An Extensible Framework for Providing Dynamic Data Structure Visualizations in a Lightweight IDE

T. Dean Hendrix, James H. Cross II, and Larry A. Barowski
Computer Science and Software Engineering
Auburn University, AL 36849
hendrix | cross | larrybar@eng.auburn.edu

ABSTRACT

A framework for producing dynamic data structure visualizations within the context of a lightweight IDE is described. Multiple synchronized visualizations of a data structure can be created with minimal coding through the use of an external viewer model. The framework supplies a customizable viewer template as well as high-level APIs to a graph drawing library and the Java Debugger Interface. Initial classroom use has demonstrated the framework's ease of use as well as its potential to as an aid to student learning.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments – *graphical environments, integrated environments, interactive environments.*

General Terms

Documentation, Experimentation, Human Factors.

Keywords

Program Visualization, Algorithm Animation, Data Structures.

1. INTRODUCTION

Software visualizations have long been at the center of efforts to provide effective learning tools for computing curricula. Although many have been shown to be effective pedagogically, they are not widely adopted. The reasons include: (1) lack of suitable automatic generation of the visualizations; (2) lack of integration among visualizations (e.g., navigation among multiple visualizations of the same software artifact); and (3) lack of integration with basic IDE support (e.g., linking the development environment with the visualization environment). To obtain the full benefit of visualizations when developing code, it is useful to automatically generate multiple synchronized views without leaving the IDE. For example, when implementing or using a data structure such as a red-black tree, appropriate views may include control structure, class structure, and a dynamic visualization of the tree itself, all of which have been shown in the literature to be effective.

In the latest version of the jGRASP environment under development (1.7.0 alpha), an extensible framework for automatically generating dynamic data structure visualizations has been added. This framework provides users with the ability to create multiple graphical views of data structures, which are synchronized, automatically generated, and updated as a program executes. Initial classroom experience with the data structure visualization framework has been very positive. Dynamic visualizations of data structures such as linked lists, heaps, and red-black trees have been produced in a matter of minutes and seamlessly integrated into the classroom lecture. Students are able to watch a red-black tree grow, shrink, and change colors as a program is stepped through and explained during class. While controlled experiments have not yet been performed to quantify the effect, if any, that these visualizations have on student learning, the anecdotal evidence collected to date is very encouraging.

2. BACKGROUND AND CONTEXT

The extant software visualization research has primarily 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 [10]. Our work overlaps the boundary between program visualization and algorithm animation, and draws on the research in both areas. While of visualizations are automatically generated from a program's runtime execution, our primary purpose is to increase the comprehensibility of the underlying algorithm and the associated program behavior. Our framework also allows one to create abstract visualizations that have only conceptual, rather than physical, relationships to the underlying source code.

Extensive research has been carried out in an effort to understand the effects that algorithm animations have on learning and comprehension. Although some results are negative, and many are conflicting, it has been shown that appropriate animations can be effective aids to student learning, given the right circumstances.

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 [6]. These results emphasize the importance of context in the effectiveness of visualizations. In subsequent empirical work, computer-supported collaborative construction and critiquing of algorithm visualizations by students was also found to lead to superior learning [7]. These results suggest that well-designed visualizations can indeed enhance student learning of algorithms. We have applied the results from this research to the teaching and learning of programming by integrating visualizations with IDE functionality. Such as seamless integration of a lightweight IDE with a set of pedagogically effective software visualizations is unique and is currently unavailable in any other environment.

3. GENERATING DYNAMIC DATA STRUCTURE VISUALIZATIONS

From the perspective of a language like Java, data structures such as linked lists, stacks, queues, and trees are simply objects of varying degrees of complexity. The current jGRASP integrated Java debugger and workbench provide an expandable nested view of objects that gives detailed information about an object and its fields. Clearly, higher-level views of complex objects would be useful for both program understanding and debugging. For example, if an object is conceptually an ordered list implemented by a tree data structure, it would be helpful to be able see three levels of abstraction: (1) the low-level expandable view of the object's members, (2) the higher level view of the underlying tree structure, and (3) the top level view as an ordered list. Note that all of these views can be visible at the same time and all are dynamic. As the user steps through the program, the views are updated appropriately (e.g, an element is added).

As an example, consider the Java TreeMap class, which provides a key-value map that is sorted by key. Figure 1 shows the source code being executed (1), the low-level object view (2), the tree view (3), and the high-level sorted-set view (4) of the red-black tree data structure that is used to maintain the sorting. Figure 2 depicts the final state of the tree view (3) after the program has completed execution.

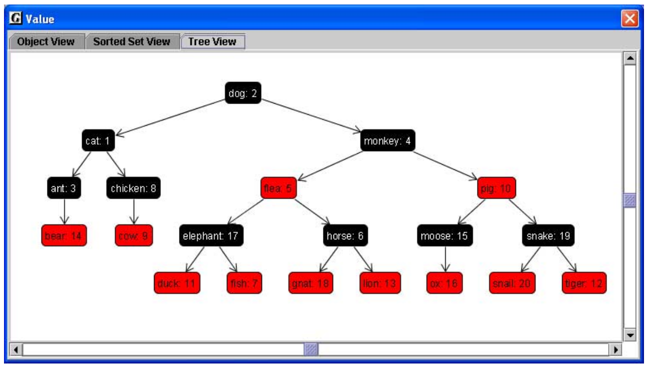
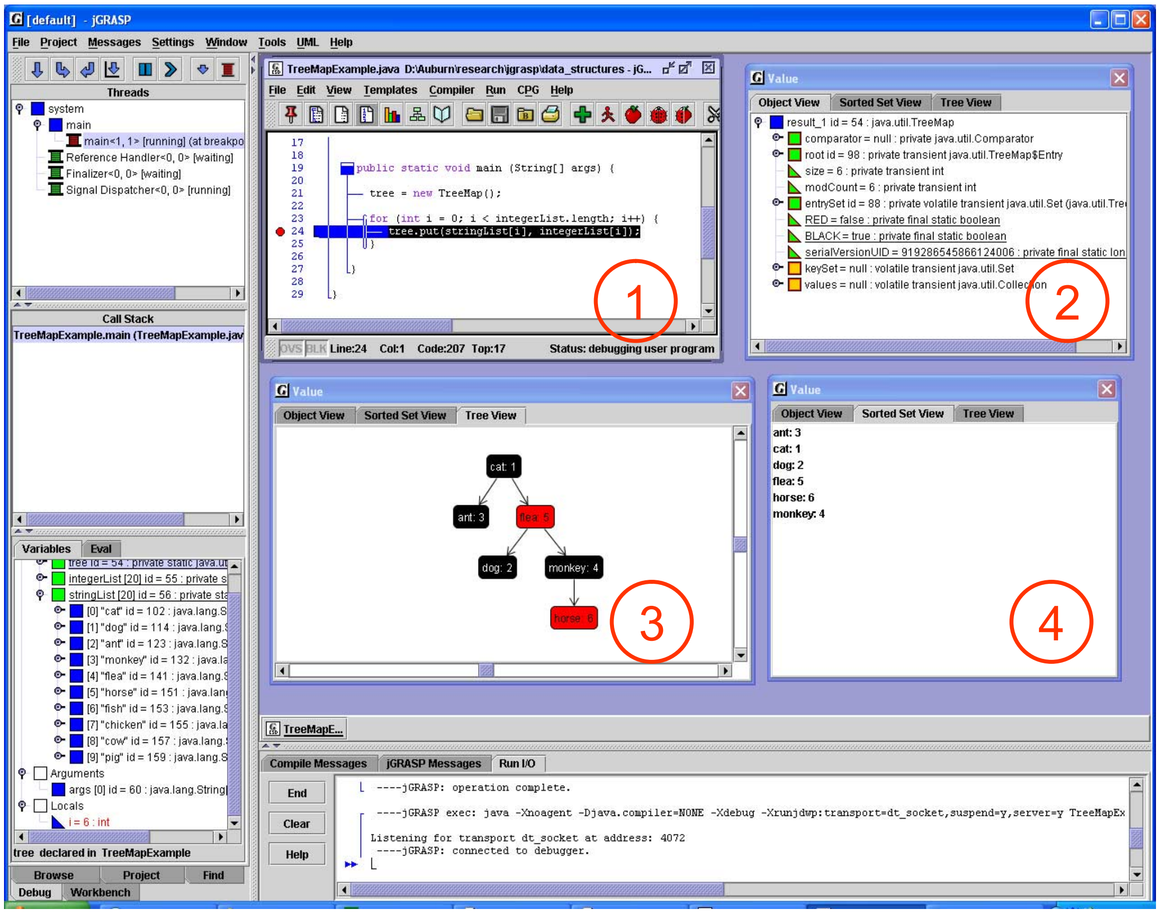


Figure 2. Tree View for an instance of java.util.TreeMap

The tree view would be appropriate if the user is interested in understanding the workings of the red-black tree, or in how the tree is organized for a particular data set and the effect this has on efficiency. However, for debugging a program that only uses the TreeMap for example, the user might be interested in seeing just the keys and values in their current order. In that case, the sorted set


[default] - jGRASP
File Project Messages Settings Window Tools UML Help
Threads
system
main
main<1, 1> [running] (at breakpo
Reference Handler<0, 0> [waiting]
Finalizer<0, 0> [waiting]
Signal Dispatcher<0, 0> [running]
Call Stack
TreeMapExample.main (TreeMapExample.jav
TreeMapExample.java D:\Auburn\research\jgrasp\data_structures - jG...
File Edit View Templates Compiler Run CPG Help
OVS BLK Line:24 Col:1 Code:207 Top:17 Status: debugging user program
1
Value
Object View Sorted Set View Tree View
2
cat: 1
ant: 3
flea: 5
dog: 2
monkey: 4
horse: 6
3
4
Variables Eval
Browse Project Find
Debug Workbench
TreeMapE...
Compile Messages jGRASP Messages Run I/O
End
Clear
Help

```
public static void main (String[] args) {

   tree = new TreeMap();

   for (int i = 0; i < integerList.length; i++) {
      tree.put(stringList[i], integerList[i]);
   }

}

}
```

```
result_1 id = 54 : java.util.TreeMap
   comparator = null : private java.util.Comparator
   root id = 98 : private transient java.util.TreeMap$Entry
   size = 6 : private transient int
   modCount = 6 : private transient int
   entrySet id = 88 : private volatile transient java.util.Set (java.util.Tre
   RED = false : private final static boolean
   BLACK = true : private final static boolean
   serialVersionUID = 919286545866124006 : private final static lon
   keySet = null : volatile transient java.util.Set
   values = null : volatile transient java.util.Collection
```

```
ant: 3
cat: 1
dog: 2
flea: 5
horse: 6
monkey: 4
```

```
   tree id = 54 : private static java.ut
   integerList [20] id = 55 : private s
   stringList [20] id = 56 : private sta
      [0] "cat" id = 102 : java.lang.S
      [1] "dog" id = 114 : java.lang.S
      [2] "ant" id = 123 : java.lang.S
      [3] "monkey" id = 132 : java.la
      [4] "flea" id = 141 : java.lang.S
      [5] "horse" id = 151 : java.lang
      [6] "fish" id = 153 : java.lang.S
      [7] "chicken" id = 155 : java.la
      [8] "cow" id = 157 : java.lang.S
      [9] "pig" id = 159 : java.lang.S
Arguments
   args [0] id = 60 : java.lang.String[
Locals
   i = 6 : int
tree declared in TreeMapExample
```

```
 ----jGRASP: operation complete.

 ----jGRASP exec: java -Xnoagent -Djava.compiler=NONE -Xdebug -Xrunjdwp:transport=dt_socket,suspend=y,server=y TreeMapEx

Listening for transport dt_socket at address: 4072
 ----jGRASP: connected to debugger.
```

Figure 1. jGRASP displaying multiple synchronized dynamic views of a TreeMap data structure.

view (4), which is based on the SortedSet interface that TreeMap implements and simply lists the keys and values in order, could be more useful. For some objects, an even higher-level view may be desired, such as a display of the image for an Image or Icon or a display of a color swatch for a Color.

The jGRASP integrated Java debugger is used to collect the runtime information necessary to render the visualizations. Thus, a program must run in the debugger or from the jGRASP workbench for its data structures to be visualized.

We use an "external viewer" model for specifying dynamic data structure visualizations. For each data structure class (e.g., java.util.TreeMap) to be visualized, one or more external viewers must be created. An external viewer is constructed from a simple class template provided in our framework. The framework has three parts: (1) a high-level interface wrapping the Java Debugger Interface; (2) a higher-level interface for FLGL, the graph drawing library that is a part of jGRASP; and (3) an interface to basic view components included in the framework (e.g., scrollable tabbed panes).

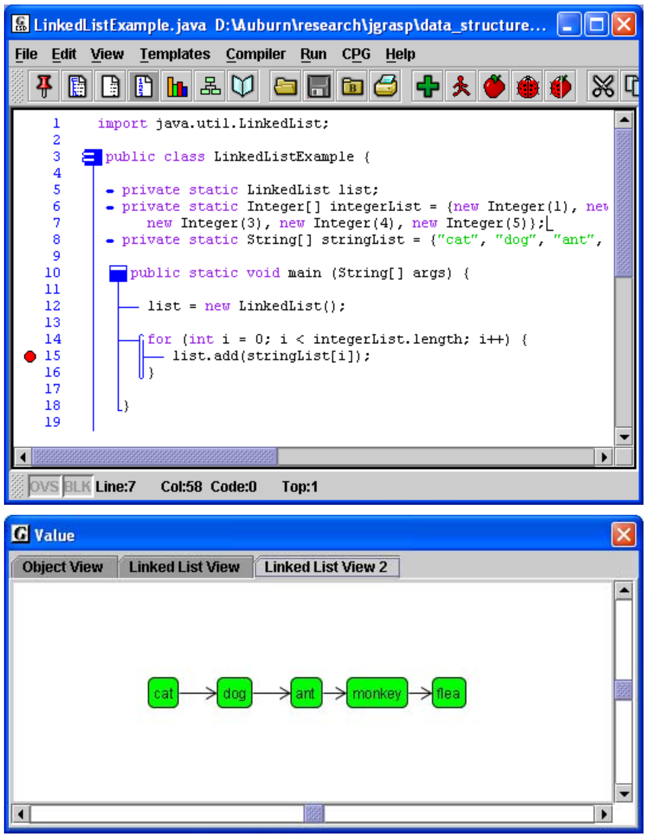
The Java Debugger Interface (JDI) is used to access Objects in the workbench and debugger. Since this interface is quite abstract and complex for the average user, the framework provides a higher-level interface. The user is able to access field values, invoke methods, convert values to strings, or retrieve primitive values using a single line of code (a method call to a framework object). In order to relieve the user of the large amount of exception handling that is necessary in the JDI, exceptions are wrapped in a single exception type, and these are caught inside jGRASP rather than in user code, and produce a useful debugging message when caught.

The Flexible Graph Drawing Library (FLGL) (www.jgrasp.org/flgl) is used in jGRASP for display of UML and for some of the current object view prototypes. FLGL allows the construction, display, and layout of graphs. The framework also provides a higher-level API to access FLGL for the display of data structure diagrams, relieving the users of the details of the use of FLGL.

In addition to this traditional source code-based API, we anticipate allowing interactive specification of data structure diagram mapping using a method similar to the one implemented in Travis [9]. Once implemented, this will provide students and teachers with an easy-to-use interactive approach to generating these visualizations at runtime. Even so, the current source code-based approach is quite easy to use, particularly for instructors.

As an example, suppose we want to produce a visualization for a linked list, say java.util.LinkedList. The first step would be to create an external viewer class from the provided framework template. This viewer class has a method named 'update' that will be called each time an instance of java.util.LinkedList running in the debugger is modified. In the update method, we simply specify the behavior that we want exhibited when the data structure changes (e.g., update the viewer display to show an added node). Since the framework provides high-level APIs to the FLGL and the JDI, this can be done very easily and without any knowledge of graphics, reflection, or the JDI. The external viewer for LinkedList shown in Figure 3 and the external viewer for TreeMap each required only 10 to 15 lines of Java code.

Users are free to create visualizations for arbitrary classes, not just predefined data structures from the Java library. Instructors and students can create viewers for their own data structures or for those provided with a textbook.

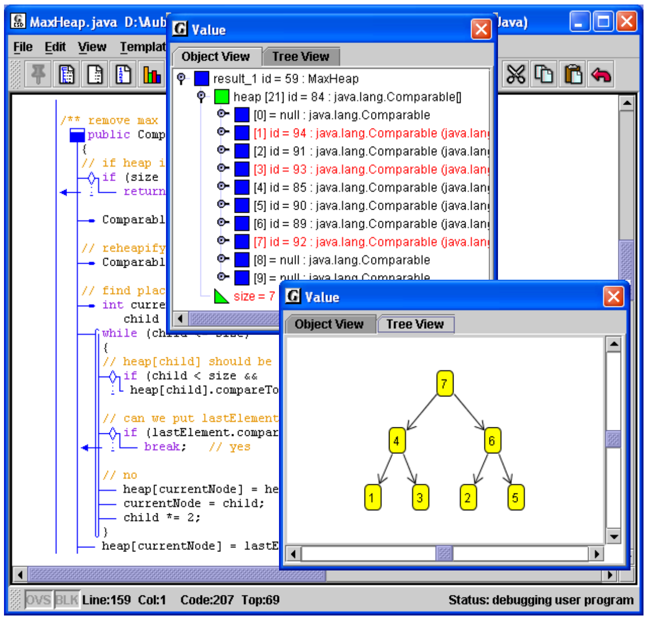


```
import java.util.LinkedList;

public class LinkedListExample {

   private static LinkedList list;
   private static Integer[] integerList = {new Integer(1), new
      new Integer(3), new Integer(4), new Integer(5)};
   private static String[] stringList = {"cat", "dog", "ant",

   public static void main (String[] args) {

      list = new LinkedList();

      for (int i = 0; i < integerList.length; i++) {
         list.add(stringList[i]);
      }

   }
```

```
OVS BLK Line:7   Col:58  Code:0   Top:1
```

Figure 3. Viewer for java.util.LinkedList

Figure 4 shows the MaxHeap class from Sartaj Sahni's CS 2 text [12] being visualized both at the concrete program level as an array (in the Object View provided by jGRASP) and at the conceptual level as a partially ordered binary tree (in the Tree View created by a user).



```
/** remove max
public Comp
{
// if heap i
if (size
   return

Comparabl

// reheapify
Comparabl

// find plac
int curre
   child
while (ch
{
// heap[child] should be
if (child < size &&
 heap[child].compareTo

// can we put lastElement
if (lastElement.compar
   break;   // yes

// no
heap[currentNode] = he
currentNode = child;
child *= 2;
}
heap[currentNode] = lastE
```

```
result_1 id = 59 : MaxHeap
   heap [21] id = 84 : java.lang.Comparable[]
      [0] = null : java.lang.Comparable
      [1] id = 94 : java.lang.Comparable (java.lan
      [2] id = 91 : java.lang.Comparable (java.lan
      [3] id = 93 : java.lang.Comparable (java.lan
      [4] id = 85 : java.lang.Comparable (java.lan
      [5] id = 90 : java.lang.Comparable (java.lan
      [6] id = 89 : java.lang.Comparable (java.lan
      [7] id = 92 : java.lang.Comparable (java.lan
      [8] = null : java.lang.Comparable
      [9] = null : java.lang.Comparable
   size = 7
```

```
OVS BLK Line:159  Col:1   Code:207  Top:69          Status: debugging user program
```

Figure 4. Low-level and high-level visualizations of a textbook MaxHeap class.

4. COMPARISON TO RELATED WORK

4.1 General Purpose Software and Algorithm Animation Systems

Balsa and Zeus [2] 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 and its successor POLKA [14] 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.

4.2 Software and Algorithm Animation Systems for Data Structures

JIVE. JIVE [8] takes the same approach to producing data structure visualizations as JVALL [4], but provides a much richer environment. JIVE includes a collection of Java classes that implement pre-built animated data structures compatible with standard Java classes from the JDSL, 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 to the animation 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. jGRASP does not require the use of special animated versions of classes in creating a data structure visualizations. Instead, jGRASP generates these using the actual classes (e.g., java.util.TreeMap).

SWAN. SWAN [13] 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. jGRASP does not require users to annotate the source code. Instead, jGRASP allows users to interactively create a visualization by specifying a mapping between a particular object structure and the graphical display.

JAWAA. JAWAA [1 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, thus 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. jGRASP data structure visualizations are produced dynamically as the program executes and do not require the use of a scripting language.

LIVE. LIVE [3] 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 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. jGRASP generates the visualizations without modifying the user's source code.

SKA. SKA [5] 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 pre-built 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. jGRASP does not require a set of pre-animated data structures and does not use a source-code annotation model for producing the visualizations.

4.3 Software and Algorithm Animation Systems for Debugging

DDD. DDD [15], a front-end for a command-line debugger, offers simple visualization of lists and trees. 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 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. jGRASP also uses a debugger-based approach. However, jGRASP is extensible and not restricted to visualizing tree structures.

Lens. Lens [10] 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. Although integrated with a debugger, jGRASP visualizations do not require source code or breakpoint annotations.

TRAVIS. 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 [9]. 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 [9]. 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. jGRASP is based on a model similar to Travis. However, in addition to the data structure visualization, jGRASP supports CSDs, UML, and other high-level object views as synchronized visualizations.

5. SUMMARY AND CONTRIBUTIONS

An extensible framework has been implemented that allows the creation of dynamic data structure visualizations within the context of a lightweight IDE. Students and instructors can implement data structure "viewers" that are automatically updated to reflect updates on the data structure as a program runs. The framework shields users from the complexities of graph drawing and using the JDI through high-level APIs. Although controlled experiments have not yet been performed to quantify the effect, if any, that these visualizations have on student learning, the anecdotal evidence collected to date is very encouraging.

REFERENCES

[1] Akingbade, A., Finley, T., Jackson, D., Patel, P., and Rodger, S. (2003). JAWAA: Easy Web-Based Animation for CS 0 to Advanced CS Courses. *Proceedings of SIGCSE 2003*, Reno, NV, February 19-23, 2003, pp. 162-166.

[2] Brown, M. H. and Sedgewick, R. (1985). Techniques for Algorithm Animation. *IEEE Software*, 1 (Jan), pp. 28-39.

[3] Campbell, A. E. R., Catto, G. L., and Hansen, E. E. (2003). Language-Independent Interactive Data Visualization. *Proceedings of SIGCSE 2003*, Reno, NV, February 19-23, 2003, pp. 215-219.

[4] Dershem, H. L., McFall, R. L., and Uti, N. (2002). Animation of Java Linked Lists. *Proceedings of SIGCSE 2002*, Covington, KY, February 27-March 3, 2002, pp. 53-57.

[5] Hamilton-Taylor, A. G., and Kraemer, E. (2002). SKA: Supporting Algorithm and Data Structure Discussion. *Proceedings of SIGCSE 2002*, Covington, KY, February 27-March 3, 2002, pp. 58-62.

[6] Hansen, S. R., Narayanan, N. H., and Hegarty, M. (2002). Designing Educationally Effective Algorithm Visualizations: Embedding Analogies and Animations in Hypermedia. *Journal of Visual Languages and Computing*, 13(2):291-317, Academic Press.

[7] Hubscher-Younger, T., and Narayanan N. H. (2003a). Designing for Divergence. To appear in *Proceedings of the Computer Support for Collaborative Learning Conference*, Kluwer Academic Publishers.

[8] Jive (2003). http://jive.dia.unisa.it

[9] Korn, J. L., and Appel, A. W. (1998). Traversal-Based Visualization of Data Structures. *Proceedings of IEEE Information Visualization '98*, October 1998, pp. 11-18.

[10] Mukherjea, S. and Stasko, J. T. (1994). Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-Level Debugger. *ACM Transactions on Computer-Human Interaction*, Vol. 1, No. 3, September 1994, pp. 215-244.

[12] Sahni, S. (2000). *Data Structures, Algorithms, and Applications in Java*. McGraw-Hill.

[13] Shaffer, C. A., Heath, L. S., and Yang, J. (1996). Using the Swan Data Structure Visualization System for Computer Science Education. *Proceedings of SIGCSE '96*, Philadelphia, PA, February 1996, pp. 140-144.

[14] Stasko, J. T. (1990). TANGO: A Framework and System for Algorithm Animation. *Computer*, 23, 9 (Sep), pp. 27-39.

[15] Zeller, A. (2001). Visual Debugging with DDD. *Dr. Dobb's Journal*, March 2001.