collect the runtime information necessary to

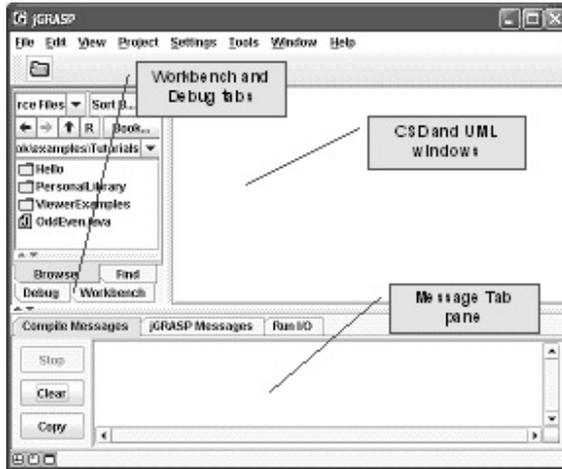


Fig. 1. jGRASP Virtual Desktop

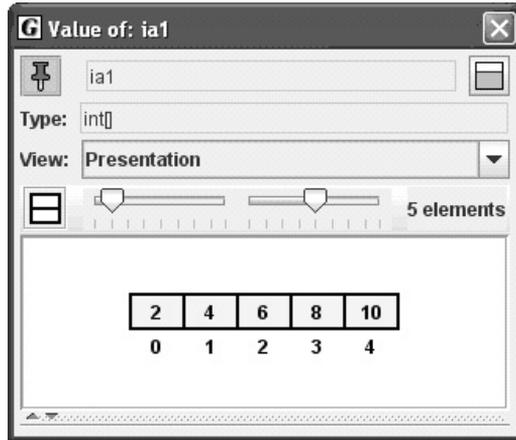
render the visualizations, a program must run in the debugger or from the jGRASP workbench for its data structures to be visualized. A separate viewer window can be opened for any primitive, object, or field of an object that is currently active on the workbench or in the debugger tab by simply dragging and dropping an icon from the debugger or workbench to the jGRASP desktop. Thus, these viewers are effortless with respect to the amount of work required of the student to create and use them.

All objects have a basic view, which is the same as the view shown in the workbench and debug tabs. This view shows all the values associated with the object in a collapsible hierarchy. Depending on their data type, some objects will have additional views. Figures 2a and 2b show object viewers for an array of integers (int) and an instance of `java.util.TreeMap`. Each is shown in a presentation view which is intended to be similar to a textbook depiction or to what an instructor might draw on the board. jGRASP provides presentation viewers for arrays, strings, and classes from the Java Collections Framework.

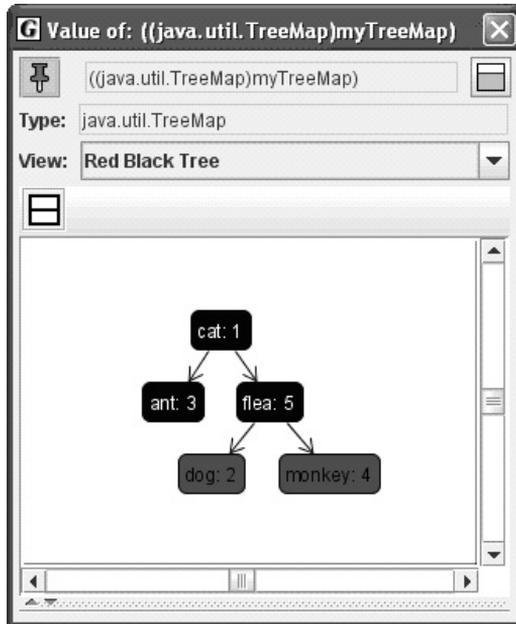
5 Animated Verifying Viewers

Viewers fall into two categories: non-verifying and verifying. The non-verifying viewers assume that the structure of the object being viewed is correct, and generally use method calls to elaborate the structure. When a structure gets beyond a certain size, the non-verifying viewers will examine only the part of the structure that is on-screen. Because of this, they can be used to examine large structures without slowing the debugging process excessively. The non-verifying viewers would generally be used to examine the contents of a structure in the context of an algorithm that uses it, rather than to examine the workings of the data structure itself. Viewers for “built-in” data structures (e.g., arrays, JCF classes) are all non-verifying. Non-verifying viewers are discussed in further detail in [3].

The purpose of the verifying viewers is to aid in the understanding of the data



(a)



(b)

Fig. 2. (a) Viewer for an array of ints. (b) Viewer for an instance of `java.util.TreeMap`.

structures themselves, and to assist in finding errors while students are developing their own implementation of a data structure. To further this intended use, any local variables of the structure's node type are also displayed, along with any nodes to which they are linked. Links between these local variable nodes or structure fragments and the main structure are displayed. This allows mechanisms of the data structure such as finding, adding, moving, and removing elements to be examined in detail by stepping through the code.

As an additional aid to understanding the mechanisms of the data structure, the verifying viewers animate structural changes. In order to do this, they store a representation of the entire data structure at each update (viewer updates happen at a breakpoint or after a step in the debugger). At each update, the value from

the previous update (which may or may not be the same as the current value) is examined for changes. If any nodes in the structure have moved, the viewer enters into animation mode. In this mode, an “animation update” occurs at regular intervals. During animation, the previous structure value and previous local variable nodes and structure fragments (which may or may not be present any longer) are displayed. Node locations are interpolated so that they move smoothly from their old locations to the new ones, within and between the main structure and local variable nodes and structure fragments. At the end of animation, the new structure value and new local variable nodes and structure fragments are displayed.

Animated verifying viewers for data structures are currently created by extending the base viewer classes provided with the jGRASP. When placed in a viewer directory, these viewers are available to any program executing in debug or workbench mode. A user can simply drag and drop the object reference anywhere on the screen. The viewer will be automatically updated as the user steps through the code. If multiple viewers are implemented for the same class, the user simply makes a selection from a drop down list in the viewer window. We are currently working towards a viewer mechanism which will attempt to identify the type of structure, if any, defined by a user’s class, and then map the internal fields of the class onto an appropriate category of viewer class (e.g., linked list). This will drastically reduce the need to manually extend the base viewer classes. When the user opens a viewer, the goal is for jGRASP to determine the inherent data structure of the object and display the most appropriate view.

Figure 3 shows three frames from an animation sequence generated by a verifying viewer. The viewer was opened on an instance of a binary search tree class used in the CS2 textbook. These frames depict the insertion of a new element (35) into an existing binary search tree. Using this viewer, students are able to watch the pointer (current) walk down the tree nodes to find the proper insertion point, and then watch as the new node “slides” into place as the left child of 40. All this is done as they are stepping through the code, thus making an immediate connection between the abstract behavior demonstrated in class and the concrete implementation embodied in the code.

6 Evaluation

We are currently conducting controlled experiments to test the following hypotheses:

- (i) Students are able to code more accurately (with fewer bugs) using the jGRASP data structure viewers.
- (ii) Students are able to find and correct “non-syntactical” bugs faster using jGRASP viewers.

Two experiments focusing on linked lists have already been performed and statistically significant results were obtained [6]. Data analysis from these experiments show that animated verifying viewers increase both accuracy and speed for students during development and debugging of their linked list code. Two follow-on experi-

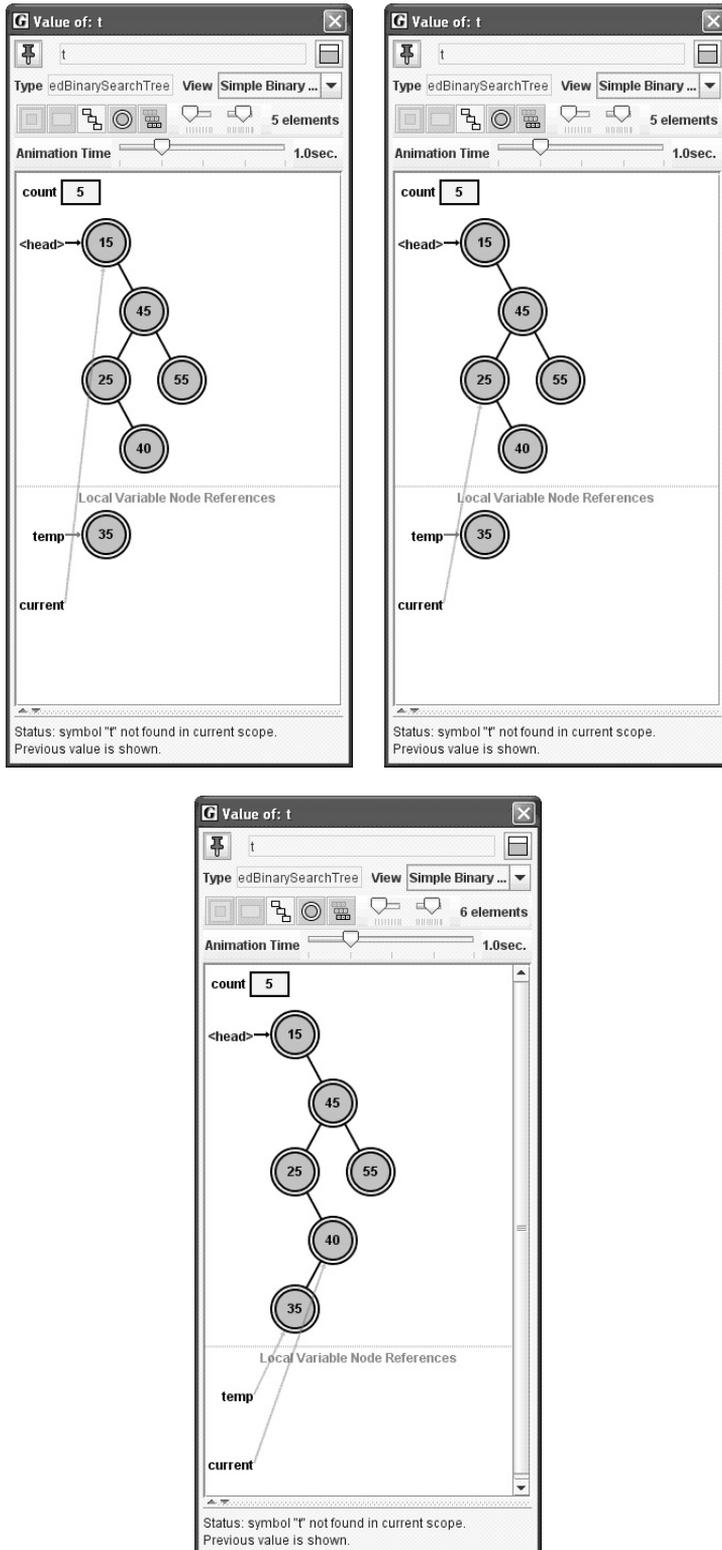


Fig. 3. Snapshots from an animated verifying viewer for a “textbook” binary search tree.

ments have just now been performed, but the data analysis is not complete. These experiments focused on binary search trees rather than linked lists.

6.1 Tree Experiment 1

The hypothesis being tested was that students will be more productive during development (will code faster and with greater accuracy) using the jGRASP data structure viewers. Students were asked to implement one operation for linked binary search trees. The class `LinkedBinaryTree.java` from the class textbook was used in this experiment. Students were provided a detailed description of the programming assignment and the grading policy. Students were required to work independently and were timed (although there was no time limit to complete the assignment). The independent variable was the visualization medium (coding using jGRASP viewers vs. without viewers). The dependent variables were: time taken to complete the assignment, and the accuracy of the assignment.

The control group implemented the method `levelOrder()` using the jGRASP visual debugger without the viewers. The driver program provided to this group contained a `toString()` method so that they could print out the contents of the list without writing additional code. The treatment group implemented the same method using the jGRASP visual debugger with the object viewers. Since our algorithm for `levelOrder()` traversal required three different data structures, we provided the students with three viewers (for `LinkedBinaryTree`, `LinkedQueue` and `ArrayUnorderedList`). The driver program given to this group did not contain the `toString()` method, so the subjects had to use the viewers in order to see the contents of the data structures. The machines in the lab were set up with permissions such that only the treatment group had access to the viewers.

6.2 Tree Experiment 2

Our hypothesis was that students are able to detect and correct logical bugs in less time when using jGRASP viewers. A Java program that implemented a linked binary search tree was provided. The program contained a total of 5 logical errors, one in each of the following five methods `addElement()`, `removeElement()`, `find()`, `preorder()`, and `postOrder()`. Students were asked to find and correct all the logical errors. The independent variable was the visualization medium (finding errors using jGRASP viewers vs. without viewers). The dependent variables were: number of bugs found, number of bugs accurately corrected, and number of new bugs introduced in the program while performing the experiment. Both the groups were first required to identify and document errors. Next, similar to experiment 1, the control group corrected the detected errors using the jGRASP visual debugger without the viewers and the treatment group corrected the errors using the jGRASP visual debugger with the object viewers.

While the data analysis for these two experiments is not yet complete, anecdotal evidence from students suggests that the positive results from the linked list experiments will be replicated in the binary search tree experiments.

7 Conclusion

jGRASP object viewers automatically generate dynamic, state-based visualizations of objects and primitive variables in Java. Multiple synchronized visualizations of an object, including complex data structures, are immediately available to users within the IDE. Multiple instructors have used these viewers in CS1 and CS 2 and have reported positive anecdotal evidence of their usefulness. Formal, repeatable experiments with linked lists have indicated statistically significant positive results on student performance. Follow-on experiments with binary search trees have just been completed, and anecdotal evidence and student feedback suggest that they will yield similar positive results.

References

- [1] Felder, R. and L. Silverman, *Learning and teaching styles in engineering education*, Engineering Education **78** (1988), pp. 674–681.
- [2] Hamer, J., *A lightweight visualizer for java*, in: *Proceedings of Third Program Visualization Workshop*, 2004, pp. 55–61.
- [3] Hendrix, D., J. Cross and L. Barowski, *An extensible framework for providing dynamic data structure visualizations in a lightweight ide*, in: *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, 2004, pp. 387–391.
- [4] Jain, J., N. Billor, D. Hendrix and J. Cross, *Survey to investigate data structure understanding*, in: *Proceedings of the International Conference on Statistics, Combinatorics, Mathematics and Applications*, Auburn, Alabama USA, 2005.
- [5] Jain, J., J. Cross, and D. Hendrix, *Qualitative assessment of systems facilitating visualization of data structures*, in: *Proceedings of 2005 ACM Southeast Conference*, Kennesaw, Georgia USA, 2005.
- [6] Jain, J., J. Cross, D. Hendrix and L. Barowski, *Experimental evaluation of animated-verifying object viewers for java*, in: *SoftViz 2006 (submitted)*, 2006.
- [7] Kannusmaki, O., A. Moreno, N. Myller and E. Sutinen, *What a novice wants: students using program visualization in distance programming course*, in: *Proceedings of Third Program Visualization Workshop*, 2004, pp. 126–133.
- [8] Naps, T., *Jhave: supporting algorithm visualization*, IEEE Computer Graphics and Applications **Sep/Oct** (2005), pp. 49–55.
- [9] Shaffer, C., L. Heath and J. Yang, *Using the swan data structure visualization system for computer science education*, in: *Proceedings of SIGCSE 1996*, 1996, pp. 140–144.
- [10] Zeller, A., *Visual debugging with ddd*, Dr. Dobb's Journal **July** (2001).