Robust Generation of Dynamic Data Structure Visualizations with Multiple Interaction Approaches

JAMES H. CROSS II, T. DEAN HENDRIX, DAVID A. UMPHRESS,
LARRY A. BAROWSKI, JHILMIL JAIN, and LACEY N. MONTGOMERY
Auburn University

jGRASP has three integrated approaches for interacting with its dynamic viewers for data structures: debugger, workbench, and text-based interactions that allow individual Java statements and expressions to be executed/evaluated. These approaches can be used together to provide a complementary set of interactions with the dynamic viewers. Data structure identification and rendering were tested by examining examples from 20 data structure textbooks. Controlled experiments with CS2 students indicate that the viewers can have a significant positive impact on student performance. The overall result is a flexible environment for interacting with effective dynamic data structure visualizations generated by a robust structure identifier.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments, Integrated environments, Interactive environments, Programmer workbench*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Interaction styles*; K.3 [**Computing Milieux**]: Computers and Education

General Terms: Design, Experimentation, Human Factors, Verification

Additional Key Words and Phrases: Program visualization, algorithm animation, data structures

ACM Reference Format:
Cross II, J. H., Hendrix, T. D., Umphress, D. A., Barowski, L. A., Jain, J., and Montgomery, L. N. 2009. Robust generation of dynamic data structure visualizations with multiple interaction approaches. ACM Trans. Comput. Educ. 9, 2, Article 13 (June 2009), 32 pages.
DOI = 10.1145.1538234.1538240. http://doi.acm.org/10.1145.1538234.1538240.

This article is an expansion of articles that we originally presented at PVW 2008 [Cross et al. 2009], ACMSE 2008 [Montgomery et al. 2008], PVW 2006 [Hendrix et al. 2006] and SoftVis 2006 [Jain et al. 2006].
The jGRASP research project is funded in part by a grant from the National Science Foundation.
J. Jain is currently affiliated with Hewlett-Packard Laboratory and L. N. Montgomery is currently affiliated with Radiance Technologies.
Author's address: J. H. Cross, Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849-5347; email: crossjh@auburn.edu.

1. INTRODUCTION

Visualizations of data structures have been used in limited ways for many years. To overcome a major obstacle to widespread use, namely lack of easy access and use, the jGRASP IDE provides *dynamic viewers* specifically intended to support fine-grained understanding of data structures and other objects. The jGRASP *structure identifier* attempts to identify and render traditional abstract visualizations for common data structures such as stacks, queues, linked lists, binary trees, heaps, and hash tables [Cross et al. 2007; 2009]. These are dynamic visualizations in that they are generated while the user's program is running in debug mode. This technique helps bridge the gap between implementation and the conceptual view of data structures, since the visualizations are based on the user's own code.

jGRASP provides three ways for the user to interact with the data structure visualizations: (1) the debugger, (2) the workbench, and (3) a text-based interactions tab.[1] Each of these is briefly described below and more fully with examples in the sections that follow.

The most common way to open a viewer on a data structure object is via the *debugger*. The user simply sets a breakpoint on a statement near the creation of the data structure and runs the program in debug mode. After the program stops at the breakpoint, the user *single steps* as necessary until the object is created, and then opens a viewer on the object by dragging it from the debug tab. As the viewer is opened, the structure identifier determines the particular type of data structure that the object represents and renders it appropriately. As the user steps through the executing program, the viewer is updated to show the effects of each step.

In addition to interacting with the data structure visualizations via the debugger, jGRASP allows the user to interact with viewers via the workbench. Objects can be created and placed on the workbench and their methods invoked via menus and buttons on the UML class diagram and/or the source code edit windows. When the user invokes a method on the object (e.g., to insert a node into a linked list), the visualization is updated to show the effect of the method.

The third approach for interacting with the viewers is a text-based interactions tab, which is essentially a Java interpreter. That is, when the user enters a Java source statement and presses the Enter key, the statement is immediately executed. If an expression is entered, it is evaluated and its value is displayed. This means that while at a breakpoint, the user can interact with any of the variables shown in the debug tab or any variables on the workbench. In addition, if the user creates an instance of a class via the interactions tab (e.g., *LinkedList list = new LinkedList();*), the instance is added to the workbench. A viewer can be opened in the usual way by dragging the reference from the debug tab or workbench. Once opened, the viewer is updated as appropriate when statements or expressions are entered in the interactions tab.

In the following sections, we discuss the context and motivation for the work, and then we provide an extended linked list example to illustrate

[1]The interactions tab was introduced in jGRASP 1.8.7.

the integration of the debugger, workbench, and text-based interactions with the viewers for data structures. This is followed by examples of other common data structures generated by jGRASP. We then provide a brief discussion of the implementation of the structure identifier, workbench, and text-based interactions. This is followed by a discussion of related work. We then discuss our evaluation of the structure identifier's capability to automatically generate data structure diagrams from examples that accompany data structure textbooks. This is followed by our evaluation of the data structure viewers based on controlled experiments with CS2 students. We close with a brief summary of the article.

2. CONTEXT AND MOTIVATION

Both the method and degree of user interaction with software visualizations have been shown to be primary contributors to their effectiveness. Research indicates that passive modes of interaction with visualizations, for example, only watching an animation of an algorithm's behavior, are not as effective as more active engagement strategies such as having the user manipulate elements of the visualization or respond to prompts [Stasko and Lawrence 1998; Naps et al. 2002; Lauer 2006]. The context in which the visualization appears has also been shown to play a vital role in effectiveness [Hansen et al. 2002]. These issues are now widely seen as fundamental to the advancement of software visualization research in education [Naps 2000; Naps et al. 2005].

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 and the user should be allowed to interact with the visualizations through commonly used features of the IDE. We hold the pragmatic view that much of the learning, and indeed much of the opportunity for learning, occurs while a student is actively working on a source code implementation of an algorithm or data structure. Thus, we provide software visualizations that are directly coupled to real source code in the context of a full-featured IDE. jGRASP is not a standalone, special-purpose visualization tool that is only used at certain times or in certain settings; rather, it is a complete, lightweight development environment that is used consistently throughout a course, in lecture, in lab, and at home.

The three basic software visualization interaction techniques that jGRASP offers are based directly on common tools and idioms available in various IDEs: the debugger, object workbench, and interactions tab. The debugging process and the use of a debugger are central to almost all courses in which programming is involved, and debugging is supported by almost all IDEs. In addition, the use of a debugger as a pedagogical tool and as an important application area for software visualization has long been recognized [Mukherjea and Stasko 1994; Baecker et al. 1997; Cross et al. 2002]. The "object workbench" paradigm has been made popular by BlueJ [Kölling et al. 2003], a popular IDE for early programming courses. In BlueJ, students use menus and dialogs to create object instances for the workbench and to invoke methods on those instances, without the need for a running program. In jGRASP, this notion of

workbench has been extended to allow the user to open type-specific viewers on objects or primitives, as well as a viewer that identifies data structures on the fly. DrJava [Allen et al. 2005] has demonstrated the utility of an "interactions tab" for experimenting with Java source statements and expressions without the necessity of compiling and running a complete program. This is especially useful in a classroom setting. In jGRASP, we have extended the basic notion of the interactions tab in several ways. Most importantly, it is fully integrated with the debugger, workbench, and viewers.

We have specifically avoided basing the visualizations in jGRASP on a scripting language, which is a common approach for algorithm visualization systems such as JHAVÉ [Naps 2005]. We also decided against requiring modifications to the user's source code as is the case in systems such as LJV [Hamer 2004] The approach we have taken for data structure viewers in jGRASP is to automatically generate the visualization from the user's executing source code and then to dynamically update it as the user steps through the source code in either debug mode or workbench mode, or as statements are executed in the interactions tab. 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 visually 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.

An overview of related work and other visualization systems is given in Section 5. Jain et al. [2005] provides an overview of many relevant systems, along with a comparison to the features available in jGRASP.

3. INTEGRATED USER INTERACTION IN jGRASP – AN EXAMPLE

To see a meaningful visualization of a data structure in a viewer requires that an instance be created and its methods be invoked. The integrated approach in jGRASP allows this to be done by (1) using the debugger in the traditional way, (2) using the workbench menus and dialogs from the UML window or the edit windows, and (3) entering source code into the interactions tab for direct execution. In each of these approaches, the viewer is opened on an instance by dragging it from the debug or workbench tab. In this section, we begin by using the debugger approach and then we show how the user can interact with the visualization using both workbench and text-based approaches.

3.1 Using the Debugger

Consider LinkedListExample.java, which is provided with the jGRASP distribution. The UML class diagram in Figure 1 shows the dependencies among LinkedListExample, LinkedList, and LinkedNode. The LinkedList class is intended to be representative of what a student or instructor may write, or of a textbook example. Figure 2 shows LinkedListExample.java stopped at a breakpoint where the *list.add* method is about to be invoked. Prior to stopping at the breakpoint, *list* was assigned an instance of LinkedList, and thus it appears in the Variables tab of the Debug window. To open a viewer on *list*, the user

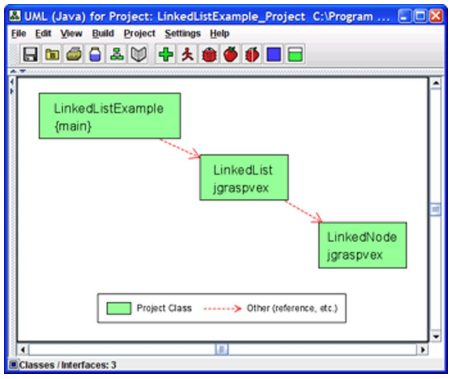

Fig. 1. UML diagram for LinkedListExample.

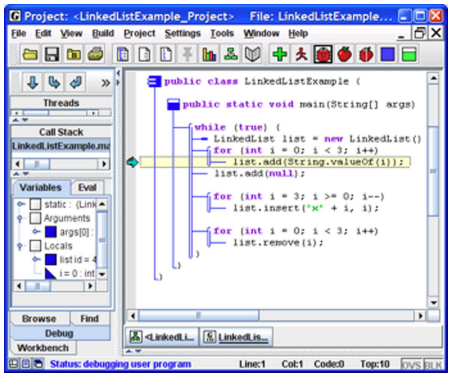

Fig. 2. Method main at breakpoint.

drags it from the Debug window. During the process of opening the viewer, the structure identifier mechanism in jGRASP determines, in this case, that the object is a linked list structure and opens the appropriate viewer. As the user steps through the program, the nodes appear in the viewer. Figure 3 shows the visualization of *list* after three elements have been added.

The debugger approach is perhaps the most natural way for students to interact with the viewers. Since it involves stepping through a program, the visualization reinforces or clarifies the effect of each step by providing live feedback at a conceptual level. For example, Figure 4 shows the program after stepping into the *insert* method. The statement about to be executed will complete the linking of *node* into the data structure. Figure 5 shows the synchronized view of *list* at this point in the program. The visualization indicates that *prev.next* is pointing to the last node in the structure (labeled "<3>"). As soon as the statement "*prev.next* = *node;*" is executed, the new node with value "x3" will move up into the structure in an animated fashion. The visualization in the viewer allows the student to make the connection between the

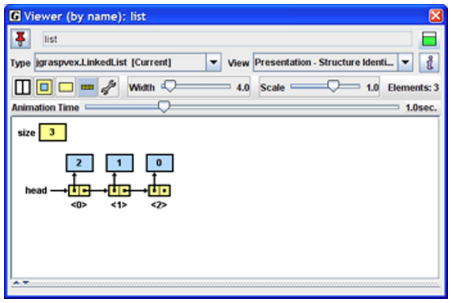

Fig. 3. View of list after three elements added.

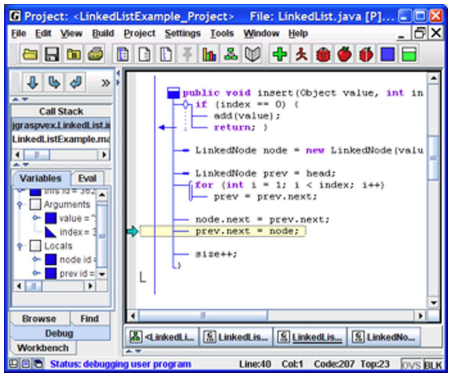

Fig. 4. Stepping in list.insert method.

implementation of the insert method and the concept of inserting an element in a linked list.

3.2 Using the Workbench

In the workbench approach for interacting with data structure visualizations, the user creates one or more instances of the class using menus or buttons on the UML class diagram or source code editing windows. From the UML diagram, this can be done by selecting *Create New Instance* on the right-click menu for the class, which pops up an object creation dialog for class LinkedList as shown in Figure 6 below. When the instance is created, it appears on the workbench tab. As before, a viewer is opened on the instance by dragging it from the workbench. The method invocation dialog for the instance is popped up by either right-clicking on the instance and selecting *Invoke Method* or by clicking the *Invoke Method* button on the viewer (upper-right corner). Figure 7 shows the method invocation dialog for *jgraspvex_LinkedList_1* with the *add* method selected. When a method is invoked, the viewer is updated to show the new state. Thus, the user can interact with the data structure visualizations in the viewer by invoking a sequence of *add*, *insert*, and *remove* methods. The operations just described could have been performed alone or

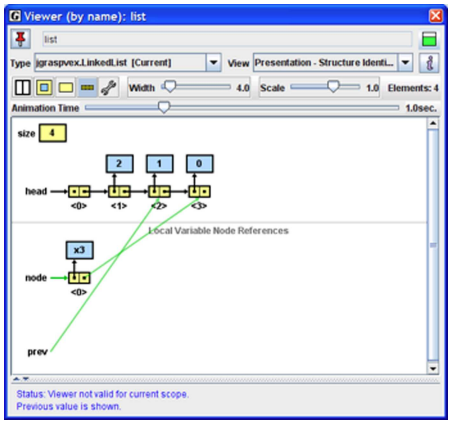

Fig. 5. View of list showing details of insert.

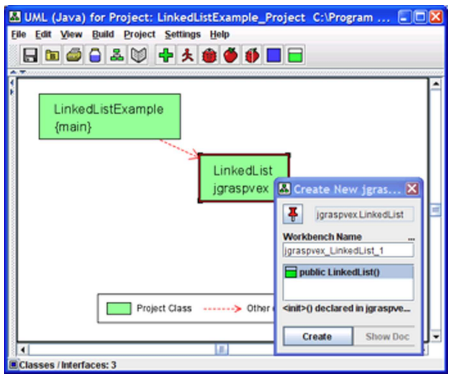

Fig. 6. Creating instance from the UML diagram.

in the context of a running program. In the latter case, the operations could also have been performed on an instance of LinkedList created by the program itself.

3.3 Using Text-Based Interactions

The interactions tab in jGRASP provides a Java interpreter that allows the user to enter expressions and statements and have them immediately evaluated and executed respectively. To use this feature in a standalone fashion with respect to data structure visualizations, the user could enter the code to create an instance and then enter statements that invoke methods on the instance. The advantage of this approach is that it allows the user to enter actual Java statements and execute them without having to enter and run an entire program, though the interactions tab can also be used to interact with elements in a running program. The integration with the workbench means that when a variable is declared and initialized

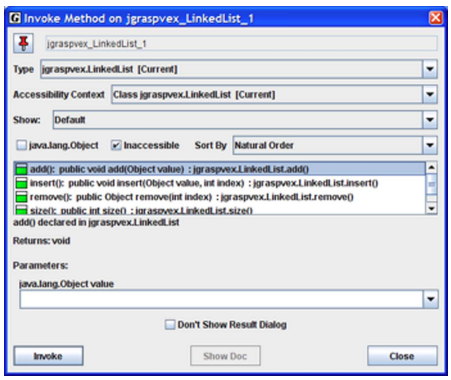

Fig. 7. Method invocation dialog.

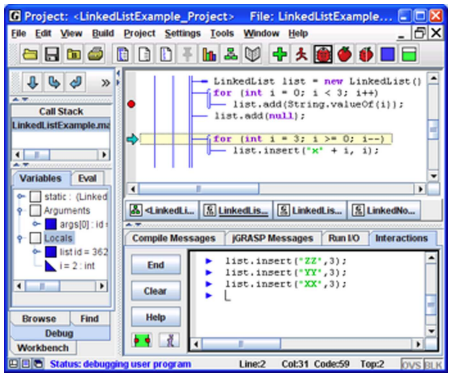

Fig. 8. Text-based interactions.

(e.g., *LinkedList myList = new LinkedList();*), it appears on the workbench. This allows the user to open a viewer on the variable. Once the viewer is opened, any effects of the statements entered in the interactions tab involving the methods invoked on the variable will be seen in the viewer. To illustrate the degree of integration of the text-based interactions, consider our original example that was running in debug mode. Figure 8 shows *LinkedListExample* stopped at the second *for* loop. Three *list.insert* method calls are shown in the interactions tab. As these statements were entered, "ZZ", "YY", and "XX" respectively were inserted at Location 3. As each node was entered, the existing nodes in the viewer were moved to the right with the final result shown in Figure 9.

3.4 Examples of Other Data Structures

Examples of visualizations recognized by the data structure identifier mechanism in jGRASP are shown in Figures 10 through 15. Figure 10 is a view of a typical circular queue implemented with an array. Figure 11 shows a view

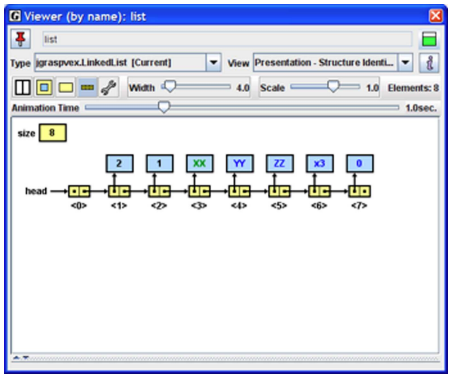

Fig. 9. Viewer for list after interactions.

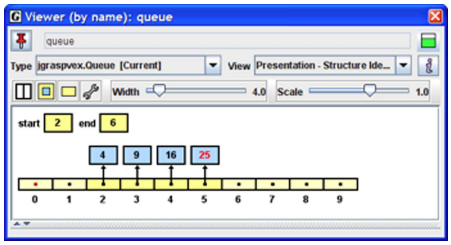

Fig. 10. View of a typical queue.

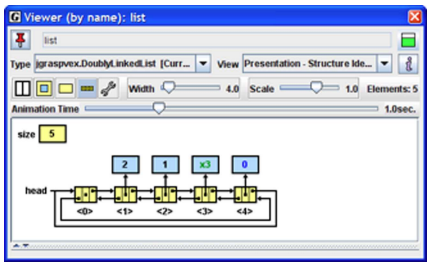

Fig. 11. View of a typical doubly linked list.

of a doubly linked list after an empty node was added to the front of the list. Figure 12 is a view of a binary tree implemented with linked nodes. Figure 13 is a view of an array-based implementation of a binary heap. Figure 14 shows an instance of TreeMap from the Java Collections Framework which is implemented as a red-black tree. Finally, Figure 15 is an instance of a hash table

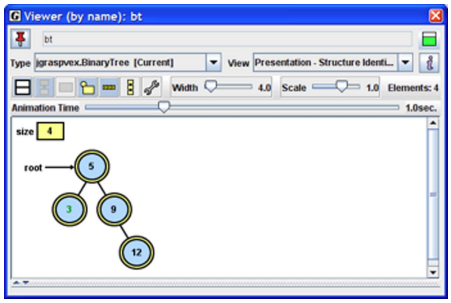

Fig. 12. View of a typical binary tree.

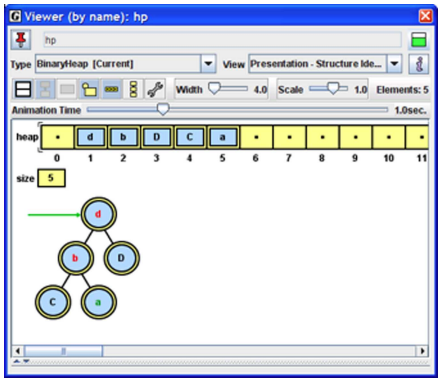

Fig. 13. View of a typical binary heap.

implemented as an array of linked lists. Each of these views was generated from the example source code provided with the jGRASP distribution.

4. IMPLEMENTATION

The procedure used by the structure identifier to recognize data structures in an executing program is briefly described in this section. This is followed by a short discussion of how the workbench was implemented. We conclude with an overview of the implementation strategy used for the text-based interactions.

4.1 Structure Identifier

The structure identifier, which is automatically invoked when a viewer is opened, works as follows. For common node-and-link implementations of structures, where nodes are objects and links are object references or accessor methods, automatic identification is done by examining class structure, and by examining links in the instance that is about to be viewed. A class and its fields and methods are first examined for same-class references, and possible structure mappings are considered. For example, a singly linked list is

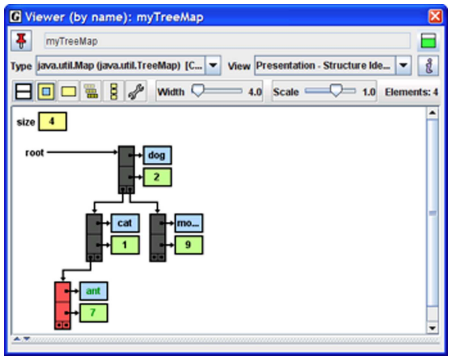

Fig. 14. TView of a typical red-black tree.

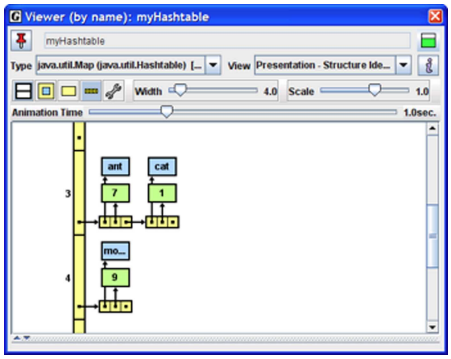

Fig. 15. View of a typical hash table.

typically implemented as a class with a field (the head node link) whose class type has one same-class reference (to the next node). This technique may lead to multiple possible structures and to multiple possible mappings from a class to a particular structure, each of which is assigned a confidence level.

The structure mapping with the highest confidence level found during automatic identification, if it is significantly higher than the confidence level for other potential mappings, will be automatically used when a viewer is first opened for a particular class. In most cases, one and only one mapping with a high confidence level will be found, and thus the mechanism will be transparent to the user. That is, an appropriate structural view will be displayed without user interaction. In cases where there are multiple mappings with similar confidence levels or where no mapping is found, the user is given the option of manually configuring the viewer (this can also be done while the viewer is in use). A configuration dialog allows the Java expressions that will be used to traverse the structure to be entered or edited. For example, for a singly linked list, expressions for the head node, next node (given a node), and

display value (given a node), are required. Any mappings that were found during the automatic analysis are made available on a drop-down list. Once the structure mapping has been selected, specified, or modified using this dialog, the new mapping will automatically be applied the next time the user opens a viewer on an instance of the same class.

The nodes used in the structure mappings need not be actual node objects in the structure. Using synthetic node values allows structures where nodes are not individual objects (or links are not object references or method return values) to be displayed. For example, a binary heap is typically implemented using an array of node values and a size value. The links are implicit. The integer index of a node value can be used as the node in the mapping expressions. This allows the implicit binary tree to be mapped and displayed as a binary tree. Automatic identification of such structures is done using name-based heuristics (currently only for English) and by examining instance characteristics for consistency with the expected structure. The heuristics are necessarily more restrictive than for node-and-link implementations, since the possible mappings are more common. Any class with an array field and an int field, for example, might be a binary heap. Unless the class and field names are suggestive of a binary heap, such a possible mapping will be ignored.

The data structure identifier mechanism, which provides for both the identification and rendering of common data structures, has been greatly improved by examining examples from data structure textbooks. The detailed results of the testing are described in Section 6.

4.2 Workbench

The workbench mechanism gathers class and method information and performs method execution and instance creation using the Java Debugger Interface (JDI). Method and constructor arguments may be any valid Java expression. These expressions are evaluated using the Java interpreter described in the following subsection. During debugging, no workbench code is "injected" into the target process, so that unintended interference with its normal operation is minimal.

Unfortunately, the JDI does not allow recursive method invocation (including instance creation). That is, when the program being debugged is at a breakpoint, a method may be invoked through JDI. If another breakpoint is encountered during that invocation, another JDI method invocation is not possible from that point. Because the jGRASP viewers, eval mechanism, and text-based interactions require method invocation through the debugger interface, we chose to ignore any breakpoints encountered during workbench method invocation. During debugging, stopping at such a breakpoint would generally not be desired anyway, but when the workbench is used in standalone mode, it is natural to think of the workbench actions performed by the user as the target program. That is, the user should be allowed to debug workbench actions as they would a normal program. To allow this, an outer layer of workbench method invocation implementation was added. This outer layer performs workbench actions through the Java reflection mechanism, using

code that is part of a shell program that runs when jGRASP is in workbench mode or text-based interactions mode. The JDI is used to issue commands to and receive results from the shell program. For the purposes of the workbench, this mechanism behaves identically to the direct JDI mechanism.

4.3 Interactions

The Java interpreter used for text-based interactions parses the entered code and builds a corresponding executable parse tree. The code may be an expression, a statement, or a sequence of statements. If the parse is unsuccessful, a compile time error is reported; otherwise, the tree is executed. If an exception propagates to interactions scope during execution and is not caught within interactions code, it is reported as a runtime error. Otherwise, if the code is an expression, the result of evaluating that expression is displayed. The same interpreter, with input restricted to expressions, is used to evaluate arguments supplied to the workbench invoke method dialog, expressions used by the object viewers, etc.

When the interactions tab is active, there is always a target process that is running in debug mode. When used in standalone mode or workbench mode, this target will be the workbench shell program described in the preceding subsection. Object and primitive values are created and stored in the VM of the target. Types are references to actual types in the target, or where needed for evaluation, synthetic types composed of references to actual types in the target. All operator evaluation takes place in the interpreter. No classes are injected into the target, so that when text-based interactions are used in conjunction with debugging, unintended interference with the normal operation of the target process is minimal.

The interactions interpreter works for all Java statements and expressions. Currently, it does not allow class or interface definitions, including anonymous class definitions. Java generics are not supported. In the near future we plan to add static method definitions at interactions scope, so that users may declare and use methods that will exist only within the interactions tab. Class definitions and generic type evaluation will also be implemented in a future version. Apart from this, the interpreter rigorously follows the specification provided by the Java Language Specification, Third Edition [Gosling et al. 2005], and in fact, several problems with the specification were found in the process of construction and testing (Sun bug ids 6684387, 6619380).

5. RELATED WORK

Software visualization research has been directed toward two main areas of interest: (1) program visualization, in which source code, data structures, or runtime behavior is represented, and (2) algorithm animation, in which views are provided of conceptual behavior at the algorithmic rather than the implementation level [Shaffer et al. 2007; Karavirta et al. 2006; Stasko and Lawrence 1998; Mukherjea and Stasko 1994; Price et al. 1993; Myers 1990]. Whether an animation is produced from source code or independently of any implementation, its primary purpose is to increase the comprehensibility of the underlying

algorithm and the associated program behavior. Extensive research has been carried out in an effort to understand the effects that algorithm animations have on learning and comprehension. Although some results were negative, it has generally been shown that appropriate animations can be effective aids to student learning, given the right circumstances [McNally et al. 2007; Lauer 2006; Parker and Mitchell 2006; Rößling et al. 2006].

5.1 Visualization Tools and Systems

The following subsections provide an overview of other work in software and algorithm animation systems in an effort to put our work in an appropriate context. Since this body of work is quite large, the overview is intended to be only representative, not exhaustive. The overview is divided into three sections. The first discusses general-purpose animation systems; the second focuses on systems developed specifically for animating data structures; the third presents animation systems designed to support program debugging and faultfinding.

5.1.1 *General Purpose Software and Algorithm Animation Systems.* Balsa [Brown and Sedgewick 1985] and Zeus [Brown 1991] are examples of early algorithm animation systems that were highly influential. Users of these systems create visualizations by inserting calls to the animation system directly into the source code being visualized. Tango [Stasko 1990] and its successor POLKA [Stasko and Lawrence 1998] were based on this work. Tango and POLKA pioneered the interesting events model of producing algorithm animations via annotated source code. An interesting event denotes a point in the algorithm at which the animation needs to react or change. Each interesting event is animated by visualization code inserted into the program's source code. This allows sophisticated visualizations to be produced, but the learning curve can be rather steep, especially for students.

ANIMAL [Rößling and Freisleben 2002] is an algorithm animation system that allows general-purpose animations to be constructed via its built-in language ANIMALSCRIPT [Rößling and Freisleben 2000; Rößling et al. 2004]. Although this approach requires users to learn a new syntax and develop animations by writing commands in this language, the result is highly flexible. ANIMAL and its scripting language have been refined and extended to provide more power and flexibility while making constructing animations faster and easier [Rößling et al. 2004; 2007].

ALVIS Live! is an algorithm development and visualization environment targeted to novice programmers [Hundhausen and Brown 2007]. The visualizations that it creates are termed "low fidelity" [Hundhausen and Douglas 2002] since the intent is to visualize algorithm behavior at a high level under specific input conditions. The development of ALVIS was informed by the results of empirical research in education [Hundhausen and Douglas 2002], and by the goal of supporting a studio approach to the teaching and learning of algorithms [Hundhausen and Brown 2007].

The Jeliot family of program visualization systems [Moreno and Myller 2003; Ben-Ari et al. 2002; Ben-Bassat et al. 2003] was developed to visualize

the data and control flows of programs. Jeliot 3 [Moreno et al. 2004] visualizes the interpretation of a Java program, showing method calls, variables, and operations so users can follow the execution of program in a step-by-step fashion. The Jeliot family is targeted primarily to novice programmers [Ben-Bassat et al. 2003].

5.1.2 *Software and Algorithm Animation Systems for Data Structures.* JIVE [Gestwiki and Jayaraman 2004] takes the same approach to producing data structure visualizations as JVALL [Dershem et al. 2002], but provides a much richer environment. JIVE includes a collection of Java classes that implement prebuilt animated data structures compatible with standard Java classes from the JDSL (jdsl.org), such as hashtables, graphs, and search trees. In addition to supporting several different animated data structures, JIVE also provides animation speed control and a novel zooming interface that allows a user to zoom in and out on large data structures. JIVE automatically creates a visualization of any program that uses one of its animated data structure classes. Users can also choose to instantiate animated data structures in isolation and interact with the animation through a graphical user interface. A distributed virtual learning environment is provided to allow multiple users, separated into teachers and students, to interact with the same animated algorithm or data structure.

SWAN [Shaffer et al. 1996] is a visualization system designed for creating animations of data structures in C and C++ programs. Rather than packaging pre-animated data structures for immediate use, SWAN allows users to create their own animations for arbitrary programs. Users must annotate source code with calls to the animation system. A separate component allows an annotated program to be viewed as an animation. Although this approach to animation is similar to that used by POLKA, SWAN is designed to be much more compact, simple, and easy to use. Unlike many animation systems, SWAN allows the user to modify the underlying data structure by interacting directly with the animation.

JAWAA [Akingbade et al. 2003; Pierson and Rodger 1998] is a scripting language for easily creating Web-based animations. Although JAWAA can be used for general-purpose animation, common data structures such as stacks are directly supported, thereby making data structure animation straightforward. To create an animation, JAWAA commands are stored in a text file which can be created by hand or produced as the output of a program. This text file is then called from a Web page which produces the animation. JAWAA is language independent and programming experience is not required. An editor is provided to allow animations to be laid out by creating graphical objects and then showing how they change over time.

LIVE [Campbell et al. 2003] is an animation system designed to produce visualizations of arbitrary data structure definitions. The system also supports viewing a given visualization in multiple languages. The graphical user interface to LIVE provides a source code window where source code can be displayed and edited. An associated canvas window allows the user to position, size, and arrange graphical objects that to correspond to the source code. The

source code can be run (interpreted) and the animation will be automatically displayed. Direct manipulation of the animation is supported, with the source code being automatically updated appropriately.

SKA [Hamilton-Taylor and Kraemer 2002] was designed to specifically address the needs of data structures instructors and students, rather than for advanced graphical capabilities. The focus on user-centered design makes SKA unique among most animation systems. SKA is composed of an extensible Java library of prebuilt animated data structures, a data structure diagram manipulation environment, and an algorithm animation system. SKA can create and manipulate instances of the animated data structures independently of any algorithm or source code, and it can also display animated algorithms. To animate an algorithm, all data structures are replaced with equivalent animated ones from the library, and a source code annotation model similar to POLKA is used.

MatrixPro [Karavirta et al. 2004; Karavirta 2007] is a visualization tool for both demonstrating and simulating algorithms. It is built on the Matrix framework, which allows users to manipulate basic data structures such as arrays, linked lists, trees, and graphs. Users can create MatrixPro animations on the fly as well as export them for later playback.

BlueJ is an IDE designed for teaching introductory courses in Java using an objects-first approach [Kölling et al. 2003]. Although it does not provide dynamic animations of runtime behavior, the workbench feature of BlueJ has been used as a visual inspection mechanism for data structures in a CS2 course [Paterson et al. 2005].

5.1.3 *Software and Algorithm Animation Systems for Debugging.* Early data structure visualization systems include Provide, Amethyst, DS-Viewer, Saber C, and VIPS [Moher 1988; Myers et al. 1988; Pazel 1989; Shimomura and Isoda 1991]. Many of these were either built directly on top of debuggers or used to aid the debugging process. The systems discussed in this section illustrate the varying design approaches that are used in visual debugging systems today.

DDD [Zeller and Lütkehaus 1996; Zeller 2001], a front-end for command-line debuggers, offers simple visualization of linked structures. The visualizations are generated directly from information available from the debugger and thus no source code annotation is required. Users control the visualization by setting breakpoints and through other standard debugger operations. Direct manipulation of the visualized data structures is also supported (e.g., expanding a linked list one element at a time by clicking on the next field of each node). Simple node layout is automated, but the user is allowed to reorganize the visualizations by dragging and dropping nodes. DDD can visualize arbitrary references between data, not just pointers. For example, the relation between an array element and the data it contains can be visualized.

Lens [Mukherjea and Stasko 1994] is an attempt to bridge the gap between debugger-based systems such as DDD and sophisticated algorithm animation systems like POLKA. Data structure visualization systems that rely completely on debugger information to produce the visualizations do not

have the capability to integrate the rich semantics of the program behavior that only a human animator could supply. Lens integrates debugger-based visualization with the interesting event annotation model. Interesting event animation commands are attached to debugger breakpoints, and thus dynamic animation-style data structure views can be created using a combination of debugger information and user control.

Criticisms of applying the interesting event model to debugger-based systems (as in Lens) include the difficulty in identifying appropriate segments of code to annotate and the inability to visualize data structures that have been created in an already running program prior to being debugged [Korn and Appel 1998]. Travis is a data structure visualization system that is built on top of a debugger (like DDD) and offers user-customizable graphical displays (like Lens). Travis is not based on the interesting event model like Lens, however. Instead it uses a traversal-based visualization scheme in which the debugger traverses a data structure and produces a visualization based on user-supplied patterns that identify how particular parts of the data structure should be displayed [Korn and Appel 1998]. Travis provides a graphical user interface for specifying these patterns. Direct manipulation of the visualization is also supported. That is, if a user modifies the data structure diagram, the underlying program objects are modified accordingly.

5.2 Visualization and Pedagogical Effectiveness

Numerous studies have been performed to address the effectiveness of software visualizations [Hundhausen et al. 2002]. Naps and others contend that unless a visualization system engages students in an active learning activity it will have little educational value [Naps et al. 2002]. A taxonomy of learner engagement with visualization tools was a significant result of this work. They go on to describe metrics for measuring a visualization's effectiveness and provide example experiments that could be performed.

It has also been argued that the effectiveness of a visualization depends on the attitudes and behaviors of instructors as well as its effect on students [Naps et al. 2003]. That is, not only must there be positive student learning outcomes, but there must also be widespread use of the visualization by the instructor. This issue was identified as a prime contributor to the disconnect between the widely perceived benefits of visualizations and the relatively low incidence of use in the classroom. A phenomenographic study of instructor use of a program visualization system identified tool integration as a primary factor in determining the degree to which an instructor will use a visualization tool [Ben-Bassat Levy and Ben-Ari 1994]. The tool must be integrated with other course materials, and it must be integrated with various course activities so that both the instructor and the student use it as a natural part of their roles in the course.

Narayanan and his students have developed a theoretical framework for pedagogically effective algorithm visualizations, in which analogies and animations are embedded within the context of a knowledge-rich hypermedia environment. HalVis, an algorithm visualization system based on this

framework, was shown to be superior to both traditional methods of instruction and algorithm animations representative of extant research in an extensive series of experiments [Hansen et al. 2002]. In subsequent empirical work, computer-supported collaborative construction and critiquing of algorithm visualizations by students was also found to lead to superior learning [Hübscher-Younger and Narayanan 2003]. These results suggest that well-designed visualizations can indeed enhance student learning of algorithms.

jGRASP applies these research results to the teaching and learning of programming by integrating visualizations with IDE functionality. That is, it offers a high degree of engagement with visualizations [Naps et al. 2002], it can be easily and directly integrated with other course materials (textbook, course notes, program development), it can be used by instructors during lectures and by students for assignments, and it provides students with an appropriate and immediate context (the source code and comments), tightly integrated with visualizations.

6. EVALUATION OF DIAGRAM GENERATION

The utility of the data structure viewers is determined by two premises: (1) that the diagrams can be generated automatically from student programs and (2) that the diagrams are useful. In this section, we examine automatic generation, and in the next section we consider the potential impact of using the diagrams.

Ensuring the robustness of the structure identifier [Cross et al. 2007] is critical to the overall success of the integrated approaches described above. Since student programs are usually patterned after textbook examples, we reasoned that if the structure identifier worked for most textbook examples, it would also work for most student programs. To this end, we identified 36 data structures textbooks to consider for testing (see Appendix A). Among these, the 20 textbooks that included working source code for each of stacks, queues, lists, and binary trees were selected for testing. Heaps and hash tables were also found in most of these 20 textbooks and these were included in the testing.

The source code for each of the textbooks selected was acquired from the publisher's or author's website. Although 16 textbooks were not selected because sample code was not readily available or was incomplete, the source code from the textbooks that were selected is thought to be representative. Our test plan for evaluating each textbook's examples included selecting data structure classes for each of the six data structure types. A single test harness was created for each textbook so that its data structures could be easily tested and retested [Montgomery et al. 2008].

6.1 Scoring

Each data structure example was evaluated in jGRASP and given one of four scores. The four outcomes included *pass*, *pass – open on field*, *pass – configure viewer*, and *fail*. *Pass* means that the jGRASP viewer accurately displayed all features of this data structure correctly without any user interaction. *Pass – open on field* indicates that although the data structure itself was not

Table I. Total Values of Each Data Structure Tested for the 20 Complete Textbooks

	Pass	Pass – open on field	Pass – configure viewer	Fail
List	36	0	1	0
Queue	20	4	0	1
Stack	20	4	0	0
Tree	28	4	0	0
Heap	11	1	1	2
Hash Table	6	2	6	1

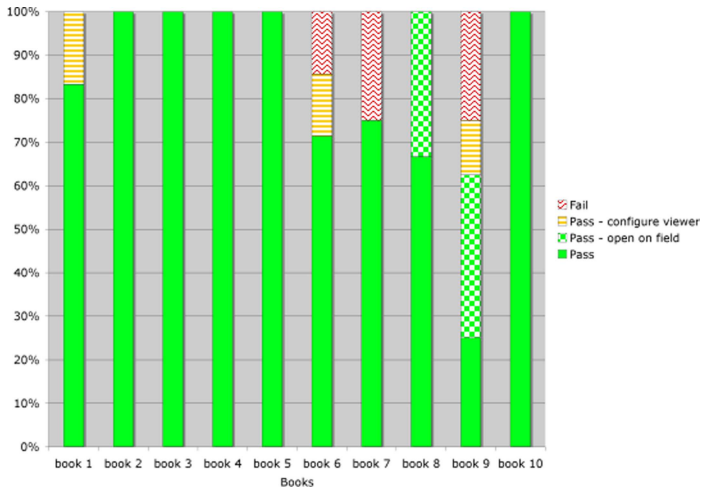

Fig. 16. Chart showing the outcome of the first ten books.

identified directly, the user could open a viewer on an appropriate field of the data structure and view the entire data structure correctly. With minor tuning the structure identifier should be able to identify these directly in a future release. *Pass – configure viewer* means that the structure type was identified, but the structure identifier had to be configured by the user via the *Configure* option on the viewer's toolbar in order to display the structure properly. *Fail* indicates that the structure identifier was not able to correctly identify the data structure in the program or that no meaningful configuration was apparent. These four outcomes are described below. Section 6.3 includes examples of *Pass – open on field* and *Pass – configure viewer*.

6.2 Results

Table I shows the number of outcomes in each category by data structure for the 20 textbooks combined. Figure 16 shows the results for textbooks 1 - 10 and Figure 17 shows the results for textbooks 11 - 20. Each bar represents the results for all of the data structures for a specific textbook. For each of the 20 textbooks tested, the percentages for *pass*, *pass – open on field*, *pass – configure viewer*, and *fail* are shown. The pie chart in Figure 18 displays the overall results of all of the data structures that were tested. The averages (over

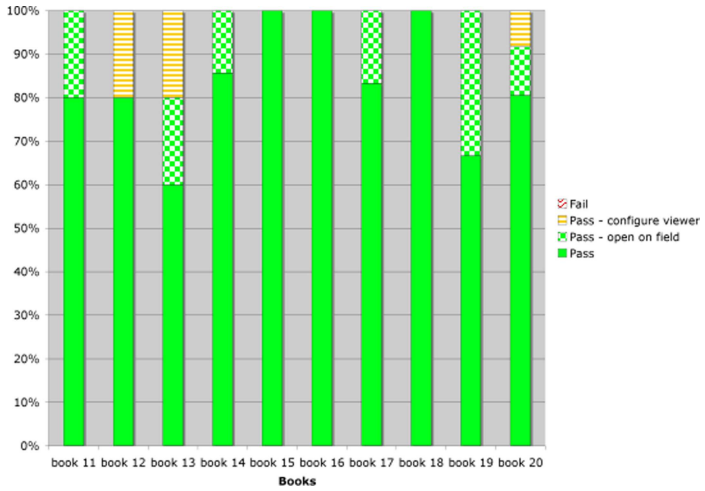

Fig. 17. Chart showing the outcome of the second ten books.

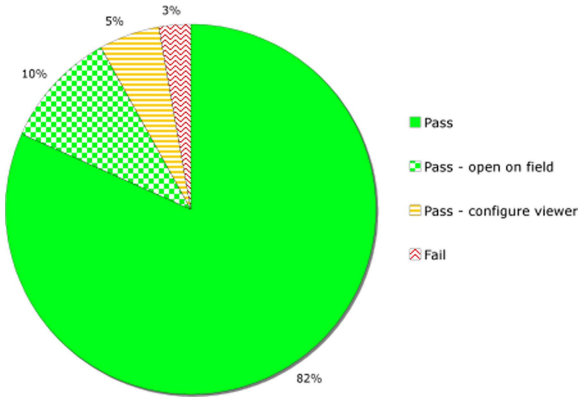

Fig. 18. Total results for 20 textbooks.

all 20 books) *pass*, *pass – open on viewer*, *pass – configure viewer*, and *fail* were 81.76%, 10.14%, 5.41%, and 2.70% respectively.

Finally, Figure 19 shows the results by type of data structure: stack, queue, list, tree, heap, and hash table over all of the data structures that we tested. For example, of all the heaps that were evaluated in the 20 textbooks, 73.33% *passed*, 6.67% *passed – open on field*, 6.67% *passed – configure viewer*, and 13.33% *failed*. The only structure identifier failures occurred with queues, heaps, and hash tables. The testing for all others resulted in one of the pass categories.

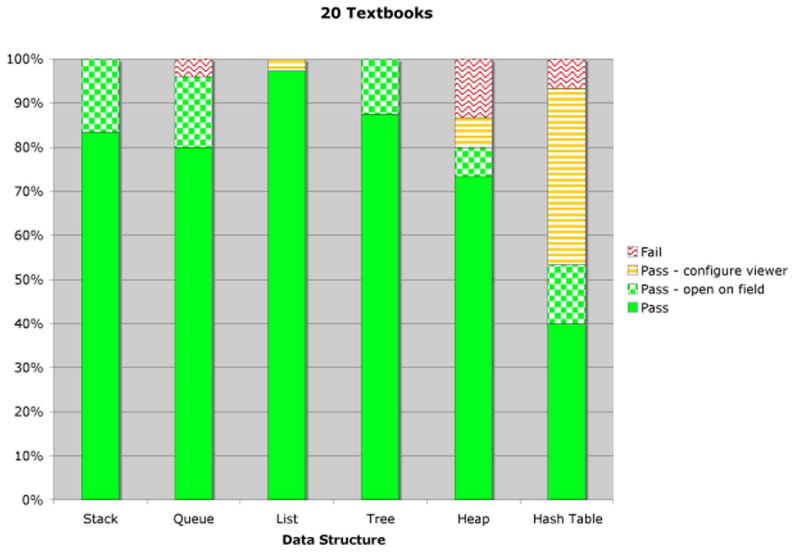

Fig. 19. Chart that displays the outcome per data structure for the 20 books.

6.3 Tuning the Structure Identifier

In cases where the data structure was not automatically identified, one of the most common problems involved wrapped data structures, where jGRASP would correctly identify the internal data structure but not the wrapper containing it. For example, if *myStack* was an instance of a stack, which has a linked list data field as opposed to subclassing a linked list, the structure identifier may not have identified it as a stack. However, if the user had opened a viewer on the linked list field and viewed it dynamically, it would have behaved as a stack, which is the same view that would have been displayed had *myStack* been recognized directly as a stack. By adding a deeper search for wrapped structures and better recognition of wrapper classes, these problems can be reduced. Because classes often contain data structure fields for which they are not wrappers and may have misleading field names, complete elimination of the problem is not possible.

In cases where viewer configuration was necessary to correctly display the structure, the most common problem was that the "value" field or method of the node type was not identified. In these cases, the viewers display a question mark as the value. Less commonly, the value field or method was misidentified. The usual reason for this is that the data structure classes contain several fields/methods that might possibly be considered to contain/provide the value (note that for types with multiple values, such as map structures with keys and values both can be identified and displayed automatically). In these cases it would be fairly easy for the user to use the configuration dialog to correct this. Through tuning of the structure identifier, the likelihood of this problem can be reduced, but due to the nature and variety of data structure implementation code, it can never be entirely eliminated.

In cases where the data structure was not automatically identified and not because of the wrapper issue discussed above, a common problem was that the data structure was implemented using a unique approach. For example, one author used a Lisp-like recursive implementation of linked lists. Many of these cases can be recognized by adding new mechanisms to the structure identifier. Misidentification of known structure implementations is rare for the straightforward examples generally used in textbooks.

7. EVALUATION OF THE DATA STRUCTURE DIAGRAMS

Numerous experiments have been conducted in the field of visualization of data structures and algorithms [Hundhausen and Douglas 2002]. These studies primarily concentrate on determining factors that affect the quality of pedagogical effectiveness using visualization techniques or on determining whether learning is enhanced using a particular system. There is yet a need for tools that can assist students in their transition from understanding a concept to being able to implement it. jGRASP viewers are designed to address this deficiency.

We conducted four controlled experiments to investigate the effect, if any, that the data structure visualizations in jGRASP had on student performance on certain programming tasks [Jain et al. 2006; Hendrix et al. 2007]. The first two experiments focused on singly linked lists, and were performed near the beginning of a CS2 course just after linked structures had been introduced. The next two experiments focused on binary search trees near the end of the same CS2 course, after the students had experience working with more complex data structures and algorithms. All four experiments were designed to test the following hypotheses:

Hypothesis 1. Students are able to code more accurately (with fewer bugs) using the jGRASP data structure viewers.

Hypothesis 2. Students are able to find and correct "non-syntactical" bugs more accurately using jGRASP viewers.

7.1 Subjects

Two criteria were important when choosing subjects for our controlled experiments. First, the subjects must be a close representation of the target population. The jGRASP viewers are being developed primarily for students enrolled in an introductory level data structure and algorithms course. Students enrolled in the CS2 at Auburn University were used as subjects since they closely represent the target population. Second, the subjects must be relatively uniform in regard to their programming abilities in order to minimize the variance between groups.

We designed experiments that were closely integrated with course requirements and that complemented the lab assignments. For example, we conducted two experiments on singly linked lists, and assigned programming projects on doubly linked lists. Students completed eight in-lab activities related to the experiments as a part of the CS2 course. All in-lab activities

were conducted during the respective lab time of each section in a particular computer lab on campus. This ensured control over the hardware and software used by the subjects, and ensured that the schedule of experiments did not conflict with the subjects' coursework.

We designed experiments based on the between-group approach to avoid the transfer of concepts learned in early experiments to a later experiment. An equal number of subjects were assigned to two separate groups. To avoid a selection bias and the corresponding threat to internal validity, we balanced the two groups with respect to two specific programming skills: the ability to detect and correct logical errors and the ability to comprehend and trace programs. Two instruments (Test 1 and Test 2) were developed and used for balancing purposes. Test 1 covered 25 common logical errors in data structures and algorithms, as identified in the literature [Eisenstadt 1997; Metzger 2003], and was administered prior to the experiment. Also administered prior to the experiment was Test 2, which was modeled on the multi-national study of reading and tracing skills in novice programmers [Lister et al. 2004]. Students were put into a list in ascending order of their combined scores on Test 1 and Test 2. The list was then divided into pairs, starting with the lowest score. Each student from a pair was randomly assigned to Group 1 or Group 2. Groups 1 and 2 were randomly assigned to be the control group (no data structure viewers) and the treatment group (using the data structure viewers) respectively. Students in Group 1 were familiarized with the jGRASP debugger and students in Group 2 were familiarized with both the debugger and jGRASP viewers. Learning how to use the viewers took less than five minutes.

7.2 Experiment 1

Our hypothesis was that students would be more productive (would code faster and with greater accuracy) using the jGRASP data structure viewers. Students were asked to implement four basic operations for singly linked lists. The program LinkedSet.java (from the class textbook [Lewis and Chase 2004]) 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 all four methods: entry(), delete(), insert(), and contains(), using the jGRASP visual debugger. 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 methods using the jGRASP object viewers. The driver program given to this group did not contain the toString() method, and the subjects used the viewers in order to see the contents of the list. The machines in the lab were set up with permissions such that only the treatment group had access to the viewers.

7.3 Experiment 2

The hypothesis for the second experiment was that students would be able to detect and correct logical bugs more accurately and in less time using jGRASP viewers. The students were provided a Java program implementing a singly linked list with 9 nonsyntactical errors in four methods: add(), insert(), delete() and contains(). Students were asked to find and correct all the 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, as in Experiment 1, the control group corrected the detected errors using the jGRASP visual debugger and the treatment group corrected the errors using the jGRASP object viewers.

7.4 Experiment 3

Our hypothesis was that students would be more productive (would code faster and with greater accuracy) using the jGRASP data structure viewers. Students were asked to implement a basic traversal operation for linked binary search trees. The program LinkedBinarySearchTree.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. Accuracy of the assignments was determined by scoring the students' submissions based on the published correctness-based grading policy.

The control group implemented the level order traversal using the jGRASP visual debugger. The driver program provided to this group contained a toString() method so that they could print out the contents of the tree without writing additional code. The treatment group implemented the same method using the jGRASP 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, and the subjects used the viewers in order to see the contents of the tree. The machines in the lab were set up with permissions such that only the treatment group had access to the viewers.

7.5 Experiment 4

The hypothesis for the fourth experiment was that students would be able to detect and correct logical bugs more accurately and in less time using jGRASP viewers. The students were provided a Java program implementing a linked binary search tree with five logical errors, one in each of the methods addElement(), findAgain(), removeElement(), inOrder() and postOrder().

Students were asked to find and correct the 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 groups were first required to identify and document errors on paper. Next, the control group corrected the detected errors using the jGRASP visual debugger, and the treatment group corrected the errors using the jGRASP debugger and object viewers.

7.6 Results and Discussion

Collection of data was strictly contingent on student consent, and student participation was rewarded with extra credit in the course. Students were eligible for the extra credit even if they decided to opt-out of data collection. Both groups were rewarded similarly regardless of the experimental treatment they received.

We used Hotelling's T^2 statistic to analyze our data since we had two dependent matched groups and more than one response variable for each experiment. Hotelling's T^2 is a multivariate counterpart of Student's *t*-test which is typically performed for univariate data. Tests were conducted to check the normality of the distribution and the population was found to be normal for all four experiments.

7.6.1 *Results of Experiment 1.* The null hypothesis was that there is no difference in the accuracy and time taken for both groups. For the 31 samples in each group, Hotelling's T^2 statistic was calculated to be 23.732087. The critical value for $\alpha = 0.05$, $p = 2$ (two response variables), and $n = 31$ (sample size) was 4.1708768. *P*-value was calculated to be 0.0000335. Since the T^2 value is much greater than the critical value, and *p*-value is much less than the alpha value, we can strongly reject the null hypothesis. Thus, there was a statistically significant difference between the two groups. The mean time taken by the group with viewers was 109 minutes while the mean time taken by the group without viewers was 112 minutes. The mean accuracy of the treatment group with viewers was 6.34 points, while the mean accuracy of the control group without viewers was 4.48 points. Students in the treatment group consistently performed better than the control group for all cases.

7.6.2 *Results of Experiment 2.* The null hypothesis was that there is no difference in the number of bugs detected, corrected, introduced, and the time taken for both groups. For the 26 samples in each group, Hotelling's T^2 statistic was calculated to be 12.833955. The critical value for $\alpha = 0.05$, $p = 4$ (four response variables), and $n = 26$ (sample size) was 7.0892211. *P*-value was calculated to be 0.0069295. Since the T^2 value is much greater than the critical value, and *p*-value is much less than the alpha value, we can strongly reject the null hypothesis. Thus, there was a statistically significant difference between the two groups. The mean time taken by the group with viewers was 88.23 minutes while the mean time taken by the group without viewers

was 87.6 minutes. On average, the group using viewers located 6.8 errors, corrected 5.6 errors, and introduced 0.65 errors, and the group without viewers located 4.9 errors, corrected 4.2 errors, and introduced 1.3 errors. Students in the treatment group consistently performed better than the control group for all cases.

7.6.3 *Results of Experiment 3.* The null hypothesis was that there would be no difference in the accuracy and time taken for both groups. The mean time taken by the group with viewers was 69 minutes while the mean time taken by the group without viewers was 82 minutes. The mean accuracy of the treatment group with viewers was 6.93 points, while the mean accuracy of the control group without viewers was 5.06 points. For the 34 samples in each group, Hotelling's T^2 statistic was calculated to be 20.565. The critical value for $\alpha = 0.05$, $p = 2$ (two response variables), and $n = 34$ (sample size) was 4.139. *P*-value was calculated to be 0.00007. Since the T^2 value is much greater than the critical value, and *p*-value is much less than the α value, we can strongly reject the null hypothesis. Thus, there was a statistically significant difference between the two groups, indicating that the viewers were helpful during code development.

7.6.4 *Results of Experiment 4.* The null hypothesis was that there would be no difference in the number of bugs detected, corrected, introduced, or in the time taken for both groups. The mean time taken by the group with viewers was 57.61 minutes, while the mean time taken by the group without viewers was 67.38 minutes. On average, the group using viewers located 3.19 errors, corrected 2.96 errors and introduced 1.66 errors, and the group without the viewers located 2.03 errors, corrected 1.69 errors and introduced 1.88 errors. For the 34 samples in each group, Hotelling's T^2 statistic was calculated to be 22.121. The critical value for $\alpha = 0.05$, $p = 4$ (four response variables), and $n = 26$ (sample size) was 7.089. *P*-value was calculated to be 0.0005. Since the T^2 value is much greater than the critical value, and *p*-value is much less than the α value, we can strongly reject the null hypothesis. Thus, there was a statistically significant difference between the two groups, indicating that the viewers were helpful during code debugging.

8. CONCLUSION

The data structure visualizations provided by jGRASP are intended to support teaching and learning activities in courses that include data structures. In U.S. computer science programs, this would typically be CS2 or CS3. The viewers are well suited for inclusion in classroom lectures and discussions. Their interactive nature has great potential to engage students, more so than traditional static lecture slides for example. The debugger, workbench, text-based interactions, or a combination of these provides great flexibility for both instructors and students. The authors have successfully used them in class as interactive teaching and learning tools.

The highly visual debugger in jGRASP, which provides a natural interface for the data structure visualizations, has also been a major strength for CS1 students as they learn the basics of object-oriented programming. Those

students who use BlueJ in their CS1 course are likely to be comfortable with the workbench approach for developing their programs, and those who use DrJava are familiar with the text-based interactions approach. By integrating these three approaches, students in CS2 and CS3 are allowed to interact with their data structure visualizations in the manner with which they are most comfortable.

The integration also allows the students to mix and match the operational aspects of each approach in a seamless manner. For example, the workbench and text-based interactions may be used alone, together, or in conjunction with a program that is being debugged. Any object visible from the debugger, workbench, or in a viewer, including any sub-component object, can be placed on the workbench. Any object or primitive value visible from the debugger, workbench, or in a viewer, including any subcomponent, can be displayed separately in a viewer. Objects created in the interactions tab appear on the workbench. Return values from workbench-initiated method invocations are displayed in viewers. Items on the workbench and in viewers are named, and those names may be referenced by source code in the interactions tab, or in expressions passed as parameters to workbench object creation and method invocation dialogs.

In addition to describing the integrated approaches for interacting with the data structure viewers, we evaluated the utility of the viewers in two ways. First, we addressed the robustness of the structure identifier by testing it with examples from 20 CS2 data structure textbooks. Results of testing textbook examples indicated that the structure identifier was extremely robust. It was able to display 82% of the examples directly, 92% by selecting an appropriate field, and 97% with a minor change to the viewer configuration. We then addressed the usefulness of the data structure viewers by conducting controlled experiments with CS2 students. The results showed that the data structure viewers had a statistically significant positive effect on student performance. Specifically, students using the data structure viewers were able to write new code faster and with fewer errors, as well as find and locate more errors in a shorter amount of time. Students using the data structure viewers also injected fewer new defects when modifying existing code.

The overall result of this work is a highly flexible approach for user interaction with a robust set of automatically generated dynamic data structure visualizations. Thus, jGRASP now provides students and educators with easy access to effective software visualizations.

APPENDIX A - Data Structure Textbooks

[1]	Andersen, S. *Data Structures in Java: A Laboratory Course*, Jones and Bartlett Publishers, 2002.
[2]	Augenstein, M., Langsam, Y., Tenenbaum, *A. Data Structures: Using Java*, Prentice Hall, 2003.
[3]	Bailey, D. *Java Structures: Data Structures in Java for the Principled Programmer*, 2nd ed., McGraw-Hill, 2002.
[4]	Budd, T. *Classic Data Structures in Java*, Addison-Wesley, 2001.

[5]	Carrano, F., Prichard, J. *Data Abstraction and Problem Solving with Java*, 2nd ed. Addison-Wesley, 2006.
[6]	Carrano, F. *Data Structures and Abstractions with Java*, 2nd ed., Prentice Hall, 2007.
[7]	Collins, W. *Data Structures and the Java Collections Framework*, 2nd ed., McGraw-Hill, 2005.
[8]	Dale, N., Joyce, D., Weems, C. *Object Oriented Data Structures Using Java*, 2nd ed., Jones and Bartlett Publishers, 2006.
[9]	Deitel, H., Deitel, P. *Java How to Program*, 7th ed., Prentice Hall, 2006.
[10]	Drake, P. *Data Structures and Algorithms in Java*, Prentice Hall, 2006.
[11]	Drozdek, A. *Data Structures and Algorithms in Java*, 2nd ed., Course Technology, 2004.
[12]	Ford, W., Topp, W. *Data Structures with Java*, Prentice Hall, 2005.
[13]	Goodrich, M., Tamassia, R. *Data Structures and Algorithms in Java*, 4th ed., John Wiley & Sons, 2005.
[14]	Gray, S. Data Structures in Java: From Abstract Data Types to the Java Collections Framework, Adedison-Wesley, 2007.
[15]	Hubbard, J. *Schaum's Outline of Data Structures with Java*, 2nd ed., McGraw-Hill, 2005.
[16]	Horstmann, C. *Big Java*, 3rd ed., John Wiley & Sons, 2007.
[17]	Hume, J., West, T., Holt, R., Barnard, D. *Programming Data* it Structures in Java, Holt Software Associates Inc., 1999.
[18]	Koffman, E., Wolfgang, P. *Objects, Abstractions, Data Structures and Designs Using Java Version 5.0*, John Wiley & Sons, 2004.
[19]	Lafore, R. *Data Structures and Algorithms in Java*, 2nd ed., Sams Publishing, 2003.
[20]	Lambert, K., Osborne, M. *Java: A Framework for Program Design and Data Structures*, 2nd ed., Course Technology, 2003.
[21]	Lewis, J., DePasquale, P., Chase, J. *Java Foundations: Introduction to Program Design and Data Structures*, Addison-Wesley, 2008.
[22]	Lewis, J., Chase, J. *Java Software Structures: Designing and Using Data Structures*, 2nd ed., Addison-Wesley, 2005.
[23]	Liang, Y. *Introduction to Java Programming*, 6th ed., Prentice Hall, 2007.
[24]	Main, M. *Data Structures and Other Objects Using Java*, 3rd ed., Addison-Wesley, 2006.
[25]	Malik, D., Nair, P. *Data Structures Using Java*, Course Technology, 2003.
[26]	Malik, D. *Java Programming: Program Design Including Data Structures*, Course Technology, 2005.
[27]	Riley, D. The Object of Data Abstraction and Structures, Addison-Wesley, 2003.
[28]	Sahni, S. *Data Structures, Algorithms, and Applications in Java*, 2nd ed., Silicon Press, 2004.

[29]	Shaffer, C. *Practical Introduction to Data Structures and Algorithms*, Java Edition, Prentice Hall, 1998.
[30]	Standish, T. *Data Structures in Java*, Addison-Wesley, 1998.
[31]	Tremblay, J., Cheston, G. *Data Structures and Software Development in an Object Oriented Domain*, Java Edition, Prentice Hall, 2003.
[32]	Tymann, P., Schneider, M. *Modern Software Development Using Java*, 2nd ed., Course Technology, 2007.
[33]	Venugopal, S., *Data Structures Outside-In with Java*, Prentice Hall, 2007.
[34]	Watt, D., Brown, D. *Java Collections: An Introduction to Abstract Data Types, Data Structures and Algorithms*, John Wiley & Sons, 2001.
[35]	Weiss, M. *Data Structures and Algorithm Analysis in Java*, 2nd ed., Addison-Wesley, 2007.
[36]	Weiss, M. *Data Structures and Problem Solving Using Java*, 3rd ed., Addison-Wesley, 2006.

ACKNOWLEDGMENT

The authors thank all of the anonymous reviewers for their useful and detailed recommendations.

REFERENCES

AKINGBADE, A., FINLEY, T., JACKSON, D., PATEL, P., AND RODGER, S. 2003. JAWAA: Easy Web-based animation for CS 0 to advanced CS courses. In *Proceedings of the 34th Technical Symposium on Computer Science Education (SIGCSE'03)*. 162–166.

ALLEN, E., CARTWRIGHT, R., AND STOLER, B. 2002. DrJava: A lightweight pedagogic environment for Java. In *Proceedings of the 33rd Technical Symposium on Computer Science Education (SIGCSE'02)*. 137–141.

BAECKER, R., DIGIANO, C., AND MARCUS, A. 1997. Software visualization for debugging. *Comm. ACM 40*, 4, 44–54.

BEN-ARI, M., MYLLER, N., SUTINEN, E., AND TARHIO, J. 2002. Perspectives on program animation with Jeliot. In *Proceedings of the International Seminar on Software Visualization*. Lecture Notes in Computer Science, vol. 2269, 31–45.

BEN-BASSAT LEVY, R. AND BEN-ARI, M. 2007. We work so hard and they don't use it: Acceptance of software tools by teachers. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ItiCSE'07)*. 246–250.

BEN-BASSAT LEVY, R., BEN-ARI, M., AND URONEN, P. 2003. The Jeliot 2000 program animation system. *Comput. Educ. 40*, 1–15.

BROWN, M. H. 1991. ZEUS: A system for algorithm animation and multi-view editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages (VL'01)*. 4–9.

BROWN, M. H. AND SEDGEWICK, R. 1985. Techniques for algorithm animation. *IEEE Softw. 1*, 28–39.

CAMPBELL, A. E. R., CATTO, G. L., AND HANSEN, E. E. 2003. Language-independent interactive data visualization. In *Proceedings of the 34th Technical Symposium on Computer Science Education (SIGCSE'03)*. 215–219.

CROSS, J. H., HENDRIX, T. D., AND BAROWSKI, L. A. 2002. Using the debugger as an integral part of teaching CS 1. In *Proceedings of the Conference on Frontiers in Education (FIE'02)*.

CROSS, J. H., HENDRIX, T. D., JAIN, J., AND BAROWSKI, L. A. 2007. Dynamic object viewers for data structures. In *Proceedings of the 38th Technical Symposium on Computer Science Education (SIGCSE'07)*. 4–8.

CROSS, J. H., HENDRIX, T. D., AND BAROWSKI, L. A. 2009. Integrating multiple approaches for interacting with dynamic data structure visualizations. In *Proceedings of the 5th Program Visualization Workshop (PVW'08). Electron. Notes Theor. Comput. Sci. 224*, 141–149

DERSHEM, H. L., MCFALL, R. L., AND UTI, N. 2002. Animation of Java linked lists. In *Proceedings of the 33rd Technical Symposium on Computer Science Education (SIGCSE'02)*. 53–57.

EISENSTADT, M. 1997. My hairiest bug war stories. *Comm. ACM 40*, 4, 30–37.

GESTWIKI, P. AND JAYARAMAN, B. 2004. JIVE: Java interactive visualization environment. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'04)*. 226–228.

GOSLING, J., JOY, B. STEEL, G., AND BRACHA, G. 2005. *Java Language Specification* 3rd Ed. Addison Wesley.

HAMER, J. 2004. A lightweight visualizer for Java. Korhonen, ed. In *Proceedings of the 3rd Program Visualization Workshop (PVW'04). Research Report CS-RR-407*, 55–61.

HAMILTON-TAYLOR, A. G. AND KRAEMER, E. 2002. SKA: Supporting algorithm and data structure discussion. In *Proceedings of the 33rd Technical Symposium on Computer Science Education (SIGCSE'02)*. 58–62.

HANSEN, S. R., NARAYANAN, N. H., AND HEGARTY, M. 2002. Designing educationally effective algorithm visualizations: Embedding analogies and animations in hypermedia. *J. Vis. Lang. Comput. 13*, 2, 291–317.

HENDRIX, T. D., CROSS, J. H., JAIN, J., AND BAROWSKI, L. A. 2007. Providing data structure animations in a lightweight IDE. In *Proceedings of the 4th Program Visualization Workshop (PVW'06). Electron. Notes Theor. Comput. Sci. 178*, 101–109.

HÜBSCHER-YOUNGER, T. AND NARAYANAN, N. H. 2003. Authority and convergence in collaborative learning. *Comput. Educ. 41*, 4, 313–334.

HUNDHAUSEN, C. AND DOUGLAS, S. A. 2002. Low fidelity algorithm visualization. *J. Vis. Lang. Comput. 13*, 449–470.

HUNDHAUSEN, C. AND BROWN, J. 2007. What you see is what you code: A "live" algorithm development and visualization environment. *J. Vis. Lang. Comput. 18*, 22–47.

HUNDHAUSEN, C., DOUGLAS, S., AND STASKO, J. T. 2002. A meta-study of algorithm visualization effectiveness. *J. Vis. Lang. Comput. 13*, 259–290.

JAIN, J., CROSS, J. H., AND HENDRIX, T. D. 2005. Qualitative comparison of systems facilitating data structure visualization. In *Proceedings of ACM Southeastern Conference (ACMSE'05*).

JAIN, J., CROSS, J. H., HENDRIX, T. D., AND BAROWSKI, L. A. 2006. Experimental evaluation of animated-verifying object viewers for Java. In *Proceedings of the ACM Symposium on Software Visualization (ACM SOFT-VIS'06)*. 27–36.

KARAVIRTA, V., KORHONEN, A., MALMI, L., AND STALNACKE, K. 2004. MatrixPro – A tool for on-the-fly demonstration of data structures and algorithms. Korhonen, ed. In *Proceedings of the 3rd Program Visualization Workshop (PVW'04), Research Report CS-RR-407*. 26–33.

KARAVIRTA, V., KORHONEN, A., AND MALMI, L. 2006. Taxonomy of algorithm animation languages. In *Proceedings of the ACM Symposium on Software Visualization (SV'06)*. 77–85.

KARAVIRTA, V. 2007. Integrating algorithm animation systems. In *Proceedings of the 4th Program Visualization Workshop (PVW'06). Electron. Notes Theor. Comput. Sci. 178*, 79–87.

KÖLLING, M., QUIG, B., PATTERSON, A., AND ROSENBERG, J. 2003. The BlueJ system and its pedagogy. *J. Comput. Sci. Educ.,* (Special Issue on Learning and Teaching Object Technology), *13*, 4, 1–12.

KORN, J. L. AND APPEL, A. W. 1998. Traversal-based visualization of data structures. In *Proceedings of IEEE Information Visualization (VIS'98)*. 11–18.

LAUER, T. 2006. Learner interaction with algorithm visualizations: Viewing vs. changing vs. constructing. In *Proceedings of the 11th Annual Conference on Innovation and Technology in Computer Science Education (SIGCSE'06)*. 202–206.

LEWIS, J. AND CHASE, J. 2005. *Java Software Structures: Designing and Using Data Structures* 2nd Ed. Addison-Wesley.

LISTER, R., WHALLEY, J., THOMPSON, E., CLEAR, T., ROBBINS, P., KUMAR, P., AND PRASAD, C. 2006. An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. In *Proceedings of the 8th Australian Conference on Computing Education (ACCE'06)*. 243–252.

MCNALLY, M., NAPS, T., FURCY, D., GRISSOM, S., AND TREFFTZ, C. 2007. Supporting rapid development of pedagogically effective algorithm visualizations. *J. Comput. Sci. Coll. 23*, 1, 80–90.

METZGER, C. 2003. *Debugging by Thinking: A Multidisciplinary Approach*. Elsevier.

MOHER, T. G. 1988. Provide: A process visualization and debugging environment. *IEEE Trans. Softw. Eng.* 849–857.

MONTGOMERY, L. N., CROSS, J. H., HENDRIX, T. D., AND BAROWSKI, L. A. 2008. Testing the jGRASP structure identifier with data structure examples from textbooks. In *Proceedings of the 46th ACM Southeast Conference (ACMSE'08)*. 198–203.

MORENO, A. AND MYLLER, N. 2003. Producing an educationally effective and usable tool for learning: The case of the Jeliot family. In *Proceedings of the International Conference on Networked E-learning for European Universities (EDEN'03)*.

MORENO, A., MYLLER, N., SUTINEN, E., AND BEN-ARI, M. 2004. Visualizing programs with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI'04)*.

MUKHERJEA, S. AND STASKO, J. T. 1994. Toward visual debugging: Integrating algorithm animation capabilities within a source-level debugger. *ACM Trans. Comput.-Hum. Interact. 1*, 3, 215–244.

MYERS, B. A., CHANDHOK, R., AND SAREEN, A. 1988. Automatic data visualization for novice pascal programmers. *IEEE Workshop on Visual Languages (VL'88)*. 192–198.

MYERS, B. A. 1990. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput. 1*, 1, 97–123.

NAPS, T. 2000. Instructional interaction with algorithm visualizations. *J. Comput. Sci. Coll. 16*, 1, 7–8.

NAPS, T. L., RÖSSLING, G., ALMSTRUM, V., DANN, W., FLEISCHER, R., HUNDHAUSEN, C., KORHONEN, A., MALMI, L., MCNALLY, M., RODGER, S., AND VELÁZQUEZ-ITURBIDE, J. Á. 2002. Exploring the role of visualization and engagement in computer science education. In *Proceeding of the Working Group Reports From ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE'02)*. 131–152.

NAPS, T., COOPER, S., KOLDEHOFE, B., LESKA, C., RÖSSLING, G., DANN, W., KORHONEN, A., MALMI, L., RANTAKOKKO, J., ROSS, R. J., ANDERSON, J., FLEISCHER, R., KUITTINEN, M., AND MCNALLY, M. 2003. Evaluating the educational impact of visualization. In *Proceeding of the Working Group Reports From ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE'03)*. 124–136.

NAPS, T. 2005. JHAVÉ: Supporting algorithm visualization. *IEEE Comput. Graph. Appl.* 49–55.

NAPS, T., RÖSSLING, G., BRUSILOVSKY, P., ENGLISH, J., JARC, D., KARAVIRTA, V., LESKA, C., MCNALLY, M., MORENO, A., ROSS, R. J., AND URQUIZA-FUENTES, J. 2005. Development of XML-based tools to support user interaction with algorithm visualization. *ACM SIGCSE Bull. 37*, 4, 123–138.

PARKER, B. AND MITCHELL, I. 2006. Effective methods for learning: A study in visualization. *J. Comput. Sci. Coll. 22*, 2, 176–182.

PATERSON, J., HADDOW, J., BIRCH, M., AND MONAGHAN, A. 2005. Using the BlueJ IDE in a data structures course. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'05)*. 349–349.

PAZEL, D. P. 1989. DS-Viewer: An interactive graphical data structure presentation facility. *IBM Syst. J. 2*, 307–323.

PIERSON, W. C. AND RODGER, S. H. 1998. Web-based animation of data structures using JAWAA. In *Proceedings of the 29th Technical Symposium on Computer Science Education (SIGCSE'98)*. 267–271.

PRICE, B. A., BAECKER, R. M., AND SMALL, I. S. 1993. A principled taxonomy of software visualization. *J. Vis. Lang. Comput. 4*, 3, 211–266.

RÖSSLING, G. AND FREISLEBEN, B. 2000. Program visualization using ANIMALSCRIPT. E. Sutinen, ed. In *Proceedings of the 1st Program Visualization Workshop (PVW'00)*. 41–52.

RÖSSLING, G. AND FREISLEBEN, B. 2002. ANIMAL: A system for supporting multiple roles in algorithm animation. *J. Vis. Lang. Comput. 13*, 3, 341–354.

RÖSSLING , G., GLIESCHE, F., JAJEH, T., AND WIDJAJA, T. 2004. Enhanced expressiveness in scripting using ANIMALSCRIPT. Korhonen, ed. In *Proceedings of the 3rd Program Visualization Workshop (PVW'04)*. 10–17.

RÖSSLING, G., KULESSA, S., AND SCHNEIDER, S. 2007. Easy, fast, and flexible algorithm animation generation. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ItiCSE'07)*. 357–357.

RÖSSLING, G., NAPS, T., HALL, M. S., KARAVIRTA, V., KERREN, A., LESKA, C., MORENO, A., OECHSLE, R., RODGER, S. H., URQUIZA-FUENTES, J., AND VELÁZQUEZ-ITURBIDE, J. A. 2006. Merging interactive visualizations with hypertextbooks and course management. *ACM SIGCSE Bull. 38*, 4, 166–181.

SHAFFER, C. A., HEATH, L. S., AND YANG, J. 1996. Using the swan data structure visualization system for computer science education. In *Proceedings of the 27th Technical Symposium on Computer Science Education (SIGCSE'96)*. 140–144.

SHAFFER, C. A., COOPER, M., AND EDWARDS, S. H. 2007. Algorithm visualization: A report on the state of the field. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'07). 39*, 1, 150–154.

SHIMOMURA, T. AND ISODA, S. 1991. Linked-list visualization for debugging. *IEEE Softw.* 44–51.

STASKO, J. T. 1990. TANGO: A framework and system for algorithm animation. *Comput. 23*, 9, 27–39.

STASKO, J. AND LAWRENCE, A. 1998. Empirically assessing algorithm animations as learning aids. In *Software Visualization: Programming as a Multimedia Experience*, 419–438. The MIT Press.

STASKO, J., BROWN, M. H. DOMINGUE, J., AND PRICE, B. A. 1998. In *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA.

ZELLER, A. AND LÜTKEHAUS, D. 1996. DDD—a free graphical front-end for UNIX debuggers. *SIGPLAN Not. 31*, 1, 22–27.

ZELLER, A. 2001. Visual debugging with DDD: Seeing is believing when it comes to tracking errors. *Dr. Dobb's J*. http://www.ddj.com/184404519.

Received September 2008; revised January 2009, February 2009; accepted April 2009