Dynamic Object Viewers for Data Structures

James H. Cross II, T. Dean Hendrix, Jhilmil Jain, and Larry A. Barowski
Computer Science and Software Engineering
Auburn University, AL 36849
crossjh | hendrtd | jainjhi | barowla@auburn.edu

ABSTRACT
The jGRASP lightweight IDE has been extended to provide *object viewers* that automatically generate dynamic, state-based visualizations of data structures in Java. These viewers provide multiple synchronized visualizations of data structures as the user steps through the source code in either debug or workbench mode. This tight integration in a lightweight IDE provides a unique and promising environment for learning data structures. Initial classroom use has demonstrated the object viewers' potential as an aid to students who are learning to write and modify classes representing data structures. Recently completed controlled experiments with CS2 students indicate that these viewers can have a significant positive impact on student performance.

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

General Terms
Documentation, Experimentation, Human Factors.

Keywords
Program Visualization, Algorithm Animation, Data Structures.

1. INTRODUCTION
The jGRASP lightweight IDE (http://jgrasp.org) has been extended to include new dynamic viewers specifically intended to generate traditional abstract views of data structures such as linked lists and binary trees. These viewers are the most recent addition to the software visualizations provided by jGRASP. The purpose of these new viewers is to provide fine grained support for understanding objects representing data structures. When a class has more than one type of view associated with it, the user can open multiple viewers in order to compare different aspects of the structure. These viewers are tightly integrated with the jGRASP workbench and debugger and can be opened for any object in the Workbench or Debug tabs of the Virtual Desktop.

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). 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 should have a positive effect on the usefulness of software visualizations in a classroom environment. Multiple instructors have reported positive anecdotal evidence of their usefulness. We conducted formal, repeatable experiments to investigate the effect of these viewers on student performance when working with binary trees. The results indicated a statistically significant improvement over traditional methods of visual debugging.

SIGCSE'07, March 7–11, 2007, Covington, Kentucky, USA.

2. RELATED WORK
The approach we have taken for the state-based viewers in jGRASP is to automatically generate the visualization from the user's executing program and then to dynamically update it as the user steps through the source code in either debug or workbench mode. This is somewhat similar to the method used in Jeliot [1]. However, jGRASP differs significantly from Jeliot in its target audience. Whereas Jeliot focuses on beginning concepts such as expression evaluation and assignment of variables, jGRASP includes visualizations for more complex structures such as linked lists and trees. In this respect, jGRASP is similar to DDD [2]. The data structure visualization in DDD shows each object with its fields and shows field pointers and reference edges in a general way that is not tailored to the type of data structure being viewed. In jGRASP, each category of data structure (e.g., linked list vs. binary tree) has its own set of views and subviews which are intended to be similar to those found in textbooks. Although we are planning to add a general linked structure view, we began with the more intuitive "textbook" views to provide the best opportunity for improving the comprehensibility of data structures.

We have specifically avoided basing the visualizations in jGRASP on a scripting language, which is a common approach for algorithm visualization systems such as JHAVÉ [3]. We also decided against modifying the user's source code as is required by systems such as LJV [4]. Our philosophy is that for visualizations to have the most impact on program understanding, they must be generated as needed from the user's actual program during routine development.

3. DATA STRUCTURE IDENTIFIER
We began this work by creating specific viewers for several of the common data structures in the Java Collections Framework

including ArrayList, LinkedList, TreeMap, and HashMap. After the development of these individual viewers reached *steady state*, we developed a Viewer API based on their common features that enables users to quickly construct a viewer for a specific class (e.g., their own linked list class). Source code for example viewers that use the API is included with the jGRASP distribution to expedite the creation of new viewers by students and/or faculty. Although a new viewer can be created by changing about 10 lines of source code in one of the examples, this approach proved somewhat impractical for the general CS2 population. While this option needs to be available for faculty, we quickly found out that it is unrealistic to expect students who are in the process of learning about data structures to be able to modify a separate viewer class in order to see an instance of their own data structure. Thus, we turned our attention to building a mechanism that could determine if an instance was a linked list or binary tree based on a set of heuristics, and then automatically generate an appropriate view.

The *Data Structure Identifier*, which is automatically invoked when a viewer is opened, works as follows. For common node-and-link implementations of structures, where nodes are objects and links are object references, automatic identification is done by examining class structure, and by examining links in the instance that is about to be viewed. A class and its fields are first examined for same-class-references and possible structure mappings are considered. For example, a singly linked list is typically implemented as a class with a field (the head node link) whose class type has one same-class-reference (to the next node). This method may lead to multiple possible structures and to multiple possible mappings from a class to a particular structure. Name-based heuristics are used to assign a confidence level to each candidate. For example, a class named *MyTree* containing a field called *root*, of a type with same-class-reference fields called *left* and *right* is highly likely to be a binary tree, and it is highly likely that the *left* and *right* fields are left and right binary tree links respectively. The same class structure could also map to a doubly-linked list, but the class and field names make it very unlikely that this was the intention. The downside of this technique is that it will only work if the language used for class and field names is known. Currently, only English-language heuristics are applied. Also, the use of unusual or meaningless class and field names will make correct identification less likely. In cases where automatic identification fails, the viewer can be configured manually.

In addition to the name-based heuristics, link-based heuristics affect the confidence level for non-empty structure instances. Links in a potential binary tree or linked list will be examined to see if they do form a binary tree or linked list structure, and the confidence level will be modified appropriately. Since the viewer may have been initiated when the structure was in the process of being modified, a small number of errors in the structure will have little effect on the confidence level. An effect of employing this method is that for some structures, a more accurate identification may be achieved for non-empty instances than for empty ones.

The structure mapping with the highest confidence level found during automatic identification, if it is significantly higher than the confidence level for other potential mappings, will be automatically used when a viewer is first opened for a particular class. In most cases, one and only one mapping with a high confidence level will be found, and thus the mechanism will be transparent to the user. That is, an appropriate structural view will be displayed without user interaction. In cases where there are multiple mappings with similar confidence levels or where no mapping is found, the user is given the option of manually configuring the viewer (this can also be done while the viewer is in use). A configuration dialog allows the Java expressions that will be used to traverse the structure to be entered or edited. For example, for a singly linked list, expressions for the head node, next node (given a node), and display value (given a node), are required. Any mappings that were found during the automatic analysis are made available on a drop-down list. Once the structure mapping has been selected, specified, or modified using this dialog, the new mapping will automatically be applied the next time the user opens a viewer on an instance of the same class.

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

4. AN EXAMPLE

BinaryTreeExample.java, which is provided with jGRASP, is intended to be representative of a "textbook" example or of what a student may write. Its *main* method creates an instance of BinaryTree and then adds instances of BinaryTreeNode to it. The UML class diagram in Figure 1 shows that BinaryTreeExample depends on BinaryTree which depends on BinaryTreeNode.

In order to open a viewer, a breakpoint is set in *main*, and then the program is run in debug mode. After an instance of BinaryTree is created, a viewer is opened by dragging the instance out of the debug window. During the process of opening the viewer, the *Data Structure Identifier* determines, in this case, that the object is a binary tree structure and opens the appropriate viewer. As the user steps though the program and into the add() method, the nodes appear in the viewer. Figure 2 shows the program while stepping in the add() method. Figure 3 shows the instance of BinaryTree after three nodes have been added and a fourth node is about to be added. Local variable *branch* indicates the position in the tree where the fourth node will be added. When the fourth node is added, the animation provided by the viewer shows the node "sliding" up into the tree. Figure 4 depicts the viewer after the node has been added but prior to *size* being incremented. Notice that *size* is incremented just below the location of the debug *step* in Figure 2. Students have indicated that seeing the links being set correctly (or incorrectly) as they step through their code is extremely helpful with respect to their understanding of exactly how the implementation relates to the abstraction of the data structure itself. That is, seeing a node added on the blackboard as links are redirected is easy, but when it comes to understanding how this happen in the actual code, it is suddenly not so easy.

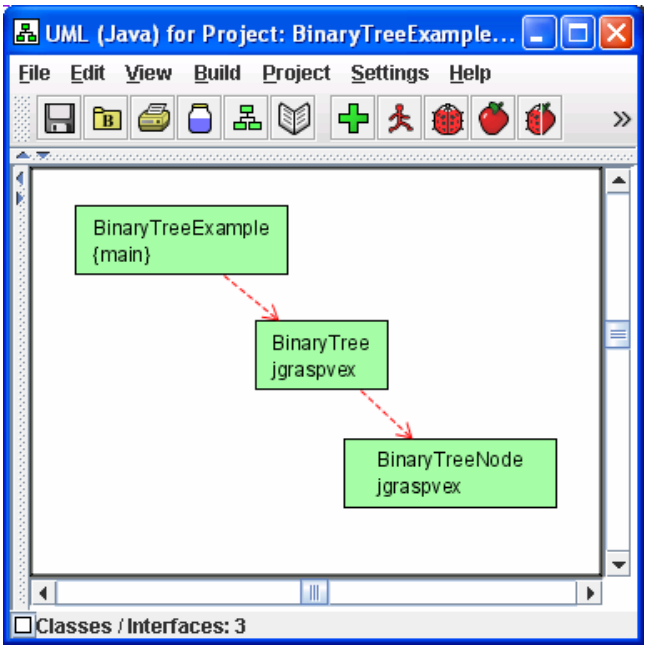

Figure 1. UML class diagram of BinaryTreeExample.java

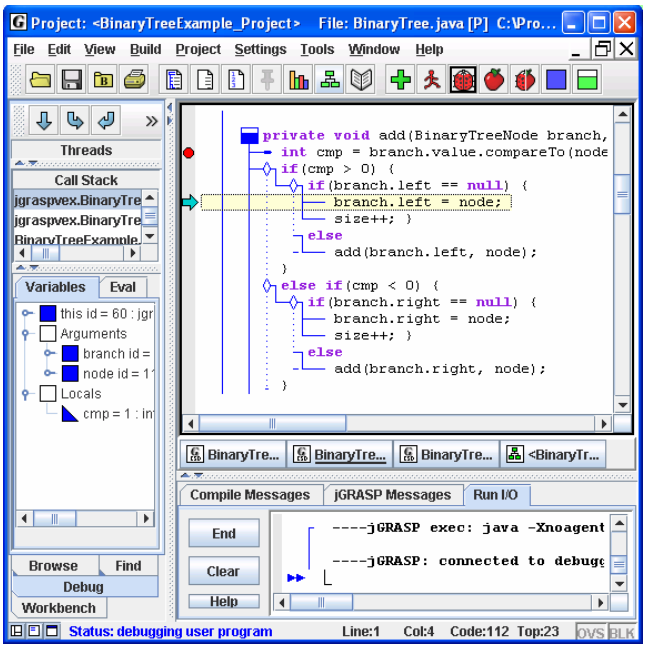

Figure 2. CSD window of jGRASP with the debugger after stopping in at a break point and then stepping in the *add()* method

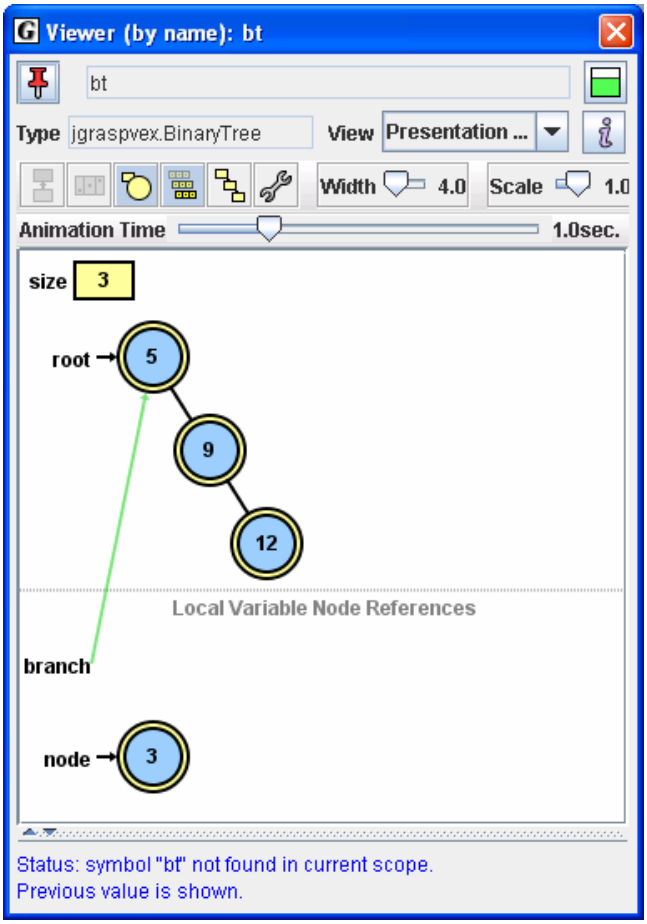

Figure 3. View after local node has been created and is about to be added to the binary tree

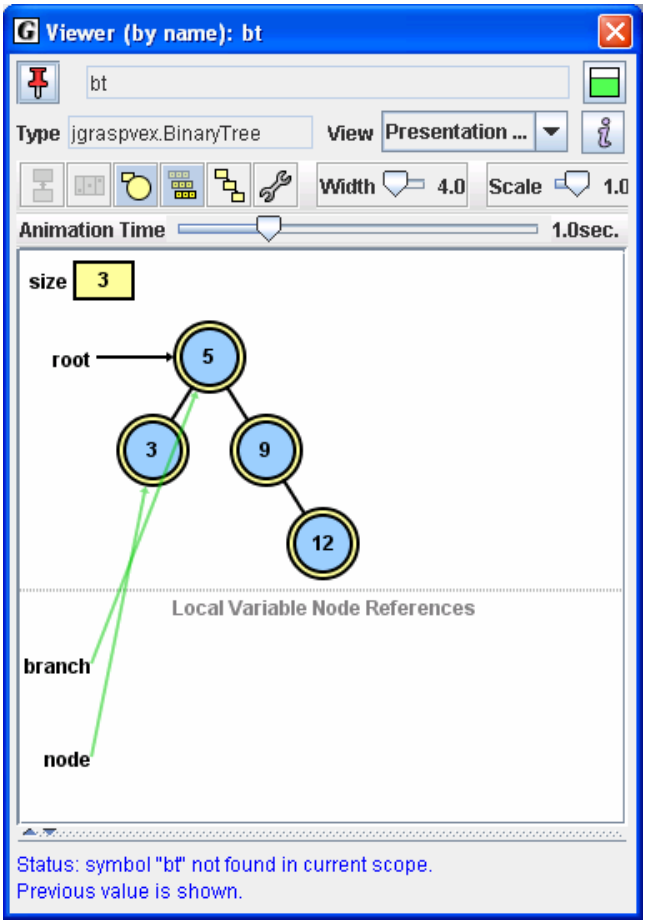

Figure 4. View after the node has "moved" from the local space into the binary tree and prior to *size* being updated

However, seeing the data structure updated in the viewer as individual statements are executed makes a direct connection between the implementation and the abstraction, and therefore provides a greater opportunity for deeper understanding. The results of two controlled experiments, described in the next section, support this informal feedback from students.

5. EVALUATION

5.1 Purpose

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

We conducted 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.

5.2 Subjects

Two criteria were important when choosing subjects for our controlled experiments. First, the subjects must be a close representation of the target population. The jGRASP viewers are being developed primarily for students enrolled in an introductory level data structure and algorithms course. Students enrolled in 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.

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 programming projects on doubly linked lists. In Spring 2006, the students completed eight in-lab activities as a part of the COMP2210 course. These were attendance-based, ungraded, in-lab activities that comprised of 5% of the course grade. All in-lab activities were conducted during the respective lab time of each section in a particular computer lab on campus. This ensured control over the hardware and software used by the subjects, and that the schedule of experiments did not conflict with the subjects' course-work.

We designed experiments based on the between-group approach to avoid the transfer of concepts learned in early experiments to a later experiment. An equal number of subjects were assigned to two separate groups. 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 [7]. Students in Group 1 were familiarized with the jGRASP debugger and students in Group 2 were familiarized with both the debugger and jGRASP viewers. Learning how to use the viewers took less than five minutes.

5.3 Experiment 1

Our hypothesis was that students would be more productive (would code faster and with greater accuracy) using the jGRASP data structure viewers. Students were asked to implement a basic traversal operation for linked binary search trees. The program LinkedBinarySearchTree.java (from the class textbook [5]) was used in this experiment. Students were provided a detailed description of the programming assignment and the grading policy. Students were required to work independently and were timed (although there was no time limit to complete the assignment). The independent variable was the visualization medium (coding using jGRASP viewers vs. without viewers). The dependent variables were: time taken to complete the assignment, and the accuracy of the assignment.

The control group implemented the level order traversal using the jGRASP visual debugger. The driver program provided to this group contained a toString() method so that they could print out the contents of the list without writing additional code. The treatment group implemented the same method using the jGRASP object viewers. Since our algorithm for levelOrder() traversal required three different data structures, we provided the students with three viewers (for LinkedBinaryTree, LinkedQueue and ArrayUnorderedList). The driver program given to this group did not contain the toString() method, 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.

5.4 Experiment 2

Our hypothesis was that students would be able to detect and correct logical bugs more accurately and in less time using jGRASP viewers. A Java program implementing a linked binary search tree with five logical errors, one in each of the following methods addElement(), findAgain(), removeElement(), inOrder() and postOrder() was provided. Students were asked to find and correct all the errors. The independent variable was the visualization medium (finding errors using jGRASP viewers vs. without viewers). The dependent variables were: number of bugs found, number of bugs accurately corrected, and number of new bugs introduced in the program while performing the experiment.

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

5.5 Results and Discussion

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. Our scoring of the students' work will constitute a grade that will be used to calculate up to three 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 three bonus points, quartile 2 will get two bonus points, quartile 3 will get one bonus point. Using this scheme both groups will be rewarded 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 [6]. Tests were conducted to check the normality of the distribution and the population was found to be normal for both experiments.

5.5.1 Results of Experiment 1

The null hypothesis was that there would be no difference in the accuracy and time taken for both groups. The mean time taken by the group with viewers was 69 minutes while the mean time taken by the group without viewers was 82 minutes. The mean accuracy of the treatment group with viewers was 6.93 points, while the mean accuracy of the control group without viewers was 5.06 points.

For the 34 samples in each group, Hotelling's T^2 statistic was calculated to be 20.565. The critical value for $\alpha = 0.05$, p=2 (two response variables), and n=34 (sample size) was 4.139. P-value was calculated to be 0.00007. Since the T^2 value is much greater than the critical value, and p-value is much less than the α value, we can strongly reject the null hypothesis. Thus, there was a statistical significant difference between the two groups, and we can say that in 95% of all cases, jGRASP object viewers helped increase the accuracy and reduce the time taken to write programs implementing data structures.

5.5.2 Results of Experiment 2

The null hypothesis was that there would be no difference in the number of bugs detected, corrected, introduced, and the time taken for both groups. The mean time taken by the group with viewers was 57.61 minutes, while the mean time taken by the group without viewers was 67.38 minutes. On average, the group using viewers located 3.19 errors, corrected 2.96 errors and introduced 1.66 errors, and the group without the viewers located 2.03 errors, corrected 1.69 errors and introduced 1.88 errors.

For the 34 samples in each group, Hotelling's T^2 statistic was calculated to be 22.121. The critical value for $\alpha = 0.05$, p=4 (four response variables), and n=26 (sample size) was 7.089. P-value was calculated to be 0.0005. Since the T^2 value is much greater than the critical value, and p-value is much less than the α value, we can strongly reject the null hypothesis. Thus, there was a statistical difference between the two groups, and we can say that in 95% of all cases, jGRASP object viewers helped increase the accuracy and decrease the time taken to write programs implementing data structures.

5.6 Other Experiments

We also carried out similar controlled experiments for singly linked lists [7]. For experiment 1, students were asked to implement four basic operations for singly linked lists – entry(), delete(), insert(), and contains(). For experiment 2, students were given a Java program implementing a singly linked list with nine errors in four methods add(), insert(), delete() and contains(). As with the experiments using linked binary search trees, the treatment group using viewers performed significantly better than the control group without viewers. Students were more productive and were able to detect and correct logical bugs more accurately using the jGRASP viewers.

6. SUMMARY AND FUTURE WORK

jGRASP object viewers automatically generate dynamic, state-based visualizations of objects and primitive variables in Java. Multiple synchronized visualizations of an object, including complex data structures, are immediately available to users within the IDE. Multiple instructors have used these viewers in CS1 and CS2 and have reported positive anecdotal evidence of their usefulness. The authors conducted formal, repeatable experiments during the Spring 2006 academic term to investigate the effect, if any, these viewers have on student learning, performance, and retention. The results indicated that the data structure viewers can indeed be beneficial for students learning about traditional data structures.

We compared the average scores of students in Group 1 and Group 2 in quizzes, exam 1, exam 2 and final exam for the course COMP 2210. In all four cases the performance of Group 2 was much better than Group 1. Early indicators suggest that jGRASP viewers help with retention of concepts as well. We are currently building a database of example Java programs from available data structure textbooks. We plan to continue to refine the heuristics in the Data Structure Identifier with a goal of automatically recognizing 95% of the classes that represent common linked data structures.

7. ACKNOWLEDGMENTS

The development of jGRASP has been supported by a research grant from the National Science Foundation.

8. REFERENCES

[1] Kannusmaki, O., Moreno, A., Myller, N., Sutinen, E. What a novice wants: students using program visualization in distance programming course. *Proc. of Third Progam Visualization Workshop*, July 1-2, 2004, 126-133.

[2] Zeller, A. Visual Debugging with DDD. *Dr. Dobb's*, July 2001 (http://www.ddj.com/184404519).

[3] Naps, T. JHAVÉ: supporting algorithm visualization. *IEEE Computer Graphics and Applications*, Sep-Oct 2005, 49-55

[4] Hamer, J. A lightweight visualizer for Java. *Proc. of Third Progam Visualization Workshop*, July 1-2, 2004, 55-61.

[5] Lewis, J and Chase, J. *Java Software Structures: Designing and Using Data Structures*, 2ed, Addison-Wesley, 2005.

[6] Johnson, R. A., and Wichern, D. W. Applied multivariate statistical analysis, 4ed, 1998, Prentice-Hall.

[7] Jain, J., Cross, J., Hendrix, D., and Barowski, L. Experimental Evaluation of Animated-Verifying Object Viewers for Java. *ACM Symposium on Software Visualization (SoftVis),* September 4-5, Brighton, UK, 2006.

[8] Hundhausen C., Douglas S., Stasko J. T. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing,* 2002, vol. 13, pp. 259-290.