Experimental Evaluation of Animated-Verifying Object Viewers for Java

Jhilmil Jain, James H. Cross II, T. Dean Hendrix, and Larry A. Barowski

Computer Science and Software Engineering
Auburn University, AL 36849
jainjhi | crossjh | hendrtd | barowla @auburn.edu

Abstract

Although many visualization techniques have been shown to be pedagogically effective, they are still not widely adopted. The reasons include: lack of suitable methods of automatic-generation of visualizations, lack of integration among visualizations, and lack of integration with basic integrated development environment (IDE) support. To effectively use visualizations when developing code, it is useful to automatically generate multiple synchronized views without leaving the IDE. The jGRASP IDE provides *object viewers* that automatically generate dynamic, state-based visualizations of objects and primitive variables in Java. Such seamless integration of a lightweight IDE with a set of pedagogically effective software visualizations is unique and is currently unavailable in any other environment. Multiple instructors have reported positive anecdotal evidence of their usefulness. We conducted formal, repeatable experiments to investigate the effect of these viewers for singly linked lists on student performance and we found a statistically significant improvement over traditional methods of visual debugging that use break-points.

Keywords: Program Visualization, Algorithm Animation, Data Structures.

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

1 Background

All Computer Science, Software Engineering, Computer Engineering, and Wireless Engineering (software option) majors at Auburn University are required to take the COMP 1210 course (an introduction to the Java programming language) followed by the COMP 2210 course (an introduction to data structures and algorithms). Data structures and algorithms are abstract concepts, and the understanding of this topic and the material covered in class can be divided into two levels: a) Conceptual – where students learn concepts of operations such as create, add, delete, sort etc; and b) Coding – where students implement the data structure and its operations using any programming language (Java in our case). Over the course of the past few years we have noticed a consistent decline in enrollment in the CS department. This trend is most noticeable during the COMP 2210 course when a majority of students decide to drop this required course. We conducted paper-based surveys and one-on-one interviews in Fall 2004 and Spring 2005 to understand the aspects of the COMP 2210 that students find most difficult. We found that students did not find fundamental concepts difficult to understand but had most trouble with the implementation. About 75% of students indicated that they had an appropriate level of expertise in Java to complete the requirements of COMP 2210. Evidently, poor Java skills is not causing the problems with implementation. Most students faced a *blank-screen syndrome* when they began implementation [Jain et al. 2005a]. The basic problem was that students have difficulty transitioning from static textbook concepts to dynamic programming implementation [Shaffer et al. 1996]. Thus, there is a need to bridge the gap from concepts to implementation.

SOFTVIS 2006, Brighton, United Kingdom, September 04–05, 2006.
© 2006 ACM 1-59593-464-2/06/0009 $5.00

We surveyed over 21 tools that are used for the purpose of data structure visualization [Jain et al 2005b] and found that most tools (more than 14 in our survey) focused on conceptual understanding. We found that only 7 implementation level tools were available to help students during program comprehension and debugging activities. But, none of these implementation tools fulfilled all of our goals, viz.,

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

Felder and Silverman [Felder and Silverman 1988], in their 1988 study report that between 75—80% of students are visual learners. Most students will retain more information when it is presented with visual elements (pictures, diagrams, flow charts, etc). In programming, visual learners can benefit from creating diagrams of problem solutions (e.g., flow charts) before coding. Similarly, visual representations of data structure states should help in data structure understanding. Thus, it would be beneficial to have a tool that enables students to visualize both the conceptual and the implementation aspects of data-structures.

2 jGRASP Object Viewers

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

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

Visualizations for data structures are created in two steps. First an external viewer class is implemented using the source code-based API provided with the jGRASP framework. In the second step the program that implements the data structure is executed using the debugger or workbench. A user can simply drag and drop the object reference anywhere on the screen to open the viewer (see Figure 2). As the user steps through the code the viewer will be automatically updated

The jGRASP viewers fall into two main categories: interface-based and structurally-based. For example, an interface-based viewer might show a HashTable as a set of keys and values, while a structurally-based viewer would show the array of hash slots, the linked list of key-value pairs at each slot, etc.

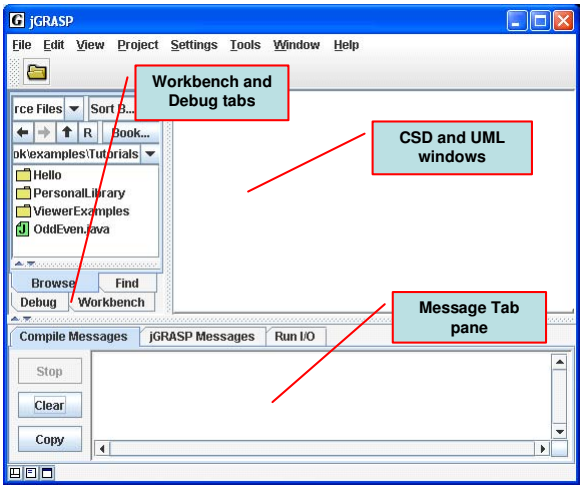

Figure 1: jGRASP Virtual Desktop

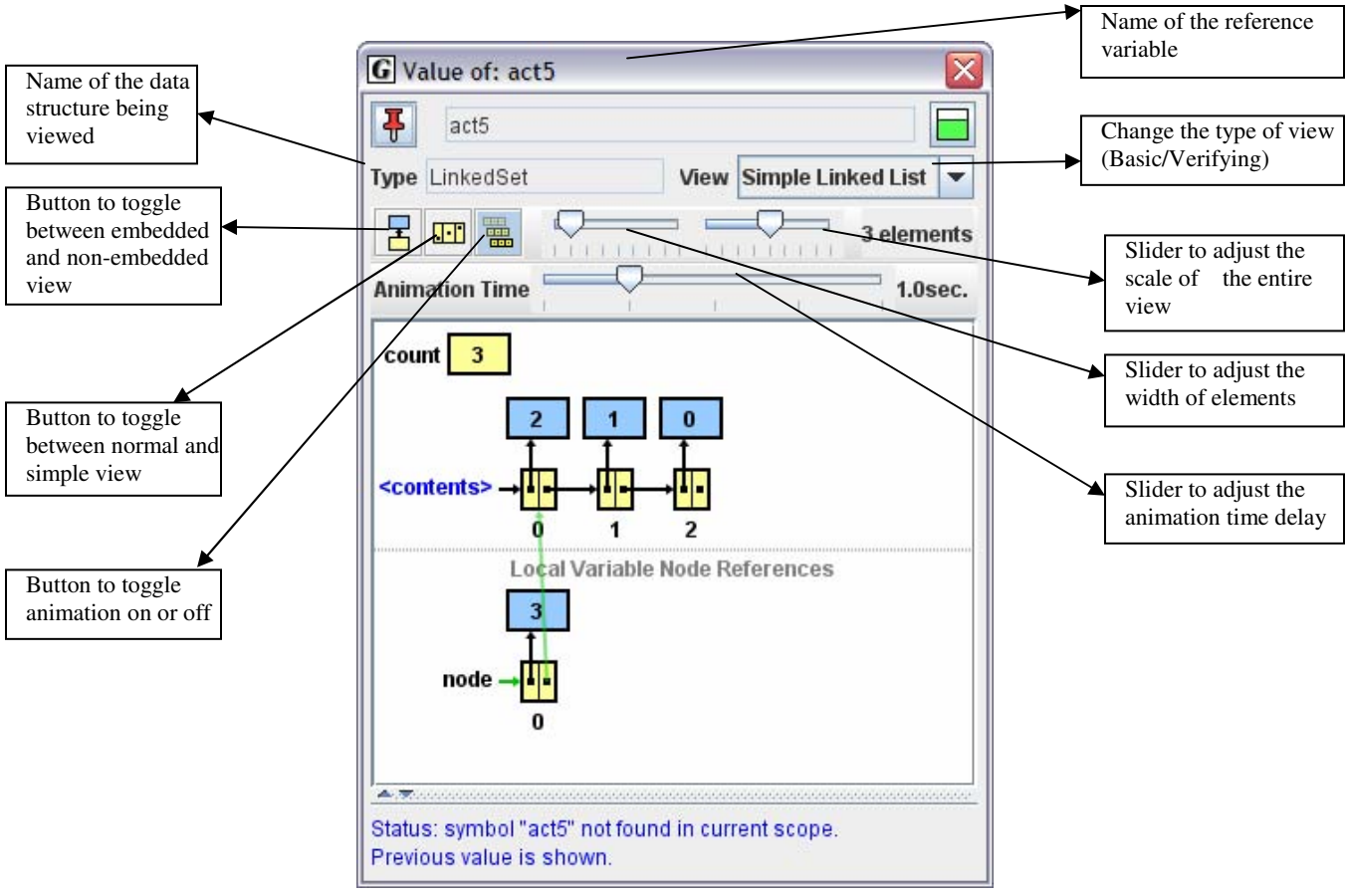

Figure 2: Details of the controls of the viewer window

The structurally-based viewers fall into two sub-categories: non-verifying and verifying (all interface-based viewers are non-verifying). The non-verifying viewers assume that the structure of the object being viewed is correct, and generally use method calls to elaborate the structure. When a structure gets beyond a certain size, the non-verifying viewers will examine only the part of the structure that is on-screen. Because of this, they may be used to examine large structures without slowing the debugging process excessively. The non-verifying viewers would generally be used to examine the contents of a structure in the context of an algorithm that uses it, rather than to examine the workings of the data structure itself. Hendrix [Hendrix et al. 2004] discusses non-verifying viewers in further details.

2.1 Advantages of Verifying Viewers

The purpose of the verifying viewers is to aid in the understanding of the data structures themselves, and to assist in finding errors while developing a data structure. To further this intended use, any local variables of the structure's node type are also displayed, along with the links between these local variable nodes or structure fragments and the main. This allows mechanisms of the data structure such as finding, adding, moving, and removing elements to be examined in detail by stepping through the code.

As an additional aid to understanding the mechanisms of the data structure, the verifying viewers animate structural changes. In order to do this, they store a representation of the entire data structure at each update (viewer updates happen at a breakpoint or after a step in the debugger). At each update, the value from the previous update (which may or may not be the same as the current value) is examined for changes. If any nodes in the structure have moved, the viewer enters into animation mode. In this mode, an "animation update" occurs at regular intervals. During animation, the previous structure value and previous local variable nodes and structure fragments (which may or may not be present any longer) are displayed. Node locations are interpolated so that they move smoothly from their old locations to the new ones, within and between the main structure and local variable nodes and structure fragments. At the end of animation, the new structure value and new local variable nodes and structure fragments are displayed.

2. 2 An Example of Animated-Verifying Viewer

To view the local variables created as a method is being executed, the user must step-into the method. This will enable the user to see an animation that depicts object creation, pointer manipulation, and the updates to variable values. Figure 2 shows the controls available on the viewer window.

jGRASP provides a library of viewers for common data structures that allow a viewer to be written using very little code. For example, a linked list viewer only needs to know how to find the first node in the list, and, given a node, how to find the next node; or, alternately, the number of nodes and access to any node by index.

Consider the following code fragments of two Java programs: a) LinkedSet.java which implements a singly linked list, and b) LinearNode.java which is the type of element contained by the class LinkedSet.

```
class LinkedSet
{
        // the current number of elements in the set
         private int count;

        //points to the last element in the list
         private LinearNode<T> contents;

}
class LinearNode
{
         //pointer to the next node
         private LinearNode<T> next;

         //generic type of element contained
         private T element;

}
```

Figure 3: Code fragments of LinkedSet.java and LinearNode.javaLinkedSet.java and

In order to create a viewer for LinkedSet.java, only the instance variables in the following methods should be updated in the template provided with the jGRASP distribution for singly linked list. We need to modifying only 5 lines of code to create a viewer for LinkedSet.java.

a) getDisplayFields() - indicates the fields of the data structure that we want to be displayed in the viewer. For our example we have passed the variable *count* which is displayed in Figure 2.

b) getFirstNodeField() - indicates the pointer (if any) to be displayed at the head of the list. For our example we have passed the variable *contents,* which is displayed in Figure 2.

c) getNodeType() - indicates to the viewer the type of the nodes contained in the linked list. For our example we have passed the variable *LinearNode*.

d) getNext() - indicates to the viewer a path to the next node in the linked list. For our example we have passed the variable *next*.

e) getNodeValue() - indicates to the viewer how to access the value of the element in the linked list. For our example we have passed the variable *element*.

Once a viewer is created for a class and the viewer path has been set, a viewer can be opened on any instance of the class during the execution of an arbitrary program.

In Figures 4-6, we will insert a node with value 6 in the index position 3 of the linked list. Figure 4 shows the insert method, and where we are in the debugging process. Figure 5 depicts the state of the object viewer for singly linked list before the line is executed, and Figure 6 shows the state after the line is executed. The local variables created in the insert method – before, after, node can be visualized in the viewer.

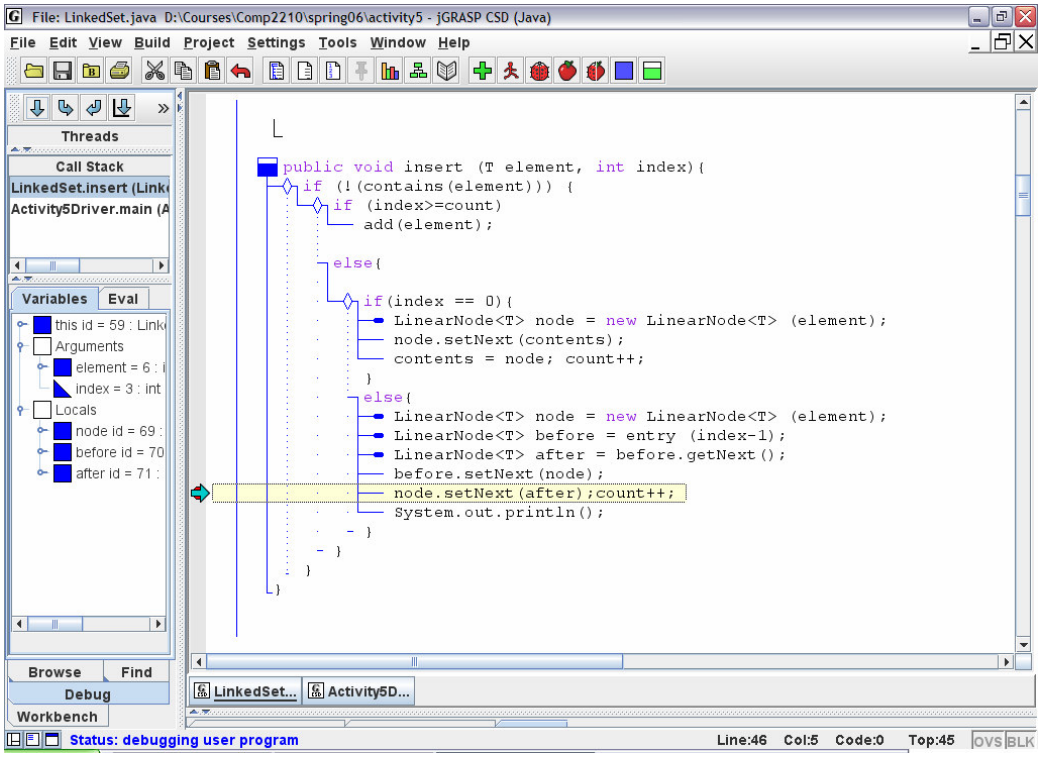

Figure 4: CSD window of jGRASP with the debugger stopped at a break point

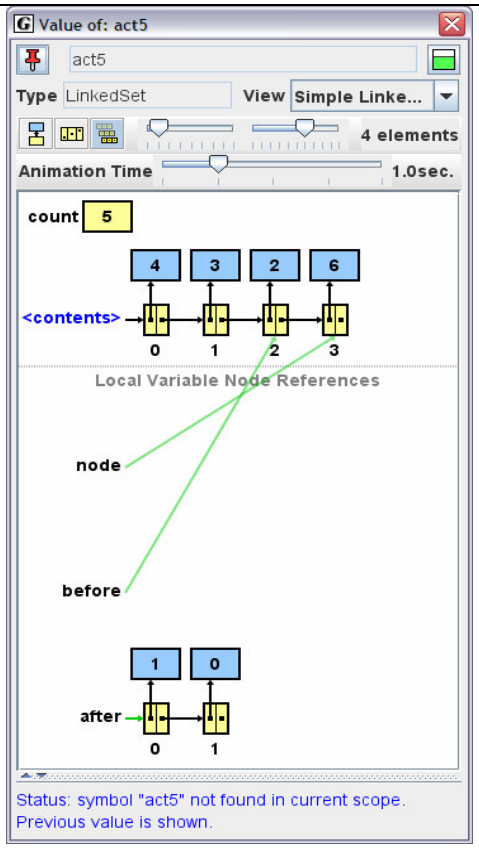

Figure 5: Before the setNext() method is called

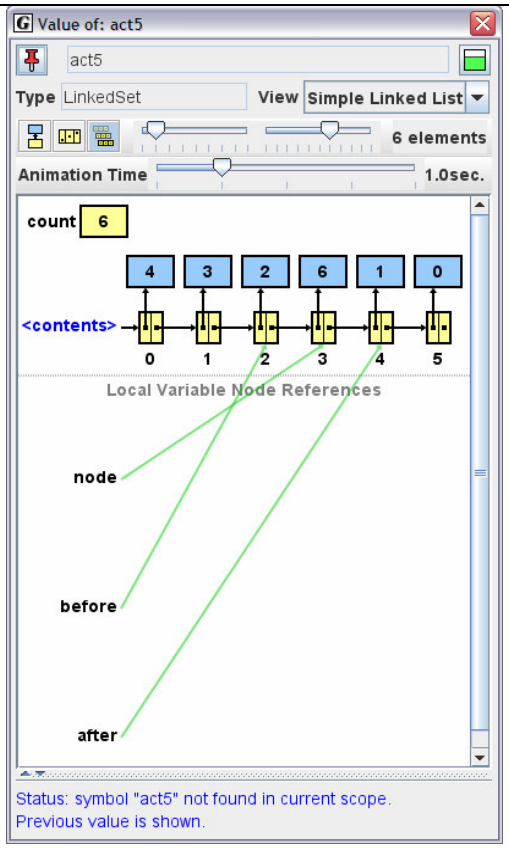

Figure 6: After the setNext() method is called

3 Experiments

3.1 Purpose

We conducted two controlled experiments to test the following hypotheses:

1. Students' are able to code more accurately (with fewer bugs) using the jGRASP data structure viewers.
2. Students are able to find and correct "non-syntactical" bugs more accurately using jGRASP viewers.

3.2 Subjects

Two criteria are important when choosing subjects for our controlled experiments. First, the subjects must be a close representation of the target population. The jGRASP viewers are being developed primarily for students enrolled in an introductory level data structure and algorithms course. Students enrolled in Fundamentals of Computing II (COMP 2210) at Auburn University were used as subjects since they closely resemble the target population. Second, the subjects must be relatively uniform in regard to their programming abilities in order to minimize the variance between groups. Additionally, we faced the following challenges while designing the repeatable experiments:

1. seamless integration with the course
2. organization of a large subject population
3. scheduling experiments such that there are no conflicts with course procedures
4. control over hardware and software used to ensure all subjects use a similar apparatus
5. plagiarism

We designed experiments that were closely integrated with course requirements and complemented the lab assignments. For example, we conducted the experiments on singly linked lists, and assigned project assignments on doubly linked lists. In Spring 2006 the students will be completing eight in-lab activities as a part of the COMP2210 course. These are attendance-based, ungraded, in-lab activities that comprise of 5% of the course grade. All in-lab activities will be conducted during the respective lab time of each section in a particular computer lab on campus. This ensured control over the hardware and software used by the subjects, and the schedule of experiments did not conflict with the subjects' course-work.

An equal number of subjects were assigned to two separate groups. The groups were balanced based on two specific programming skills – the ability to detect and correct logical errors and the ability to comprehend and trace programs. To test for the above abilities we conducted two tests as a part of the first in-lab activity. We identified common logical errors that are specific to the implementation of data structures [Eisenstadt 1997; Hristova et al. 2003; Metzger 2003; Rubey 1975; Youngs 1974]. In the first test we created problems designed to test for each of the common logical errors (a total of 25). In the second test, we chose eight questions from the twelve created by the multi-national study of reading and tracing skills in novice programmers [Lister et al. 2004]. We omitted questions on sorting since students had not been taught these concepts. Groups were balanced in week 5 of the course.

Internal validity implies the presence of evidence to indicate that the special conditions imposed in an experiment caused the observed outcome. Selection-bias is said to exist if distinct groups are not comparable before an experiment. Selection-bias is a major threat to internal validity for multiple-group experiment design. In our case a selection-bias would imply that factors other than the viewers that were used in the experiments caused different outcomes for the two groups.

We designed experiments based on the *between-group* approach to avoid the transfer of concepts learned in early experiments to a later experiment. Using the steps described above we balanced the programming expertise of all the groups thus having two comparable groups.

We address the above issued by forming two comparable, groups balanced on the basis of programming expertise. The following steps were taken to determine group assignments:

1. Students were sorted in a list in ascending order of their combined scores in Lab Activity 1.
2. The list was divided into pairs starting from the lowest score. Each student from a pair was randomly assigned to group 1 or 2.
3. Groups 1 and 2 were randomly assigned as the control group (no viewers) and the treatment group (state based viewers).

Students in Group 1 were familiarized with the jGRASP debugger and students in Group 2 were familiarized with both the debugger and jGRASP viewers. We conducted in-lab activity 2 to accomplish these goals.

3.3 Experiment 1

Our hypothesis was that students will be more productive (will code faster and with greater accuracy) using the jGRASP data structure viewers.

3.3.1 Materials and Procedure

Students were asked to implement four basic operations for singly linked lists. The program LinkedSet.java (from the class textbook [Lewis and Chase 2004]) was used in this experiment. Students were provided a detailed description of the programming assignment and the grading policy. Students were required to work independently and were timed (although there was no time limit to complete the assignment). The independent variable was the visualization medium (coding using jGRASP viewers vs. without viewers). The dependent variables were: time taken to complete the assignment, and the accuracy of the assignment.

The control group implemented all the four methods – *entry(), delete(), insert(),* and *contains()* using the jGRASP visual debugger. Details of these methods are provided in Appendix A. The driver program provided to this group contained a toString() method so that they could print out the contents of the list without writing additional code. The treatment group implemented the same methods using the jGRASP object viewers. The driver program given to this group did not contain

the toString() method, so the subjects had to use the viewers in order to see the contents of the list. The machines in the lab were set up with permissions such that only the treatment group had access to the viewers.

3.4 Experiment 2

Our hypothesis was that students are able to detect and correct logical bugs more accurately and in less time when using jGRASP viewers.

3.4.1 Materials and Procedure

A Java program implementing a singly linked list with 9 errors in four methods *add(), insert(), delete()* and *contains()* was provided. Details of these methods are provided in Appendix A. Students were asked to find and correct all the non-syntactical errors. The independent variable was the visualization medium (finding errors using jGRASP viewers vs. without viewers). The dependent variables were: number of bugs found, number of bugs accurately corrected, and number of new bugs introduced in the program while performing the experiment.

Both the groups were first required to identify and document errors. Next, similar to experiment 1, the control group corrected the detected errors using the jGRASP visual debugger and the treatment group corrected the errors using the jGRASP object viewers.

4 Results and Discussions

Collection of data was strictly contingent on student consent. Students were eligible for 5% of the course grade for the in-lab activities even if they decided to opt-out of data collection. Students that decided to opt-in for data collection were eligible for a 3% grade bonus. Our scoring of the students' work will constitute a grade that will be used to calculate up to 3 extra points on their final numeric average. For each group, we will create four quartiles. Quartile 1 (i.e. top 25% of the students) will get 3 bonus points, quartile 2 will get 2 bonus points, quartile 3 will get 1 bonus point. Using this scheme both groups will be awarded similarly regardless of the experimental treatment they receive.

We used Hotelling's T^2 statistic to analyze our data since we have two dependent matched groups and more than one response variable for each experiment. Hotelling's T^2 is a multivariate counterpart of Student's-t test which is typically performed for univariate data [Johnson and Wichern 1998].

4.1 Results of Experiment 1

The null hypothesis is that there is no difference in the accuracy and time taken for both groups. For 31 samples in each group, Hotelling's T^2 statistic was calculated to be 23.732087. The critical value for $\alpha = 0.05$, p=2 (two response variables), and n=31 (sample size) was 4.1708768. P-value was calculated to be 0.0000335. Since the T^2 value is much greater than the critical value, and p-value is much less than the alpha value, we can strongly reject the null hypothesis. Thus, there is a statistical significant difference between the two groups. Figure 7 shows that the mean time taken by the group with viewers is 109 minutes while the mean time taken by the group without viewers is 112 minutes.

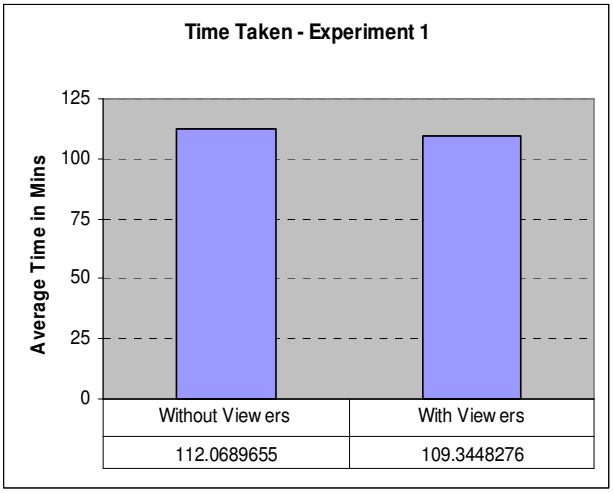

Figure 7: Comparison of mean time

Figure 8 shows that the mean accuracy of the treatment group with viewers is 6.34 points, while the mean accuracy of the control group without viewers is 4.48 points.

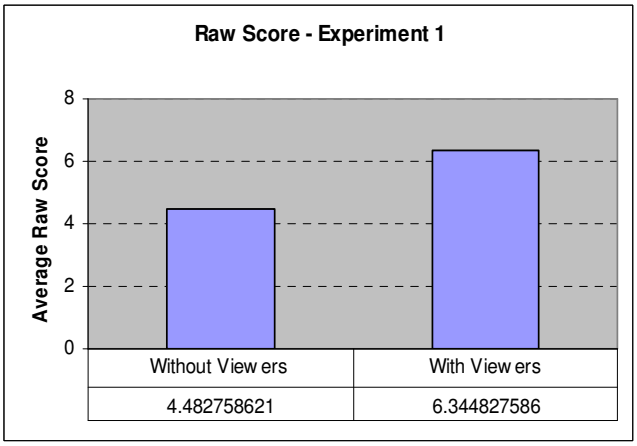

Figure 8: Comparison of mean accuracy

Table 1 and 2 shows the breakdown of the number of students in each group that correctly implemented each of the given method. We see that students in the treatment group consistently performed better than the control group for all cases.

Thus, we can say that in 95% of all cases, jGRASP object viewers will help increase the accuracy and reduce time taken for programs implementing data structures.

Table 1 – Details for each method for Experiment 1 (Group 1)

Group 1 (Without Viewers) – Control Group				
	1. Entry	**2. Insert**	**3. Delete**	**4. Contains**
No. Correct	12	4	4	15
% Correct	38.71%	12.9%	12.9%	51.61%

Table 2 – Details for each method for Experiment 1 (Group 2)

Group 2 (With Viewers) – Treatment Group				
	1. Entry	**2. Insert**	**3. Delete**	**4. Contains**
No. Correct	15	8	7	18
% Correct	48.39%	25.81%	22.58%	58.06%

4.2 Results of Experiment 2

The null hypothesis is that there is no difference in the number of bugs detected, corrected, introduced, and the time taken for both groups. For 26 samples in each group, Hotelling's T^2 statistic was calculated to be 12.833955. The critical value for $\alpha = 0.05$, $p=4$ (four response variables), and $n=26$ (sample size) was 7.0892211. P-value was calculated to be 0.0069295. Since the T^2 value is much greater than the critical value, and p-value is much less than the alpha value, we can strongly reject the null hypothesis. Thus, there is a statistical difference between the two groups.

Figure 9 shows that the mean time taken by the group with viewers is 88.23 minutes while the mean time taken by the group without viewers is 87.6 minutes. Figure 10 shows that the group with viewers is able to detect and correct more errors. In addition this group introduced fewer errors.

Table 3 and 4 shows the breakdown of the number of students in each group that correctly implemented each of the given method. We see that students in the treatment group consistently performed better than the control group for all cases.

Thus, we can say that in 95% of all cases, jGRASP object viewers will help increase the accuracy, but the time taken to write programs implementing data structures is a bit more. We will need to perform further analysis to explore this issue.

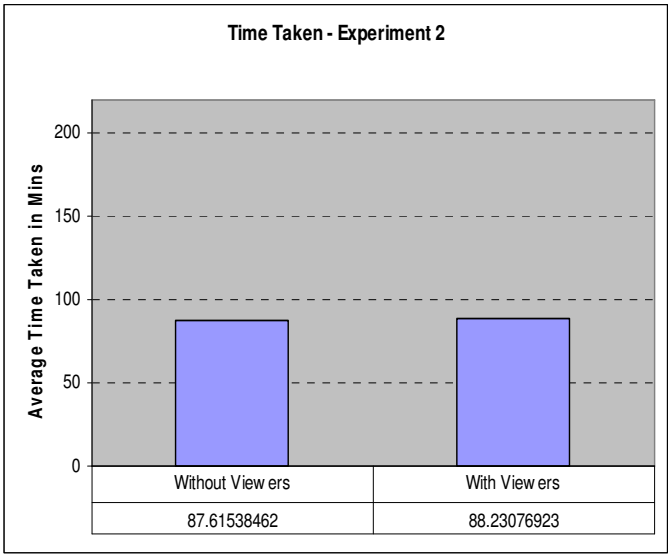

Figure 9: Comparison of mean time

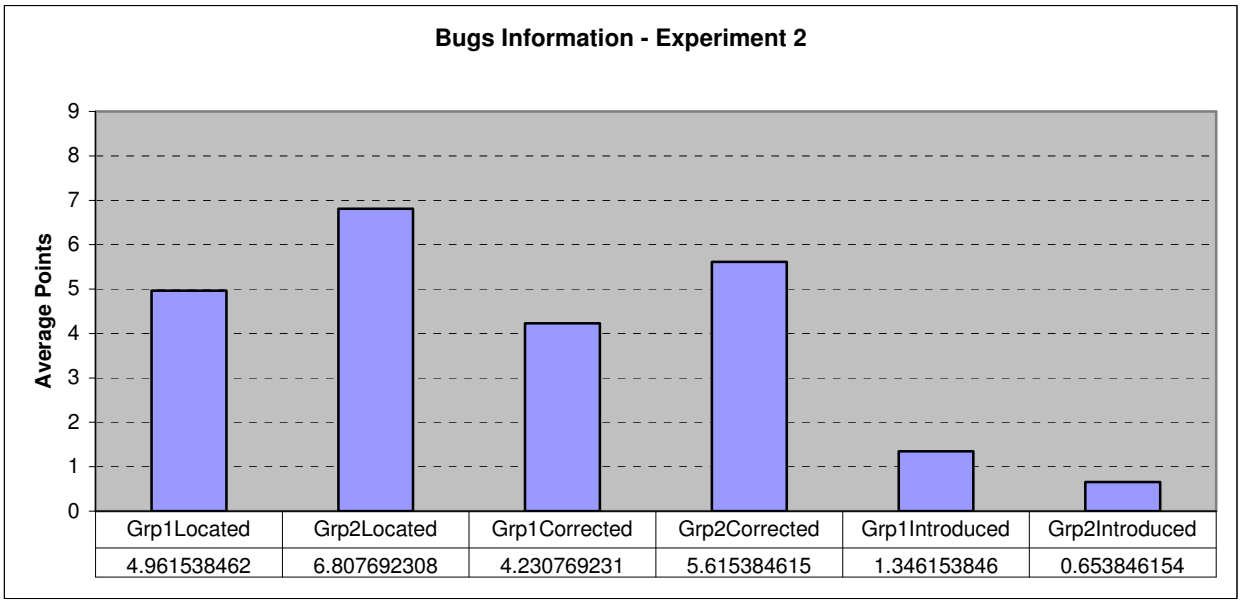

Figure 10: Comparison of mean bugs located, corrected and introduced

Table 3: Details for each method for Experiment 2 (Group 1)

Group 1 (Without Viewers) – Control Group				
	Add	**Insert**	**Delete**	**Contains**
Located	16	14	11	14
	61.54%	53.85%	42.31%	53.85%
Corrected	9	9	10	15
	34.62%	34.62%	38.46%	57.69%
Introduced	4	2	3	4
	15.38%	7.69%	11.54%	15.38%

Table 4: Details for each method for Experiment 2 (Group 2)

Group 2 (With Viewers) – Treatment Group				
	Add	**Insert**	**Delete**	**Contains**
Located	22	18	15	18
	84.62%	69.23%	57.69%	69.23%
Corrected	16	14	14	18
	61.54%	53.85%	53.85%	69.23%
Introduced	3	1	0	2
	11.54%	3.85%	0.00%	7.69%

Conclusions and Future Work

The decline in CS students over the years at Auburn University, especially in COMP 2210 prompted us to do conduct extensive surveys and interviews. We discovered that the main problem that the students were facing in this course was transitioning from abstract, static text book concepts to actual, dynamic programming implementation. So solve this problem we design and implemented a set of jGRASP object viewer framework.

In this paper we discussed the advantages of the animated verifying viewers. We have designed and conducted formal, repeatable experiments to investigate the effect of these viewers for singly linked lists on student performance. We found a statistically significant improvement over traditional methods of visual debugging that use break-points. Students were more productive and were able to detect and correct logical bugs more accurately using the jGRASP viewers.

We plan to repeat these experiments for linked binary search trees. Currently the scoring of student solutions for Experiment 2 was done only by one grader. In the next round of experiments we will use multiple grades and then run an inter-reliability analysis [Shrout and Fleiss 1979] to ensure that our grading scheme is reliably reproduced. We need to explore why the group with viewers takes a slightly longer time to finish the tasks in experiment 2. We also plan to conduct these experiments in multiple schools with different instructors, different environment variables, and students to determine if we get repeatable results.

We are currently working on a set of generalized viewers that will work for most linked structures. The goal is to eliminate the need for a user to create custom viewers. However, the viewer API will be published to encourage users to create their own viewers if the desire to do so.

References

EISENSTADT, M. 1997. My hairiest bug war stories. *Communications of the ACM*, vol. 40, issue 4, pp. 30-37.

FELDER, R.M., AND SILVERMAN, L.K. 1988. Learning and Teaching Styles in Engineering Education. *Engr. Education*, 1988, vol. 78, pp. 674-681.

HENDRIX, D.T., CROSS, J.H., AND BAROWSKI, L.A. 2004. 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, pp.387-391.

HRISTOVA, M., MISRA, A., RUTTER, M., AND MERCURI, R. 2003. Identifying and correcting Java programming errors for introductory computer science students. . In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, Reno, Nevada, USA, pp. 153-156.

JAIN, J., BILLOR, N., HENDRIX, D., AND CROSS, J. H. 2005. Survey to Investigate Data Structure Understanding. Submitted to the *International Conference on Statistics, Combinatorics, Mathematics and Applications*, Auburn, AL, December 2-4, 2005.

JAIN, J., CROSS, J. H., AND HENDRIX, D. 2005. Qualitative Assessment of Systems Facilitating Visualization of Data Structures. In *Proceedings of 2005 ACM Southeast Conference,* Kennesaw, GA, March 18-20, 2005.

JOHNSON, R. A., AND WICHERN, D. W. 1998. Applied multivariate statistical analysis, 4th Edition, ISBN: 0130925535.
LEWIS, J. AND CHASE J. 2004. Java Software Structures : Designing and Using Data Structures, 2nd Edition, ISBN: 0321245849.

LISTER, R., ADAMS, E. S., FITZGERALD, S., FONE, W., HAMER, J., LINDHOLM, M., MCCARTNEY, R., MOSTRÖM, J. E., SANDERS, K., SEPPÄLÄ, O., SIMON, B., AND THOMAS, L. 2004. A multi-national study of reading and tracing skills in novice programmers. In *Working Group Reports From ITiCSE on innovation and Technology in Computer Science Education*, pp. 119-15.

METZGER, R. C. 2003. Debugging by Thinking: A Multidisciplinary Approach (HP Technologies), ISBN: 1555583075.

RUBEY, R. J. 1975. Quantitative aspects of software validation. In *Proceedings of the International Conference on Reliable Software,* Los Angeles, California, pp. 246-251.

SHAFFER, C., HEATH, L.S., AND YANG, J. 1996. Using the swan data structure visualization system for computer science education. In *Proceedings of the SIGCSE*, ACM Press, 1996, pp. 140-144.

SHROUT, PATRICK E., AND FLEISS, JOSEPH L. Intraclass correlations: Uses in assessing rater reliability. *Psychological Bulletin*. vol. 86, issue 2, Mar 1979, pp. 420-428.

YOUNGS, E. 1974. Human Errors in Programming. *International Journal of Man-Machine Studies*, vol. 6, pp. 361- 376.

Appendix A

Methods used for Experiment 1 and Experiment 2

1) void **add** (element) – this method adds a new node to the end of the linked list. (Note: The list can have duplicates).

For example, if the list contains the following elements in the given order: “a”, “b”, “b”, “c”, “d”. After the method add(“e”) is called, node “e” should be added to the END of the list. So after the add(“e”) method is executed, the contents of the list are: “a”, “b”, “b”, “c”, “d”, “e”

2) void **insert** (element, position) – should insert a given element at the given position (it is added before the element which is currently in that position). (Note: The list can have duplicates).

Example 1) For example, if the list contains the following elements in the given order: “a”, “b”, “c”, “d”, “e”. After the method insert (“f”, 0) is called, node “f” should be inserted before “a” (which is at index 0). So after the insert(“f”, 0) method is executed, the contents of the list are: “f”, “a”, “b”, “c”, “d”, “e”

Example 2) For example, if the list contains the following elements in the given order: “a”, “b”, “c”, “d”, “e”. After the method insert (“f”, 5) is called, node “f” should be inserted after “e” (which is at index 4). So after the insert(“f”, 5) method is executed, the contents of the list are: “a”, “b”, “c”, “d”, “e”, “f”

Example 3) For example, if the list contains the following elements in the given order: “a”, “b”, “c”, “d”, “e”. After the method insert (“f”, 1) is called, node “f” should be inserted between “a” (which is at index 0) and “b” (which is at index 1). So after the insert(“f”, 1) method is executed, the contents of the list are:
“a”, “f”, “b”, “c”, “d”, “e”

3) boolean **contains** (element) – this method returns true is the list contains this element and false otherwise.

For example, if the list contains the following elements in the given order: “a”, “b”, “c”, “d”. The method call contain(“e”) will return false. The method call contain(“b”) will return true.

4) void **delete** (index) – this method deletes the node at a given index.

Example 1) For example, if the list contains the following elements in the given order: “a”, “b”, “c”, “d”, “e”. After the method delete (0) is called, the node “a” which is at index 0 should be deleted. So after the delete(0) method is executed, the contents of the list are: “b”, “c”, “d”, “e”

Example 2) For example, if the list contains the following elements in the given order: “a”, “b”, “c”, “d”, “e”. After the method delete (4) is called, the node “e” which is it index 4 should be deleted. So after the delete(4) method is executed, the contents of the list are: “a”, “b”, “c”, “d”

Example 3) For example, if the list contains the following elements in the given order: “a”, “b”, “c”, “d”, “e”. After the method delete (1) is called, the node “b” which is at index 1 should be deleted. So after the delete(1) method is executed, the contents of the list are:
“a”, “c”, “d”, “e”

5) LinearNode<T> **entry** (index) – this method returns the object reference of the node at given index position. This method will be used by insert and delete methods

Example 1) For example, if the list contains the following elements in the given order: “a”, “b”, “c”, “d”, “e”. After the method entry (0) is called, the object reference for node “a”, which is at index 0 should be returned.

Example 2) For example, if the list contains the following elements in the given order: “a”, “b”, “c”, “d”, “e”. After the method entry (4) is called, the object reference for node “e”, which is at index 4 should be returned.

Example 3) For example, if the list contains the following elements in the given order: “a”, “b”, “c”, “d”, “e”. After the method entry (2) is called, the object reference for node “c”, which is at index 2 should be returned.