

A Low-Effort Animated Data Structure Visualization Creation System

by

Larry A Barowski

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama

August 2, 2014

Keywords: visualization, data structure, jGRASP

Approved by

James H. Cross II, Chair, Professor of Computer Science and Software Engineering
Richard O. Chapman, Associate Professor of Computer Science and Software Engineering
T. Dean Hendrix, Associate Professor of Computer Science and Software Engineering

Abstract

Algorithm animations and high level data structure visualizations are not widely used, in part because producing them is difficult and time consuming. Algorithm animation and data structure visualization systems attempt to minimize this effort, but none has made the process as simple as it could be. The purpose of the system described in this work, the jGRASP Visualization System, is to allow the creation and use of dynamic data structure visualizations from working source code with minimal effort. Ideally, the only work required to create an animated visualization should be selecting values from a program running in debug mode in an IDE, selecting the way each value will be displayed from a list of available choices, and physically arranging the display elements. This has been achieved for arbitrary implementations of common data structures in Java.

This dissertation begins by discussing existing work on the usefulness of algorithm animation and data structure visualization for algorithm and code understanding, and existing work on systems with similar organization, features, and construction to the one described here. The jGRASP Visualization System is described in detail from the user's perspective. Implementation details are discussed. Previously published results on the effectiveness of these visualizations when applied to arbitrary data structure code, and on code understanding experiments performed with the visualizations are presented. Finally, visualization feature usage data collected from users of the system is analyzed and discussed.

Acknowledgements

I'd like to thank my advisor, Dr. Cross, who began the jGRASP project and has fought to keep it alive for many years. We have worked extremely well together for over a decade. Any disagreements we've had on the direction of the project have been friendly, and our differing opinions tended to merge quickly to produce useful results.

Thanks also to my advisory committee members, Dr. Chapman and Dr. Hendrix, to Jhilmil Jain and Lacey Montgomery, whose validation experiments were an important contribution to this work, and to all other official and unofficial jGRASP contributors, past and present, who are too numerous to name here. Among those are a few unique jGRASP users who have repeatedly provided insightful feature suggestions and problem reports.

Table of Contents

Abstract	ii
Acknowledgements.....	iii
Table of Contents.....	iv
List of Figures.....	vi
List of Tables.....	viii
1 Introduction	1
2 Literature Review	3
2.1 Usefulness of Visualizations in Algorithm and Code Understanding	3
2.2 Algorithm Animation and Data Structure Visualization Systems.....	7
2.3 High Level Data Structure Detection.....	13
3 System Overview (User Perspective).....	15
3.1 Viewers.....	15
3.2 Available Viewers.....	25
3.3 The Structure Identifier Viewer.....	30
3.4 Subviewers	34
3.5 Auto-Step and Auto-Resume.....	35
3.6 Canvas	36
3.7 Steps for the Creation and Use of an Animation	41
3.8 Weaknesses.....	42

4	Viewer Implementation and Viewer API	44
4.1	The jGRASP Debugger Interface (jgrdi)	44
4.2	Viewer Interface	48
4.3	Viewer Implementation Base Classes	51
4.4	Utility Classes	52
5	The Structure Identifier Mechanism.....	54
5.1	Mapping Expressions	54
5.2	Automatic Determination of Mapping Expressions	58
5.3	Automated Testing of the Structure Identifier.....	62
5.4	Automated Tuning of the Structure Identifier	65
6	Evaluation.....	68
6.1	Structure Identifier Applicability to Arbitrary Source Code.....	68
6.2	Code Understanding.....	70
6.3	Usage.....	74
7	Summary and Conclusion.....	80
8	Future Goals	87
9	References	89
Appendix A	Structure Identifier Mapping Expressions.....	94
Appendix B	Viewer Class Naming Scheme	99
Appendix C	Selected Viewer API Method Summaries.....	101

List of Figures

Figure 1. DDD view of a C++ linked list.....	10
Figure 2. Online Python Visualizer view of a class-based linked list.....	11
Figure 3. Values in the (a) variables display, (b) expression evaluation table, (c) workbench.....	16
Figure 4. Viewer dialog for a linked list showing viewer selection list.....	19
Figure 5. Example method where variable name "x" is reused.....	24
Figure 6. Basic (a) and detail (b) viewers for a linked list node, and detail viewer settings (c). ..	26
Figure 7. Interface-based (a), presentation (b), and bar graph (c) viewers for a linked list.....	29
Figure 8. Various specialized viewers.....	30
Figure 9. Structure identifier viewer and source code for a binary tree.....	31
Figure 10. Structure identifier linked list and array-based configuration.	32
Figure 11. Node display modes for a binary tree map node.....	34
Figure 12. Viewer dialog displaying a view of a stack with subviewer open.	35
Figure 13. Canvas for a text scanning algorithm.....	36
Figure 14. Viewer expression and scope editing dialog.....	38
Figure 15. Canvas name change dialog.....	39
Figure 16. Canvas for a binary search animation showing animation controls.....	40
Figure 17. Sequence diagram for a viewer update.....	51
Figure 18. Viewer for an integer-encoded binary tree.	58
Figure 19. Structure identifier data structure recognition module dependencies.	59
Figure 20. Type reference patterns for linked structures.	61

Figure 21. Structure identifier test file for a java.util.LinkedList.....	64
Figure 22. Test source file for a java.util.LinkedList.....	65
Figure 23. Example mapping confidence result sets.....	67
Figure 24. Structure identifier total and per-data-structure test result percentages [43].....	69
Figure 25. Number of students who correctly implemented each method in first experiment.	72
Figure 26. Number of students who correctly completed each task in second experiment.	73
Figure 27. Average accuracy scores over all students for third and fourth experiments.....	73
Figure 28. Average raw accuracy scores over all students for fifth and sixth experiments.....	74
Figure 29. Viewer use as a percentage of debugger use, seven day moving average.....	76
Figure 30. Correlation between viewer use and debugger use.	77
Figure 31. Debugger, viewer, and structure identifier use trends.....	79

List of Tables

Table 1. Viewer target type examples.....	18
Table 2. Detail viewer icon properties.	27
Table 3. Structure identifier per-data-structure test result numbers [43].	70
Table 4. jGRASP feature use in 2013.	78
Table 5. jGRASP feature use in 2012.	79
Table 6. jGRASP feature use January 1 to May 13, 2014.....	79
Table 7. Array-and-list-based structure expressions.....	94
Table 8. Array-and-list-based structure synthetic variables.	95
Table 9. Linked list structure expressions.	95
Table 10. Linked list synthetic variables.....	95
Table 11. Binary tree structure expressions.....	96
Table 12. Binary tree synthetic variables.	97
Table 13. Chained hash table structure expressions.	97
Table 14. Chained hash table synthetic variables.	98
Table 15. Viewer class name string replacements.	99
Table 16. Example viewer class names and target types.	100
Table 17. Viewer interface method summaries.	101
Table 18. Value interface method summaries.	102

1 Introduction

Algorithm animation is the process of abstracting a program's data, operations, and semantics, and creating dynamic graphical views of those abstractions [1]. By contrast, data structure visualization systems produce visualizations based only on runtime data structure state or state history and do not exhibit algorithm semantics that are not encoded in that state or state history. Thus, the automation of data structure visualization tends to be more straightforward, and algorithm animation systems usually require some degree of scripting or other manual configuration for each animation. The purpose of an algorithm animation is most often to aid in the understanding of an algorithm and the data structures used to implement it, and sometimes also to aid in the understanding of source code that implements the algorithm. Data structure visualizations can also be used to aid algorithm and data structure understanding. The higher potential for automation generally makes them better suited than algorithm animations for aiding in code understanding during software development and debugging, code review, and reverse engineering of software. For code understanding purposes, detailed data-structure-specific graphical visualizations provided by a data structure visualization system are potentially superior to text representations or the straightforward generic graphical representations provided by a typical debugger, and dynamic visualizations that show smooth transitions from state to state are potentially superior to ones that only show static states in sequence.

Algorithm animations and high level data structure visualizations are not widely used, in part because producing them is difficult and time consuming. Algorithm animation and data structure

visualization systems attempt to minimize this effort, but none has made the process as simple as it could be. The purpose of the system described in this work, the jGRASP Visualization System, is to allow the creation and use of dynamic data structure visualizations from working source code with minimal effort. Ideally, the only work required to create a dynamic visualization should be selecting values from a program running in debug mode in an IDE, selecting the way each value will be displayed from a list of available choices, and physically arranging the display elements if it is desirable to view multiple values in a single display. Achieving this goal requires several key elements: a comprehensive set of visualizations for common data structures and other program values, animation control in the IDE's debugger (automatic repeated stepping with an adjustable delay between steps, etc.), automatic analysis of runtime elements to detect known data structures and render them using the visualizations, and a way to combine and arrange visualizations for multiple data structures on a single display and save the results for later use. All of these have been implemented, and are integrated into the jGRASP IDE. No other algorithm animation or data structure visualization system combines all of these elements, and none provides automatic detection and rendering of high level data structures in arbitrary code.

Although the system was designed to be independent of the target programming language, currently it operates only on Java code. For simplicity and to avoid overly general language, it will be described here using Java terminology where convenient. Any description of language-specific source code elements should be assumed to be referring to Java unless otherwise specified.

2 Literature Review

Three categories of literature are reviewed here. First, studies of the usefulness of data structure visualizations for code understanding are examined. Some of these studies investigate visualizations in isolation, while others do so in the context of an algorithm animation or data structure visualization system or tool. Second, algorithm animation and data structure visualization systems and tools are described, with a focus on those that share functionality or implementation techniques with the system described in this work. Finally, the lack of literature related to identifying high level data structures from type information alone is noted.

2.1 Usefulness of Visualizations in Algorithm and Code Understanding

In the first formal study of the effectiveness of algorithm animation for algorithm understanding, published in 1993, Stasko et al. [2] used XTango [3] to animate various operations on a binary tree representation of a pairing heap. The animations used were fairly advanced, providing smooth transitions between states, and were not significantly different in form from those provided by current algorithm animation systems. Two groups of computer science graduate students were compared, with ten students in each group. The first group was given only a textual description of the data structure and operations, while the second was also given access to the animation system and allowed to view animations of the different operations in whatever order and as many times as they felt necessary. Both groups were given 45 minutes of learning time, followed by a 45 minute exam testing their understanding of the algorithms. The pairing heap was a fairly new data structure at the time and is still a fairly obscure one, so

the subjects were unlikely to have prior exposure to it. Results showed a small but not statistically significant edge for the visualization group. In addition to noting some specific problems with the study, such as a mismatch between the focus of the learning materials and testing for both groups, the authors suggest that a more general reason for the lack of positive results may be that the subjects did not have an adequate mental mapping between the abstract representation of the data structure and algorithms and their visual representations. They hypothesize that the animations used may therefore be more useful to someone who already has a basic understanding of the algorithms.

With the goal of more closely modeling a typical learning environment, Kehoe et al. [4] conducted a study in which subjects were given unlimited study time and test time. Two groups were provided with study materials and test questions related to the binomial heap data structure. A previous algorithm animation study by Byrne et al. in 1999 [5] using the same data structure and a more traditional learning and testing environment did not find a statistically significant learning benefit for the study group. In this study, the materials were available during both study time and test time. One group had access to animations implemented using POLKA [6] which were hyperlinked at appropriate places in explanatory HTML pages. The animations allowed the participants to step forward and backward, and they were able to view the text and work with the animations simultaneously. The control group was provided with still images that showed key points in the algorithms and that were similar in form to the animation display. In the 23 question test, the study group answered 20.5 questions correctly on average compared to 16 for the control group. Results were statistically significant. The study group performed particularly well on questions that required them to perform operations on binomial heaps, as opposed to questions related to mathematical properties, time complexity, and structure definitions. The

study group also spent more time than the control group on preparation before the test and in reviewing the study materials during testing. Subjects in this study were also observed and asked to comment during the study and interviewed afterward. The authors found that the study group subjects were more relaxed and seemed to enjoy the process much more than the control group subjects. The authors suggest that in addition to improving understanding, algorithm animations can improve motivation by making algorithms less intimidating.

A study by the jGRASP group [7] [8] using an earlier version of the data structure visualizations described in this work was targeted primarily at code understanding with a small aspect of algorithm and data structure understanding. This is described more fully below in section 6.2. Subjects were asked to implement methods for a linked list in one experiment and to find and correct errors in a linked list in a second experiment. A third and fourth experiment required the same tasks for binary trees. The subjects presumably already had a firm understanding of the data structures and algorithms involved. The test group members were allowed to use visualizations and the control group members were prevented from doing so. For all four experiments, the test group performed significantly better quantitatively, and differences between groups were statistically significant. A fifth experiment used binary heap data structures, for which the subjects were unfamiliar with implementation, and a sixth used linked priority queues, for which the subjects were unfamiliar with both implementation and concept. Results for these two experiments were also positive.

In addition to the Kehoe et al. [4] study described above, other studies have found through formal or informal surveys that the use of algorithm animation or data structure visualizations for algorithm or code understanding resulted in improved learning attitudes and motivation. This may not always translate into improved understanding in the short term, but over the long term

learning should be improved and more students should be retained if they feel that the process is less taxing and more enjoyable. Stasko [9] conducted a survey of students who were required to construct algorithm animations using Samba during an algorithms course. The students largely reported that the assignments were fun and useful for algorithm understanding. A study by Lauer [10] found that students' opinions of the usefulness of visualizations created using the MA&DA framework [11] were very positive. Rößling et al. [12] [13] found that the use of the ANIMAL algorithm animation system in introductory programming course lectures was motivating to the students. In a study by Urquiza-Fuentes and Velázquez-Iturbide [14] in which subjects created or viewed WinHIPE animations, results of a formal survey before testing showed that all students believed the animations would help them in understanding algorithms, and results of a survey after testing showed that all subjects believed the animations had helped. A formal survey of CS2 students using an early version of the jGRASP Visualization System [15] found that students believed that the visualizations were helping them to understand data structures and algorithms.

In a 2002 meta-study [16], Hundhausen et al. examined 24 experimental studies of algorithm visualization effectiveness. Eleven were found to have positive and statistically significant results, ten to have non-significant results, two to have positive results that could be due to confounding factors, and one to have significant negative results. Some later studies have shown positive results. Hübscher-Younger and Narayanan [17] found that static or dynamic explanatory visualizations created by other students had a positive and statistically significant effect on student understanding of Fibonacci, exponentiation, and binary tree insertion algorithms. Buchanan and Laviola [18] found that students exposed to lectures using CSTutor, a data structure visualization and visual creation system, performed better on a final exam than students taught using traditional lectures, though differences in other exam and lab quiz scores were not statistically significant, and in

one lab quiz the control group performed better to a statistically significant degree (indicating perhaps that use of the tool may have taken up too much lecture time). A second experiment added a third group whose members used the tool to follow along during lectures. There were no statistically significant differences in exam and quiz grades between the two test groups, but both groups scored better than the control on one exam/quiz to a statistically significant degree.

Studies comparing the efficacy of creating visualizations to simply viewing them also show mixed results. Hundhausen et al. [16] suggest that higher levels of engagement are more important than the nature or quality of visualizations. A study by Lauer [10] tested student performance on questions and exercises involving a Fibonacci heap for three groups that viewed, changed, or constructed visualizations using the MA&DA framework [11]. No significant differences between groups were found. A study by Urquiza-Fuentes and Velázquez-Iturbide [14] found statistically significant differences in the understanding of a breadth-first tree traversal algorithm for a visualization creation group versus a viewing group using WinHIPE. In the Hübscher-Younger and Narayanan study [17] cited above, it was found that creating visualizations improved student learning to a greater degree than using visualizations created by others for many of the tests used. Positive results were also more strongly correlated with the pleasurability of the visualizations than with how well they represented the algorithms (as scored by the students).

2.2 Algorithm Animation and Data Structure Visualization Systems

Systems in which visualizations are constructed from notifications of interesting events inserted into the code through method calls, method call mappings, or textual annotations, are generally referred to as algorithm animations systems. Events in these systems are focused on the algorithm or the visual representation of the algorithm and data structure state. Systems in which

visualizations are constructed from debugger or interpreter information about the runtime state of data structures are called data structure visualization systems. Other systems offer a hybrid approach in which both runtime state information and event notifications are used.

Many of the systems described below are no longer available or no longer used or updated, but for consistency they will all be described in the present tense.

BALSA [19] [20] is an early and influential algorithm animation system. Visualizations called "views" are created using display primitives. Changes to the views are signaled by adding interesting event notification calls directly to the code. When running an animation, these calls can be hidden in the displayed source code. An interpreter system operates on the sequence of interesting events generated during a run of the program. The system can be used in interactive or scripted mode. In interactive mode, the user can run the algorithm in an interpreter, with debugger-like control over execution (stepping, breakpoints, etc.). Reverse execution is also supported if supported by the views to be used. Views can be opened for data structures, and view windows arranged on "algorithm windows". Window configurations can be saved and restored. Scripts are created by recording an interactive mode session and saving it to file for later playback. Some script-specific actions can also be specified during an interactive mode session, such as waiting for the user to press a button to proceed.

The Tango [1] algorithm animation system decouples the running program from the animation system and provides a graphical library for visualizations including support for animated transitions (moving graphical objects, etc.). Interesting event notifications can be supplied through procedure calls inserted into the code, or through an external annotation system. Inter-process communication is used to convey event notifications to the animation

system, which runs in a separate process from the code. Because of this decoupling and because interesting event notifications are typically focused on visualization actions or conceptual algorithm actions, the system is applicable to source code in any language and built under any compilation system. Two-way communication is also provided, so that user manipulation of visualization elements can influence program flow. Tango's successor, XTango [21] adds higher-level graphics and animation to the animation library, while POLKA [6] adds support for concurrency in order to allow the visualization of parallel algorithms.

In CATAI [22], C++ source code is modified by adding directives at places in the code where a data structure is modified. These directives map changes in the structure to changes in its animated representation. The mapping is two-way, enabling interactive animations where changes made to the visualizations are mirrored in the running code. Thus, this is essentially a layer on top of an interesting event notification system that decreases the work needed for two-way interaction. Other directives may be added to algorithmic code to change the animated representation in illustrative ways, such as changing a node color to indicate that it is the current node in an active search. A library of visualizations is provided for common textbook data structures and algorithms, and custom visualizations may be added.

DDD [23] [24] provides straightforward visualizations from code during debugging, for various target languages and target debuggers. These show aggregate type instances as nodes displaying the instance's sub-elements, with references as edges between them. Thus, they are not data structure specific, but can be used to display linked data structures. Figure 1 shows a DDD view of a C++ linked list. Visualizations can be added to a display window using a context menu on variables in the source code or by double clicking on them, and dragged to arrange them on the display window. They can be animated by stepping through the code in the

debugger, though there are no smooth intra-step transitions. Field or array element values can be dragged out into their own nodes in order to expand the display of a data structure node-by-node. Earlier systems that display aggregate type instances as nodes with sub-element information and references between them as links, and that create these visualizations automatically from code running in debug or interpreted mode, include VIPS [25] [26] which also displays linked structures in a more compact node-and-edge diagram, and GDBX [27].

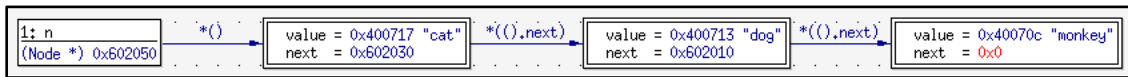


Figure 1. DDD view of a C++ linked list.

The Online Python Visualizer [28] is an online tutoring system with a visualization tool that produces diagrams similar to those of DDD for Python code, but in a more automated fashion. The current call stack frames are always shown graphically as elements containing all local variables and method arguments. For each local variable or method argument that is a composite type, links to a node representing that value are shown, as are reference links between instances. Class field values that are class instances, however, are always shown embedded in the class instance node unless there is another reference to them, so that linked structures implemented using classes are not displayed in the traditional way as nodes and links at all times, but instead are often shown recursively embedded within a single node. Linked structures implemented using Python lists or dicts are always displayed as separate nodes with reference links by default. The only controls are forward and backward stepping, and home and end functions. All source code must be pasted into a single window, so this is intended to be used for small examples. The line of code that is about to be executed is displayed, but there are no additional debugger capabilities. Displayed elements cannot be manually resized or arranged. Figure 2 shows a

visualization of a class-based linked list using this tool, as a node is being inserted. Node displays elements are separated because of the two references to the previous head node that exist.

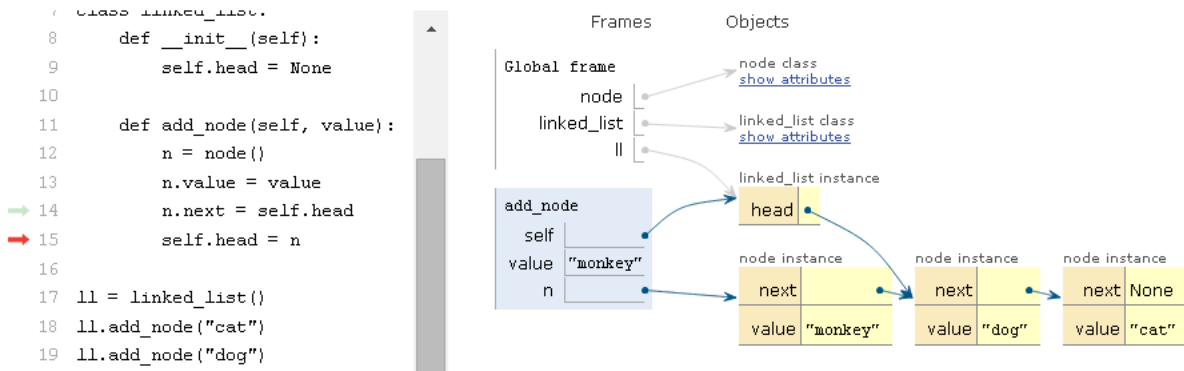


Figure 2. Online Python Visualizer view of a class-based linked list.

Incense [29] is an early data structure display system that provides a small number of visualizations for the display of code-level structures in the Mesa programming language. It also provides a library of display primitives and extendable prototype visualizations that can be used to display high level data structures. Visualizations are generated automatically from a program running in debug mode, with the user required only to enter the name of the variable to be displayed. For any code value displayed, the visualization used can be selected from a list of applicable ones based on the value's type.

The Travis [30] data structure visualization system uses debugger state information to construct the visualizations. A traversal-based specification language defines the relationship between data structure code elements and their graphical representations. Patterns in the language specify how parts of a data structure should be traversed and how the associated visualization should be constructed or modified. Control over the visualizations is at the

graphical element level (nodes, edges, etc.). A visual interface is also provided so that patterns can be constructed by selecting from lists of types, type fields, etc. in the running program.

The Lens [31] system integrates debugger-state-based data structure visualizations with visualization changes triggered by interesting event notifications. Animation commands can be added to a program through annotations using an integrated source code editor. This can be done interactively as the program is running in a design debug mode, using a graphical editor to draw and position graphical entities corresponding to program variables. Positioning can be absolute or relative to other entities. In these commands, simple formulas using variable references from the running debugger can also be used for positions and other attributes. In addition to low-level-entity graphics manipulation commands (adding or moving shapes, etc.), templates for the display of scalars, arrays, binary trees, and linked lists are provided. These templates are linked to the source code representations of data structures through field name mappings, such as the field names for the left and right children of a node in a binary tree. Thus, visualizations for common data structures can be created with minimal effort, and their behavior can then be augmented by adding interesting event notifications.

The systems described above cover the majority of the basic techniques used to generate algorithm animations and data structure visualizations, and the majority of the categories of features provided. Some others that are relevant in relation to this work will be described briefly here. ANIMAL [12] allows algorithm animations to be created through visual editing, scripting, or through an API, and for all animations to be edited visually. Animations are constructed using graphics primitives. JIVE [32] is a plugin to the Eclipse IDE that provides object diagrams and sequence diagrams of execution history. It supports the viewing of visualizations during debugging or later through stored execution records. CSTutor [18] allows users to create and

modify linked data structures by sketching nodes and edges using a stylus input. Source code for the corresponding actions is automatically created. Changes to the code will animate the visualization.

Many of these systems provide some of the features of the jGRASP Visualization System, and use some of the same or similar techniques for generating visualizations. Of the ones that display abstract data structures using debugger or interpreter state information and field names or other mapping and traversal expressions to relate that information to the visualizations, none generates these expressions automatically for previously unencountered data types. That is, none of them displays high level representations of user-created data structures with no effort other than clicking and dragging the mouse while debugging. Also, none of them automatically displays arguments and local variable values that are likely to become part of a data structure or to have recently been part of a data structure (as opposed to all arguments and local variables), as the jGRASP Visualization System does for nodes of linked structures. There is also little discussion in the literature of scoping issues when values are displayed based on program variable names. The jGRASP Visualization System provides tight control over the scope in which variable names and other expressions used to generate values for display are evaluated, and the flexibility to use more relaxed scoping rules for the examination of recursive algorithms.

2.3 High Level Data Structure Detection

High level data structure detection systems such as the Data Structure Detection Tool (DDT) [33] and MemPick [34] attempt to identify high level data structures in binary code for which no type information is available. This is generally done through analysis of running code, by finding probable data structure nodes and edges in memory, and analyzing the way probable data

structure access methods modify these memory graphs. There are also systems for identifying data structures in procedural (non object oriented) languages using source code analysis, such as OBAD [35] and work by Dekker and Ververs [36]. These categories of data structure detection use different input information from the data structure detection discussed in this work, and the techniques used have little in common. There appears to be no published work related to identifying high level data structures from debugger type information, type information in object code compiled in debug mode, or type information in source code (all of which would essentially be equivalent if only type information were used in the analysis).

3 System Overview (User Perspective)

Here the visualization system is described primarily from a user perspective, with minimum exposition of the inner workings sufficient to facilitate understanding and discussion of the user interface. The system is built on top of a debugger and works in conjunction with it. "Viewers" [37] [38] [39] are used to display data structures and other values, and may be used individually in a "viewer dialog" or combined on a viewer "canvas". In the remainder of this work, the word "viewer" will be used to refer to the system elements that generate these GUI displays, and to the display GUI elements themselves, as the meaning will always be clear from context. Values can be opened in a viewer dialog or added to a canvas from their display in the debugger using drag-and-drop or through the use of context menus. User controlled stepping, automatic repeated stepping, or resuming from breakpoint to breakpoint in the debugger enables program animation, where viewer displays are updated after each step or at each breakpoint. The arrangement of viewers on a canvas can be saved to a canvas file.

3.1 Viewers

The viewer is the basic component of visualization in the system. A viewer graphically displays a single program entity, providing a representation of its state and possibly showing smooth transitions from one debugger state to the next and/or indicating changes between one debugger state and the next (changed values, new data structure elements, etc.). Currently, the only program entities supported are object, array, and primitive values. Note that in Java, arrays are objects, and the viewer system treats them as both. Other non-value entities may be added in

the future, such as the program call stack, a call stack frame, a memory block, the set of local variables, etc. These potential entities will be ignored in the rest of this discussion and viewers will be described as a display mechanism for program values only.

A viewer may be created for any value displayed in the debugger. Places where values are displayed include the variables display, which for the current or selected scope shows the "this" value or class value if in a static method context, method arguments, and local variables. The debugger also includes a list of values created through the use of an interpreter-like "interactions" system or by invoking constructors through a "workbench" [40] [8] system, and a table of expression evaluation results. All of these values are displayed in the typical way as trees where objects and arrays can be expanded to show their fields and elements respectively. Various places where values are available in the debugger are shown in Figure 3.

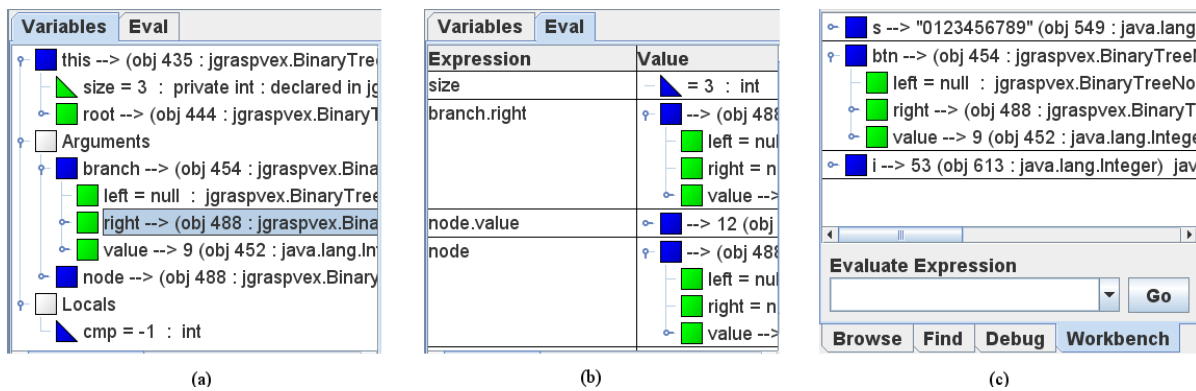


Figure 3. Values in the (a) variables display, (b) expression evaluation table, (c) workbench.

Each viewer implementation class applies to a program type or set of program types. When a value is displayed in the viewer system, only viewers that apply to its runtime type will be made available for it. The applicable types are specified by a target type encoded in the viewer implementation class name. A wildcard name is specified by an asterisk (*). Appendix B describes the class naming system used. The types that are applicable to the viewer depend on

the category of the viewer target type. For Java, a viewer with a non-wildcard target type applies to a specific primitive type if the target type is a primitive type, or to any value that is assignable to the target type otherwise. Assignability was not used for primitive types because typically it is not desirable. This is because viewers for specific primitive types would most often be used to show binary structure of the value, which would change on assignment, or numeric value, which would be confusing if altered due to casting conversion and superfluous if not altered since an identical viewer for the specific numeric type would generally also be available. The wildcard target type indicates viewer applicability to all primitive, object, and array types. Adding array brackets to a wildcard target type specifies a minimum number of array dimensions but there is no maximum, since multidimensional arrays in Java are implemented as arrays of arrays. The runtime type of a value, as opposed to the declared type, is used to determine which viewers apply. For any particular value, multiple viewers may be applicable. Table 1 shows some example target types and resulting viewer applicability in the context of the Java language.

Table 1. Viewer target type examples.

Example Target Type	Category of Target Type	Applicability
int	Primitive	Primitive values of type int.
MyClass	Class	Assignable to a MyClass: instances of MyClass or any subclass of MyClass.
MyInterface	Interface	Assignable to a MyInterface: any object instance of a type that implements MyInterface directly or indirectly.
MyClass[]	Array of Class	Assignable to an array of MyClass: one dimensional arrays of MyClass or any subclass of MyClass.
MyInterface[]	Array of Interface	Assignable to an array of MyInterface: one dimensional arrays of MyInterface, any subinterface of MyInterface, or any class that implements MyInterface directly or indirectly.
*	Any Type	Any value.
*[]	Any Array	Any array value of dimension one or higher.
*[][]	Any 2D Array	Any array value of dimension two or higher.

Since the runtime type of a value determines the applicable viewers, the selected viewer may no longer apply when that type changes. For example, if a variable declared as a Java Object has a String value and a String-specific viewer is selected to display it, then later a LinkedList is assigned to the variable, the String viewer will no longer be applicable. When this happens, a viewer applicable to the new value's type is automatically selected while the user's choice of viewer for the previous runtime type is stored, and if the variable is later set to a value of that previous type, that previous choice of viewer will be restored. If the viewer is in a canvas, this association between runtime types and selected viewers will be maintained between debug sessions and IDE sessions.

Viewers are used to display program values in viewer dialogs, which display a single value, or in viewer canvases which combine the display of multiple values. In either case, for each value the viewer used for display can be selected from a list of all viewers that are applicable to the value's type, as shown in Figure 4. Each viewer provides a short, descriptive name which will appear in the list. Since viewers are implemented as plugins, name conflicts are possible, and they are tolerated. Conflicting names will be displayed with parenthetical index numbers, as, for example "Array Viewer (1)" and "Array Viewer (2)". The viewer names for viewers distributed with the system, of course, have no conflicts. Viewer settings are indexed internally by viewer class name, so if new viewers with the same display name are added to the system this will not cause any conflicts. Figure 4 shows a viewer dialog displaying a linked list with the viewer selection list open.

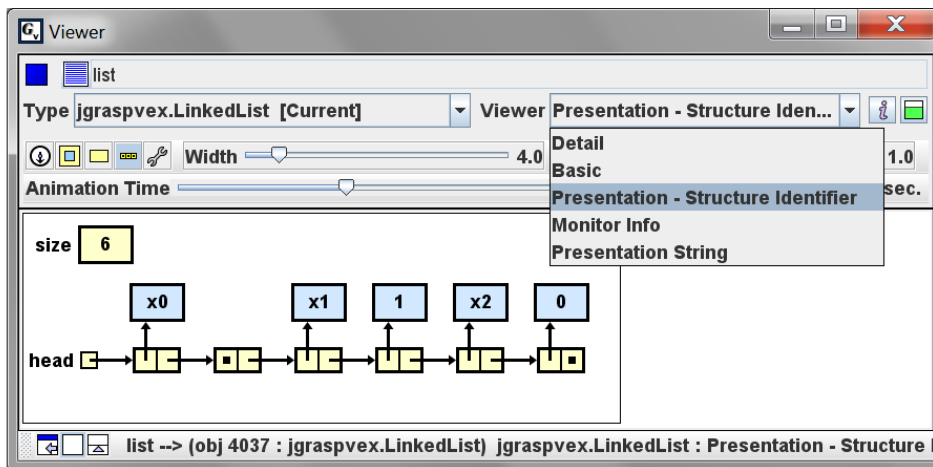


Figure 4. Viewer dialog for a linked list showing viewer selection list.

There are various ways to launch a viewer. A value may be dragged from its display in the debugger and dropped on a canvas or elsewhere. If it is dropped on a canvas, it will be added to that canvas; otherwise a viewer dialog will be created for it. Values displayed anywhere in the

debugger also have context (right click) menus from which an associated viewer dialog can be launched. Invoking a method with a non-void return type using the workbench mechanism always displays the result in a viewer dialog. The debugger also includes an "Evaluate Expression" function where the result of evaluation is shown in a viewer dialog. Viewer dialogs and viewer frames on the canvas also have an icon from which a copy may be dragged. Many viewers allow sub-values to be dragged out from elements of the display, or launched in a viewer dialog through the use of a context menu applied to elements of the display. On a canvas, viewers may also be created directly by specifying an expression and optional scope.

Viewers are associated with values in one of two ways. A value-based association fixes the value when the viewer dialog or canvas element is created. The viewer will then show changes to the value if it is an object but will not reflect changes to the reference or expression from which it was created. For example if local variable 'x' is assigned to the integer array {1, 2, 3} and at that point a value-based viewer dialog is created for 'x', any changes to the elements of the array will be reflected in the viewer display, but if 'x' is assigned to a different array, the viewer will continue to show the original one. By contrast, an expression-based viewer displays the result of evaluating an expression. Most often the expression would be a simple or compound name with casting applied where necessary, but it can be any arbitrary expression in the source code language that is being debugged. When a viewer is created by dragging a value, by default it will be created using expression-based association. Holding down the control key at the time of the drop will cause value-based association to be used, and the form of the drag icon indicates whether the drop will be by value or by name. Context menus on displayed values in the debugger provide a choice of opening a viewer with either value-based or expression-based association. The choice of value-based association is useful for observing changes to a particular

object after the variable, field, or array element from which the viewer was created is assigned a new value. For some values, such as subvalues in a viewer, only value-based association may be available. This is generally true whenever there is no available expression or no easily determinable expression for the value, or when the expression could be excessively long. For instance, the expression for the 1000th element in a linked list for which there is no indexed element access method would be very long (something like `list.head.next.next.next.next ...`). Values in value-based viewers are given internal names (`value_1`, `value_2`, etc.) which can be used in the expressions for expression-based viewers, as well as in the workbench and interactions systems, and in the expression evaluation table. Thus, these values become a part of the workbench and interactions systems. They cannot be used from within the program that is being debugged, however.

Values in viewers, whether opened in a value-based viewer or generated by evaluation of the expression in an expression-based viewer, are prevented from being garbage collected when the target language is Java. They are released when a value-based viewer is closed or an expression in an expression-based viewer is reevaluated. Thus, opening any value-based viewer may alter program execution, by delaying garbage collection of the value if it would otherwise be eligible. Since the timing of garbage collection is arbitrary, this does not change program semantics. The evaluation of the expression in an expression-based viewer could alter internal program state, just as it can for expressions in the expression evaluation table of the debugger. The only expressions of concern are ones that would be created manually by the user, and it is the user's responsibility to avoid using expressions that will cause problems. Expressions created by dragging values will always be compound names, possibly with array accesses, the evaluation of which will not alter internal program state. The viewers themselves can alter program state in

almost any way, and they must be designed in a way that prevents this unless intended. The entire system and the viewers themselves do create values in the heap of the running program, so they could have a minor effect on heap memory limits.

Value-based viewers have no associated scope. They are valid and will display changes regardless of the current program scope, as long as the value continues to exist. Expression-based viewers may have an associated scope. By default, they will have the scope of the program (or thread) that applied to the value display from which they were created, at the time they were created. So for example a viewer created from the display of the local variable 'x' in the method 'test()' in class 'C', called from the method 'start()' in class C, which was called from 'main()' in class 'C' would have the scope {C.main(String[]) : C.start() : C.test()} by default. Viewers created from places where scope does not apply, such as the workbench or expression evaluation table, will not have a scope by default. The expression used to generate the value in an expression-based viewer is only evaluated when the debugger is stopped in the associated scope. When the debugger is stopped elsewhere, the most recently determined value will be used and the viewer dialog or canvas will display an "out of scope" message. Thus, changes to the most recently determined value will continue to be displayed. So for example, if variable 'x' contains the integer array value {1, 2, 3} in method 'test()' and a viewer is created at that point, if a new array is assigned to 'x' in method 'test()', the new value will be displayed, but if the debugger next stops in method 'test2()' which also has a local variable 'x', the most recently determined value of 'x' from 'test()' will continue to be displayed, and changes to it will be reflected in the viewer. Changes to the 'x' in 'test2()' will have no effect on the viewer.

For practical implementation reasons, the scope elements used for scope comparison consist only of the class and method signature at the top of the call stack and the call stack depth, rather

than the class and method signatures for every frame of the call stack. The scope `{C.main(String[]) : C.start() : C.test() }` would be recorded internally as `{C.test() : 3}` and the expression in an expression-based viewer created with this scope would be reevaluated whenever the debugger stops in `C.test()` with a stack depth of three. This is generally sufficient to prevent confusion in the display of values, since to arrive at the same method with the same call stack depth through a different path, the program would first have to exit the scope in which the viewer was created, so any local variables in the expression would no longer exist on the call stack. Thus, a differently-scoped evaluation will only be done in cases where the originally-scoped evaluation is no longer valid.

Because there is no practical way to determine intra-method scope in Java, language-extensions have been added to the expression evaluation in expression-based viewers. Consider a method where local variable 'x' is used as a loop index within the scope of a loop, and in a loop that follows a different local 'x' is in scope and holds a string value, as shown in Figure 5. It would be useful and most often desirable if the viewer displayed the value of only one of those local 'x' variables. There is no reliable and repeatable way within the Java debugger API to identify one specific local variable of several with the same name and type in the same method, but if they have different types they can be distinguished by type. For that reason, when a viewer is created from a local variable in the debugger, it will be referenced by name and type, and tagged as a local variable. For Java, the language extension that enables this uses back ticks to avoid conflict with any valid Java syntax. The expression used for the local int variable 'x' would be `"local int` x"`, and the expression would be (successfully) evaluated only when 'x' was of type 'int' and the current scope matched other viewer scope requirements. Unsuccessful evaluation has a similar effect on the viewers as does a scope mismatch: the most recent successfully

determined value continues to be used and only internal changes to it are reflected in the display, while an "out of scope" message is shown. Debugger interfaces for languages other than Java may have similar issues, and similar language extensions could be needed.

```
void test() {  
    for (int x = 0; x < 10; x++) {  
        update(x);  
    }  
  
    for (int i = 0; i < 10; i++) {  
        String x = "x";  
        process(x + i);  
    }  
}
```

Figure 5. Example method where variable name "x" is reused.

Most viewers are updated whenever the debugger stops at a breakpoint, watchpoint, after a step, or through a manual thread suspension by the user. For some viewers though, periodic updating is useful and the viewer system has this capability. For example, in a viewer for a GUI window class it may be desirable to see changes as the window is moved or resized while in use, without having to stop the debugger repeatedly to see those changes. Viewers that make use of this feature may request any frequency of updating.

For all viewers, the effective declared type can be changed to any type to which the current runtime type of the value could be assigned. This allows the value to be displayed as if it were assigned to a variable of a different type. For example a viewer created from a variable with declared type String could be displayed as if the declared type were Object. A list of all possible assignment target types is provided. For viewers that closely reflect code structure, such as the

"Detail" viewer, this can change properties of the display. This is explained in the description of the "Detail" viewer in the following section.

3.2 Available Viewers

Viewers are added to the system using a plugin mechanism. Users may create their own viewers using an API described later in this work. In order to support dynamic data structure visualization with minimal work, a comprehensive set of viewers for the Java language and Java library classes is provided, including a viewer that automatically detects and analyzes user-created data structures.

The "Basic" viewer applies to all values and shows the instance fields of an object, the elements of an array, or a meaningful text representation for a primitive value. It is the default viewer for values added to a canvas for which a more specialized viewer does not apply. Fields and array elements can be dragged out of the basic viewer by expression or by value. Figure 6 shows the basic viewer for a linked list node.

The "Detail" viewer applies to all values, and displays the fields of an object or the elements of an array as an expandable tree. This is the default viewer for values displayed in a viewer dialog for which a more specialized viewer does not apply. Fields and array elements can be dragged out of the detail viewer by expression or by value. Like the value displays in the debugger, the detail viewer distinguishes primitive values from objects using icon shape. Objects and arrays are marked with a square icon while primitives are marked with a triangle icon. Icon color is used to show the relationship between the declaring type of a field and the declared type of the object being displayed. These icon properties are shown in Table 2. The declared type and runtime type of each field is shown. For Java objects, a unique id for each object is shown, so

that multiple references to the same object can easily be identified. Figure 6 shows the detail viewer for a linked list node.

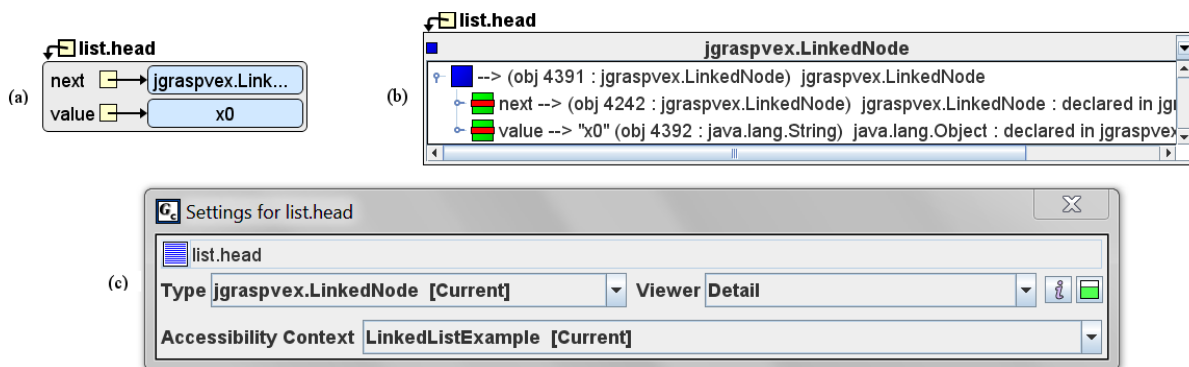














Figure 6. Basic (a) and detail (b) viewers for a linked list node, and detail viewer settings (c).

For each field or array element, the detail viewer shows a red bar if a field is inaccessible from the current program scope, a gray bar if it is not visible in the current scope, and a red and gray bar if it is both inaccessible and not visible. The definition of "visible" for this purpose is somewhat different from the Java Language Specification's [41] definition. A field or array element is considered "not visible" if, assuming that it was accessible, a cast would be needed to refer to it in the current program scope and given the current declared type of the object. This definition has a practical use since in some cases, such as using Java reflection methods or in the interactions and workbench features of the jGRASP IDE, accessibility restrictions can be overridden. Thus, a gray bar indicates "Assuming there is access to this entity, a cast is needed at this point to reference it", while a red bar indicates "In the current program scope, this entity is inaccessible from ordinary source code."

In addition to changing the effective declared type, which can be done for all viewers as described above, the effective current scope for a detail viewer can also be changed to see the effects of scope on accessibility and visibility [42]. A useful selection based on the runtime type

and current program scope is provided, and includes general categories, such as "public" and "public, protected, or package x" where x is the package of the runtime type of the value. Field icon colors and the accessibility/scope bars will reflect these changes. The workbench system allows methods to be invoked on the value in a viewer, and display and use of those methods will also reflect these changes. Thus, changes to the effective declared type can be used to answer the question "How could I use this value if it were assigned to a variable of type X?" and changes to the effective scope to answer the question "How could I use this value if it were referenced in class X?"

Table 2. Detail viewer icon properties.

Property	Value	Meaning
Icon Shape	 Triangle	Declared type is primitive.
Icon Shape	 Square	Declared type is object.
Icon Color	 Green	Field declared in same type as object's declared type.
Icon Color	 Orange	Field declared in supertype of object's declared type.
Icon Color	 Yellow	Field declared in subtype of object's declared type.
Icon Color	 Cyan	Field declared in interface implemented directly or indirectly by object's declared (class) type.
Icon Color	 Light Cyan	Field declared in interface implemented directly or indirectly by subclass of object's declared (class) type.
Icon Color	 Magenta	Field declared in class that implements object's declared (interface) type.
Icon Color	 Gray	Declared type of field and object are not linearly related through class or interface hierarchy.
Icon Color	 Blue	Element is not a field.
Bar Color	 Gray	Field cannot be referred to without casting the type of the containing object.
Bar Color	 Red	Field is not accessible in the current debugger context.

For classes that implement the `java.util.Collection` interface, the "Collection Elements" viewer displays the contents as a simple list. For those that implement the `java.util.Map` interface, the "Key/Value" viewer displays a simple list of the keys and values in the map. These viewers are referred to as "interface-based", because they do not depend on the internal data structure representation but apply to high level list and map abstractions. They are useful for viewing the contents of list and map structures when their internal structure is not of interest. Figure 7 shows an interface-based viewer for a linked list of Integers.

For linked lists, binary trees, hash tables, and array-based data structure classes such as `java.util.ArrayList` that are in the Java collections classes, the "Presentation" viewer displays the internal structure. The presentation viewers for linked structures do not actually analyze the structure, but show it as it would appear if it were complete and correct. Thus, they are not useful for understanding the internal workings of these structures, but do show their internal structure when in a complete and correct state, which should always be the case unless the debugger is used to step into library methods. They also only examine structure elements that are visible in the viewer (not beyond the edges of its window), and so are fast enough to display structures with large numbers of elements. The structure identifier viewer described in the next section does not have these limitations or this speed advantage, and can also be used for these same Java collections classes. Figure 7 shows a presentation viewer for a linked list of Integers.

A "Bar Graph" viewer displays a bar graph for arrays or lists of numeric values. This viewer is useful for understanding sorting algorithms and search algorithms that operate on sorted values. Figure 7 shows a bar graph viewer for a linked list of Integers.

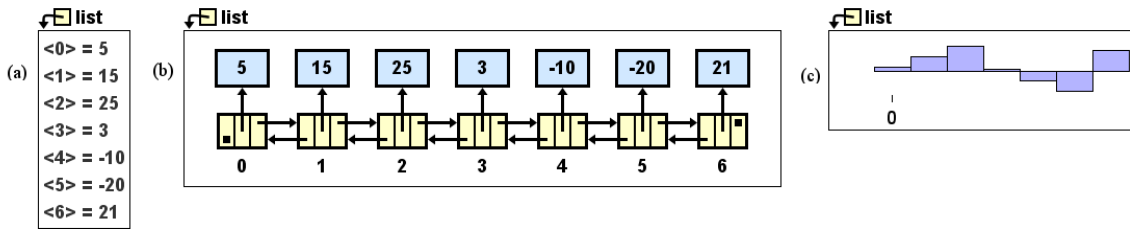


Figure 7. Interface-based (a), presentation (b), and bar graph (c) viewers for a linked list.

Numerous other specialized viewers are provided. Java strings may be viewed as formatted strings or in source code format (with escape sequences shown). Instances of `java.awt.Color` may be viewed as swatches of color. A viewer for `Images` and `ImageIcons` displays the actual image. A numeric viewer shows the numeric value in different number bases for integral types, and the number's binary representation and the details of how the value is determined from that representation for floating point types. The "Component" viewer shows the positions and sizes of a `java.awt.Component` GUI element and its subcomponents, and updates itself periodically while the debugger is running to show changes to the component hierarchy and to component sizes and positions. This is useful for examining GUI layout problems. All of these viewers are shown in Figure 8. The "Monitor Info" viewer lists the thread that owns and the threads that are currently waiting on a Java monitor object, and can be used to assist in identifying the cause of a deadlock.

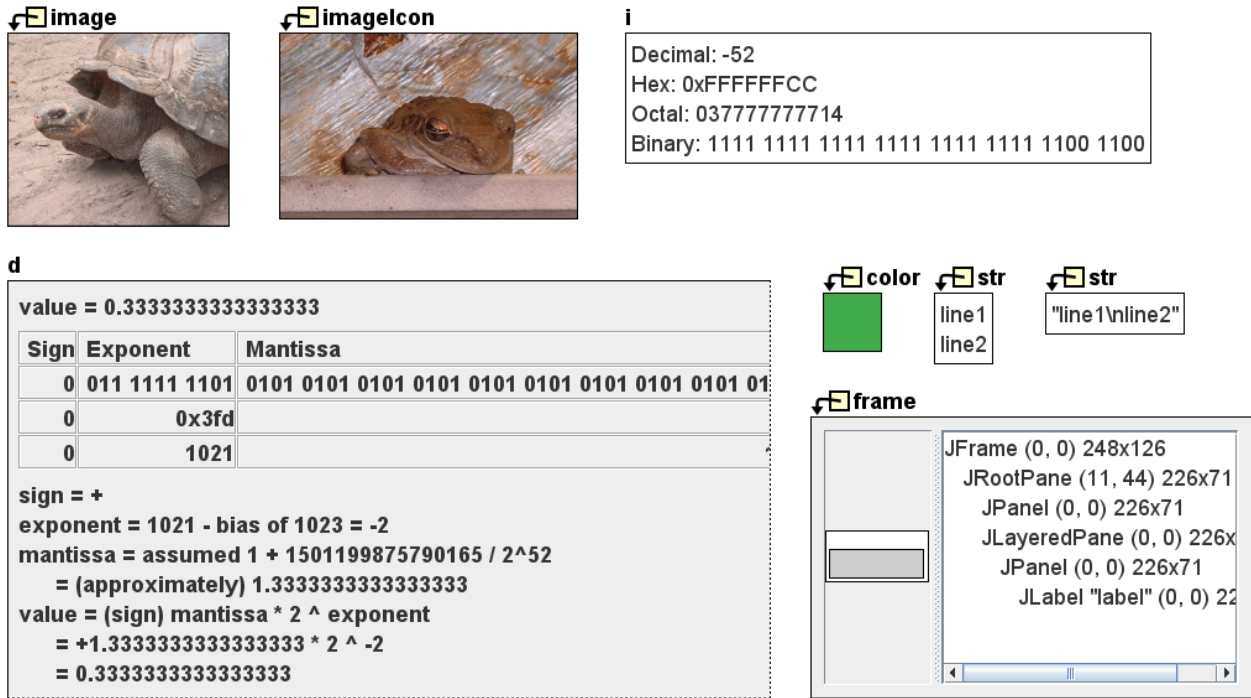


Figure 8. Various specialized viewers.

3.3 The Structure Identifier Viewer

The structure identifier viewer is the key to visualizing arbitrary data structure instances without requiring user effort. This viewer analyzes classes and attempts (most often successfully) to detect linked lists, binary trees, binary heaps, chained hash tables, and array or list-based structures such as stacks, queues, and lists. The display is similar to the presentation viewers for Java collections class data structures described above. Unlike those viewers, the structure identifier viewer traverses the links in linked structures to determine their internal structure and extent. Thus, these viewers are useful for exploring the inner workings of linked data structures as elements are added or removed, and in finding bugs in data structure and algorithm implementations. These viewers also display any nodes, sub-lists, or sub-trees referenced by method arguments or local variables in the current context, as well as method arguments or local variables that reference nodes in the structure itself, as shown in Figure 9. Nodes are shown

moving between and among the main structure display and local variable/argument display in an animated fashion as the debugger advances from one state to the next. The speed of these inter-step transitions is adjustable, and it may be disabled entirely if desired.

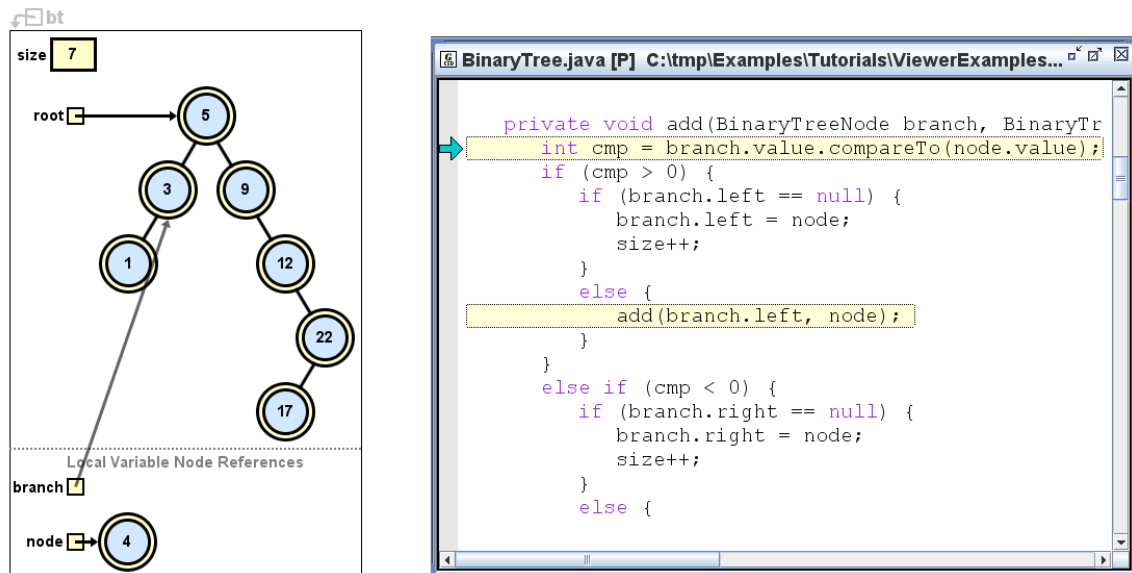


Figure 9. Structure identifier viewer and source code for a binary tree.

When a value is opened in a canvas or viewer dialog, if the structure identifier determines with a high degree of certainty that it is one of the target data structure types, the structure identifier will be the default and initial viewer for it. Thus, for a typical user-implemented data structure class or textbook example, no configuration is necessary. For each structure analyzed, one or more mappings between sets of code elements and data structure elements are determined, and for each mapping a confidence level is computed. The list of mappings found by the structure identifier other than the one with highest confidence that was used by default, if multiple mappings were found, may also be selected from a list. Configuration options allow the mapping to be fully specified, so that if automatic structure identification fails for a particular class, the viewer can still be used but with significant user effort. More often, structure identification may fail in a small way, such as in not identifying the "value" of a node correctly,

or missing the dummy node in binary tree with a dummy sink node. In that case, little effort may be required to correct the problem. Configuration also allows the text shown for a node value to be specified and to be different from the default display based on the value. For example, if a node has an integral value and the parity of that value is significant to its use, the parity could be emphasized in the text by showing the values 41 and 92 as "Odd:41" and "Even:92" rather than the default "41" and "92". By default only one value is displayed unless the structure is determined to be a map, in which case both the key and value are shown. Configuration allows an arbitrary number of values to be displayed for each node. Figure 10 shows the structure identifier configuration dialog for linked list and array-based structure mappings.

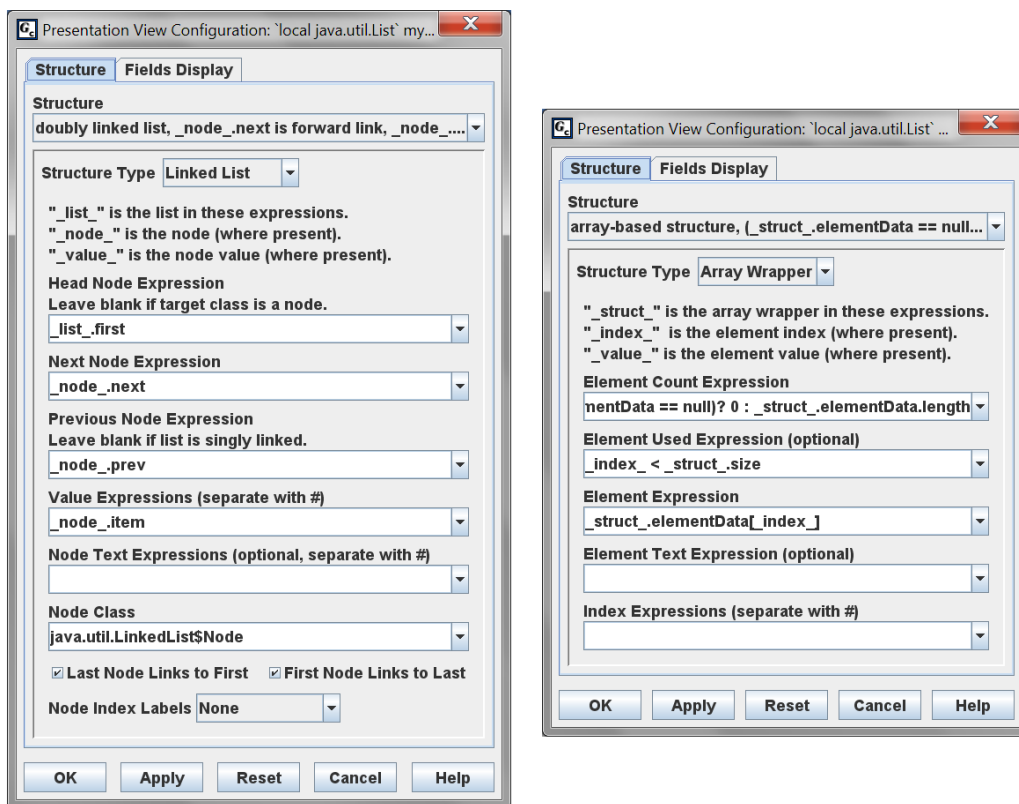


Figure 10. Structure identifier linked list and array-based configuration.

Because of the full structure traversal used, the structure identifier viewer may be too slow for instances with a large number of nodes. For example, if a linked list has one million nodes,

the structure identifier will traverse all of those nodes before displaying the list, and doing this traversal through a debugger interface is many times slower than it would be if done in the code itself. Initial limits of a few thousand nodes, the exact number depending on the particular structure type, prevent accidental long delays when a viewer is launched. When a limit is exceeded, the user is given the option of specifying a larger limit or keeping the current one and only viewing a portion of the data structure.

Full structure traversal allows the structure identifier viewer to indicate state changes at each step in a program, even for portions of the structure that are not visible (scrolled beyond the visible window). For example, text for a value that has changed in a structure may be shown in red rather than black. In addition to showing the intermediate state as nodes are moved into, out of, and among a linked structure, incorrect structure can also be identified. Incorrect links such as a bad reverse link in a doubly linked list or a link that forms a cycle in a binary tree are shown in red. These may be links that are temporarily incorrect as the structure is being modified, or they may be due to bugs in the data structure implementation.

Structure identifier viewers may be shown in any of four orientations: horizontal and left to right or right to left, and vertical top to bottom or bottom to top. Most linked list and array-based structures are shown left-to-right by default, but if a stack is recognized it will initially be shown bottom-to-top as it typically would be in a textbook diagram. Binary trees are shown with the root on top by default.

Binary trees may be shown with round nodes if the nodes are single-valued, and that is the default. For linked lists, array-based structures, and binary trees with rectangular nodes selected, two settings options can be changed for a total of four display modes. The default is to show

values embedded within nodes and to show the node divided into a rectangle for each value and one for each outgoing edge (which will display a black square if the outgoing edge reference is null). With the non-embedded option, values are shown externally with links from the node that contains them. With the simple option, nodes are displayed as undivided rectangles with outgoing edges anchored at node borders, and no indication of null edges other than the absence of the edge. Figure 11 shows the four display modes applied to a binary tree map node. Since this is a map, the node display shows both the key and value.

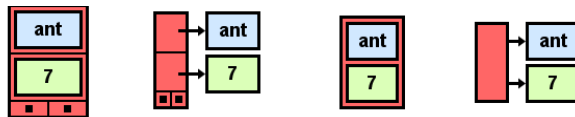


Figure 11. Node display modes for a binary tree map node.

3.4 Subviewers

In viewer dialogs and on the canvas, a subviewer pane may be opened. In this pane, any sub-value selected in a viewer will be displayed as if it were in a viewer dialog of its own. Typically, sub-values are selected by clicking on elements of the viewer display. Nodes and values may be selected separately where both exist. By default, the detail viewer is used to display sub-values, but any applicable viewer can be chosen and this selection will be remembered. The subviewer allows sub-elements of a structure to be quickly examined. Subviewers do not have subviewers of their own, since the display would become crowded and confusing. Figure 12 shows the structure identifier viewer for a linked list displayed in a viewer dialog, with one element value selected and the subviewer open. Note that the subviewer has the same controls that it would have if it were a separate viewer dialog.

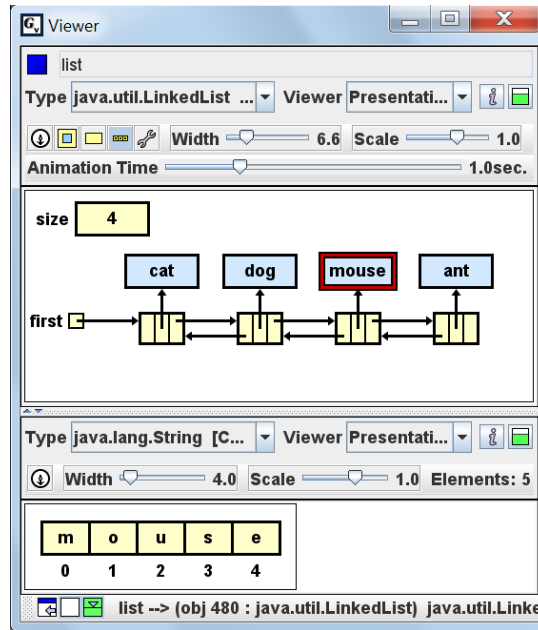


Figure 12. Viewer dialog displaying a view of a stack with subviewer open.

3.5 Auto-Step and Auto-Resume

In order to animate debugger operations, auto-step and auto-resume functions are provided. When auto-step is on, the next step operation in the debugger, whether it be a "step in", "step over", or "step out", will continue repeatedly with a pause after each step. When auto-resume is on, the next resume operation will repeatedly resume after the debugger stops at each breakpoint encountered. The speed of stepping or resuming is adjustable. If any viewers, such as the structure identifier, that provide smooth inter-step transitions are updated after a step or breakpoint, there will be an additional delay while that animation takes place.

With a canvas or viewer dialog open, the effect of auto-stepping or auto-resuming is to show an animation of the data structures and other values displayed in viewers, synchronized with display of the source code being executed, where the next line to be executed is highlighted at each step or breakpoint. Thus, a strong connection between source code and conceptual data structure display is maintained. Auto-resume with breakpoints at significant points in the

program can be used for an "interesting location" animation, and auto-stepping can be used for a line-by-line animation, though with the canvas certain (uninteresting) regions of code can be excluded, as described in the section that follows.

3.6 Canvas

The viewer canvas provides a way to combine multiple viewers in a single window and to store any changes to viewer configuration for later use. Viewers on the canvas can be arbitrarily arranged and sized. All but the selected viewer, if any, are shown "frameless" so that they appear to be combined in a single display. The frame of a selected viewer allows it to be moved, resized, or removed from the canvas, and provides a settings menu for it. Figure 9 shows a canvas containing viewers for an integer, stream scanner, string scanner, and list, which is used to visualize an algorithm that scans text and counts and stores individual words.

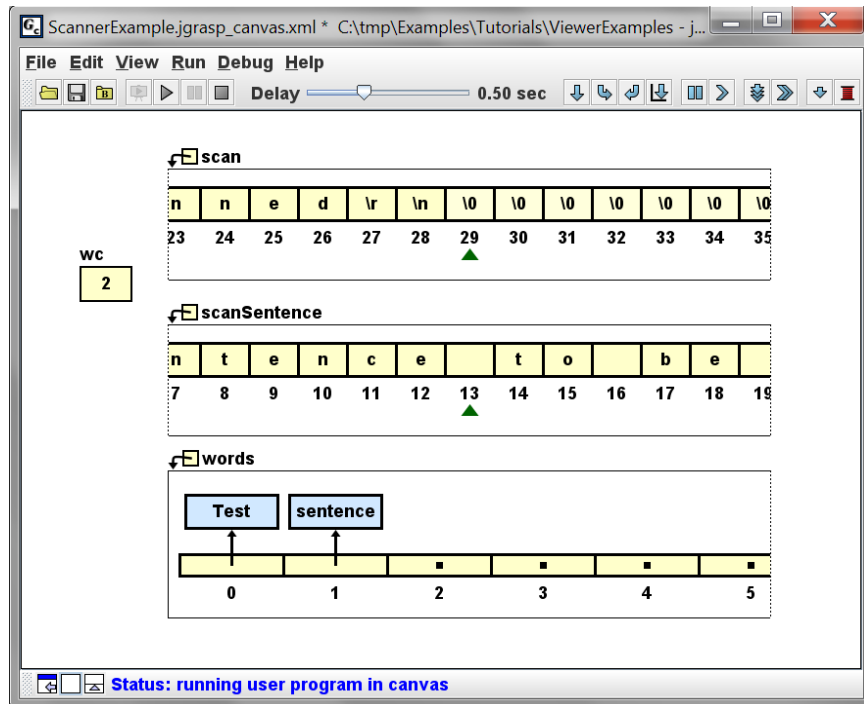


Figure 13. Canvas for a text scanning algorithm.

By default, viewers on the canvas are "auto sized". They will resize automatically based on content, up to some maximum size that is based on the OS desktop size. Dragging the frame of a viewer to resize it will turn "auto size" off, to allow a fixed size for the viewer. Typically, if a viewer is made larger than is currently necessary, the unused portion will be transparent, so that it still appears to be auto-sized unless selected. A settings menu item allows a resized viewer to be reset to the auto-size state.

Viewers on the canvas can be made transparent. For example, by default the structure identifier viewer for a linked list has an opaque background and a thin border around the extent of the list. In transparent mode, the border is not shown and the background is transparent, so that any viewers it overlaps will "show through". This can be useful for viewers that may have large areas of empty display space for particular code examples, so that smaller viewers can be placed in this empty space and a more compact display can be achieved.

The scope test for a viewer on the canvas can be specified as "full", "ignore depth", or "none". The scope of a viewer on the canvas can also be directly edited as shown in Figure 14. Ignoring the call stack depth can be useful when using a viewer to show the workings of recursive algorithms. For example, if a viewer is opened on an argument named "node" in a recursive binary tree traversal method named "visit", then in full scope test mode the viewer will continue to show the value of "node" at the recursive depth of "visit" in which the viewer was opened. In "ignore depth" scope test mode, the viewer will show the value of "node" at the current recursive depth of "visit" as the traversal progresses. There is no confusion about which "node" is being displayed, since the one with full scope testing would display an "out of scope" message when the recursive depth is different from the one in which the viewer was created.

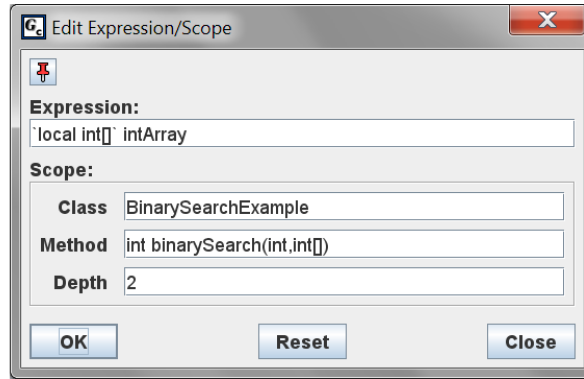


Figure 14. Viewer expression and scope editing dialog.

The contents of the canvas and configuration of its contents persist between debug sessions. Since there is no way to restore value-based viewers after a session ends, these will not persist if added to a canvas. A "canvas file" is used to store the arrangement of items on the canvas, as well as any per-viewer canvas settings (selected viewer, scope test, transparency, auto-sizing, etc.) and individual viewer configuration such as a change to the display text expression in a structure identifier viewer.

Text boxes for use as labels or to contain explanatory text may be added to a canvas. These may contain either plain text or html. Text boxes can be moved but not resized, and like the viewers they can be made transparent.

In order to simplify the continued use of a canvas as changes to class names, method names, and method arguments are made to its associated program, a name change dialog is provided, as shown in Figure 15. This dialog lists all the class names and method names and signatures referenced in viewer scopes on the canvas, and many of those referenced in viewer expressions. For each class name or method name and signature, a replacement may be specified, and a wholesale change can be made to the canvas. The dialog also displays the expression and scope

for each viewer on the canvas, so that the effect of changes can be examined before the change is finalized.

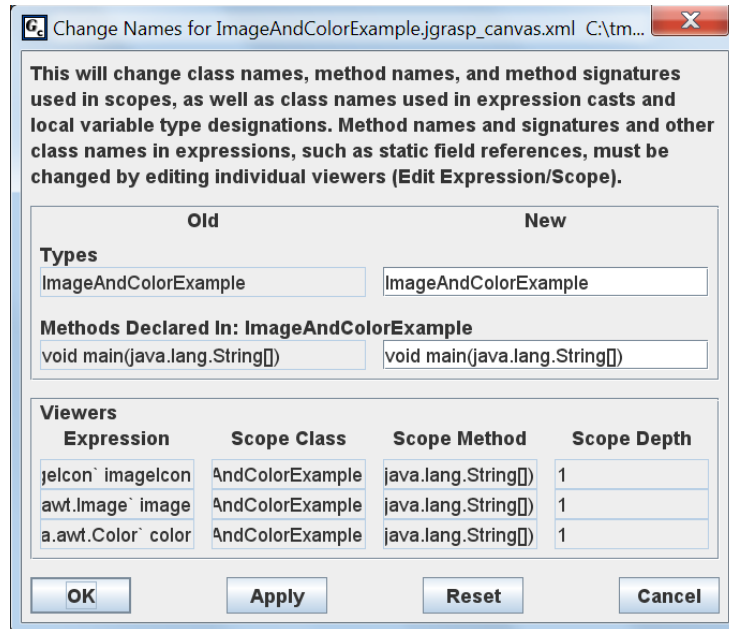


Figure 15. Canvas name change dialog.

In order to allow uninteresting segments of code to be skipped while animating using auto-stepping, an exclusions dialog is provided. This allows individual methods or entire classes to be skipped. A list of classes and methods in the file or project simplifies selection. This also includes general categories such as all constructors for a class and all constructors for all classes. These exclusions are saved with the canvas file. By default, Java library classes are skipped since stepping into library classes is not often instructive for any debugging operation. This is controlled by a separate debugger setting in the IDE that can be changed if necessary.

The canvas includes controls that reduce the number of steps necessary to launch an animation and automatically step through it. A "Run in Canvas" button starts the program which will then pause at its entry point (the first line of "main" for a Java application). A "Play" button then turns auto-step on and begins stepping-in. A "Pause" button turns auto-step off so that the

animation will stop after the next step. A "Stop" button stops the running program. When started using "Run in Canvas", the animation will be paused at the end of the program so that the final state of the animation can continue to be examined. In addition to these animation controls, the canvas has full debugging controls. These are compatible with the play/pause/stop controls, so that after a "Run in Canvas", "Play", and "Pause" to start an animation and stop at an interesting place, directed stepping can be done using "Step-in", "Step-over", or "Step-out", or the animation can be "fast forwarded" to a desired code location by using "Run to Cursor" or "Resumed" to a breakpoint. These controls are shown on the canvas in Figure 16. The IDE also provides a "Run in Canvas" button that will launch the program (which will then pause at its entry point) and open its associated canvas. If multiple canvases are associated with the program, the desired one can be selected from a list.

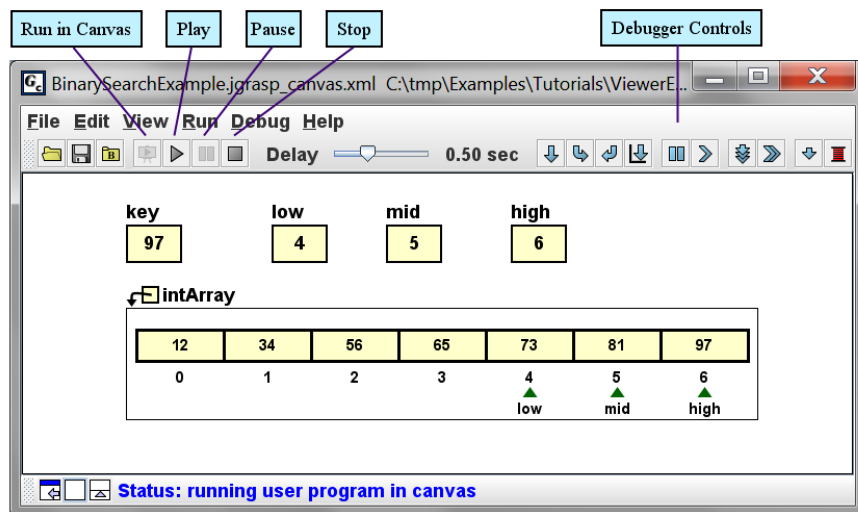


Figure 16. Canvas for a binary search animation showing animation controls.

Currently there are no reverse debugging controls on the canvas or in the debugger, but if a back end debugger interface with reverse debugging capability is integrated into the system in the future, reverse controls will be added. The current viewers would require no modification to handle reverse debugging, since they all depend on the current state of the program and for some,

the previous state, and the effects of the difference in past and present state (moving nodes, different colors for modified values, etc.) would have a useful meaning in either direction. If in the future viewers are developed that show the results of a longer past history in some way, they would need to be alerted when the debugging direction is reversed, so that any necessary changes to the display and internal state could be made.

3.7 Steps for the Creation and Use of an Animation

Following is a typical scenario for creating an animation. The debugger is started on the program of interest and stepped or run to a breakpoint at a useful place, at which point the "Open Canvas" debugger control is used to create a new, empty canvas. Alternately, "Run in Canvas" may be used to start the debugger and open an empty canvas in one step. Values are then dragged from the debugger variables display and dropped on the canvas. The items on the canvas are arranged by dragging them, resized if necessary, and configured if necessary such as by selecting the viewer for a value or changing per-viewer configuration. Exclusions are set if desired. The canvas file is saved. By default, the canvas filename will be such that it will be associated with the running program.

Launching an animation from a source code window in the IDE starts with clicking the "Run in Canvas" button. The canvas associated with the program is opened and the program starts and pauses at its entry point. Clicking the "Play" button starts the animation, which can then be paused at any time using the "Pause" button and resumed using the "Play" button. Launching an animation from the canvas starts with opening the canvas file by using the IDE's "File" > "Open" function or double clicking on the canvas file in the IDE's "Browse" pane, then clicking the "Run in Canvas" button on the canvas and continuing as described above.

3.8 Weaknesses

The visualizations in this system are generated from debugger information. This requires that the base debugger interface provide all information needed to render a visualization and for the structure identifier to examine type information. Most debugger interfaces do provide enough information, though code compiled in debug mode would be necessary for most languages and compilers. For Java this is not the case, as necessary information is included in ordinary class files. If Java code is compiled normally (not in debug mode), the local variable and argument names for local variable nodes in the structure identifier viewers will be replaced by numbered arguments, but otherwise all viewers will have full capability. Generally, any language and compilation system for which it is possible to produce object code that can be usefully debugged is a candidate for use in this system.

Because the system operates on top of a debugger, system capabilities may also be limited by debugger capabilities. As an example, the Java debugger interface currently provides no way to reverse execution, and so there is no way to step in reverse through an animated visualization. The system could record relevant program state and require all viewers to record their states at each update, but this would add a lot of complexity and would make the development of viewers much more difficult.

By strict definition, the jGRASP Visualization System is a data structure display system rather than an algorithm animation system. When using the system specifically for algorithm understanding, it may be desirable for a viewer to display information relevant to the state of the algorithm that cannot be determined or cannot easily be determined from the data structure and program state. For example, in an animation of a binary tree traversal algorithm, it may be useful to display visited and unvisited nodes in different colors, and there may be no straightforward

way to determine which nodes are visited from the program state alone. There is nothing to prevent a viewer from accepting viewer-specific communication added to the code. A "visited" field could be added to the node class for example, and the viewer could look specifically for a boolean field with that name. A more general mechanism could also be used. For example, a binary tree viewer might look for a class named "JGRASP_BT_Info", which would provide a list of visited nodes, and to which new nodes would be added as they are visited. Viewers that use such a mechanism for interesting event notifications could also be constructed, thus producing a true algorithm animation system. The current viewers are not designed to read interesting event notifications though, and the viewer API provides no predefined means of communication. The IDE also provides no way to automatically hide such communication in the source code display, which would obviously be desirable.

4 Viewer Implementation and Viewer API

The viewer implementation mechanism is best understood in conjunction with the viewer API. Each viewer is a plugin to the jGRASP IDE. Viewers must be Java classes, and may optionally be packaged in jar files. When placed in the plugins directory of the IDE or in a default or user-selected per-user plugins directory, they will be automatically detected and made available. To facilitate the development and testing of viewers, a "reload plugins" function allows all viewers to be reloaded without shutting down and restarting the IDE, and without stopping the debugger.

4.1 The jGRASP Debugger Interface (jgrdi)

The jgrdi is a sub-component of the viewer API that allows program entities to be evaluated and used to construct the viewer display. For example, in Java a variable's compile time type and run time type may be determined, array elements and object fields may be enumerated and examined, methods may be invoked and their return values examined, etc. The goal was to produce an API that is simpler to use than most debugger APIs, such as the Java Debugger Interface (JDI) or GNU Debugger Machine Interface (GDB/MI), targeted specifically for use in the viewer system, and largely language-independent. The level of detail needed to construct, for example, a typical data structure viewer, does not require much beyond examining object fields and possibly invoking accessor methods. The steps needed to explore such a structure may be identical for that structure's representation in many languages. It is entirely possible that a viewer created to display a particular data structure representation in one language may work without

modification for a representation in another language if the two representations use the same element names.

Using a debugger API often involves a lot of error handling. In the jgrdi, much of this error handling is internalized and automated in a way that is targeted at program visualization. For example, if the viewer will display a text representation of a field and that field value is not available because of some condition in the debugger (session ended, object no longer exists, etc.) it is usually sufficient that the viewer display an error message instead, such as "<not available, object has been garbage collected>". The jgrdi will return error text when a text value is requested and is not available, an empty list of fields when the fields of an object are requested and are not available, etc. A viewer created without consideration for error conditions will usually display something meaningful or nothing at all in those cases. Most of the viewers provided with the system use no error handling in their use of the jgrdi. When there is an error while evaluating the expression itself in an expression-based viewer, the viewer is not updated (the previous value will continue to be shown) and an error message is shown in a message bar at the bottom of the viewer dialog or canvas. Reducing the need for error handling greatly simplifies the development of viewers.

For viewers typically needed for data structure visualization, the jgrdi provides all necessary detail and capability needed to examine program values and types. For extremely language and target-specific viewers, a way to determine the underlying debugger API and access to it or to a direct wrapper around it is provided. For example, for Java targets running under the JDI, access to the JDI and the JDI values underlying program classes, values, and types is provided. This is used in the Java-specific "Monitor Info" viewer which shows the thread that owns and the threads that are waiting on an object that is a Java synchronization monitor.

Following are descriptions of key jgrdi interfaces.

DebugContext: a largely transparent interface representing the current debugger state. This also provides methods for the creation of primitive values in the debugger, access to program types by name, expression evaluation in the target language, access to local variables and arguments and their values in the current context, and access to scope information for the current context.

Local Variable: descriptive interface for a local variable or method argument. Methods provide the variable name, distinguish an argument from a local variable, and retrieve the declared type of the variable. Values in the current context can be retrieved through the DebugContext class using

`DebugContext.getValue(LocalVariable).`

ValueAndType: a wrapper class for a value and its declared type. This is the way a value to be displayed is passed to a viewer. This combined wrapper ensures that additional properties can be added as necessary to support languages other than Java without breaking existing viewers.

Value: a value interface. This interface can represent an object instance, primitive value, array, or other value. A summary of the methods in this interface is provided in Appendix C. There are no subinterfaces for more specific value categories. This facilitates language independence, since different languages have different categories of value, some of which may overlap in some languages but not in others (arrays are objects in Java but not in some other languages, for example). Viewers like the structure identifier can operate on this general concept of a value without dealing with language-specific issues. It is anticipated that

if the jgrdi is extended to languages that have both pointers and references, this distinction will be auxiliary information provided by the Value interface, and in most cases viewers will treat references and pointers identically. Methods of the Value interface provide information about the forms and subcomponents of the value that are available, and access to those forms and subcomponents. For example, a Java int form of a Value is accessible through

Value.toInt(DebugContext)

if the value is representable as a Java int. For Java, that would include primitive byte, char, short, and int values. Other methods provide the array size for arrays, array values by index, field values by name, and method references by name and signature. Because viewers may need to create and work with values that do not already exist in the program, methods are also provided to set field and array element values. This also allows for the possibility of viewers with more interactive capabilities.

Type: a type reference. This can represent an object, primitive, array, or other type. As in the Value interface, there are no subclasses for specific type categories in order to facilitate language-independent viewers. Methods provide general information about the type and its supertypes, access to fields and field values if applicable, the ability to set static field values for an object type, and the ability to compare types for assignability. The generality of this class should allow it to be used for aggregate types in non-object-oriented languages or in object oriented languages that also have non-object aggregate types.

Member: supertype for Fields and Method. Methods provide the member name, declaring type, accessibility information, etc.

Field: descriptive interface for a field or other structure member. Methods provide the declared type and access modifiers. Field values in the current context can be retrieved through

```
Value.getFieldValue(DebugContext, Field).
```

Method: descriptive interface for a method or function. Methods provide the return type and argument types. Instance methods may be invoked through

```
Value.invokeMethod(DebugContext, Method, Value[])
```

and static methods through

```
Type.invokeMethod(DebugContext, Method, Value[]).
```

All of the interfaces described above also provide access to their counterparts (if present) in the underlying debugger interface or debugger interface wrapper.

4.2 Viewer Interface

The Viewer interface provides the framework for interacting with the viewer system. A summary of the methods in this interface is provided in Appendix C. An encoded naming system relates a Viewer implementation class to the broadest source code type or category to which it applies, as described in section 3.1. Abstract methods require an implementation to provide a viewer name and priority. An abstract build() method is called when an entity becomes available for display (is added to a canvas or viewer dialog). At this point the viewer can opt out of displaying the entity or adjust its priority based on specific properties of the entity, such as its specific runtime type or some properties of that type. For example, the structure identifier will supply a very high priority if it identifies a data structure with high confidence and a very low priority otherwise. An abstract update() method is called whenever a program value is newly

available, such as when the debugger has stopped at a breakpoint or after a step, and provides a reference to the representation of the value in the debugger API and to its declared type. A typical implementation of this method would analyze the program value and produce an internal representation with the minimum detail needed for later display, possibly comparing it to the previously stored representation in order to mark or record changes to the structure (so that, for example, the text description of a field that has changed from one debugger step to the next could be displayed in red rather than black, or so that changes to a linked structure could be animated). A `toXML()` method allows the internal configuration state of the viewer to be stored along with a canvas. For example, if the user is allowed to display an array either vertically or horizontally, that orientation could be saved. The saved XML, if available, is provided in the constructor when the Viewer is created as part of a previously saved canvas.

If an intra-step transition is in progress, a parameter of the `update()` method indicates the total number of transition steps to be performed and the index of the current step. During an intra-step transition, the viewer would generally operate on the previous and new data structure state representations, and not re-analyze the current value, but the current value is still available if needed. An intra-step transition is initiated during a non-animation update, by calling a method on an `update()` control parameter to set the number of animation steps, time between steps, and initial delay. An initial delay longer than the transition step time can be useful to give the user time to recognize a pre-transition layout change. For example, when a node is added to a linked list, the size allocated to the list and to the area occupied by each local variable node should each be the larger of the size needed at the start and end of a transition, in order to allow unaffected nodes to remain in one position during the process. This makes changes during the transition clearer, but may require a readjustment of the layout before it starts and after it ends. The extra

initial delay gives the user time to mentally absorb this readjustment. After the last transition step, the delay for the auto-step or auto-resume that is being processed is generally sufficient for the same purpose, but an additional final step delay can be specified if desired.

Update notifications occur on a single debugger thread and all jgrdi method calls must be made from this thread, except for requests for later method invocations on the debugger thread. Java requires that all GUI method calls be made from the (single) Java Abstract Window Toolkit (AWT) event dispatch thread, except for requests for later method invocations on that thread and a limited number of other general purpose method calls. This thread separation is most easily enforced by determining and storing state information necessary for display during an update and using that information during a GUI update, where the state information contains neither jgrdi object references nor Java GUI object references. This also allows asynchronous viewer updates and GUI updates, which prevents potential deadlocks and unnecessary delays when either the debugger or GUI is busy. A development version of jGRASP includes thread checking on every jgrdi and Java GUI call (added globally using an aspect oriented programming system) that reports any violations of these threading restrictions, so the viewer framework and viewers included with the system are well tested for threading violations. This is important since such threading violations can cause errors that are timing-related or whose effects are delayed, and can therefore be very hard to diagnose. Figure 17 shows a sequence diagram for a simple viewer and viewer GUI update. The thread used for each message is indicated by color. In this example, a breakpoint occurs, the viewer requests one debugger value, and the viewer calls one GUI method.

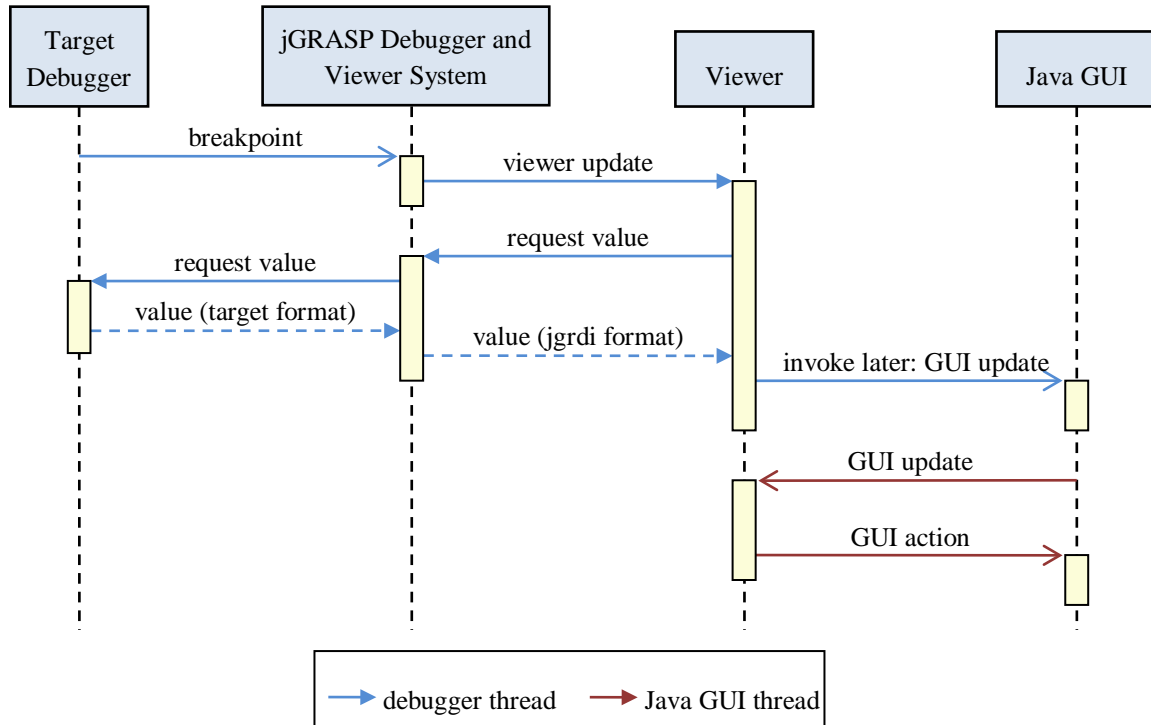


Figure 17. Sequence diagram for a viewer update.

4.3 Viewer Implementation Base Classes

The `ViewerRoot` class is an implementation of `Viewer` that provides the base GUI functionality for a typical viewer. It creates a scrollable or non-scrollable main window and for grid-like or table-like viewers, optional row and column headers. An `updateGui()` method is called after each `update()` or at other times when the display should change, such as after the viewer window is obscured and revealed, or after it is resized. A viewer that relies on Java GUI classes for display would only need to update those elements using the stored representation of the program entity. A viewer that displays text using a Swing text component that was added to the main panel, for instance, would just set the text for that component. For viewers that handle

their own display by painting on the main panel, a `paintMainView()` method is called after `updateGui()`.

Stock viewer classes are provided so that common types of viewers can be created with little effort. A text viewer, for example, only requires that the Viewer implementation set the display text when the target entity is updated. Similarly a text list view requires the implementation to provide a list of text values, and can be used to create a viewer that displays the contents of an ordered or unordered list structure such as an array, linked list, or hashed set, as a list of text representations of the structure elements. Viewer base classes that display structural representations of linked lists, arrays and array-based or list-based structures (stacks, queues, etc.), binary trees, and chained hash tables, only require an implementation to enumerate the structure elements. Multiple values may be supplied for each element so that keys and values in a map structure can be displayed separately.

4.4 Utility Classes

Colors and Sizes classes help to ensure uniformity among viewers by providing standard colors for nodes, values, disabled or unused nodes and values, and standard sizes for node edges, borders, separators, margins, etc.

The `PresentationElement` class simplifies node display in linked structures and element display in arrays and lists. Given the properties of the node, its values, and number of outgoing edges, a consistent node layout will be produced with standard colors and component sizes. This also handles nodes in different structure orientations (vertical vs. horizontal and forward vs. backward) automatically, and handles the various node display modes (embedded vs. non-embedded values, normal vs. simple view).

Other utility classes include a Subviewer so that viewers may provide their own subviewers in addition to the one in the viewer dialog or canvas, classes to assist in making viewer sub-element values draggable, a class that provides standard icons, drawing utility classes, and various GUI components that are useful in viewers.

5 The Structure Identifier Mechanism

Here the internal mechanics of the structure identifier are described, in terms of the way it associates data structure implementations to the graphical representations that will be used to display them. The methods used to automatically determine those associations by examining type information are explained. An automatic regression testing system for the structure identifier and its extension for tuning the system are also described.

5.1 Mapping Expressions

In order to examine a structure at runtime for future display, a set of mapping expressions in the target language is used for each structure category. The structure identifier supports four structure mapping categories: array-based or list-based structures such as lists, sets, stacks, and queues; linked lists; binary trees including binary heaps; and chained hash tables. Note that these categories are closely tied to display format and there is not a one-to-one correspondence between mapping categories and categories used for structure identification. For example, binary trees and binary heaps use entirely separate identification systems but the same mapping category. For each expression, relevant known structure elements are made available through synthetic variables. In the case of the binary tree, the root node expression has synthetic variables for the tree structure itself, and the left and right child expressions have that variable as well as a synthetic variable for the node being traversed. Some examples of how these expressions are used are presented below. The full list of mapping expressions and synthetic variables for each mapping category is described in Appendix A.

Many of the mapping expressions are necessary for traversing the structures. These map code elements to data structure elements. For example, for a binary tree, the traversal expressions are used to determine 1) root node of the tree, 2) left child of a node, and 3) right child of a node. In the case of the Java collections class `java.util.TreeMap`, which is a map implemented as a red-black tree, the following traversal expressions could be used:

Root Node: `_tree_.root`

Left Node: `_node_.left`

Right Node: `_node_.right`

where `_tree_` is the synthetic variable for the tree structure, and `_node_` is the synthetic variable for the node being traversed. For binary trees, an optional dummy node expression is also useful, since some binary tree representations use such a dummy "sink" node instead of null children. Unlike other nodes, the dummy node may have multiple parent nodes, so it is displayed differently, centered at the bottom of the tree.

Additional expressions are useful for specifying structure display elements and properties. The primary of these is the value expression for a node in a node-based structure or an element in an array-based structure. Since each node or array element may have multiple relevant values, any number of expressions is allowed. If there are multiple values, the expressions are separated by hash symbols (`#`). In the case of `java.util.TreeMap`, the value expressions could be:

`_node_.key # _node_.value`

Thus, for `java.util.TreeMap`, both the map key and value associated with each node would be displayed. By default, these values are displayed using a useful text representation such as the

toString() result for Java objects, the String.valueOf(primitive_value_type) result for primitives, and the first few elements for arrays. Future plans include display of image values as images.

Another useful expression for node-based structures is a node class. This allows nodes outside the structure that are referenced by local variables and method arguments to be displayed, as well as local variable and method argument references to nodes in the structure. Nodes can then be shown moving in and out of the main structure while stepping through a program. In the case of java.util.TreeMap, the node class would be java.util.TreeMap\$Entry.

To illustrate a more complex example of mapping expressions, consider a Java int array used to hold a binary tree. Each three consecutive ints in the array specify a node. The first is the index of the left child, the second is the index of the right child, and the third holds the value. A negative index is used to represent a null link. So the int array { 3, 6, 0, -1, -1, 1, -1, -1, 9 } would represent a tree containing three nodes, with "0" at the root value, "1" as the left child value, and "9" as the right child value. In this case, there is no node class in the implementation, so Integer can be used as a stand-in node class, to hold the node index. The root expression would then be an Integer with value 0, or null if the array is empty. The left and right expressions would produce the Integer index of the left or right link, using the synthetic variable `_node_` which in this case would be an Integer holding the index of the node being traversed, or they would produce null if the index of the left or right link is -1. These expressions could be specified as:

```
Root Node: (_tree_.length > 0)? Integer.valueOf(0) : null
```

```
Left Node: (_tree_.length > _node_.intValue() &&  
           _tree_[_node_.intValue()] >= 0)?  
           Integer.valueOf(_tree_[_node_.intValue()]) : null
```



```
Right Node: (_tree_.length > _node_.intValue() + 1 &&
             _tree_[_node_.intValue() + 1] >= 0)?
             Integer.valueOf(_tree_[_node_.intValue() + 1]) : null
```

```
Value Expression: (_tree_.length > _node_.intValue() + 2)?
                  _tree_[_node_.intValue() + 2] : 0
```

In this case there is no node class and no way to distinguish local variables and method arguments that refer to tree node indices from other integer variables, so there is no meaningful node class expression, and the structure identifier will not show local variable nodes and subtrees or smooth transitions between states.

Note the conditional tests in the expressions. They are there to avoid exceptions when the structure is in an invalid state. In all mapping expressions, care must be taken to avoid exceptions during evaluation. Evaluation must be possible even when the structure is in an incomplete or invalid state, such as when it is not yet initialized or when an element is in the process of being added, since the viewer that is using these expressions could be updated for display at any time. The structure itself will never be null during evaluation though, so there is no need in this example to test for (`_tree_ != null`).

For this example, it would also be useful to see the array representation in the viewer. To do this, the following expressions could be specified:

```
Array Size: _tree_.length
```

```
Array Element: Integer.valueOf(_tree_[_index_])
```

The viewer resulting from supplying these expressions and applied to the int array { 3, 6, 0, -1, -1, 1, -1, -1, 9 } is shown in Figure 18.

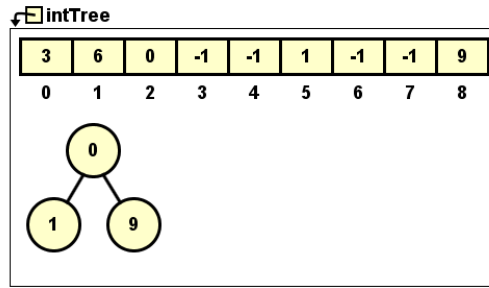


Figure 18. Viewer for an integer-encoded binary tree.

5.2 Automatic Determination of Mapping Expressions

The structure identifier mechanism analyzes classes or other code structures to search for potential data structures. Potential mappings from code structure to data structure are developed and for each one found, a confidence value is computed. This is done primarily through the analysis of type structure and type component names, though runtime structure may be used in some cases. Name analysis is language dependent, and the current structure identifier targets English language source code only. Relevant names and name components and their effect on confidence in different contexts are all stored as tabular data, so that tables for other languages could be substituted or added in the future.

The structure identifier has modules for recognizing singly and doubly linked lists, linked binary trees, chained hash tables, binary heaps, and array-based structures such as stacks and queues. Some of these modules may identify data structure sub-components using other modules. For example, in a chained hash table, the chain for each slot is a linked list, and the linked list is identified using the same module that recognizes a linked list value for a viewer. The array and list-based recognizer module is recursive, since stacks and queues may use the array-based lists that it also recognizes to hold their contents. The modules and dependencies among them are shown in Figure 19.

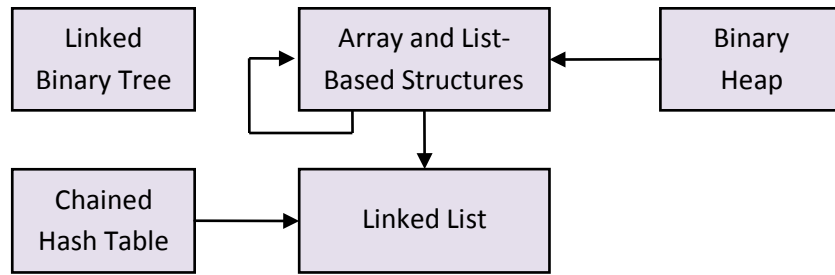


Figure 19. Structure identifier data structure recognition module dependencies.

For each module, the runtime type of a value being evaluated by the structure identifier is analyzed to see if its configuration matches a known type reference pattern. As this is done, potential mapping expressions are determined and confidence values computed. For example, a linked list typically is a wrapper class with a field that has a reference to the head node (and possibly one to the tail node), while the node class has a same-class reference to the next node (and possibly a back reference). These references might be through fields or accessor methods, and both fields and apparent accessor methods are examined. The type reference patterns for the supported linked structures are shown in Figure 20. The dark orange links represent optional references and the dark orange dashed rectangles show optional portions of each pattern. Linked lists and linked binary trees can be recognized without wrapper classes (so that the head node or root node "is" the list or tree, respectively), so these wrapper sections are optional. Note that these are the patterns currently used by the structure identifier, but changes to these patterns may be made in the future in order to recognize a wider variety of data structure implementations.

Each type examined by the structure identifier may match each pattern in multiple ways (using different fields or accessor methods, and different optional pattern components), and multiple patterns may be matched. For each match, the confidence value depends on class names, field names, the number of optional components matched, and for node classes, the number of self-references that were not matched to the pattern. For example, a class named

"LinkedList" with a field named "head" of a class type named "LinkedList", where the LinkedList class has a field named "next" of type LinkedList, would have a high confidence as a linked list, because those are very common and expected class and field names for a linked list. In the linked list recognition module analysis, a potential node class with three same-class fields would have a lower contribution to confidence than one with one or two same-class fields, and one with "node" or "element" as part of the class name would have a higher contribution to confidence than one without. So for linked structures, there is some dependence on class, field, and method names in addition to type structure. Generally though, common implementations of linked data structures will be correctly identified regardless of the names used.

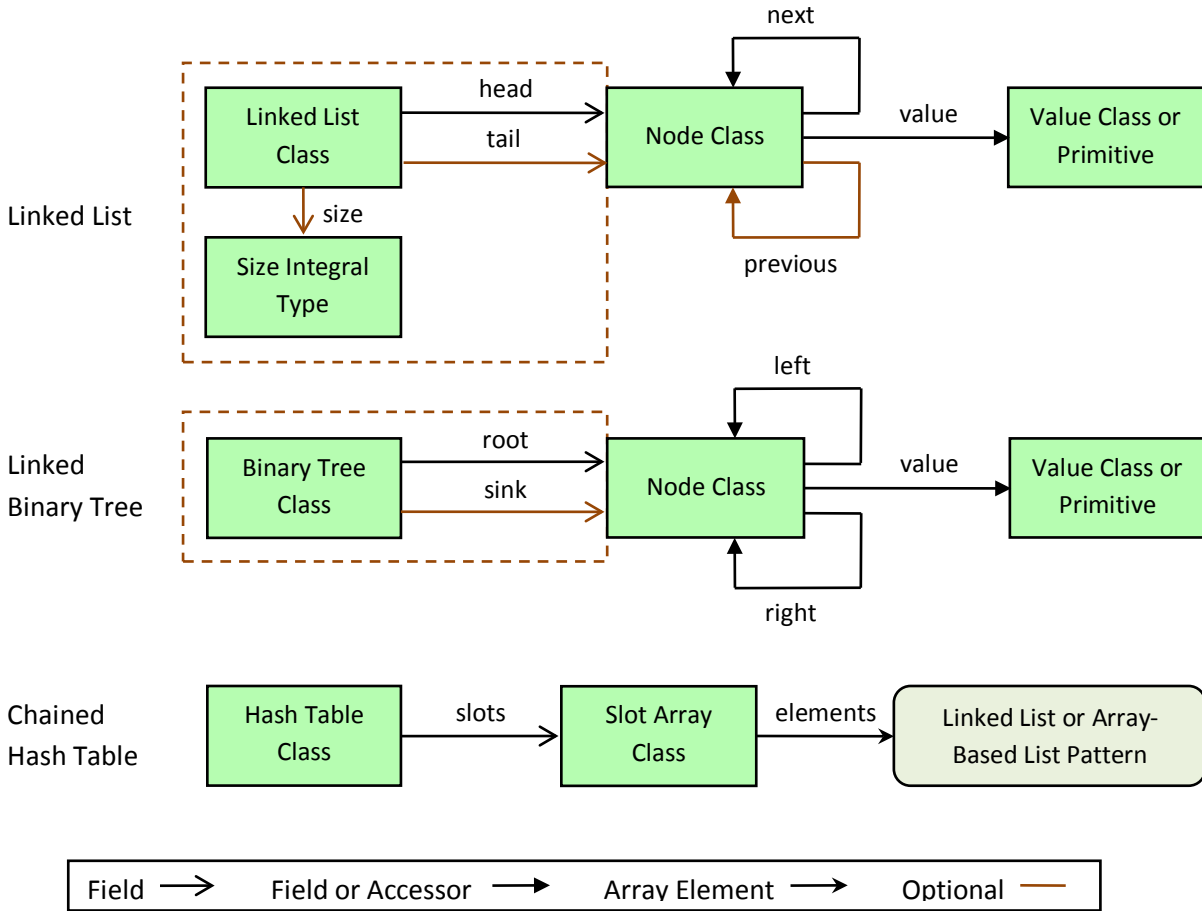


Figure 20. Type reference patterns for linked structures.

For array and list-based structures such as stacks and queues, the type reference pattern is simply a class containing an array, linked list, or array-based list, and one or more integral indices. Confidence levels for these potential mappings depend highly on the class and field names. To handle the case where elements may be stored in a linked list, the linked list analysis component of the structure identifier is applied to potential element list fields, and similarly the array-based list analysis component is applied to handle the case where elements are in a wrapped array.

Although binary heaps are conceptually binary trees, their implementation is array or list-based, so their identification is also strongly name-dependent. A separate structure identifier

module handles binary heaps. The analysis uses runtime values to distinguish heaps where the zero-element is used from those where it is not. For empty heaps, it is assumed to be unused (the most common implementation). Thus, for binary heaps where the zero element is used, the structure identifier will incorrectly identify it if it is empty when the viewer is created. Currently, this is the only module in the structure identifier that makes use of data structure contents in its analysis.

Analysis results for a particular type are cached during a debug session, so that analysis is done only the first time a type is encountered. To reduce the analysis time, analysis paths with low confidence are terminated before being fully explored in many cases. Analysis time is generally not disruptive to the user, taking a quarter second or less on modern hardware. For a typical data structure class, a few dozen possible structure mappings are found. Those with very low confidence are discarded. If all have low confidence, the structure identifier viewer will decrease its priority so that it will not be the viewer used by default for that class. Otherwise, the mapping with the highest confidence will become the one used by default by the structure identifier viewer. The others are made available in a list on the structure identifier viewer configuration dialog, so that it is easy for the user to correct the case where the structure identifier has correctly identified and mapped the data structure, but has given an incorrect mapping the highest confidence.

5.3 Automated Testing of the Structure Identifier

In order to allow previously unrecognized data structure code to be successfully identified and mapped, changes to the structure identifier mechanism, either in the weights involved in computing confidence levels or in the type reference patterns used, will occasionally be made.

These changes may cause data structures that were previously identified and mapped successfully to be misidentified or to fail to be identified. To help reduce performance degradation, an automated regression testing system was developed.

The test database consists of a large number of data structure code examples. Each program contains one or more data structures to be tested. An XML test file specifies a series of tests. For each test, the program to be run and line number at which to stop is specified, along with the expressions, typically local variable or field references, for the values to be tested with the data structure identifier. For each expression, one or more acceptable data structure mappings are specified. The testing system is integrated into the jGRASP IDE. In processing a test file, the debugger is automatically started for each test and stopped at the specified line number. Each corresponding test expression is evaluated and the result is sent to the structure identifier. If the structure mapping with the highest confidence does not match one of the acceptable mappings, an error is reported. In this way, hundreds of data structure examples can be tested with no manual intervention. To simplify adding new examples to a test file, a developer version of the IDE has a feature that produces an XML version of the current structure mapping for a structure identifier viewer.

Figure 21 shows an XML test file that verifies correct operation of the structure identifier when applied to a Java `LinkedList` instance. The test system is directed to debug `ViewerTest.java`, stop at line 7, apply the structure identifier to the value in the local variable `"javaUtilLinkedList"`, and check that the mapping specified is identical to the highest-confidence mapping that the structure identifier produces. If the mapping does not match the highest-confidence result, an error message will be reported. The test source file, shown in Figure 22, simply creates an empty `LinkedList`. A dummy assignment statement is added so that the

debugger has a place to stop after the linked list is created. This example is minimal for illustrative purposes. Starting the debugger repeatedly is time consuming compared to running a structure identifier evaluation, so typically, hundreds of test objects for different test classes would be created in a single source file, and the XML test file would specify tests for each of them from a single breakpoint location if practical.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<SITestData>
  <Filename>ViewerTest.java</Filename>
  <ClassName>ViewerTest</ClassName>
  <Breakpoint>7</Breakpoint>
  <Variable>
    <Name>javaUtilLinkedList</Name>
    <Mapping>
      <Type>LinkedList</Type>
      <HeadExpression>_list_.header</HeadExpression>
      <NextExpression>_node_.next</NextExpression>
      <PreviousExpression>_node_.previous</PreviousExpression>
      <ValueExpression>_node_.element</ValueExpression>
      <NodeType>java.util.LinkedList$Entry</NodeType>
    </Mapping>
  </Variable>
</SITestData>
```

Figure 21. Structure identifier test file for a java.util.LinkedList.


```
/** Structure identifier test file for LinkedList. */
public class ViewerTest {

    public static void main(final String[] args) {
        java.util.List javaUtilLinkedList = new java.util.LinkedList();
        // Breakpoint on the following line. */
        int dummy = 0;
    }
}
```

Figure 22. Test source file for a java.util.LinkedList.

5.4 Automated Tuning of the Structure Identifier

The automated testing system described above has been extended, on an investigative basis, for use in the automated tuning of the structure identifier. The structure identifier mechanism uses various weights to determine confidence values. These may be related to structural or name-based properties. In the development of the structure identifier, these weights were determined manually by trial and error, guided by intuition. Automated tuning attempts to determine values for these weights that will result in better structure identifier performance.

The tuning system was implemented for the linked list identifying module of the structure identifier. The system operates in a similar way to the testing system, but the entire test file is run multiple times while the weights are adjusted between runs using simulated annealing. The objective function has the goal of high confidence for acceptable mappings and low confidence for others. The idea is that even if the structure identifier is successful when applied to a particular data structure implementation, if there is a large gap in confidence between correct and

incorrect mappings then similar data structure implementations are more likely to be correctly recognized. For this to be meaningful, the test file must contain all possible acceptable mappings (while the testing system only needs those that the structure identifier is highly likely to produce). Thus, all mappings produced by the structure identifier for each example must be categorized as acceptable or unacceptable, most easily by selecting each of them in the structure identifier viewer configuration dialog while debugging, and examining the result.

The objective function for the simulated annealing gives a high penalty when the confidence gap is below a certain value, which will be referred to as the "penalty minimum", for a test case, and essentially an infinite penalty when the structure identifier fails completely for a test case. Complete failure happens when all acceptable mappings are not recognized, or when there is at least one unacceptable mapping with a higher confidence than one acceptable mapping. These results are illustrated in Figure 23 for a test case with two acceptable mappings. If no failure conditions occur and the penalty minimum is not violated, then the objective function is simply the sum of the confidence gap for each test case, normalized to the minimum confidence of the acceptable mappings (this is a maximization function). The penalty for each gap that is greater than the penalty minimum is large enough so that results are stratified by the number of such penalties. In other words, all possible objective function values with n penalties are higher than all possible values with $n+1$ penalties.

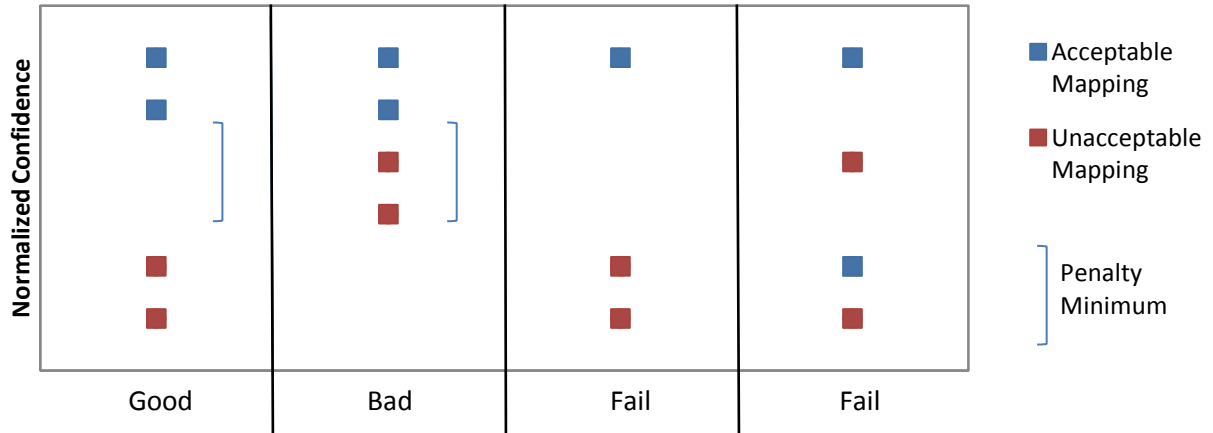


Figure 23. Example mapping confidence result sets.

The tuning system was used to determine the weights used in the current implementation of the structure identifier's linked list analysis module. For the testing set that was used, this resulted in a significant improvement in the value of the objective function over its value using the manually determined weights, even if penalty values are ignored. The number of penalty values went from a few percent of test cases to none. No validation was done to see if this improved the performance of the structure identifier on previously untested linked list examples, though intuitively this seems likely.

6 Evaluation

The jGRASP Visualization System has been evaluated in three ways. First, the effectiveness of the structure identifier in recognizing data structures in previously unencountered code was formally tested. Second, formal code understanding experiments using the viewers were conducted. Third, feature utilization data from users was collected and analyzed.

6.1 Structure Identifier Applicability to Arbitrary Source Code

An early version of the structure identifier was tested against previously unencountered textbook examples in an experiment conducted by Lacey Montgomery in conjunction with other jGRASP team members including the author of this work [43]. The goal was to determine how well the structure identifier would perform against typical textbook data structure source code, which should be similar to student implementations. Examples from twenty English language data structures textbooks that provided working Java source code were used. Each of these had at least one stack, queue, linked list, and binary tree example, while some also had binary heap and/or chained hash table examples. For some textbooks, multiple examples for a single data structure that were judged to be sufficiently distinct were used. Each of these examples was tested by compiling the code, setting a breakpoint, and opening the relevant data structure variable in a viewer dialog. Where the initial test failed and the data structure was a wrapper class, the field referencing the contained data structure was opened in a viewer dialog. Where the initial test identified the structure type correctly but had some minor errors in the structure mappings, configuration using the structure identifier configuration dialog was attempted.

Before this testing, the structure identifier did not attempt to recognize stacks, queues, and binary heaps that used structures other than arrays to contain the elements. This obvious omission was noted and corrected before determining final results. Once this was done, 82% of the examples worked correctly. Another 10% were wrapper classes for which the structure identifier worked correctly on the field referencing the contained data structure. Another 5% worked correctly with minor manual configuration, most frequently the specification of the value expression. The remaining 3% failed with no simple correction evident through manual configuration. In Figure 24 these four categories of results are shown as "Pass", "Pass - open on field", "Pass - configure viewer", and "Fail" respectively, and the figure shows total and per-data-structure result percentages in these categories. Table 3 shows per-data-structure result numbers for each result category.

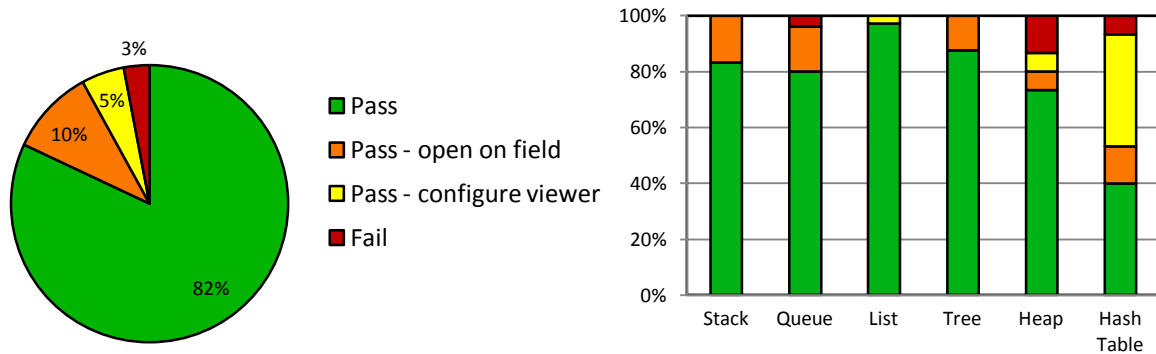


Figure 24. Structure identifier total and per-data-structure test result percentages [43].

Table 3. Structure identifier per-data-structure test result numbers [43].

	Pass	Pass - open on field	Pass - configure viewer	Fail	Total
Stack	20	4	0	0	24
Queue	20	4	0	1	25
List	36	0	1	0	37
Tree	28	4	0	0	32
Heap	11	1	1	2	15
Hash Table	6	2	6	1	15

Apart from the change noted above, there were some correctible weaknesses in the handling of wrapper classes for all data structures. These issues have since been corrected, and the structure identifier has been improved and tuned in many ways so that all of these examples are now correctly handled. The examples are still used in the automated regression testing for the structure identifier. Of course, the structure identifier will never recognize all possible data structures, since some may have unusual structure and/or unusual element names. Few cases have been found where it did not work with textbook data structure implementations or student code. Problems that were found have all been correctible, and have been corrected when encountered.

6.2 Code Understanding

Code understanding experiments involving the viewers were conducted by Jhilmil Jain in conjunction with other jGRASP team members including the author of this work [44] [7] [8], as the focus of her dissertation research. Although the structure identifier had not yet been developed when these experiments were performed, the viewers used were very similar in form

and capability to the current structure identifier viewer. They parsed the links in linked data structures to show actual structure including incorrect links, showed structure nodes referenced by local variables and method arguments, and used smooth transitions between states to show nodes moving into, out of, and among linked structures.

For all six experiments described below, test and control groups were balanced based on two programming skills tests that were developed and given to the subjects prior to the experiment. The first test scored the subjects based on their ability to detect and correct 25 common logical errors in data structure implementations that were reported in the literature [45] [46]. The second test scored the subjects based on their ability to read and trace code, and used eight of the ten questions from the test given in a multi-institutional study of code reading and tracing skills in novice programmers [47]. Subjects were sorted and divided into pairs based on their total scores. For each pair, one subject was randomly selected and assigned to the test group, and the other was assigned to the control group.

The first experiment tested the effect of the viewers on code development. Two groups of undergraduate computer science students were compared. Both had access to the jGRASP IDE, but for the control group the ability to launch viewers was disabled. Subjects were asked to implement `delete()`, `insert()`, `contains()`, and `entry()` methods for a linked list, where `entry()` is an indexed element access method. A skeleton linked list class was provided. For the control group only, this class included a method for generating a string representation of the list contents. Subjects were given unlimited time. The test group produced correct implementations more often than the control group for each one of the implemented methods. The difference was easily observable, as the test group had 23% more correct implementations on average. Average time taken by the test group was also slightly lower than for the control group. The difference in

accuracy between groups was statistically significant. Figure 25 shows the number of students in each group who correctly implemented each method.

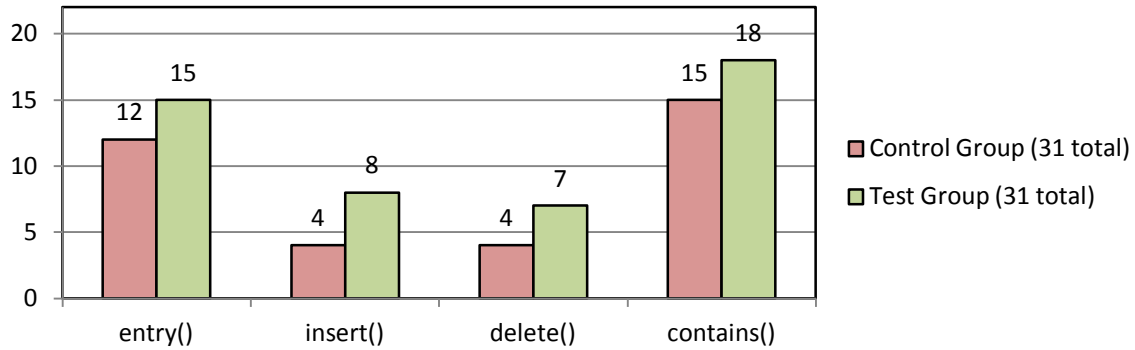


Figure 25. Number of students who correctly implemented each method in first experiment.

The second experiment tested the effectiveness of the viewers for debugging purposes. As in the first experiment, both groups had access to the jGRASP IDE, but for the control group the ability to launch viewers was disabled. Subjects were asked to identify and correct bugs in `add()`, `insert()`, `delete()`, and `contains()` methods of a linked list, which was seeded with nine bugs. Subjects were again given unlimited time. The test group identified more bugs, corrected more bugs, and introduced fewer new bugs than the control group. Differences were easily observable, as the test group identified bugs 20% more accurately and corrected bugs 15% more successfully than the control group, and introduced half as many new bugs as did the control group. Average time taken by the test group was slightly higher than for the control group. The difference in accuracy between groups was statistically significant. Figure 26 shows the number of students in each group who correctly performed each task (identified all bugs, corrected all bugs) and who did not introduce new bugs in each method.

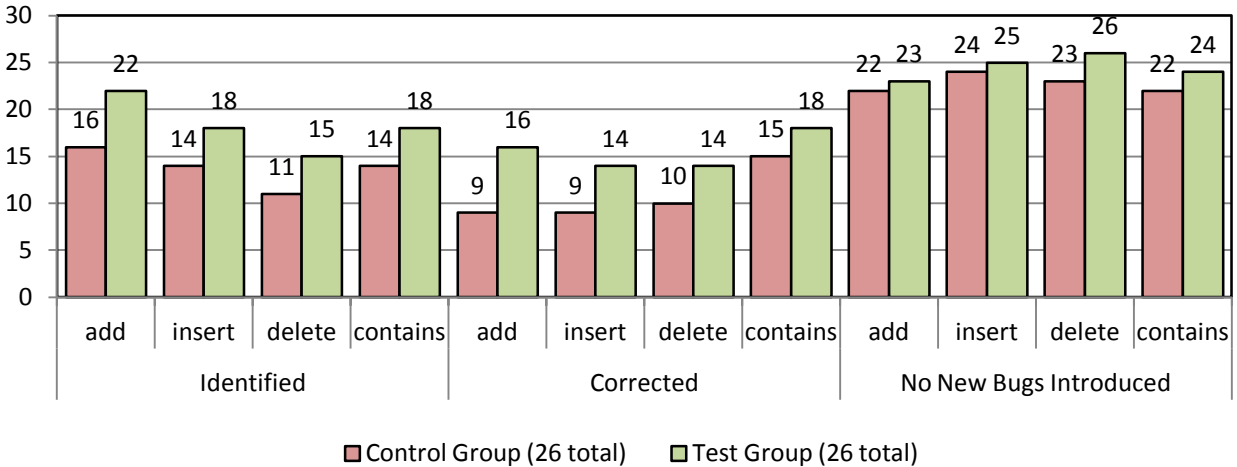


Figure 26. Number of students who correctly completed each task in second experiment.

The third and fourth experiments performed focused on binary tree data structures. In the third, subjects were asked to implement a breadth-first traversal. In the fourth, they were asked to identify and correct bugs in methods for adding an element, removing an element, in-order traversal, post-order traversal, and searching the tree. As in the first two experiments, differences in average accuracy/success between the test and control groups were significant in size and were statistically significant. Unlike the first two experiments, average time taken by the test group was considerably lower than for the control group. Figure 27 shows the average raw score for the third experiment and the average score on each task in the fourth experiment.

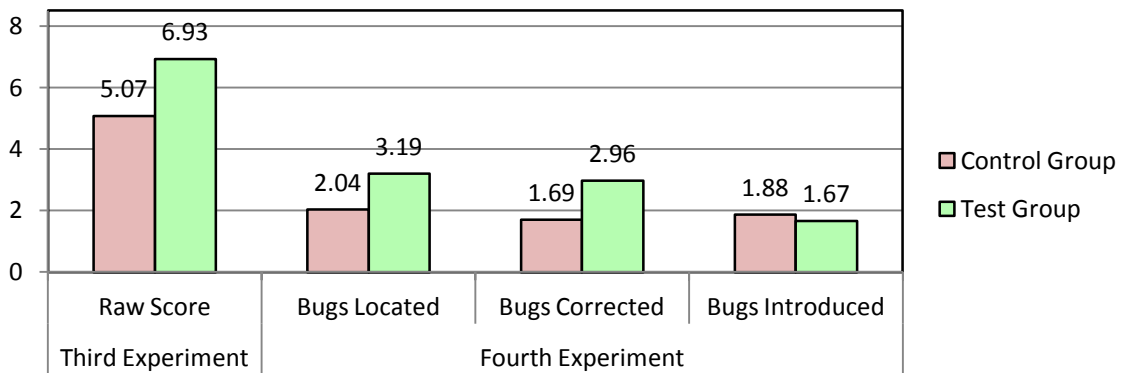


Figure 27. Average accuracy scores over all students for third and fourth experiments.

In the four experiments described above, students had prior instruction and presumed understanding of the concepts related to the data structures used and experience with their implementation. In a fifth experiment, subjects were asked to convert a min heap to a max heap and implement `addElement()`, `removeMax()`, and `findMax()` methods. Subjects had received classroom instruction on the concept of min-max heaps, but had no experience with their implementation. In a sixth experiment, subjects were asked to implement the add method for a linked priority queue, for which they had no prior conceptual knowledge or implementation experience. Results for these two experiments were also positive and differences in accuracy between groups were statistically significant. Average raw accuracy scores for these two experiments are shown in Figure 28.

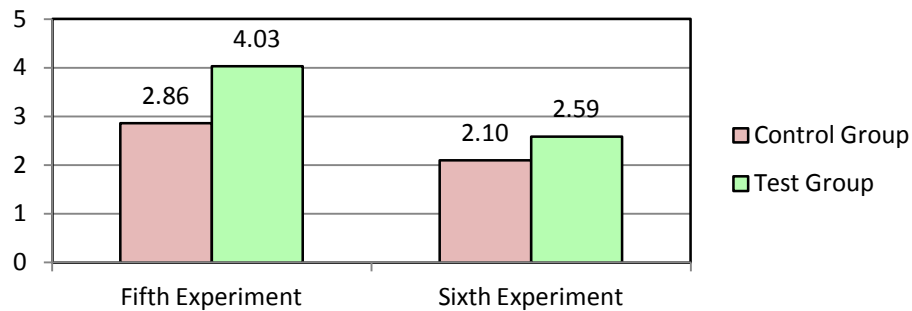


Figure 28. Average raw accuracy scores over all students for fifth and sixth experiments.

6.3 Usage

When jGRASP is first started after installation, users are given the choice of allowing or disallowing data collection. Approximately 60% of users choose to allow it. Versions that included data collection were first released in November of 2011. Because some older versions are still in use, data was collected for about 50% of all users for the first four months of 2014. Below we will discuss data for the period of January 1, 2012 to May 13, 2014. During this period, our data collection server was down for a significant amount of time on three days. On

days where it was down for a period of time that was long enough to cause a noticeable change in reported statistics, the total number of daily data collection sets recorded was adjusted based on weekly trends. Some of the feature usage numbers below were similarly adjusted if necessary, for example to prevent the three misleading outliers that would otherwise appear in a per-day scatter plot. Other numbers were not adjusted. Down time was less than 0.3% so the effect on reported total numbers and percentages below is not significant.

The data collected consists of counts of the number of times various features are used by one account in one day. Multiple users may use one account and single users may use multiple accounts, but this should be infrequent, so the counts should approximate per-user-per-day usage and they will be discussed below in these terms. For the analysis here, the counts of interest are number of Java compiles, number of Java debug sessions started, and number of viewers launched. Figure 29 shows per-day viewer use as a percentage of per-day debugger use from January 1, 2011 to May 13, 2014, as a seven day moving average. Of the users who debugged a Java program one or more times on a particular day, this is the percentage who opened at least one viewer. Yearly averages for this value are 25% in 2012 and 2013 and 28% for January 1 to May 13 of 2014. Thus, among users who do use the debugger, the viewers are fairly well utilized and have been for several years. Full implementation of the canvas is recent enough that any correlation with viewer utilization cannot yet be determined.

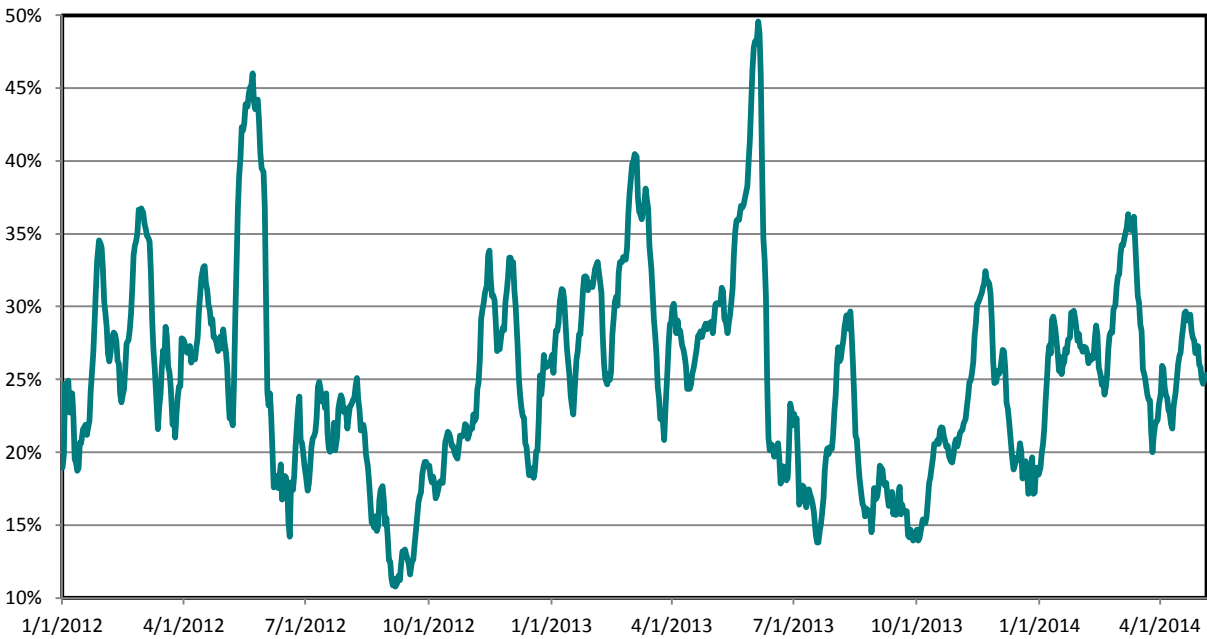


Figure 29. Viewer use as a percentage of debugger use, seven day moving average.

Because the number of jGRASP users is quite large (approximately 15,000 per weekday in the middle of academic semesters), the variability in viewer use as a percentage of debugger use per day seen in the graph is somewhat unexpected. Since most users are college or high school students, one possible cause is that some programming assignments benefit more from viewers than others, and many classes cover identical topics at approximately the same time. Several variables were examined for correlation with viewer use. The strongest correlation is with debugger use itself. That is, on days when the debugger is heavily used, viewers are used more often than average per debug session. Figure 30 shows the relationship between total number of viewers launched and total number of debugger sessions started, from January 1, 2012 to May 13, 2014. Each dot represents one day. The x value is the total number of times the debugger was started as a fraction of the total number of times a Java program was compiled. The y value is the total number of times a viewer was opened as a fraction of the total number of times the debugger was started. The correlation between these values is fairly strong, $r = .76$. One possible

contributing factor to this correlation may be that assignments that benefit from debugger use also benefit from viewer use. Another factor may be that in some classes students are encouraged by their instructors to use the viewers for some assignments, and thus viewer use is driving debugger use during times when those assignments are given.

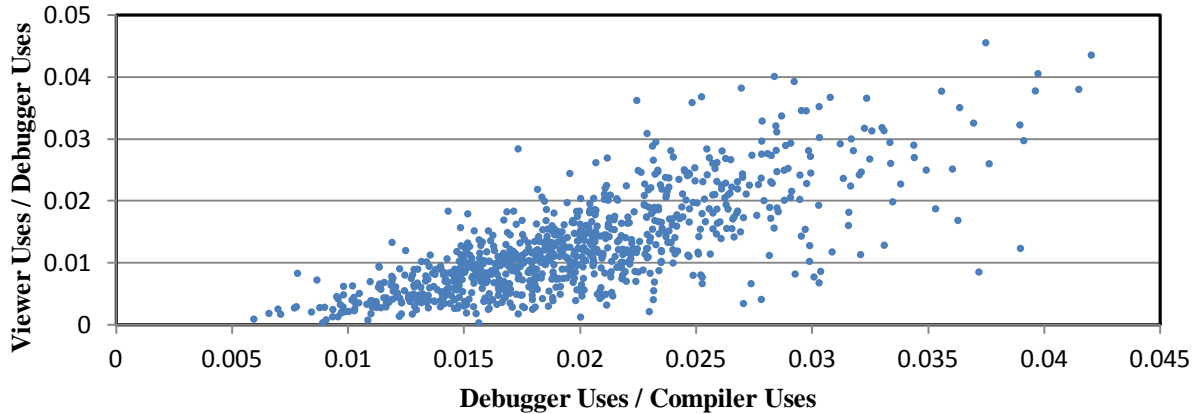


Figure 30. Correlation between viewer use and debugger use.

Total counts and at-least-once-per-user-per-day counts for Java compiles performed, debug sessions started, viewers launched, and structure identifier viewers launched for 2013 are shown in Table 4. Debugger use per-day was slightly less than 10% of Java compiler use per-day, so use of the debugger among users (who are mostly students) is fairly low, though this has improved somewhat in 2014. Viewer use per day was 25% of debugger use per day, as reported above, so when the debugger is used, utilization of the viewers is quite high. 46% of viewers opened were structure identifier viewers. Note that the structure identifier viewer is the default when a viewer is opened on a value for which the structure identifier mechanism does recognize a data structure, so this gives some sense of the number of viewers that are opened on the data structures supported by the structure identifier relative to those opened on values with simpler types or unsupported data structure types. In total, 340,717 viewers including 155,910 structure

identifier viewers were opened. Note that these counts only include users who agreed to data collection. In 2013 recorded data collection uses comprised 44% of total recorded uses, so the actual totals should be over twice as high. This indicates that the visualization system and structure identifier viewers are very well tested by users. The jGRASP IDE automatically reports (with user approval) uncaught exceptions due to internal bugs, so undetected coding bugs in the system should be rare. Minor behavioral bugs in the viewers may persist because users will typically not actively report bugs unless they have a significant impact on usability.

Table 4. jGRASP feature use in 2013.

	Java Compiles	Debug Sessions	Viewers Opened	Structure Identifier
At Least Once Per-User Per-Day	976,911	96,415	24,557	16,295
Total Count	26,277,766	512,820	340,717	155,910

Table 5 and Table 6 show feature use in 2012 and 2014 (January 1 to May 13) respectively. Figure 31 shows yearly trends in per-day debugger use relative to compiler use (at least one use per-user per-day), total number of viewers launched relative to total number of compile operations done, and total number of structure identifier viewers launched relative to total number of viewers launched. Each of these numbers has increased somewhat from year to year.

Table 5. jGRASP feature use in 2012.

	Java Compiles	Debug Sessions	Viewers Opened	Structure Identifier
At Least Once Per-User Per-Day	701,016	60,759	15,350	8,612
Total Count	19,941,355	340,538	212,714	91,416

Table 6. jGRASP feature use January 1 to May 13, 2014.

	Java Compiles	Debug Sessions	Viewers Opened	Structure Identifier
At Least Once Per-User Per-Day	473,097	55,617	15,393	12,089
Total Count	12,591,850	310,851	220,952	111,822

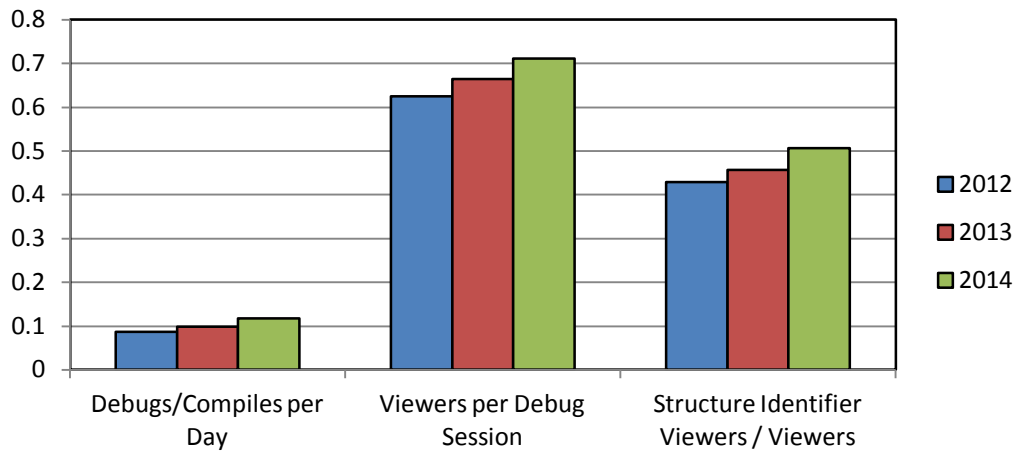


Figure 31. Debugger, viewer, and structure identifier use trends.

7 Summary and Conclusion

The jGRASP Visualization System is a unique tool that produces high-level data structure visualizations, and other visualizations of program values. Studies on the effects of algorithm animations and data structure visualizations on algorithm and code understanding have shown mixed results, though a majority of them appear to be positive and few are strongly negative. Algorithm animations have been more heavily studied than data structure visualizations. The limitations of testing and the overhead of using a visualization system may have had an adverse effect on some studies. Experiments using the jGRASP Visualization System that focused primarily on code understanding showed positive results.

Many tools have similar features to the jGRASP Visualization System. Those that produce similar high level data structure diagrams automatically do so only for built-in or previously known program types, or require considerable user effort to do so for arbitrary program types. Those that produce visualizations for arbitrary program types automatically or with little effort typically display them only as object nodes with reference links between them. The system described in this work is unique in that it automatically produces high-level textbook-like data structure diagrams from arbitrary source code for common data structures. The visualization system is also tightly integrated into the jGRASP IDE, allowing the visualizations to interact with many aspects of the debugger, including the workbench system and interpreter-like interactions system. The IDE is very much an integrated whole rather than a collection of separate tools that are launched from a common user interface.

Data structure and other type visualizations in the system are provided through viewers. Each viewer displays a single value. Values can be associated with a viewer directly (value-based) or through a source code expression that is evaluated to produce the value as needed (expression-based). By default, the expression for an expression-based viewer is only evaluated when the program is in the same scope as it was when the viewer was created. When it is in a different scope, only changes to the internal structure of the value are shown.

Viewers may be displayed separately in viewer dialogs, or multiple viewers may be combined on a canvas. In either case a subviewer allows sub-components of displayed values to be quickly examined. The canvas allows viewers to be arranged for the purpose of understanding data structures, algorithms, and their implementations in source code, or for repeatedly debugging a program or evaluating aspects of its performance. This arrangement can be saved to file for later use or for distribution. The canvas also provides video-player style controls (play, pause, etc.) so that an example program and associated canvas can be treated as a program animation to be used for understanding or illustrative purposes. Such an animation can be created simply by running a program in debug mode, opening a new canvas, dragging debugger values onto the canvas, arranging the resulting viewers and configuring them if necessary, and saving the arrangement as a canvas file.

An extensive set of viewers for Java is provided. General purpose "Basic" and "Detail" viewers display object fields and array elements in rows and in an expandable tree format, respectively, as well as a simple text representation for primitive values. The "Detail" viewer uses icon color to indicate the relationship between the declaring type of a field and the declared type of the object that contains it. A colored bar shows field accessibility and visibility. The effective declared type and effective debugger scope for the detail viewer can be changed in

order to explore the effects of such changes on accessibility, visibility, and type relationships. For Java library classes and user classes that implement the `java.util.Collection` or `java.util.Map` interfaces, interface-based viewers display text values for the elements or keys and values, respectively. For linked lists, binary trees, hash tables, and array-based data structure classes in the Java collections classes, "Presentation" viewers display their internal structure, though this is for illustrative purposes only and cannot be used to explore the internal workings of these structures. This restriction allows these viewers to be fast enough to display structures with arbitrarily large numbers of elements. Other type specific viewers for primitive types, strings, colors, images, etc. are provided.

The structure identifier viewer automatically detects linked lists, binary trees, chained hash tables, and array and list-based structures such as stacks and queues, in arbitrary data structure code. It then displays these values in a format similar to that of the presentation viewers, by showing internal structure in the way a typical textbook data structure diagram would. Unlike the presentation viewers, the structure identifier viewer fully traverses data structure links and other internal structure, so that it can be used to explore the workings of these data structures and associated algorithms. To further that purpose, the structure identifier viewer uses color to highlight incorrect links in linked structures and to show changes to the structure elements from one debugger state to the next. For linked structures, any structure nodes referenced by local variables or method arguments in the current debugger context are displayed along with the main structure in the viewer, or reference links are shown if the referenced nodes are in the main structure. Nodes are shown moving smoothly between and among the main structure, local variables, and method arguments as the debugger progresses from state to state.

Viewers are added to the system through a plugin API. The API provides many support classes to simplify the creation of viewers and encourage consistency of display (colors, edge thicknesses, node display format, etc.) among different viewers. The jgrdi portion of the API is a debugger API specifically targeted at viewer use. This frees viewer implementation code from having to deal with the extensive error handling needed when working with many debugger APIs, and the details of type structure that are irrelevant to typical use in a viewer. It also allows viewer implementations to be written in way that may be completely or largely independent of the target language. The underlying target-dependent debugger API and API values can be accessed, if necessary, for viewers that are purposefully dependent on the target language or system implementation.

The structure identifier viewer maintains a set of mapping expressions used for data structure traversal and to determine data structure display properties. The data structure display category (linked list, binary tree, chained hash table, array or list-based structure) and mapping expressions are automatically determined by matching the structure of the runtime type of a value to be displayed with a type reference pattern. The system contains recognition modules for linked lists, linked binary trees, chained hash tables, binary heaps (to be displayed as binary trees), and array or list-based structures. Multiple mapping candidates are produced, each with a confidence value that depends on the strength of the match to the type reference pattern used, and to the expected class and field (or accessor method) names of the matched elements. The mapping with the highest confidence will be used by default, but the user may select from others that are sufficiently high. When a value is initially opened in a viewer, if that maximum structure identifier mapping confidence is below a critical value, another general-purpose viewer will be

used instead. Users may also directly edit the mapping expressions that were automatically determined, or they may select a display category and specify all of the expressions manually.

Automated regression testing aids in the maintenance and development of the structure identifier mechanism. A test file contains mapping expressions for hundreds of different data structure implementations, as well as directions for running the debugger on classes that contain those data structures. During a testing operation, the structure identifier is applied to each data structure example and if the highest-confidence mapping that it finds is not one specified in the test file (more than one may be acceptable for each data structure example), an error message is reported. This system was experimentally extended to optimize the weights used internally by the structure identifier linked list recognition module, by running it repeatedly while adjusting the weights using simulated annealing. The objective function had the goal of a maximum average gap in confidence between acceptable and unacceptable mappings. The value of the objective function on the test cases used was greatly improved, and the weights determined by this experiment were used in the final system. No verification was done to determine whether or not this improved data structure identifier performance on previously unencountered linked list code, though intuitively this seems likely.

The jGRASP viewer system was evaluated in several ways. The structure identifier mechanism was tested against previously unencountered data structure examples from twenty textbooks (the author of this work was not the primary investigator for these tests). After correcting one obvious shortcoming of the system that was not specific to these examples, 82% of the data structures were identified correctly. Another 10% were displayed correctly when a viewer was opened on a field of the structure (the wrapper class was not correctly recognized), 5% had problems that were easily correctible through manual configuration, and 3% failed

completely. The system has been improved since then and now works correctly for all of these examples (all of which are used in the automated regression testing).

Code understanding experiments (for which the author of this work was not the primary investigator) were done to evaluate the usefulness of viewers in an earlier version of the system, which were similar in form to the current structure identifier viewers. For each experiment, subjects were separated into a test group that had access to the viewers and a control group that did not. For various data structure examples, they were asked to implement data structure methods or find and correct bugs in data structure methods. In four of these experiments all subjects had prior conceptual knowledge of and implementation experience with the data structures, for one they had conceptual knowledge only, and for one they had neither conceptual knowledge nor implementation experience. Results for all experiments were positive and statistically significant.

Viewer utilization by end users was also evaluated. Approximately 60% of users consent to data collection the first time they run jGRASP, and this represents about 50% of current users (because older versions that did not have data collection are still in use). Data from January 1, 2012 to May 13, 2014 was analyzed. Viewer use relative to debugger use was encouragingly high. On average, when a user started the debugger at least once in one day, 25% of the time they also opened at least one viewer on that day. Debugger use itself was quite low however, at about 10% of Java compiler use on a per-user per-day basis. Viewer use as a fraction of debugger use was highest on days when the debugger itself was heavily used. Hundreds of thousands of viewers, including hundreds of thousands of structure identifier viewers, were opened by users each year, indicating that the viewer system and structure identifier mechanism are well tested.

Debugger usage and viewer usage have been gradually increasing since data collection was initiated.

8 Future Goals

The most important goal for the future of the jGRASP Viewer System is to extend its use to languages other than Java, and thereby greatly expand its potential user base. Plans are to create a low-level Java wrapper (debugger API) for the text-based GDB/MI [48] (GNU debugger machine interface) -based debugger, create a debugger core using the wrapper API, and add support to the viewer system for C, C++, and Objective-C at a minimum. Although the viewer system has been created with language-independence in mind, it has only been used for Java, and changes will need to be made to support languages with features, such as multiple implementation inheritance, that Java does not have. Such language features were planned for during development of the system but never tested, and many mechanisms that would be needed only for languages other than Java were left unimplemented.

A second major goal is to embed the viewer system in plugins for the most popular IDEs in use, most importantly, Eclipse. Current commercial-strength IDEs such as Eclipse have enormous numbers of features and plugins. Professional users are unlikely to use a light-to-medium-weight IDE such as jGRASP, even if it does have unique and useful features that are not available elsewhere. Many instructors also select an IDE for their students, as a recommendation or requirement, with an eye to future professional use. Using jGRASP for visualization in parallel with another IDE is possible, but awkward for large projects. Providing the viewers through IDE plugins would greatly expand their potential user base. This goal could be largely achieved by creating plugins for the viewer system only. The debuggers built into these IDEs do

not have all the capabilities necessary for full-featured and tightly-integrated viewers though, so full jGRASP debugger plugins may be a better option. This would also allow other debugger-related features, such as the workbench and interactions, to be included with the plugins.

A minor but more immediate goal is to add the capability for display of dependencies between viewers in a canvas. This would be optional on a per-viewer basis. Viewer implementation classes could provide a list of exposed external references and a list of internal object targets (where one target would generally be the value associated with the viewer, if that value is an object) and target locations. The viewer system would automatically display reference arrows between matching references and targets. The most likely default option would be for references to be shown only for the viewer, if any, which is selected on the canvas. For values dragged from other viewers onto the same canvas, the reference arrows between them would be displayed at all times by default. Thus, the basic viewer could operate similarly to the visualizations in DDD, where the display of linked structures can be expanded node-by-node. The resulting structure displayed by a group of viewers would also operate in some sense as a single entity. For example, deleting the initial viewer would delete the rest of the group, and previously expanded reference node chains would be recorded and restored as the data structure loses and gains nodes.

9 References

- [1] J. T. Stasko, "Tango: a framework and system for algorithm animation," *IEEE Computer*, vol. 23, no. 9, pp. 27-39, September 1990.
- [2] J. Stasko, A. Badre and C. Lewis, "Do algorithm animations assist learning? An empirical study and analysis.," in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, Amsterdam, Netherlands, 1993.
- [3] J. Stasko, "Animating algorithms with XTANGO," *ACM SIGACT News*, vol. 23, no. 2, pp. 67-71, Spring 1992.
- [4] C. Keyhoe, J. Stasko and A. Taylor, "Rethinking the evaluation of algorithm animations as learning aids: an observational study," *International Journal of Human-Computer Studies*, vol. 54, no. 2, pp. 265-284, Feb 2001.
- [5] M. D. Byrne, R. Catrambone and J. T. Stasko, "Evaluating animations as student aids in learning computer algorithms," *Computers & Education*, vol. 33, no. 4, pp. 253-278, Dec 1999.
- [6] J. T. Stasko and E. Kraemer, "A methodology for building application-specific visualizations of parallel programs," *Journal of Parallel and Distributed Computing*, vol. 18, no. 2, pp. 258-264, June 1993.
- [7] J. Jain, J. H. Cross, T. D. Hendrix and L. A. Barowski, "Experimental Evaluation of Animated-Verifying Object Viewers for Java," in *Proceeding of SoftVis 2006*, Brighton, UK, 2006.
- [8] J. H. Cross, T. D. Hendrix, D. A. Umphress, L. A. Barowski, J. Jain and L. N. Montgomery, "Robust Generation of Dynamic Data Structure Visualizations with Multiple Interaction Approaches," *ACM Transactions on Computing Education*, vol. 9, no. 2, pp. 13:1-13:32, June 2009.

- [9] J. T. Stasko, "Using student-built algorithm animations as learning aids," in *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, San Jose, California, 1997.
- [10] T. Lauer, "Learner interaction with algorithm visualizations: viewing vs. changing vs. constructing," in *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, Bologna, Italy, 2006.
- [11] M. Krebs, T. Lauer, T. Ottmann and S. Trahasch, "Student-built algorithm visualizations for assessment: flexible generation, feedback and grading," in *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, Monte de Caparica, Portugal, 2005.
- [12] G. Röbling, M. Schüer and B. Freisleben, "The ANIMAL algorithm animation tool," in *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, Helsinki, Finland, 2000.
- [13] G. Röbling and B. Freisleben, "Experiences in using animations in introductory computer science lectures," in *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, Austin, Texas, 2000.
- [14] J. Urquiza-Fuentes and J. Á. Velázquez-Iturbide, "An evaluation of the effortless approach to build algorithm animations with WinHIPE," *Electronic Notes in Theoretical Computer Science*, vol. 178, pp. 2-13, July 2007.
- [15] J. H. Cross, T. D. Hendrix, L. A. Barowski and D. A. Umphress, "Dynamic Program Visualizations – An Experience Report," in *Proceedings of the 45th ACM technical symposium on Computer science education*, Atlanta, GA, 2014.
- [16] C. D. Hundhausen, S. A. Douglas and J. T. Stasko, "A meta-study of algorithm visualization effectiveness," *Journal of Visual Languages and Computing*, vol. 13, no. 3, pp. 259-290, 2002.
- [17] T. Hübscher-Younger and H. N. Narayanan, "Dancing hamsters and marble statues: characterizing student visualizations of algorithms," in *Proceedings of the 2003 ACM symposium on Software visualization*, San Diego, California, 2003.
- [18] S. Buchanan and J. J. J. Laviola, "CSTutor: A Sketch-Based Tool for Visualizing Data Structures," *ACM Transactions on Computing Education (TOCE)*, vol. 14, no. 1, 2014.

- [19] M. H. Brown and R. Sedgewick, "A system for algorithm animation," in *SIGGRAPH '84 Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, Minneapolis, MN, 1984.
- [20] M. H. Brown and R. Sedgewick, "Techniques for Algorithm Animation," *IEEE Software*, pp. 28 - 39, January 1985.
- [21] J. Stasko, "Animating Algorithms with XTANGO," *SIGACT News*, vol. 23, no. 2, pp. 67-71, May 1992.
- [22] G. Cattaneo, G. F. Italiano and U. Ferraro-Petrillo, "CATAI: Concurrent Algorithms and Data Types Animation over the Internet," *Journal of Visual Languages & Computing*, vol. 13, no. 4, pp. 391-419, 2002.
- [23] A. Zeller and D. Lütkehaus, "DDD—a free graphical front-end for UNIX debuggers," *ACM SIGPLAN NOTICES*, vol. 31, no. 1, pp. 22-27, January 1996.
- [24] A. Zeller, "Visual Debugging with DDD," *Dr. Dobbs*, 1 March 2001.
- [25] S. Isoda, T. Shimomura and Y. Ono, "VIPS: A Visual Debugger," *IEEE Software*, pp. 8-19, May 1987.
- [26] T. Shimomura and S. Isoda, "VIPS: a visual debugger for list structures," in *COMPSAC 90: Proceedings, the Fourteenth Annual International Computer Software & Applications Conference*, Chicago, IL, 1990.
- [27] D. B. Baskerville, "Graphic Presentation of Data Structures in the DBX Debugger: Technical Report CSD-86-260," 1985.
- [28] P. J. Guo, "Online python tutor: embeddable web-based program visualization for cs education," in *Proceeding of the 44th ACM technical symposium on Computer science education*, Denver, Colorado, 2013.
- [29] B. A. Myers, "INCENSE: A system for displaying data structures," in *SIGGRAPH '83 Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, Detroit, MI, 1983.
- [30] J. L. Korn and A. W. Appel, "Traversal-based visualization of data structures," in *Proceedings of the IEEE Symposium on Information Visualization*, Research Triangle, CA, 1998.

- [31] S. Makherjea and J. T. Stasko, "Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-Level Debugger," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 1, no. 3, pp. 215-244, September 1994.
- [32] P. V. Gestwicki and B. Jayaraman, "JIVE: java interactive visualization environment," in *OOPSLA '04 Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, Vancouver, British Columbia, 2004.
- [33] C. Jung and N. Clark, "DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage," in *MICRO 42 Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, 2009.
- [34] I. Haller, A. Slowinska and H. Bos, "MemPick: High-Level Data Structure Detection in C/C++ Binaries," in *20th Working Conference on Reverse Engineering (WCRE)*, Koblenz, Germany, 2013.
- [35] A. S. Yeh, D. R. Harris and H. B. Reubenstein, "Recovering abstract data types and object instances from a conventional procedural language," in *Proceedings of 2nd Working Conference on Reverse Engineering, 1995.*, Toronto, Ont., 1995.
- [36] R. Dekker and F. Ververs, "Abstract data structure recognition," in *Proceedings of the Ninth Knowledge-Based Software Engineering Conference*, Monterey, CA, 1994.
- [37] T. D. Hendrix, J. H. Cross and L. A. Barowski, "An extensible framework for providing dynamic data structure visualizations in a lightweight IDE," in *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, Norfolk, VA, 2004.
- [38] T. D. Hendrix, J. H. Cross, J. Jain and L. A. Barowski, "Providing Data Structure Animations in Lightweight IDE," *Electronic Notes in Theoretical Computer Science*, vol. 178, pp. 101-109, 2007.
- [39] J. H. Cross, T. D. Hendrix, J. Jain and L. A. Barowski, "Dynamic Object Viewers for Data Structures," in *Proceedings SIGCSE 2007*, Covington, KY, 2007.
- [40] J. H. Cross, T. D. Hendrix and L. A. Barowski, "Integrating Multiple Approaches for Interacting with Dynamic Data Structure Visualizations," *Electronic Notes in Theoretical Computer Science*, vol. 224, pp. 141-149, January 2009.
- [41] J. Gosling, B. Joy, G. L. Steele Jr. and G. Bracha, *The Java Language Specification*, 3rd Edition, Addison Wesley, 2005.

- [42] J. H. Cross, T. D. Hendrix and L. A. Barowski, "Exploring Accessibility and Visibility Relationships in Java," in *Proceedings of ITiCSE*, Madrid, Spain, 2008.
- [43] L. Montgomery, J. H. Cross, T. D. Hendrix and L. A. Barowski, "Testing the jGRASP Structure Identifier with Data Structure Examples from Textbooks," in *Proceedings of the 46th ACM Southeast Conference*, Auburn, AL, 2008.
- [44] J. Jain, "User Experience Design and Experimental Evaluation of Extensible and Dynamic Viewers for Data Structures," Ph.D. Dissertaion, Auburn University, 2007.
- [45] M. Eisenstadt, "My hairiest bug war stories," *Communications of the ACM*, vol. 40, no. 4, April 1997.
- [46] R. C. Metzger, *Debugging by Thinking: A Multidisciplinary Approach*, Elsevier, 2003.
- [47] J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. K. Ajith Kumar and C. Prasad, "An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies," in *Proceedings of the 8th Australasian Conference on Computing Education*, Hobart, Australia, 2006.
- [48] R. M. Stallman, R. Pesch and S. Shebs, *Debugging with GDB: The GNU Source-Level Debugger*, 9th ed., Free Software Foundation, 2002.

Appendix A Structure Identifier Mapping Expressions

For each structure type, the mapping expressions, their meanings, and the synthetic variables available in the expressions are specified. Expressions marked (multiple) can have an arbitrary number of expressions separated by hashes (#), and the number of expressions must be consistent among such expressions. For each structure, the meanings of the synthetic variables are described. Note that the hash table mappings support non-chained hash tables, but the structure identifier does not look for them.

Table 7. Array-and-list-based structure expressions.

Expression	Meaning of Evaluation Result	Synthetic Variables
Element Count	The number of elements.	_struct_
Element Used	For each element, true if its value is significant and false otherwise. If blank, all elements will be displayed as significant.	_struct_, _index_
Element	The value of an element.	_struct_, _index_
Element Text	The display text for an element. If blank the default will be used.	_struct_, _index_, _value_
Index	A list of expressions for which index pointers into the structure will be shown. If blank, there will be no index pointers.	

Table 8. Array-and-list-based structure synthetic variables.

Synthetic Variable	Meaning
struct	The structure itself.
index	The index of the relevant element in the array or list.
value	The value of the relevant element.

Table 9. Linked list structure expressions.

Expression	Meaning of Evaluation Result	Available Synthetic Variables
Head Node	The head of the list. If blank, the list value itself is the head node (there is no wrapper).	_list_
Next Node	For each node, the next node in the list, or null if there is no next node.	_list_, _node_
Previous Node	For each node, the previous node in the list, or null if there is no previous node. If blank, the list is singly-linked.	_list_, _node_
Value (multiple)	The value or values associated with each node.	_list_, _node_, _value_
Node Text (multiple)	The display text for the value or values associated with a node. If blank, the defaults will be used.	_list_, _node_, _value_
Node Class	The class name for nodes. If blank, nodes referenced by local variables will not be displayed.	

Table 10. Linked list synthetic variables.

Synthetic Variable	Meaning
list	The list structure itself.
node	The relevant node.
value	The value of the relevant node.

Table 11. Binary tree structure expressions.

Expression	Meaning of Evaluation Result	Available Synthetic Variables
Root Node	The root of the tree. If blank, the tree value itself is the root node (there is no wrapper).	_tree_
Left Node	For each node, the left child in the tree, or null if there is no left child.	_tree_, _node_
Right Node	For each node, the right child in the tree, or null if there is no right child.	_tree_, _node_
Value (multiple)	The value or values associated with each node.	_tree_, _node_
Node Text (multiple)	The display text for the value or values associated with a node. If blank, the defaults will be used.	_tree_, _node_, _value_
Dummy Node	The dummy sink node for the tree. If blank, the tree does not have a sink node.	_tree_
Node Color	The display color for the node, in rgb integer format. If blank, the default color will be used.	_tree_, _node_
Node Class	The class name for nodes. If blank, nodes referenced by local variables will not be displayed.	
Array Size	If the tree has an associated array or list (as a binary heap would), the array or list size. If blank, there is no associated array.	_tree_
Array Element	If the tree has an associated array, the value for an element. If the expression is preceded by a hash symbol (#), then the result is a node in the tree and the value expression will be applied to it to determine the display value.	_tree_, _index_
Array Field	If the tree has an associated array which is stored in a field of the tree structure, the name of that field. If blank, there is no associated array or it is not stored in a field of the structure.	_tree_

Table 12. Binary tree synthetic variables.

Synthetic Variable	Meaning
tree	The tree structure itself.
node	The relevant node.
value	The value of the relevant node.
index	The relevant index in the associated array.

Table 13. Chained hash table structure expressions.

Expression	Meaning of Evaluation Result	Available Synthetic Variables
Element Count	The number of list slots in the table.	_table_
Element	The element chain for each slot.	_table_, _index_
First Node	The first node in a chain for a slot. If blank, the element chain is the first node (there is no wrapper).	_table_, _element_
Next Node	The next node in a chain. If blank, the hash table is not chained.	_table_, _element_, _node
Value (multiple)	The value or values associated with each node.	_table_, _element_, _node
Node Text (multiple)	The display text for the value or values associated with a node. If blank, the defaults will be used.	_table_, _element_, _node_, _value_
Node Class	The class name for nodes. If blank, nodes referenced by local variables will not be displayed.	

Table 14. Chained hash table synthetic variables.

Synthetic Variable	Meaning
table	The hash table structure itself.
element	The relevant chain.
node	The relevant node.
value	The value of the relevant node.
index	The relevant slot index.

Appendix B Viewer Class Naming Scheme

The class name of a viewer plugin indicates the target type for the viewer, which in turn specifies the types to which the viewer will apply as described in section 3.1. In order to avoid conflict with unrelated portions of the class filename and to support characters that are not allowed in filenames on some systems, type names that appear in these class names use replacements for some characters and strings. The wildcard type is represented internally by an asterisk (*), and also has a string replacement in the class name. The replacement strings are shown in Table 15.

Table 15. Viewer class name string replacements.

String Name	String	Replacement String
dot	.	—
underscore	_	_U
dollar sign	\$	_S
array braces	[]	_A
wildcard	*	_X

Viewer class names consist of the target type name followed by a single underscore, arbitrary text, and finally the word "View". The requirement for the word "View" prevents any possibility of supporting classes (which should not end in "View") being misidentified as top-level viewer classes. Some example viewer class names and their corresponding target type names are shown in Table 16.

Table 16. Example viewer class names and target types.

Viewer Class Name	Target Type
byte_ByteView	byte
java__lang__String_FormattedView	java.lang.String
long_A_A_SimpleView	long[][]
_X_BasicView	*
_X_A_ToStringView	*[]

Appendix C Selected Viewer API Method Summaries

Table 17. Viewer interface method summaries.

Modifier and Type	Method and Description
void	build(ViewerInitData vid, org.w3c.dom.Element initDataIn) Builds the viewer non-GUI internals.
void	destroy() Called when the viewer is closed or frozen.
void	getInfo(ViewerInfo vi) Retrieves optional information about the viewer, such as a text description.
int	getPriority(ViewerPriorityData vpd) Gets the viewer priority.
java.lang.String	getViewName() Gets the display name of the viewer.
boolean	toXML(org.w3c.dom.Document doc, org.w3c.dom.Element e) Stores the state of the viewer in an XML dom element.
void	update(ViewerValueData valueData, ViewerUpdateData data, DebugContext context) Updates the view.

Table 18. Value interface method summaries.

Modifier and Type	Method and Description
void	disableCollection() Prevents garbage collection for languages that have garbage collection.
void	enableCollection() Allows garbage collection for languages that have garbage collection.
Type	getArrayComponentType(DebugContext context) Gets the type of the elements of this value if it represents an array.
Value	getArrayElement(DebugContext context, int index) Gets the value of an array element if this value represents an array.
java.util.List<Value>	getArrayElements(DebugContext context) Gets the values in an array if this value represents an array.
java.util.List<Value>	getArrayElements(DebugContext context, int offset, int length) Gets the values in an array if this value represents an array.
int	getArrayLength(DebugContext context) Gets the array length if this value represents an array.
java.lang.String	getDescription(java.lang.String title, Type declaredType, java.lang.String refPrefix) Gets a text description of the value.
Value	getFieldValue(DebugContext context, Field f) Gets the value of a field.
Value	getFieldValue(DebugContext context, java.lang.String fieldName) Gets the value of a field.
Method	getMethod(DebugContext context, java.lang.String methodName, java.lang.String returnType, java.lang.String[] argumentTypes) Gets a method that can be invoked on the object.
java.lang.Object	getNativeValue() Gets the native representation of this value, for use with viewers that need to do special-case handling of debugger-specific values.
Type	getType(DebugContext context) Gets the type of this Value.
long	getUniqueID() Gets a unique id if applicable.

java.lang.String	getValueString() Gets a source code representation of the value, if there is one.
Value	invokeMethod(DebugContext context, Method method, Value[] arguments) Invokes a method.
boolean	isArray() Determines if this value is an array.
boolean	isInstanceOf(DebugContext context, java.lang.String typeName) Determines if this value is an instance of some class or interface.
boolean	isNull() Determines if this value is the null object.
boolean	isObject() Determines if this value is an object, as opposed to a primitive.
boolean	isPrimitive() Determines if this value is a primitive.
boolean	isSame(Value v) Determines if this value is the same as another.
boolean	isSameNaN(Value v) Determines if this value is the same as another, where all numeric values with identical binary representations are considered equal.
void	setFieldValue(DebugContext context, Field f, Value value) Sets the value of a field.
void	setFieldValue(DebugContext context, java.lang.String fieldName, Value value) Sets the value of a field.
boolean	toBoolean(DebugContext context) If the value is a type that can have true and false values, returns the true or false value.
byte	toByte(DebugContext context) If the value is a type that can be represented as a byte, returns that value.
char	toChar(DebugContext context) If the value is a type that can be represented as a char, returns that value.

double	toDouble(DebugContext context) If the value is a type that can be represented as a double, returns that value.
float	toFloat(DebugContext context) If the value is a type that can be represented as a float, returns that value.
int	toInt(DebugContext context) If the value is a type that can be represented as an int, returns that value.
long	toLong(DebugContext context) If the value is a type that can be represented as a long, returns that value.
short	toShort(DebugContext context) If the value is a type that can be represented as a short, returns that value.
java.lang.String	toString(DebugContext context) Gets a string representation of the value, suitable for display.