

USER EXPERIENCE DESIGN AND EXPERIMENTAL EVALUATION OF
EXTENSIBLE AND DYNAMIC VIEWERS FOR DATA STRUCTURES

Except where reference is made to the work of others, the work described in this dissertation is my own or was done in collaboration with my advisory committee. This dissertation does not include proprietary or classified information.

Jhilmil Jain

Certificate of Approval:

Nedret Billor
Associate Professor
Mathematics and Statistics

James H. Cross II, Chair
Professor
Computer Science and Software
Engineering

Dean Hendrix
Associate Professor
Computer Science and Software
Engineering

David Umphress
Associate Professor
Computer Science and Software
Engineering

Joe F. Pittman
Interim Dean
Graduate School

USER EXPERIENCE DESIGN AND EXPERIMENTAL EVALUATION OF
EXTENSIBLE AND DYNAMIC VIEWERS FOR DATA STRUCTURES

Jhilmil Jain

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
May 10, 2007

USER EXPERIENCE DESIGN AND EXPERIMENTAL EVALUATION OF
EXTENSIBLE AND DYNAMIC VIEWERS FOR DATA STRUCTURES

Jhilmil Jain

Permission is granted to Auburn University to make copies of this dissertation at its discretion,
upon request of individuals or institutions and at their expense. The author reserves all
publication rights.

Signature of Author

Date of Graduation

DISSERTATION ABSTRACT

USER EXPERIENCE DESIGN AND EXPERIMENTAL EVALUATION OF EXTENSIBLE AND DYNAMIC VIEWERS FOR DATA STRUCTURES

Jhilmil Jain

Doctor of Philosophy, May 10, 2007
(M.S.W.E., Auburn University, 2002)
(B.E., Govt. College of Engineering, India, 2000)

207 Typed Pages

Directed by James H. Cross II

Many techniques for the visualization of data structures and algorithms have been proposed and shown to be pedagogically effective. Yet, they are not widely adopted because they lack suitable methods for automatically generating the visualizations, lack integration among visualizations, and lack integration with basic integrated development environment (IDE) support. In this work, additionally it was identified that the lack of adoption was because these tools do not focus on one of the main problems students in an introductory level data structures and algorithms class face; that is, the transition from abstract and static concepts to dynamic implementation.

To effectively use visualizations when developing code, it is useful to automatically generate multiple synchronized views without leaving the IDE. The jGRASP IDE has been extended to provide *object viewers* that automatically generate dynamic, state-based visualizations of data structures 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.

Formal and repeatable controlled experiments were conducted to investigate the effect of these viewers on the performance of students. These studies indicated a statistically significant improvement over traditional methods of visual debugging that use breakpoints. Six controlled experiments were conducted to test various hypotheses. The goal of Experiments I and III was to determine if students would be able to code more accurately and in less time using the jGRASP data structure viewers for a relatively easy (singly linked list) and a relatively hard (linked binary tree) to understand data structure. The goal of Experiments II and IV was to determine if students would be able to find and correct more logical errors accurately and faster using jGRASP viewers for a relatively easy (singly linked list) and a relatively hard (linked binary tree) to understand data structure. Experiment V was conducted using min-max heap to test if students would be able to transition from concept to implementation faster and more accurately using jGRASP viewers for data structures that are covered only conceptually in lectures. Experiment VI was conducted using linked priority queue to test if students would be able to apply concepts for data structures that were not covered in lectures faster and more accurately using jGRASP viewers. In all six experiments, the group using jGRASP viewers performed significantly better than the other group.

Thus, this research has shown that automatic generation of visualizations for data structures tightly integrated within an IDE helps students not only learn concepts but also aids in transitioning from static concept to dynamic implementation for both relatively easy and hard to learn data structures.

ACKNOWLEDGEMENTS

I have been very fortunate to have had Dr. James H. Cross II as my Ph.D. advisor. His clarity of thought and methodological approach of solving complicated issues was always an inspiration. I am grateful to Dr. Cross not only for his technical advice, but also for the encouragement and support he gave me when they were most needed.

I would like to thank rest of my faculty advisory committee: Nedret Billor, Dean Hendrix and David Umphress for their helpful comments and suggestions. I would also like to thank Larry Barowski for integrating the viewers in jGRASP. Additionally, I am thankful to all the students of COMP2210 who participated in the experiments.

I would also like to thank my father, who passed away in 2005, for getting me interested in science and technology at an early age, and my mother for believing her daughter could do anything she chose to do and always encouraged me.

Style manual or journal used: ACM Computing Surveys.

Computer software used: Microsoft Word, Microsoft Excel, SAS, NCSS/PASS, and jGRASP integrated development environment.

TABLE OF CONTENTS

| | |
|--|-----|
| LIST OF FIGURES | xii |
| LIST OF TABLES..... | xvi |
| 1. INTRODUCTION | 1 |
| 2. LITERATURE REVIEW | 6 |
| 2.1. Tools for learning data structures | 6 |
| 2.1.1. Conceptual level | 6 |
| 2.1.2. Implementation level | 11 |
| 2.1.2.1. Visual debugging | 15 |
| 2.1.3. Summary of DSV tools | 18 |
| 2.2. Guidelines for design of DSV tools | 18 |
| 2.2.1. Methods for visualization generation | 22 |
| 2.3. Guidelines for pedagogical effectiveness of DSV tools | 23 |
| 2.3.1. Guidelines for improving learning for students | 24 |
| 2.3.2. Guidelines for increasing adoptability by instructors | 27 |
| 2.4. Conclusions: design and pedagogical requirements | 27 |
| 3. SURVEY DESIGN AND ANALYSIS TO INVESTIGATE DATA STRUCTURE UNDERSTANDING | 30 |
| 3.1. Survey design | 30 |
| 3.2. Participants and deployment | 32 |

| | |
|--|----|
| 3.3. Results and discussions | 32 |
| 3.4. Conclusions | 42 |
| 4. jGRASP DATA STRUCTURE VIEWERS | 43 |
| 4.1. Technology used to create viewers | 44 |
| 4.2. Types of viewers | 45 |
| 4.2.1. Animated verifying viewers | 51 |
| 4.3. Types of viewer generation | 53 |
| 4.3.1. API based | 53 |
| 4.3.1.1. An example | 53 |
| 4.3.2. Automatic identification | 57 |
| 4.3.2.1. Data structure identifier | 58 |
| 4.3.2.2. An example | 60 |
| 5. EXPERIMENTAL EVALUATION | 64 |
| 5.1. Experimental design issues | 65 |
| 5.1.1. Subject selection | 66 |
| 5.1.2. Grading and compensation | 67 |
| 5.1.3. Data analysis | 68 |
| 5.2. Experiment 1 – linked list | 69 |
| 5.2.1. Method | 69 |
| 5.2.1.1. Participants | 69 |
| 5.2.1.2. Materials | 69 |
| 5.2.1.3. Design and procedure | 70 |
| 5.2.2. Results | 72 |
| 5.3. Experiment 2 – linked list | 74 |
| 5.3.1. Method | 74 |
| 5.3.1.1. Participants | 74 |
| 5.3.1.2. Materials | 74 |

| | | |
|----------|---|-----|
| 5.3.1.3. | Design and procedure | 78 |
| 5.3.2. | Results | 79 |
| 5.4. | Experiment 3 – linked binary tree | 82 |
| 5.4.1. | Method | 82 |
| 5.4.1.1. | Participants | 82 |
| 5.4.1.2. | Materials | 82 |
| 5.4.1.3. | Design and procedure | 83 |
| 5.4.2. | Results | 84 |
| 5.5. | Experiment 4 – linked binary tree | 86 |
| 5.5.1. | Method | 86 |
| 5.5.1.1. | Participants | 86 |
| 5.5.1.2. | Materials | 86 |
| 5.5.1.3. | Design and procedure | 89 |
| 5.5.2. | Results | 90 |
| 5.6. | Experiment 5 – min-max heap | 92 |
| 5.6.1. | Method | 92 |
| 5.6.1.1. | Participants | 92 |
| 5.6.1.2. | Materials | 93 |
| 5.6.1.3. | Design and procedure | 93 |
| 5.6.2. | Results | 95 |
| 5.7. | Experiment 6 – linked priority queue..... | 97 |
| 5.7.1. | Method | 97 |
| 5.7.1.1. | Participants | 97 |
| 5.7.1.2. | Materials | 97 |
| 5.7.1.3. | Design and procedure | 100 |
| 5.7.2. | Results | 100 |
| 5.8. | Sample size analysis | 102 |
| 5.9. | Retention of concept..... | 105 |

| | |
|--|---------|
| 6. QUESTIONNAIRE TO EVALUATE THE USER INTERFACE ASPECTS OF jGRASP VIEWERS AND DEBUGGER..... | 106 |
| 6.1. Debugger questionnaire | 106 |
| 6.1.1. The debug buttons | 107 |
| 6.1.2. Results and discussions | 109 |
| 6.1.3. Interface layout recommendations | 113 |
| 6.2. jGRASP viewer questionnaire | 115 |
| 6.2.1. Results and discussions | 116 |
| 6.2.2. Interface layout recommendations | 120 |
| 7. SUMMARY AND CONCLUSIONS..... | 122 |
| BIBLIOGRAPHY..... | 127 |
| APPENDICES..... | 137 |
| A – Survey for data structure understanding (Fall 2004/Spring 2005) | 138 |
| B – Interview results for data structure understanding (Fall 2004/Spring 2005) | 140 |
| C – Test 1: Questions to test error detection and correction..... | 141 |
| D – Test 2: Questions to test program understanding and tracing..... | 151 |
| E – Activity to familiarize students with jGRASP debugger..... | 155 |
| F – Activity to familiarize students with jGRASP debugger and viewers..... | 162 |
| G – Program LinkedSet.java provided for Experiment II..... | 170 |
| H – Program LinkedBinarySearchTree.java provided for Experiment IV..... | 173 |
| I – Program Heap.java provided for Experiment V..... | 179 |
| J - Program PriorityQueueLinked.java provided for Experiment VI..... | 184 |
| K - SAS code for Experiment I, III, V and VI: 2 response variables..... | 185 |
| L - SAS code for Experiment II and IV: 4 response variables..... | 186 |
| M – Questionnaire on jGRASP debugger –Group 1..... | 187 |
| N – Questionnaire on jGRASP viewers –Group 2 | 189 |

LIST OF FIGURES

| | |
|---|----|
| Fig. 1.1: Dissertation research focus | 1 |
| Fig. 3.1: Pie charts of year in the undergraduate degree..... | 33 |
| Fig. 3.2: Pie charts of COMP 2210 majors..... | 33 |
| Fig. 3.3: Pie charts of Java level..... | 34 |
| Fig. 3.4: Fall 2004 – Java Level grouped by major..... | 35 |
| Fig. 3.5: Spring 2005 – Java Level grouped by major..... | 36 |
| Fig. 3.6: Fall 2004 – Java level grouped by year..... | 37 |
| Fig. 3.7: Spring 2005 – Java level grouped by year..... | 38 |
| Fig. 3.8: Bar chart comparing programming language experience..... | 39 |
| Fig. 3.9: SAS code used to determine association between each understanding level where <i>usg</i> was assigned a value “Concept”/”Impl”/”Appl”, and <i>atd</i> was assigned the numerical rating..... | 40 |
| Fig. 3.10: SAS code used to determine association between all the data structures where <i>sname</i> was assigned the name of the data structure “list-array”/ “stack-pointer”/ “tree” etc, and <i>atd</i> was assigned the numerical rating..... | 41 |
| Fig. 4.1: jGRASP virtual desktop | 44 |
| Fig. 4.2: Interface-based viewer for Stack collection class..... | 46 |
| Fig. 4.3: Structure-based viewer for Stack collection class..... | 46 |
| Fig. 4.4: Structure-based viewer for ArrayList collection class..... | 47 |
| Fig. 4.5: Structure-based viewer for LinkedList collection class..... | 47 |
| Fig. 4.6: Structure-based viewer for PriorityQueue collection class..... | 48 |
| Fig. 4.7: Interface-based v viewer for TreeMap collection class..... | 49 |
| Fig. 4.8: Structure-based viewer for TreeMap collection class..... | 49 |
| Fig. 4.9: Interface-based viewer for HashMap collection class..... | 50 |
| Fig. 4.10: Structure-based viewer for HashMap collection class..... | 50 |

| | |
|--|----|
| Fig. 4.11: Structure-based viewer for Vector collection class..... | 51 |
| Fig. 4.12: Details of the controls of the viewer window..... | 54 |
| Fig. 4.13: Code fragments of LinkedSet.java and LinearNode.java..... | 55 |
| Fig. 4.14: LinkedSet.java – CSD window of jGRASP with the debugger stopped at break point..... | 56 |
| Fig. 4.15: View when the node with value 6 has been attached to the <i>previous</i> node <i>before</i> in the linked list..... | 56 |
| Fig. 4.16: View after the next pointer of the new node is set to point to the rest of the list (pointed to by <i>after</i>). The remaining list slides up from the local space to the main structure..... | 56 |
| Fig. 4.17: Code fragments of LinkedBinarySearchTree.java and BinaryTreeNode.java..... | 60 |
| Fig. 4.18: Configuration dialog box for automatic structure identification..... | 61 |
| Fig. 4.19: Possible structural mappings identified for the given LinkedBinarySeachTree and BinaryTreeNode code fragments..... | 62 |
| Fig. 4.20: <i>LinkedBinarySearchTree</i> – CSD window of jGRASP with the debugger stopped at a break point | 63 |
| Fig. 4.21: View after local node has been created and is about to be added to the binary tree..... | 63 |
| Fig. 4.22: View after the node has “moved” from the local space into the binary tree and prior to <i>size</i> being updated..... | 63 |
| Fig. 5.1: Methods used for Experiment I and Experiment II..... | 71 |
| Fig. 5.2: Experiment I – comparison of mean time..... | 72 |
| Fig. 5.3: Experiment I – comparison of mean accuracy..... | 73 |
| Fig. 5.4: Experiment II – CSD window of jGRASP with debugger stopped at a breakpoint in the add() method..... | 75 |
| Fig. 5.5: View after two nodes have been added and the next pointer of the third node (to be added) is set to the head node..... | 75 |
| Fig. 5.6: View after head of the list has been set to the third node being added. The node has “moved” from the local space into the linked list and prior to the <i>count</i> being incremented | 75 |
| Fig. 5.7: Experiment II – CSD window with the debugger stopped at a breakpoint in the insert() method | 76 |

| | |
|---|-----|
| Fig. 5.8: The next pointer of the node to be inserted is set to index 1 instead of index 0..... | 76 |
| Fig. 5.9: The next pointer of the node at index 0 is set to the new node <i>tmpNode</i> , and the node slides in from the local space into the linked list and prior to <i>count</i> being incremented..... | 76 |
| Fig. 5.10: Experiment II – Debugger breakpoint stopped in the delete() method..... | 77 |
| Fig. 5.11: Node at index 1 was supposed to be deleted. <i>current</i> points at the node that will be deleted..... | 77 |
| Fig. 5.12: View after the next pointer of <i>previous</i> is set to the node pointed by <i>current</i> . The node “c” slides down into the local space since it is no longer a part of the linked list. The <i>count</i> variable has not been decremented yet..... | 77 |
| Fig. 5.13: Experiment II – comparison of mean time..... | 80 |
| Fig. 5.14: Experiment II – comparison of mean bugs (logical errors) located, corrected and introduced | 80 |
| Fig. 5.15: Methods used in Experiment III | 83 |
| Fig. 5.16: Experiment III – average time taken by the treatment group (with viewers) and the control group (without viewers) | 85 |
| Fig. 5.17: Experiment III – average accuracy of the treatment group (with viewers) and the control group (without viewers) | 85 |
| Fig. 5.18: Methods used in Experiment IV..... | 89 |
| Fig. 5.19: Experiment IV: Debugger stopped at a breakpoint in the addElement() method.. | 87 |
| Fig. 5.20: View after local node with value 8 has been created but not yet added to the tree..... | 87 |
| Fig. 5.21: View after newNode is added to the tree, and is incorrectly set as <i>root</i> | 87 |
| Fig. 5.22: Experiment IV – average time taken by the treatment group (with viewers) and the control group (without viewers) | 91 |
| Fig. 5.23: Experiment IV – average accuracy of the treatment group (with viewers) and the control group (without viewers) | 91 |
| Fig. 5.24: Details of max heap used in Experiment V..... | 95 |
| Fig. 5.25: Experiment V – comparison of mean time..... | 96 |
| Fig. 5.26: Experiment V – comparison of mean accuracy..... | 96 |
| Fig. 5.27: Details of priority queue used in Experiment VI..... | 99 |
| Fig. 5.28: Experiment VI – comparison of mean time..... | 101 |

| | |
|---|-----|
| Fig. 5.29: Experiment VI - comparison of mean accuracy..... | 101 |
| Fig. 6.1: The debug button panel in jGRASP..... | 107 |
| Fig. 6.2: Usefulness of the <i>Debug Tab</i> features..... | 110 |
| Fig. 6.3: Frequency of use of the <i>Debug Tab</i> features..... | 110 |
| Fig. 6.4: Frequency of use of Debug controls..... | 111 |
| Fig. 6.5: Icon representation of Debug controls..... | 111 |
| Fig. 6.6: Layout recommendation 1 for debug panel..... | 114 |
| Fig. 6.7: Layout recommendation 2 for debug panel..... | 114 |
| Fig. 6.8: Usefulness of viewer features..... | 117 |
| Fig. 6.9: Frequency of use of viewer features..... | 117 |
| Fig. 6.10: Icon representation of viewer features..... | 118 |
| Fig. 6.11: Layout recommendation for the viewer controls based on usefulness and frequency of use..... | 120 |

LIST OF TABLES

| | |
|--|-----|
| Table 2.1: Summary of eight common software visualization taxonomies..... | 19 |
| Table 3.1: Excerpt from Question 6. Legend indicates ratings that are used to data structure understanding | 32 |
| Table 3.2: Average association among different levels of understanding for each data structure..... | 41 |
| Table 3.3: Association among the different data structures for each level of understanding..... | 41 |
| Table 3.4: Ratings of data structures covered in less time during lectures..... | 42 |
| Table 5.1: Students that correctly implemented methods for Experiment 1 (Group 1)..... | 73 |
| Table 5.2: Students that correctly implemented methods for Experiment 1 (Group 2)..... | 73 |
| Table 5.3: Students that correctly completed methods for Experiment 2 (Group 1)..... | 81 |
| Table 5.4: Students that correctly completed methods for Experiment 2 (Group 2)..... | 81 |
| Table 5.5: Method implemented for Experiment VI..... | 98 |
| Table 5.6: Comparison of average scores of Group 1 and Group 2 in the COMP 2210 course..... | 105 |
| Table 6.1: Results of jGRASP debugger questionnaire for Group 1..... | 112 |
| Table 6.2: Results of jGRASP viewers questionnaire for Group 2..... | 118 |
| Table 6.3: Results of open ended questions for jGRASP viewers for Group 2..... | 121 |

CHAPTER 1

INTRODUCTION

The research focus of this dissertation is a unique intersection of three prevailing areas of computer science education: program visualization, data structure visualization and algorithm animation (Figure 1.1). These three areas fall into a general category called software visualization. Software visualization (SV) is a technique of using imagery to manage the complexity of program artifacts to improve understanding and facilitate efficient learning. Program visualization (PV) uses graphical elements to increase program comprehension and illustrate the program's run time behavior. Algorithm animation (AA) is the use of graphics to show how the program works at a conceptual level. Data structure visualization (DSV) falls between PV and AA – the goal here is to increase comprehensibility of the underlying algorithm and the associated program behavior. The difference between visualization and animation is that visualization is typically static and animation is a dynamic representation of the domain. Since both types of images are implemented, the jGRASP visualizations will refer to these as *views* and the components that provide them will be referred to as *viewers*.

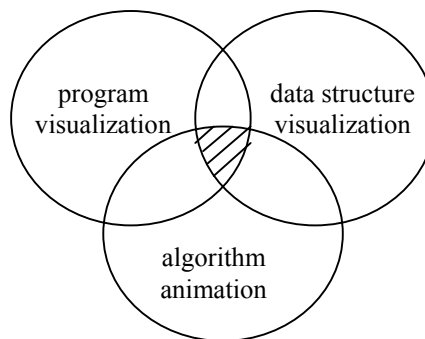


Fig. 1.1: Dissertation research focus

All computer science, software engineering, computer engineering, and wireless engineering (software option) majors at Auburn University are required to take COMP 1210 Fundamentals of Computing I. COMP 1210 provides an introduction to the Java programming language. This course is followed by COMP 2210 Fundamentals of Computing II, which is the introductory level data structure course. It uses an object-oriented approach to introduce the basic concepts, design, implementation and application of fundamental data structures.

Data structures and algorithms are abstract concepts, and the understanding of these topics and the material covered in class can be divided into three levels: a) Conceptual – where students learn concepts of operations such as create, add, delete, sort etc; b) Coding – where students implement the data structure and its operations using any programming language (Java in this case); and c) Application - where students choose the most appropriate data structure to solve a programming exercise. Over the course of the past few years a consistent decline in enrollment in the computer science department has been observed. This trend is most noticeable during COMP 2210 when quite a few students decide to drop this required course. Paper-based surveys and one-on-one interviews were conducted in Fall 2004 and Spring 2005 to understand the aspects of the COMP 2210 course that students find most difficult. It was determined that students did not find fundamental concepts difficult to understand but had the 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. Hence, poor Java skills may not be causing the problems with implementation. Most students faced a blank-screen syndrome when they began implementation [Jain et al. 2005a]. The basic problem is that students have difficulty transitioning from static textbook concepts to dynamic programming implementation. Thus, there is a need to bridge the gap between concepts and implementation.

Felder and Silverman 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 such as pictures, diagrams, flow charts, etc. In programming, visual learners can benefit from creating diagrams of problem solutions (e.g., flow charts) before coding [Felder and Silverman 1988]. 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.

Over 21 tools that are used for the purpose of data structure visualization were surveyed [Jain et al. 2005b] and it was found that most tools (more than 14 in the survey) focused on conceptual understanding. Only seven implementation level tools included in the survey were intended to help students during program comprehension and debugging activities. But, none of these implementation tools fulfilled all of the following research goals:

Pedagogical goals:

1. Actively engage students.
2. Reduce cognitive load of the short term memory so that efforts can be directed to problem solving.
3. Easy transition from static textbook concepts to dynamic implementation.
4. Reduce number of tools required by using one tool that serves the dual purpose of classroom demonstration and development environment.

Design goals:

1. Provide automatic generation of views.
2. Provide multiple and synchronized views.
3. Provide full control over the speed of the visualization.

The jGRASP lightweight IDE (<http://jgrasp.org>) has been extended to include 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 viewers is to provide fine grained

support for understanding instances of classes representing data structures. When a class has more than one view associated with it, the user 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.

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 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. Formal and repeatable experiments were conducted to investigate the effect of these viewers on student performance for a relatively easy to understand data structure (linked list using pointers) [Jain et al. 2006], a relatively hard to understand data structure (linked binary tree), data structures that are covered only conceptually during lectures, and data structures that are not covered in lectures at all. A statistically significant improvement over traditional methods of visual debugging that use breakpoints was found in all cases.

In Chapter 2, various data structure and algorithm visualization systems developed in academia, and guidelines provided for their design and pedagogical effectiveness are explored. Based on these two aspects, goals and requirements for this research are outlined.

In Chapter 3, design and analysis of surveys conducted to investigate data structure understanding are discussed. Various factors affecting students in COMP 2210 are also explored.

In Chapter 4, user interface design of jGRASP viewers is discussed. The details of two types of viewers interface-based and structure-based are given. This is followed by a discussion of viewer generation in jGRASP using an API based approach and automatic generation using detailed examples.

In Chapter 5, details of experimental evaluation conducted to test the various hypotheses are given. Experiment I and II were conducted using singly linked lists, Experiment III and IV were conducted using linked binary trees, Experiment V was conducted using min and max heaps, and finally Experiment VI was conducted using linked priority queues.

In Chapter 6, the design and analysis of a questionnaire conducted to evaluate the user interface aspects of jGRASP debugger and viewers are discussed. This is followed by recommendations for reworking the user interface of both features.

In Chapter 7, the following are summarized: i) motivation for this research, ii) results from the analysis of surveys for data structure understanding and literature review that were used to define the goals of this research, iii) jGRASP viewer design, iv) empirical evaluation conducted to test the effectiveness of the viewers, and v) questionnaire to evaluate the usability of jGRASP debugger and viewers. This chapter concludes with a description of current research activities and recommendations for follow-on experiments.

CHAPTER 2

LITERATURE REVIEW

This chapter reviews the current literature on data structure and algorithm visualization (DSV). First, the various academic data structure and algorithm visualization systems are reviewed. Second, system design and pedagogical effectiveness guidelines are explored. Based on these two aspects goals and requirements for this research are outlined.

2.1. TOOLS FOR LEARNING DATA STRUCTURES

There are two levels of data structure understanding: conceptual and implementation. At the conceptual level of understanding, students are required to learn the algorithm of operations used to construct a data structure such as add or delete nodes, search for a node, sort the data structure etc. At the implementation level of understanding students are required to write a program that correctly implements the data structure and the various related operations. In this section, a number of data structure and algorithm visualization systems are reviewed and categorized as either conceptual or implementation.

2.1.1. Conceptual Level

1. **ANIMAL: A New Interactive Modeller for Animations in Lectures** [Rößling and Freisleben 2000a, 2000b, 2002] is a system for creating algorithm and data structure visualizations using a visual editor or scripting commands. Using the editor, novice users can generate or edit

animations visually without using any programming code. Objects such as points, polygon, polylines, text, list elements, and arcs can be added to the animation using drag and drop. Advanced users can also use ANIMAL's scripting language for creating animations. Using this tool, animations can be displayed using video-player like features such as play, pause, rewind, or jump to a given step. Source code or pseudo code and textual descriptions can be embedded within the animation. The system's flexibility does not restrict it to introductory computer science courses, and also provides platform independence.

2. **JAWAA**: The *Java And Web* based *Algorithm Animation* [Akingbade 2003] [Pierson and Rodger 1998] [Rodger 2002] is a scripting language that facilitates easy creation of web-based animations. General-purpose animations as well as data structure animations can be created in a matter of minutes. First, a .anim file containing JAWAA commands or scripts is created by hand or by using the JAWAA editor. The JAWAA editor allows creation of animations using a GUI by laying out objects. This .anim text file is then called as an applet from an html web page to generate animations on the web. JAWAA is language independent and no prior programming experience is required to use it.
3. **JIVE**: The *Java Interactive software Visualization Environment* [Cattaneo et al. 2002] [Jive 2002] is a highly interactive system for automatically creating visualizations of programs using its library of pre-coded animated data structures such as graphs, hashtables, and search trees. The graphs and binary search trees are based on the JDSL library. Users can also create stand-alone Java applets with interactive GUIs. JIVE provides an excellent interface for visualizing large data sets using an innovative zooming graphical framework. It also provides a multi-user distributed learning environment such that teachers and students can interact with the same animation or data structure synchronously.

4. **JVALL**: *Java Visual Automated Linked List* [Dershem et al. 2002] provides animation of linked list operations and it is fully compatible with the Java LinkedList class. The GUI consists of three areas. The top part contains user controls such as colors, speed and implementation model. The middle part displays the actual visualization of the linked list. The bottom part consists of text reporting, animation status, and redo/undo buttons. The JVALL system can be used in the data structures course as an interactive linked list client, for laboratory activities, for classroom demonstration, for debugging programs and lastly for visualization of classes implemented using the Java LinkedList class. The advantages of JVALL include flexibility of animating any program using the LinkedList class, easy integration of algorithm text and visualization, ease of modification of color and visual display components (such as nodes, pointers, arrows and background color), ease of controlling execution speed (such as redo, undo, and rewind), and support for animation of algorithm given by the user.

5. **JSAVE**: The *Java Simple Automated Visualization Environment* [Jsave 2003] is an interactive system for the visualization of Java Collection classes. Currently, only the List interface is supported. It provides a library of classes that can be directly used in Java programs or XML scripts can be written for visualization purposes. The specialty of JSAVE is the flexibility of user interaction in terms of excellent user control of color, navigation, and multiple representations of the data structure visualizations. Dynamic color customization of components is possible while interacting with the visualizations. The user can play the visualization as a movie, or step through it. JSAVE also allows rewinding the visualization or stepping back through it. The user can dynamically switch between singly linked list, circular list, array, and relative comparison representations as the visualization is running in order to compare the data structures. The ultimate goal of JSAVE is to provide a complete visualization of the functionality of the Java Collection classes.

6. **Jarc's Web-based courseware** (Ada 95 based course) - Jarc and Feldman [1998] developed an interactive multimedia environment for data structure visualization and algorithm animation. The courseware consists of eleven laboratory exercises to be used for a four-week period. The course covers topics such as graphs, binary trees, and sorting. There are three components for each exercise, a) page containing explanatory text; b) page containing Ada 95 code, and c) applet visualizing the data structure. The unique features of this system are the two interactive modes: *Show Me* and *I'll Try*. The *Show Me* mode allows the students to explore the solutions of exercises. The *I'll Try* mode gives the student the full interaction capability to try to replicate the steps of the algorithm. Experimental studies performed on the system reported the following: quick sort and graph search problems are most difficult, there was no significant difference in learning between the students with active and reflective learning styles, and there was no statistically significant difference in the performance between the *I'll Try* and *Show Me* modes.

7. **JHAVÉ** - The **J**ava **H**osted **A**lgorithm **V**isualization **E**nvironment [Nap et al. 2000] is not a visualization system itself but serves as a client into which algorithm visualization engines can be plugged in. JHAVÉ currently supports three such engines – (a) Samba animation-scripting language designed by Stasko [1990, 1998]; (b) the GAIGS data structure visualization language developed by Naps and Bressler [1998]; and (c) the ANIMAL scripting language developed by Röbling [Röbling and Freisleben 2002]. The environment provides four pedagogical tools – context-sensitive documentation in a browser window, stop-and-think questions, input generators, and rewind capability. Any algorithm visualization engine, which produces visualizations using a script file, can be plugged into JHAVÉ. The server of JHAVÉ manages available algorithms and generates script files that the client can display. The user can access JHAVÉ by using any web browser, which launches an AVClient applet. The user can select any of the available algorithms and the

request is sent to the server, which generates a script file for it. The client then uses the appropriate engine to render this script file. If the user had requested inputs to the algorithm, an input generator object is sent to the server. The server uses this object to run the algorithm. It is not clear from the documentation how customized visualizations can be created.

8. **MRUDS - Multiple Representations for Understanding Data Structures** [Hanciles et al. 1997] is a system built using Microsoft Visual Basic 3.0 for the Windows platform. The two goals of MURDS are the effective use of multiple representations for linear data structures such as arrays, stacks, queues and linked lists and integration of learning strategies such as elaboration and metacognition. MURDS consists of three modules: Domain, Presentation and Interface. The domain module is subdivided into three parts: analogy, representation and algorithm. The analogy part consists of every day metaphors used to describe concepts of data structures. The representation part consists of diagrammatic illustrations of data structures and the algorithm part contains algorithms for the respective data structures. The presentation module determines how the concepts should be displayed to the student. It is subdivided into four parts: analogy (animation of metaphors), representation (animation of structural representation of concepts), algorithm (pseudo code added to representation), and self-assessment (measuring students knowledge of the domain). The interface part allows the user to interact with MURDS using mouse clicks and drag-and-drop actions. MURDS was evaluated using formative and summative techniques. The authors reported that all multiple representations (i.e., analogy, structural representation and algorithm) helped students learn more efficiently than any other combination.
9. **JDSL Visualizer Tool - Java Data Structures Library** [Baker et al. 1999] consists of APIs, which can be used to create visualizers for data structures using Java. The JDSL visualizer GUI consists of several components. The top-left panel contains the visualization, the top-

right panel contains the history, and the bottom panel consists of a number of buttons each of which corresponds to a method of the data structure being displayed. Two separate windows contain the exceptions thrown and the online help respectively. Six data structures are currently supported: enumerations, sequences, binary trees, binary trees with rotations, heaps and red-black trees. The unique feature of this tool is the history panel, which allows the user to compare any two states of the data structure. Customized visualizers can be created but to achieve this additional code must be inserted into the program implementing the data structure.

10. **MatrixPro**: This system is based on the Matrix algorithm simulation application framework [Karavirta et al. 2002, 2004a, 2004b][Korhonen et al. 2004]. The goal of the system is to enable instructors to use on-the-fly direct manipulation to demonstrate data structures and related algorithms. The tool also allows the instructor to ask “what-if” type of questions and incorporate exercises. The main GUI consists of three components: menubar, toolbar, and area of visualizations. The menubar allows the user to add and solve problems, although in a typical scenario instructors will use the functionality to add problems and students will use the functionality to solve problems. The toolbar contains functionality to manipulated animations, such as controlling the speed, adding and removing breaks, and changing the granularity of the animation. The area of visualization contain visual entities such as nodes, keys, and hierarchies which can be dragged and dropped, flipped, rotated, resized and customized.

2.1.2. Implementation Level

1. **LIVE**: The *L*anguage-*I*ndependent *V*isualization *E*nvironment [Campbell et al. 2003] is a system that enables visualization and manipulation of programs and data structures for

multiple languages such as subset of Java, C++, and ÜberLanguage (in-house Pascal like language). The GUI of LIVE consists of two main components: a canvas (on the left hand side) and a source code area (on the right hand side). The user can enter and edit code in the source code panel. When the code “Runs”, LIVE parses the program, creates a syntax tree, and generates the animation automatically. Since, animations are created by interpreting the syntax tree the user can switch between various code modes, thus allowing the user to view the same code in the syntax of multiple languages. The user can also directly and dynamically manipulate data structures displayed on the canvas and generate source code statements for the same. LIVE is especially useful in understanding the concepts of pointers, linked structures, recursion and effects of the scope of nested variables.

2. **JavaMy** – The system developed by Chen et al. [2003] provides data structure visualization for operations such as insertion and deletion for arrays, stacks, queues, trees, heaps and graphs; and rudimentary animations of simple user defined algorithms using the JavaMy programming language. A toolbar is provided at the bottom of the GUI, which allows the student to pause or resume, step- through, or control the speed of the animation. The student can interactively delete or add a random or a user-specified node to the data structure. The system was successfully used for applications using data structures such as: balances symbol checking, conversion of infix expressions to prefix expressions and vice versa, breadth first, and depth first search and sorting.
3. **SKA** – The **S**upport **K**it for **A**nimation [Hamilton-Taylor and Kraemer 2002] consists of a visual data structure library, a visual data structure manipulation environment and an algorithm animation system. The SKA canvas allows creation and manipulation of built in data structures. The available operations include adding or deleting a node, highlighting parts and deleting sub trees (if applicable). The user can run, pause and resume, and step through

- and add breakpoints to a pseudo code version of an algorithm. SKA supports parallel execution of multiple algorithms. The ultimate goal of SKA is to significantly reduce time required to create, manipulate and trace data structure diagrams. SKA can be used for one-on-one tutorials, group discussions and self-study.
4. **Swan** - This system [Shaffer et al. 1996] allows users to visualize data structures and basic execution process of C/C++ programs. In Swan a data structure is treated as a single or collection of directed or undirected graphs. The program implementing a data structure has to be physically annotated before the data structure can be visualized. The steps include adding calls to the Swan Annotation Interface Library (SAIL), then compiling the program, and then running the program, which results in the visualization. Swan can also be used as a graphical debugging tool at the abstract level since a two-way communication is possible between the annotations and views. Three execution controls are available to the user: run, step, and pause. Textual descriptions for visualizations can be added manually during annotation. The tool is easy to use, and the annotation system can be learnt quickly.
 5. **INCENSE** - This system [Myers 1983] allows users to design and display pictorial representations of data structures for programs written in a Pascal-like language called Mesa. Incense contains built-in displays for all the basic data types of Mesa, two-dimensional representation of arrays and records and pointers. The user can create very simple customized *Artists* by using the Mesa language. Multiple displays, called *Formats*, can be created for a single data structure. The user can specify the string name of the data type to be visualized during debug time. The system however does not have an integrated and interactive debugger, so it is not possible to set breakpoints and step through the code easily. Also, this system requires the user to write low-level graphics code to implement the viewers.

6. **DRUIDS** - *Display Resource for Understanding Internal Data Structures* [Whale 1994] consists of two modules: an *algorithm animator* (that focuses on searching and sorting) and a *program animator* (that focuses on displaying detailed structure and state of data structures). The program can be written in C or Modula-2. The system runs only on UNIX and Mac platforms. The student can create limited customized data structure visualizations, although multiple representations of a particular data structure are not possible. The system can be used in a classroom to replace static lecture slides or in a laboratory to animate student's code. In the program animator mode, DRUIDS enables the student to view the current execution point in the code, and the current values of variables in active procedures. The primary focus of the program animator however is to display basic linked structures such as stacks, general linked lists, self-organizing lists and rings, and balanced and unbalanced binary search trees. The algorithms can be viewed in a step-wise or continuous fashion pausing only when input is required from the user. Thus, the user does not have the capability to pause and resume.
7. **LJV**: *Lightweight Java Visualizer* [Hamer 2004a, 2004b] generates a textual description of the connectivity of the data structure using Java reflection, and passes these to the graph drawing software GraphViz. The user needs to write code in order to generate visualizations, which are static in nature. The layout is generated automatically, but it does not contain semantic meaning. This causes some confusion, as the representation can be different each time it is generated and does not match the textbook description. Also, since currently there is no mechanism to interact with the visualization, any changes made to the program are not seen immediately.
8. **Jeliot 3**: This system [Lattu et al. 2000] [Ben-Ari et al. 2002] [Moreno and Niko 2003] [Moreno et al. 2004] is very useful for basic memory-level visualization of programs written

in Java. It generates automatic visualization of data, control structures and method calls. The system uses the theater metaphor for displaying objects and variables. The user interface consists of four panels: left panel contains the source code, right panel contains the visualization, bottom left panel contains control buttons, and bottom right panel contains the output area. The visualization or animation area is further subdivided into four areas: method frame area, constants area, expression evaluation area and instance area. The only data structures that can be visualized using Jeliot 3 are one, two and three dimensional arrays. The system does not scale well as the number of dimensions and elements of an array are increased.

2.1.2.1. Visual Debugging Systems

This is a subcategory of implementation level systems. These systems specifically help the student in debugging activities using visual debugging techniques.

1. **MVT – Matrix Visual Tester** [Lönnberg et al. 2004] is a prototype-debugging tool based on interactive graphical testing. It allows the user to visually control program execution and provides visual manipulation of the program data structures. Data elision and abstraction can be used to control execution details such that the user can find and understand the information that is of interest. MVT has been used to visually manipulate Java programs without touching the target source code. The visualization consists of four areas – the topmost part contains the visualization manipulation controls, the second part is split into two areas the data view and structure panel. The data view consists of data containers depicted as tables and program variables that are grouped as table elements. Primitive values are shown as text and object references as arrows from the referring variable to the object that is referred to. The structure panel depicts the package tree, which contains the packages and classes used by the program

- being debugged. The bottom part contains code view, which depicts the execution position in the program. The user can either step forward or backward in the current executing thread.
2. **VIPS** – *V*isualization and *I*nteractive *P*rogramming *S*upport [Shimomura and Isoda 1990, 1991] tool can be used to automatically display list data structures of programs written in C. The visualization consists of seven windows – (1) monitor window: accepts debugging commands and displays the responses; (2) program-text window: displays the execution line in the program being debugged; (3) list window: displays list structure using rectangles and arrows; (4) input-output window: displays data inputted to the program being debugged and its output; (5) editor window: source code can be edited here; (6) variable display window: variables used by the program; and (7) stack display window: call stack of the program being debugged. The visualizations created cannot be customized and are available for the list structures only. Preliminary evaluation shows that VIPS reduces debugging time and number of commands used by 25-30%.
 3. **Lens** – This tool [Mukherjea and Stasko 1993, 1994] provides a combination of program visualization and algorithm animation for source code written in C. The user interface comprises of three areas: the left section contains the source code, the right section is divided into an upper, and lower area. The upper area contains the graphical editor, and the lower area contains the debugger command window. The user can issue “*attach event*” animation commands to debugger breakpoints. Using a combination of information provided by the debugger and user control dynamic animation-style data structure views can be created. These visualizations are not just static representations but also contain the rich semantics of program behavior that is annotated by the user. The problem with this approach is that the user must be able to correctly identify the code segments which need to be annotated. Since the creation of the data structure may be spread out in the program, this approach will result

in many related but separated annotations. Also, this approach cannot be used to debug a program that is already running.

4. **DDD** – This system [Zeller and Lütkehaus 1996] [Zeller 2001] provides visualizations of how a data structure is laid out in memory. Visualizations are created from the information given by the debugger; therefore annotating the source code is not required. The user can manipulate the visualization directly (using clicking) or by setting breakpoints in the source code. The visualization is created and laid out automatically and cannot be customized, although the user can use drag-and-drop to restructure the nodes. DDD is used for visualizing any linked data structures.

5. **TRAVIS** – *Traversal-based Visualization of Data Structures* [Korn and Appel 1998] uses a technique called *traversal-based visualization* to generate high-level and informative viewers while debugging. Advanced algorithm animation systems depend on user augmented source code to produce visualizations. Debuggers on the other hand use information obtained from symbol tables of the target program. Therefore, visualizations produced by debuggers often lack important semantic content, making them inferior to algorithm animation systems. TRAVIS aims to fix this problem. The debugger traverses a data structure using a set of patterns specified by the user to identify parts of the data structure to be drawn in a similar way. A declarative language is used to specify the patterns and the actions to be taken when the patterns are encountered. The user can also construct traversal specifications using a graphical user interface that will be translated to the declarative language. The debugger also supports modification of data. Thus changes made to the live visualization are reflected in the underlying data. This technique can be used to create visualizations for programs written in Java, C, and C++.

2.1.3. Summary of DSV Tools

The reason why some students struggle with understanding code is because visualizing the translation of the static description to a dynamic process is difficult. Numerous systems are available to address needs of students and instructors at various levels. In the literature review it was found that conceptual level systems that help in understanding the basics of data structures were in abundance, but the disadvantages are that most systems handle very few or a specialized set of data structures; they are typically stand-alone with no tracing abilities; and extensive amount of effort and time is required to create visualizations. In comparison, fewer implementation level tools that help the student in program comprehension activities and debugging were found in the literature review. The disadvantages of these tools are that programs had to be written in pseudo code or a subset of languages such as C++ or Java; only a limited number of data structures have debug support; tremendous effort (in terms of scripting or code annotations) is required to generate visualizations; Windows platform and Java language support is not available in most tools; and finally most are simply not available for download or are not supported anymore.

The focus of jGRASP viewers was to address all of the disadvantages mentioned above. In addition, the viewers are focused on helping the student transition from concept to implementation and are tightly integrated in an integrated development environment; both of these features are unique and are currently unavailable in any other system.

2.2. GUIDELINES FOR DESIGN OF DSV TOOLS

Various taxonomies for SV can be found in the literature, with Myers [1986, 1990] publishing one of the first in 1990. He suggested classifying systems based on a 2 x 3 grid of aspect vs.

display style. Aspect consists of what is being visualized (code, data or algorithm) and display style consists of static or dynamic illustrations. Shu described in her book [Shu 1988] a classification of SV systems based on what they present (data presentation, program construction and/or execution, software design), and their use as visual coaching systems (systems that bridge the gap between the process of creating a mental model and a program while solving a problem). Singh and Chignell [1992] published a taxonomy for SV systems very similar to Myers. They use aspect and form for classification purposes. Stasko and Patterson [1992] used four measures – aspect, abstraction, animation, and automation. Kraemar and Stasko [1993] classified systems using two dimensions: visualization task (data collection, data analysis, storage, display) and purpose (debugging, performance evaluation, program visualization). Brown [1988] used three measures: content (direct, synthetic), persistence (current, history), and transformation (incremental or discrete). Roman and Cox [1993] used five classification dimensions - scope, abstraction, specification method, interface and presentation. In 1992, Price et al. [1992] published a comprehensive taxonomy that was later extended in 1993 [Price et al. 1993]. This seems to be the most complete taxonomy found in our research. They used six dimensions (scope, content, form, method, interaction, and effectiveness) to categorize software visualization systems. See Table 2.1 for a summary of the eight taxonomies.

Table 2.1: Summary of eight common software visualization taxonomies

| Taxonomy | Classification Measures |
|--------------------------------|---|
| I. [Myers 1986, 1990] | <ol style="list-style-type: none"> 1. <i>Aspect</i> <ul style="list-style-type: none"> • code, data, algorithm 2. <i>Display Style</i> <ul style="list-style-type: none"> • Static, dynamic |
| II. [Shu 1988] | <ol style="list-style-type: none"> 1. <i>Visualization of</i> <ul style="list-style-type: none"> • data presentation, program construction and/or execution, software design 2. <i>Visual coaching</i> |
| III. [Singh and Chignell 1992] | <ol style="list-style-type: none"> 1. <i>Aspect</i> <ul style="list-style-type: none"> • program, algorithm, data |

| Taxonomy | Classification Measures |
|---------------------------------|---|
| | 2. <i>Form</i> <ul style="list-style-type: none"> • Static, dynamic |
| IV. [Stasko and Patterson 1992] | 1. <i>Aspect</i> 2. <i>Abstraction</i> 3. <i>Animation</i> 4. <i>Automation</i> |
| V. [Kraemer and Stasko 1993] | 1. <i>Task</i> <ul style="list-style-type: none"> • data collection, data analysis, storage, display 2. <i>Purpose</i> <ul style="list-style-type: none"> • debugging, performance evaluation or optimization, program visualization |
| VI. [Brown 1988] | 1. <i>Content</i> <ul style="list-style-type: none"> • direct, synthetic 2. <i>Transformation</i> <ul style="list-style-type: none"> • discrete, incremental 3. <i>Persistence</i> <ul style="list-style-type: none"> • current, history |
| VII. [Roman and Cox 1993] | 1. <i>Scope</i> <ul style="list-style-type: none"> • code, data state, control state, behavior 2. <i>Abstraction</i> <ul style="list-style-type: none"> • representation (direct, structural, synthesized) 3. <i>Specification method</i> <ul style="list-style-type: none"> • predefinition, annotation, declaration, manipulation 4. <i>Interface</i> <ul style="list-style-type: none"> • graphical vocabulary, interaction 5. <i>Presentation</i> <ul style="list-style-type: none"> • interpretation of graphics, analytical, explanatory, orchestration |
| VII. [Price et al. 1993] | 1. <i>Scope</i> <ul style="list-style-type: none"> • generality, scalability 2. <i>Content</i> <ul style="list-style-type: none"> • program, algorithm, code, fidelity and completeness, data gathering time 3. <i>Form</i> <ul style="list-style-type: none"> • medium, presentation style, granularity, multiple views, program synchronization 4. <i>Method</i> <ul style="list-style-type: none"> • visualization specification style, connection technique 5. <i>Interaction</i> <ul style="list-style-type: none"> • style, navigation, scripting facilities 6. <i>Effectiveness</i> <ul style="list-style-type: none"> • purpose, appropriateness and clarity, experimental evaluation, production use |

In [Jain et al. 2004] a qualitative analysis of six systems (ANIMAL, JAWAA, LIVE, JSAVE, LIVE, jGRASP viewers) was performed and it was concluded that the future development of tools should consider the following design guidelines.

1. Minimize learning curve and time by having one tool for classroom demonstration and development.
2. Enable visualization of concurrent programming features.
3. Provide multiple synchronized views of data structures.
4. Provide program synchronization.
5. Explore the benefits of features such as sound and multi-dimensional rendering.
6. Provide the ability to save the interactions with visualizations for future playback would aid students in revisiting material covered in class.
7. Provide visualization of large data sets and trace program data flow.
8. Provide full control over the speed and direction of the visualization.
9. Perform empirical evaluations must be carried out to gauge the effectiveness.

In addition, other design guidelines summarized by [Khuri 2001] are as follows:

1. Use consistent graphical layout of buttons, menus etc.
2. Provide help files to explain the tools itself and the organization of the interface within the tool.
3. Provide effective use of shape, size, color and texture. They can be used to call attention to specific data or process step, identify elements, depict logical structure, and highlight relationships. Use of more than four colors is not recommended since they will overload the short-term memory of the user. Similar background colors can be used to conceptually link two areas.

[Rößling 2003] suggests that a feature supporting the import and export (in various formats) of the visualizations should be available in the tool since content reuse is made possible.

2.2.1. Methods for Visualization Generation

1. **Manual:** The advantage of this approach is extreme flexibility of visualizations. The disadvantages are that visualization creation is time-consuming, requires a lot of expertise, and will not be used by most undergrad students to create customized views.
2. **Declarative:** Special logical commands are embedded into the source code (perhaps as formal comments). The advantage of this approach is that it is clean, while the disadvantage is that it requires expertise, and will not be used by most undergrad students to create customized views. Examples include Leonardo, Swan.
3. **API-based:** The advantage of this approach is that it is clean. The disadvantage is that it requires expertise, and will not be used by most undergrad students to create customized views. Examples include XTANGO, JVALL, JSAVE, JDSL Visualizer Tool.
4. **Scripting:** Very simple commands are used to create animations, which are saved as a “specialized” file. The advantage of this approach is that reuse of visualization is possible. The disadvantage is that it is may be somewhat time consuming. Examples include JAWAA, JSamba, ANIMAL.
5. **Topic-specific:** These visualizations focus of a particular area, e.g., graphs. The advantage of this approach is that good support in area of expertise is provided. The disadvantages of this approach are that the visualizations are “hard-wired”, not re-usable, and not customizable. Examples include JIVE, Jarc’s Web-based courseware, MURDS, JavaMy.

6. **Code interpretation:** The system evaluates and visualizes code automatically. The advantage of this approach is that no extra step to create visualization. The disadvantage is that the user may have little or no control over the appearance. Examples include Zstep, LIVE, JavaMy, INCENSE, DRUIDS, Jeliot, DDD.
7. **GUI:** A graphical user interface is used to interactively build visualizations. The advantage of this approach is that visualizations are flexible. The disadvantage is that it maybe difficult to learn how to use the interface. Examples: SKA (only circles, lines,text), ANIMAL.

Design issues which will help accomplish the research goal of having one tool for classroom demonstration and in-lab development are: (1) full control over the speed and direction of the visualization, (2) ease of visualization of large data sets, and (3) ease of creating custom viewers with minimal effort on the part of the user by using an interactive and intelligent GUI builder or automatic recognition of data structures.

2.3. GUIDELINES FOR PEDAGOGICAL EFFECTIVENESS OF DSV TOOLS

Before detailing the research conducted in the area of visualization effectiveness, it is important to understand the term *effectiveness* in the context of algorithm animation. Hundhausen et al. [2002] broadly describe software visualization artifact effectiveness as when any tool is shown to satisfy both the *usefulness* and *usability* criteria. Usefulness means that the tool must provide functionality that people actually would like to use and usability means that the tool must provide easy access and interactivity with that functionality. In order for the tool to be useful, it is important to identify: (1) the target users: their general background, the knowledge they have and what can they learn, (2) the goals and tasks the users want to accomplish, (3) the context in which

the user is working, and (4) determine how much automation is sufficient i.e., what has to be left to the machine and what to the user. Section 2 of this chapter and chapter 3 cover the steps taken to identify the goals and context of the target users (i.e., undergraduate students enrolled in CS2 level courses) for this research project. Usability of a system includes measuring well-known heuristics such as learnability (e.g., intuitive navigation), efficiency of use, memorability, good error recoverability, and subjective satisfaction. Chapter 5 and 6 detail the steps taken to measure usability of jGRASP viewers.

2.3.1. Guidelines for Improving Learning for Students

Listed below are pedagogical suggestions for improving data structure visualization tools for students.

- 1) Provide resources that help learners interpret the graphical representations [Khuri 2001].
- 2) Adapt to the knowledge level of the user.
- 3) Provide multiple views (of data, program and algorithm) [Khuri 2001].
- 4) Provide simultaneously identical views of different algorithms manipulating the same data [Bergin et al. 1996].
- 5) Include performance information.
- 6) Include execution history. However, Saraiya et al. [2004] demonstrate that this does not help.
- 7) Support flexible execution control (direction, speed). Bergin et al. [1996] demonstrate that providing support for high degree and flexible interaction control helps. Saraiya et al. [2004] report that control of pace/speed of visualization helps but control of direction does not. Rößling [2003] suggests that functionality similar to a video player (like forward and reverse) will help during lectures and office hours.
- 8) Support learner-built visualizations. Stasko [1997] demonstrated that actively engaging students in building their own visualizations helps. Hundhausen and Douglas [2000] report

- otherwise, although they suggest the reason could be that the students were given limited time for the task. In addition, Hundhausen [1998] suggested that when students create customized visualizations, conversations with an expert enables students understand more about the correctness and efficiency of algorithms.
- 9) Support custom input data sets. Naps et al. [2003a], Lawrence [1993, 1994] report that this feature is helpful, although Saraiya et al. [2004] report otherwise.
 - 10) Provide an example data set that covers the important cases in an algorithm. Saraiya et al. [2004] report that it is helpful, but Hansen et al. [2000] and Lawrence [1994] report otherwise. Intuitively, the latter makes sense, since the student does not get the opportunity to think about all the various test cases.
 - 11) Support dynamic questions. Saraiya et al. [2004] report that a question guide used by students to answers questions requiring exploration such that students are engaged intellectually actually does not help students learn. Jarc et al. [2000] also reported that predictive questions not effective. But the reason for the failure cited was that academically poor students using interactive visualizations tend to guess the questions rather than trying to understand. On the other hand, Hundhausen et al. [2002] and Byrne et al. [1996, 1999] report that questions used to predict future behavior of algorithm helps. [Grissom et al. 2003] [Khuri 2001] reported that responding to questions integrated into the visualization tool during their exploration of a DSV showed a significant improvement between pre- and post-test. Answering strategic questions about the visualization was shown to be effective [Hansen et al. 2000] [Naps et al. 2000].
 - 12) Support dynamic feedback.
 - 13) Complement visualizations with explanations [Naps et al. 2003a]. For example, use a text window to make sure the user understands the visualization [Bergin et al. 1996] [Khuri 2001]. Stasko et al. [1993] reported that visualization may be *ineffective* because instructor

- creating the visualization already understands the algorithm, the students on the other hand have no background or foundation on which to understand the algorithm. To construct this mapping, the instructor must explain it in words. Also, Colaso et al. [2002] demonstrated that text and visualization help in retention, which is the ultimate goal of education.
- 14) Visualizations should be consistent with the ones used in textbook.
 - 15) Provide user with standard GUI to interact with components within the visualization system. Bergin et al. [1996] and Khuri [2001] report that this improves usability, which is consistent with the basic rules or heuristics of user interface design.
 - 16) Substantial screen real estate will be needed for the most effective visualizations. Bergin et al. [1996] demonstrated that using innovative techniques to use real estate (like graphs, zooming etc) instead of multiple windows might be more effective. Rößling [2003] also suggested that the visualization canvas should be resizable.
 - 17) Strive to draw the user's attention to the critical area of the visualization. Bergin et al. [1996] report that this can be achieved by putting emphasis on node being modified. Douglas et al. [1996] demonstrated that perceptual features such as motion, color, sound, and size against an unchanging background, and perceptual economy (demonstrated by using simple geometric and stick figures) can be used to provide focus of attention.
 - 18) Provide pseudocode display. Saraiya et al. [2004] demonstrate that this is not useful.
 - 19) Have students present visualizations to the audience for feedback and discussion (visualization may be custom built by students or not). Grissom et al. [2003] suggested this, although it has not been tested yet.
 - 20) Engage the student actively. Byrne et al. [1996] demonstrated that animations aid learning of "procedural" knowledge by encouraging learners to predict algorithm behavior and Kann et al. [1997] reported that programming the respective algorithm can be used to engage the student.

22) Use analogies to explain the concepts. Khuri [2001] and Hanciles et al. [1997] demonstrated that this helps students learn better. Douglas et al. [1996] reported that in addition to using metaphors, the system should be flexible enough so that customized visualization reflecting cultural expectations can be created.

2.3.2. Guidelines for Increasing Adoptability by Instructors

Naps et al. [2003b] and Khuri [2001] suggest that DSV tools for instructors use can be improved by reducing the time to download, install, learn, and maintain/upgrade the tool. They also suggest that the DSV tools will be used more frequently if instructors are able to quickly and easily develop (customized) visualizations, adapt and integrate the visualizations into course materials, and teach students how to use visualizations. To help with the issue of course integration, tools must be platform independent and the visualizations produced by the tool should be consistent with the ones used in the textbooks.

2.4. CONCLUSIONS: DESIGN AND PEDAGOGICAL REQUIREMENTS

Based on the literature review, the following design and pedagogical requirements were determined.

Design requirements:

1) Minimal effort should be required to create custom viewers [Stasko 1997] [Hundhausen and Douglas 1998, 2000]. The viewers must be generated automatically. Yet, full control over customization should also be available using API-based approach.

- 2) Visualization speed should be controllable (ability to pause, play, step over) [Bergin et. al. 1996] [Rößling 2003].
- 3) Viewers should be scalable such that they are able to recognize all data structure in Java and should be available for all platforms.
- 4) Multiple and synchronized views should be made available to show different conceptual views of the same data structure since this has been shown to improve learning Khuri [2001] and Narayanan [2003, 2004].
- 5) Viewers should be consistent and tightly incorporated with an integrated development environment.

Pedagogical requirements:

- 1) Viewers should actively engage students [Stasko et al. 1993, 1997][Lawrence et al. 1994] [Byrne et al. 1996] [Kann et al. 1997]
- 2) Cognitive load of the short term memory should be reduced so that efforts can be directed to problem solving [Bergin et al. 1996] [Khuri 2001]. The correct visualization state should always be visible, and structural changes should be highlighted, and the visualization must be a similar representation of the textbook figures.
- 3) Number of tools required should be reduced by providing a single tool that serves the dual purpose of classroom demonstration and development environment (i.e., can be used for lab exercises and assignments).
- 4) Materials should be provided to seamlessly integrate viewers with existing course material.

5) Learning to use and interacting with viewers must taken only a few minutes such that instructors and quickly teach students how to use these viewers.

6) Transition from static textbook concepts to dynamic implementation should be effortless and effective [Shaffer et. al. 1996]. Animation of variables interacting with the data structure is shown to be useful to accomplish this goal. For example: when debugging a linked list, it will be useful to see a node being created and how it is inserted into a particular position in the list.

With respect to pedagogical issues, this research project will explore how the viewers can help students code faster (i.e., finish assignments faster), with greater accuracy, find and correct logical bugs. The ease of creating viewers together with ease of integration with course materials which make the jGRASP viewers easy to use by instructors and students is also illustrated.

CHAPTER 3

SURVEY DESIGN AND ANALYSIS TO INVESTIGATE DATA STRUCTURE UNDERSTANDING

In this chapter, the design and analysis of a survey that was conducted to identify data structures that are most difficult to understand conceptually and most difficult to implement using Java will be discussed. Students rated each data structure at three levels of understanding (abstract, implementation, and application) using a 5-point Likert scale. The Cochran-Mantel-Haenszel method was used to analyze associations among the different understanding levels for each data structure and to analyze the association among the different data structures for each understanding level.

3.1. SURVEY DESIGN

Two surveys and multiple interviews were conducted to understand the typical difficulties students have with introductory level data structures and algorithms course. The first survey was conducted in Fall 2004, and then it was slightly modified (questions regarding programming languages were added) and conducted again in Spring 2005 [Appendix A]. Both surveys were conducted using a paper-based questionnaire. These were distributed at beginning of the lab session for each section of COMP 2210 Fundamentals of Computing II.

The first two questions of the survey were about students' background – major, degree and year in the program. Question three asked if they felt their Java experience was appropriate for the class. Questions four and five asked students to choose the programming languages that

they can use unassisted. Question six asked them to rate data structures at three levels using the 5-point Likert response scale shown in Table 3.1. A Likert scale [Likert 1932] is a bipolar psychometric scale that is often used in questionnaires to measure attitudes, by either positive or negative responses statements. Respondents are asked to specify their level of agreement to each of a list of statements. For this survey, the respondents rated each data structure for three levels of understanding. The levels were: (a) conceptual: how easy is it to understand the basic working and concepts of the data structure? (b) implementation: how easy is it to write a program implementing the operations of the data structure? (c) application: how easy is it to use the data structure in an application?

Survey design primarily consists of two types of questions: (1) close-ended questions with a finite set of answers from which to choose, and (2) open-ended questions which do not have one definite answer [Davis 1971, Rea 1997]. The advantage of close-ended questions is that they are easy to standardize and lend themselves to statistical analysis, while the disadvantage is that they are difficult to write since the designer must consider all possible choices. All questions other than two and four were closed-ended. Question two was designed to be open-ended since there were varied majors of students in this course. Question four was also designed to be open-ended since the programming language backgrounds of students was not known.

Table 3.1: Excerpt from Question 6. Legend indicates ratings that are used to data structure understanding table

| Data Structures | Understanding Level | | |
|-----------------|---------------------|----------------|-------------|
| | Conceptual | Implementation | Application |
| List | | | |
| Stack | | | |
| Queue | | | |
| Hash Table | | | |
| Graph | | | |
| Tree | | | |
| | | | |

| Legend | | | | | |
|----------------------|-------------------------|--------------------|---|--------------------|-------------------------|
| 0 | 1 | 2 | 3 | 4 | 5 |
| not covered in class | very hard to understand | hard to understand | not too hard yet not easy to understand | easy to understand | very easy to understand |

3.2. PARTICIPANTS AND DEPLOYMENT

The survey was not compulsory, there was no time limit to complete it, and it was administered anonymously. In Fall 2004, 92 students were registered in COMP 2210, 86 surveys were distributed and a total of 77 were returned for a completion rate of 89.5%. In Spring 2005, 60 students were registered in COMP 2210, 60 surveys were distributed, and a total of 50 were returned for a completion rate of 83.3%. In both semesters, data used for analysis were randomly selected from completed surveys.

3.3. RESULTS AND DISCUSSIONS

All students in both semesters were pursuing an undergraduate degree. Figure 3.1 shows the pie chart of their year in the degree program. Ideally COMP 2210 should be taken in the sophomore

year [CS curriculum 2005], but it was seen that approximately 25.64% of the students in Fall 2004 and 28% of students in Spring 2005 were in their senior year. The high percentage of seniors was most likely a result of transfer students from community colleges as well as students who change majors sometime after their freshman year.

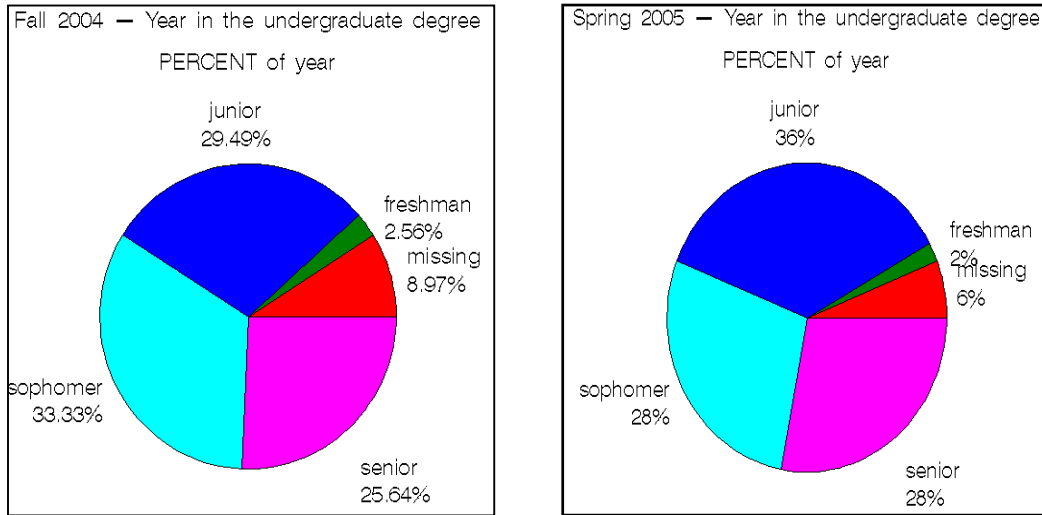


Fig. 3.1: Pie charts of year in the undergraduate degree

COMP 2210 is required to be taken by all Computer Science, Software Engineering and Wireless Engineering majors and as expected Figure 3.2 depicts an approximately equal proportion of each major in both semesters.

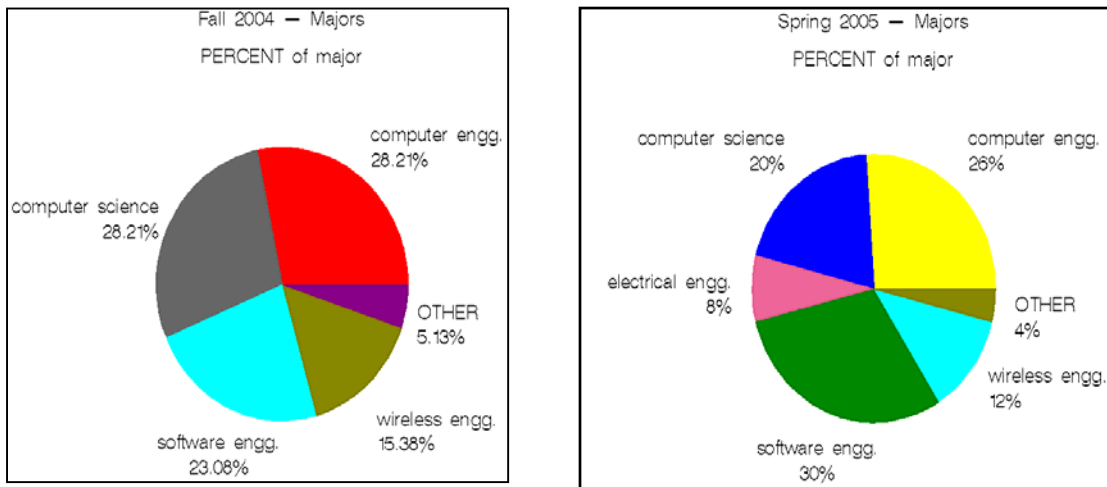


Fig. 3.2: Pie charts of COMP 2210 majors

COMP 1210 is an introduction to Java and is a pre-requisite for COMP 2210, thus as expected, Figure 3.3 shows that approximately 75% of students over both semesters felt that their level of Java proficiency was sufficient for COMP 2210.

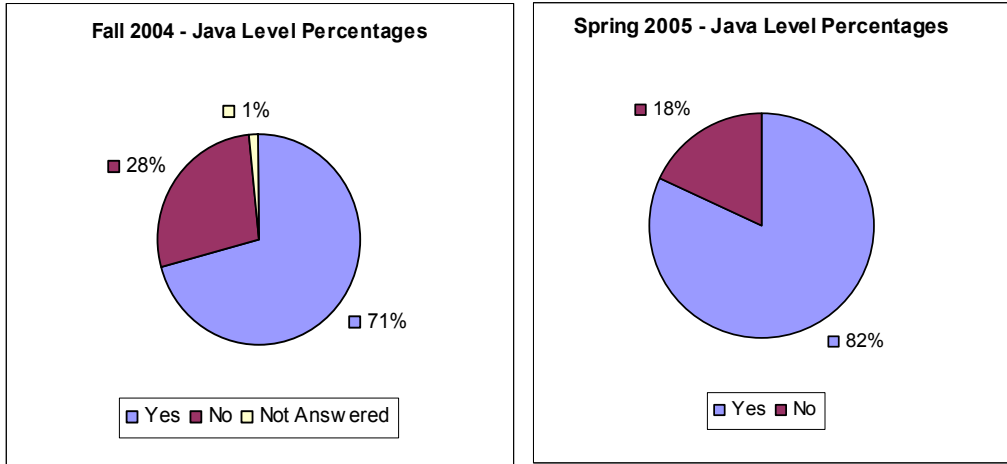


Fig. 3.3: Pie charts of Java level

Figures 3.4 and 3.5 show a stacked bar chart of the Java level appropriateness grouped by majors. It was observed, that the wireless engineering majors consistently felt that they lacked enough Java experience. Similarly, Figures 3.6 and 3.7 show a stacked bar chart of the Java level appropriateness grouped by years. It was observed that most juniors felt that their Java level was not appropriate for COMP 2210. Both of these issues need further investigation.

Fall 2004 — Java Level grouped by Major

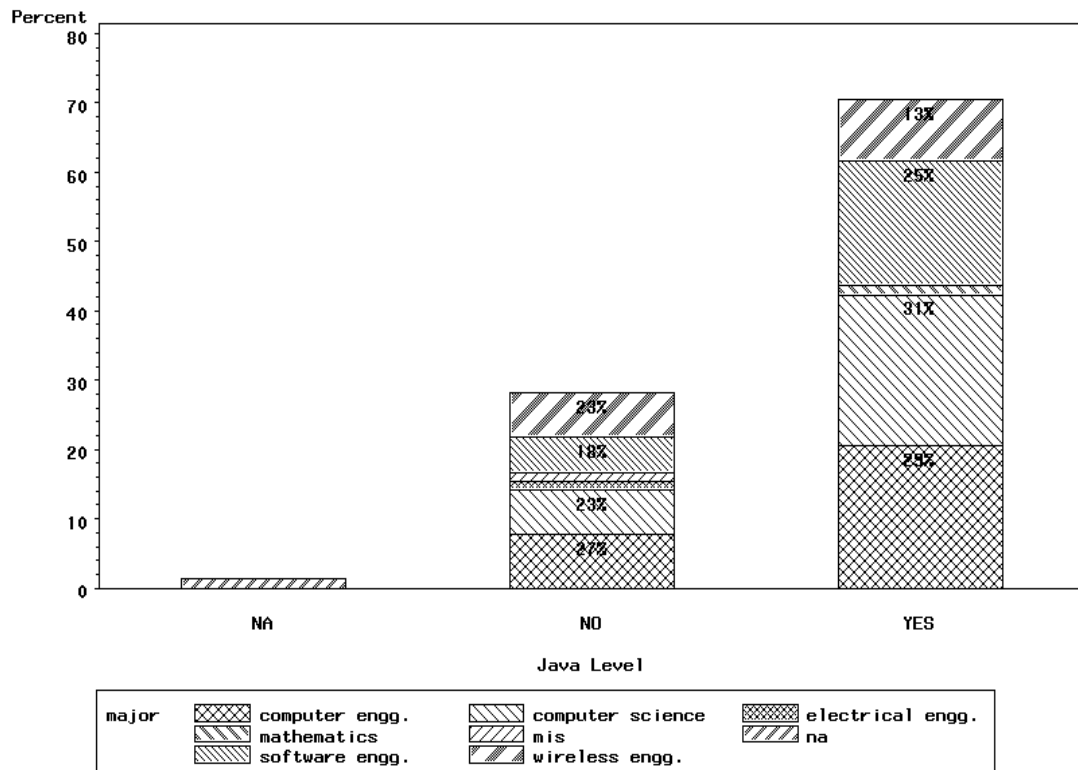


Fig. 3.4: Fall 2004 - Java Level grouped by major

Spring 2005 — Java Level grouped by Major

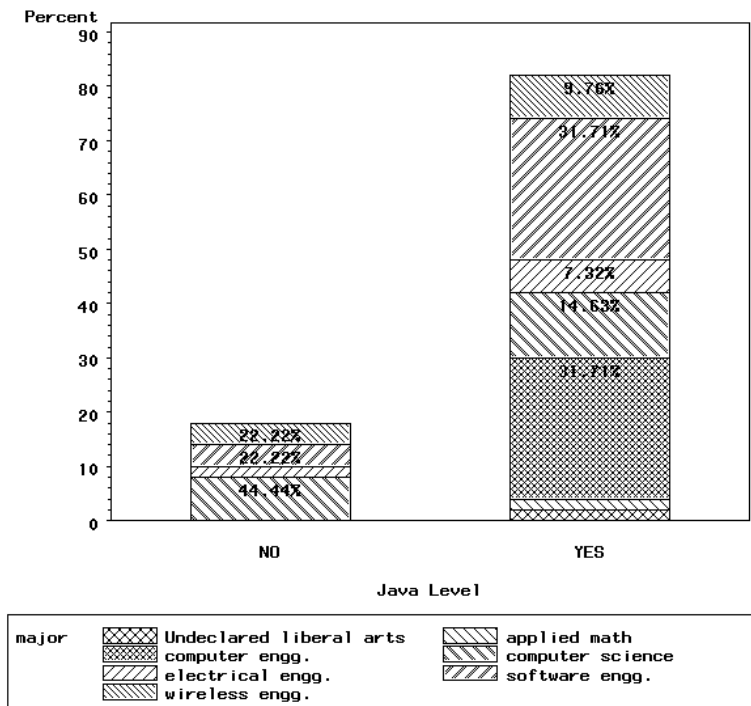


Fig. 3.5: Spring 2005 - Java Level grouped by major

Fall 2004 — Java Level grouped by Year

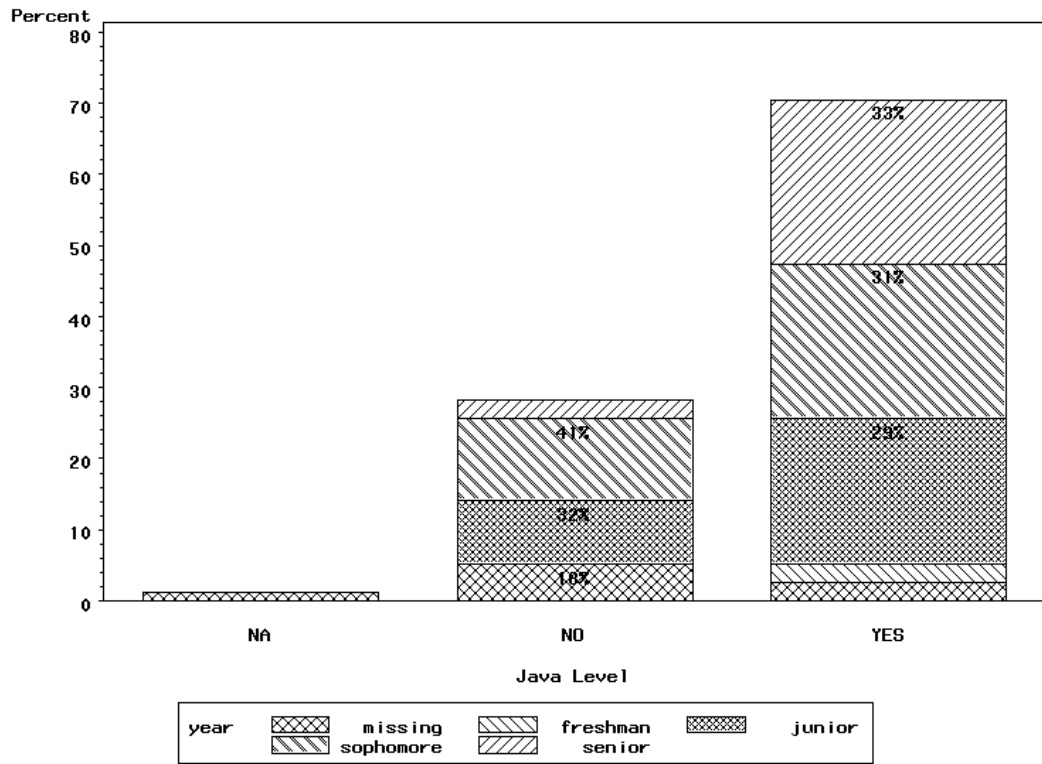


Fig. 3.6: Fall 2004 - Java level grouped by year

Spring 2005 — Java Level grouped by Year

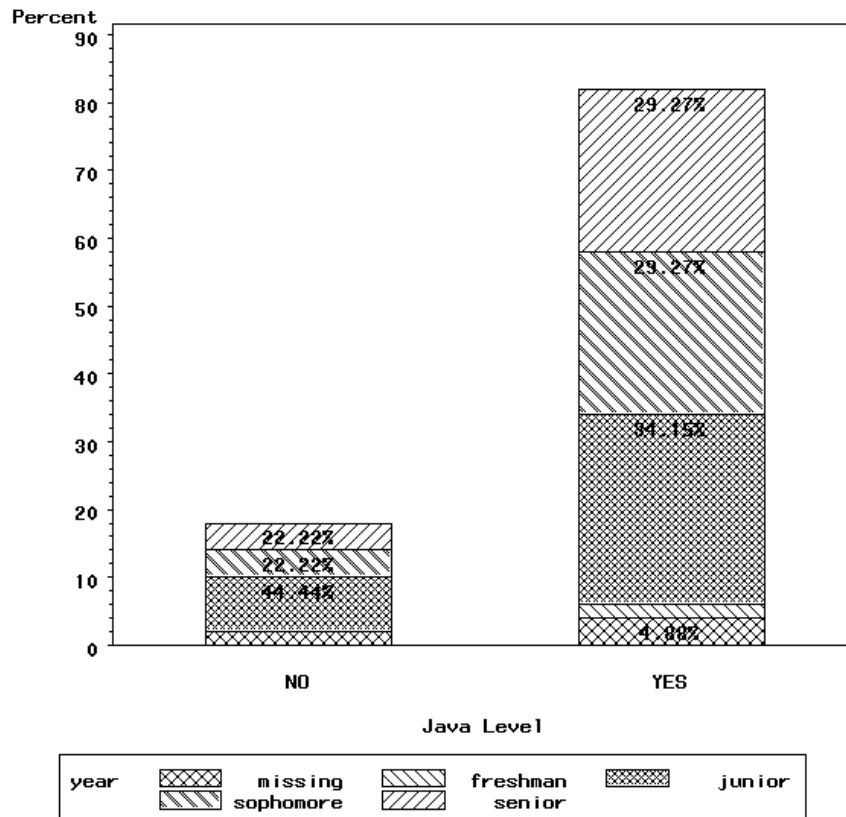


Fig. 3.7: Spring 2005 - Java level grouped by year

In questions four and five, students were asked to choose the number of languages that they can program in (except Java) without the assistant of a teaching assistant. Figure 3.8 shows the results in percentages, and HTML [48%] and C [36%] were the top two languages. Since neither one of these is object oriented, it could be a factor for students poor implementation ability.

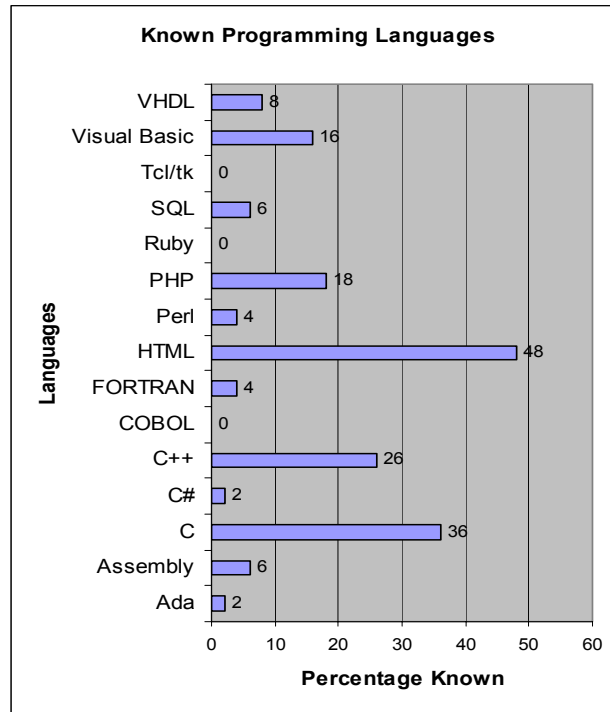


Fig. 3.8: Bar chart comparing programming language experience

Cochran-Mantel-Haenszel (CMH) statistical method was used to test associations in Question 6 in which students were asked to rate multiple data structures are three levels of understanding: conceptual, implementation, and application. CMH is a non-model-based test of the null hypothesis. The stratified analysis strategy was used for examining the association between two variables while adjusting the effects of explanatory variables. Since this approach requires minimal assumptions, it allows researchers to perform hypothesis tests on data that do not conform to the strict random sampling assumption, thus the conclusions of these analyses are generally restricted to the sample population at hand. Hence this is an ideal statistic for retrospective and observational studies [Agresti 1996, Lawal 2003, Stokes 2001, Upton 1978].

The independent variable in the CMH test for associations is the type of data structure, which is a nominal scale and the three levels of understanding are the dependent variables, each of which is an ordinal scale. The null hypothesis for the CMH statistic is that the row and column

variables are independent. The p-value was 0.0001 for all associations detailed below. There was significant association in the following two cases since the null hypothesis is rejected (the p-value is less than the alpha value of 0.05). In the first case association among the different levels of understanding for each data structure was tested. For each data structure, the association between each level was found by adjusting the effect of individual student.

```
proc freq;  
  tables student*usg*atd/cmh;  
run;
```

Fig. 3.9: SAS code used to determine association between each understanding level where usg was assigned a value "Concept"/"Impl"/"Appl", and atd was assigned the numerical rating

In Figure 3.9, the CMH option in the TABLES statement gives a stratified statistical analysis of the relationship between *usg* (it is a nominal scale consisting of one of the three labels -"Concept"/"Impl"/"Appl") and *atd* (it is an ordinal scale consisting of the rating) after controlling for student. The stratified analysis provides a way to adjust for the possible confounding effects of individual student without being forced to estimate parameters for them.

Table 3.2 lists the results of the SAS code for each level of understanding. Since rating 1 corresponded to most difficult to understand and 5 to easiest to understand, it can be seen from Table 3.2 that *Implementation* level is the toughest to understand followed by *Application* level followed by *Conceptual* level in both semesters. This result was supported by data from the course grades. Yet, it was observed in the literature review that most tools target conceptual understanding, and the tools that are available for implementation only provide very limited support.

Table 3.2: Average association among different levels of understanding for each data structure

| | Fall 2004 | Spring 2005 |
|----------------|-----------|-------------|
| Conceptual | 3.46 | 3.31 |
| Implementation | 2.42 | 2.35 |
| Application | 2.44 | 2.40 |

Next, association among the different data structures for each level of understanding was tested. For each level, association between all the data structures was found by adjusting the effect of individual students. Figure 3.10 shows the SAS code used and Table 3.3 shows the list of data structures that were found to be relatively easy and hard to understand at each level for both semesters.

```
proc freq ;
    tables student*sname*atd/cmh;
run;
```

Fig. 3.10: SAS code used to determine association between all the data structures where *sname* was assigned the name of the data structure “list-array”/ “stack-pointer”/ “tree” etc, and *atd* was assigned the numerical rating

Table 3.3: Association among the different data structures for each level of understanding

| | Fall 2004 | Spring 2005 |
|----------------|---|--|
| Conceptual | <i>Easy</i> : Array based (linked list, stack) | <i>Easy</i> : Array based (linked list, stack) |
| | <i>Hard</i> : Linked adjacency list, Minimum cost spanning tree | <i>Hard</i> : Minimum cost spanning tree, Spanning tree, Linked adjacency list |
| Implementation | <i>Easy</i> : Linked list (array, pointer) | <i>Easy</i> : Array based (linked list, stack) |
| | <i>Hard</i> : Spanning tree, Linked adjacency list | <i>Hard</i> : Minimum cost spanning tree, Spanning tree, Linked adjacency list |
| Application | <i>Easy</i> : Linked list (array, pointer) | <i>Easy</i> : Linked list (array, pointer) |
| | <i>Hard</i> : Linked adjacency list, Minimum cost spanning tree | <i>Hard</i> : Linked adjacency list, Minimum cost spanning tree |

It was also observed that data structures that were covered in less time during lectures ranked as “hardest” consistently at all levels of understanding (conceptual, implementation and application). Table 3.4 lists the data structures along with their scores. These data structures are relatively easier to understand than spanning tree, yet students felt that they had trouble learning them.

Table 3.4: Ratings of data structures covered in less time during lectures

| Data Structure | Average ratings of all three levels |
|-----------------|-------------------------------------|
| Game Tree | 1.062 |
| Parse Tree | 1.121 |
| Expression Tree | 1.452 |

3.4. CONCLUSIONS

Using the results of this survey, three relatively easy to understand data structures (array-based linked list, array-based stack and pointer-based linked list) and three relatively hard to understand data structures (minimum cost spanning tree, spanning tree and linked adjacency list) were identified. It was found that students with a Wireless Engineering major and juniors felt that their Java knowledge was not appropriate for COMP2210, and also object oriented experience seems to be lacking. Both of these issues need further investigation to determine if there is a correlation between these issues and students’ performance in COMP2210.

Based on the survey and interviews (interview results are listed in Appendix B) two important issues were discovered: (1) students struggle more with implementation than with conceptual understanding and (2) the gap between transitioning from static textbook concepts to dynamic implementation needs to be addressed. In the following chapters it will be illustrated how jGRASP viewers were designed and evaluated to solve both of the problems listed above.

CHAPTER 4

jGRASP DATA STRUCTURE VIEWERS

Based on the survey and interviews, it was established that implementation part of the introductory data structures and algorithms course is usually what the students find most difficult. Since 75-80% of students are visual learners [Felder and Silverman 1988] visualization of data structures while writing code might be useful. 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.

Starting with version 1.8, the jGRASP lightweight IDE has been extended to provide dynamic viewers for data structures classes in Java. The goal of a viewer is to provide multiple and synchronized views of a particular data structure. When a class has more than one view associated with it, multiple viewers can be opened 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 from Debug tabs from the Virtual Desktop (see Figure 4.1).

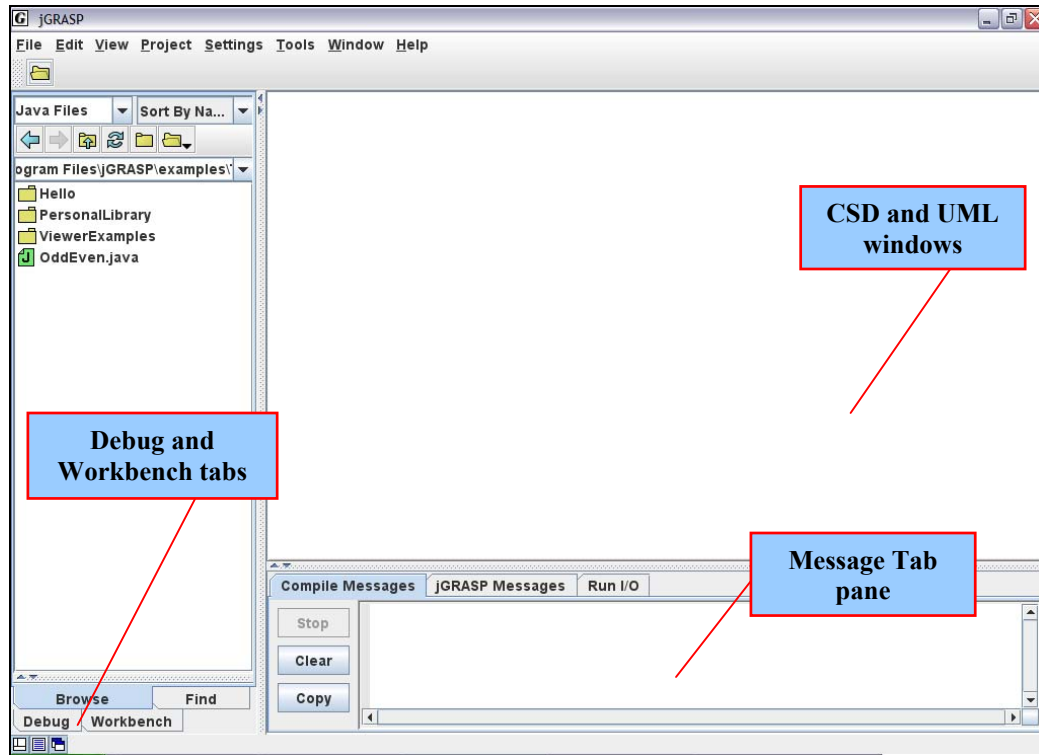


Fig. 4.1: jGRASP virtual desktop

A program must run in the debugger or from the jGRASP workbench for its data structures to be visualized since the jGRASP integrated debugger is used to collect the runtime information necessary to render the visualizations. A separate viewer can be opened for any object that is currently active on the workbench or in the debugger tab by simply dragging it from the debugger or workbench and dropping it to the jGRASP desktop. Thus, these viewers are effortless with respect to the amount of work required by the student to open and use them.

4.1. TECHNOLOGY USED TO CREATE VIEWERS

jGRASP viewers use a debugger interface called *jgrdi* (jGRASP Debugger Interface) that provides access to fields and allows methods to be called in the target process. This is similar to the Java reflection interface (“java.lang.reflect” package), but it is intended to be language-neutral, so that the same interface may be used for languages other than Java. In the case of Java,

jgrdi is a wrapper around the parts of the *jdi* (Java Debugger Interface available in the package “com.sun.jdi”) that provide these functions. It is much easier to use than either reflection or the *jdi*, and it allows the code to be much more compact and readable. However, mistakes are more likely to cause runtime errors than compile time errors. Use of the interface in a particular viewer will typically be quite minimal, and any errors will quickly be triggered when testing. Uncaught exceptions in viewer code do not cause jGRASP to crash, but are caught by jGRASP and reported in a dialog along with a call stack dump. At that point the user is given the option to disable the viewer and continue debugging or using the workbench.

All viewer code can be reloaded at any time. This allows viewers to be developed and debugged without shutting down the debugger or workbench. Previous versions of viewers that are open continue to run, so each time viewers are reloaded, old and new versions can be compared. This feature is quite useful in viewer development. For special-case viewers that need more detailed and extensive communication with the debugger, the name of the underlying debug interface i.e., “com.sun.jdi” in the case of Java and access to values in their “native” form (the form they take in that underlying interface) are available.

4.2. TYPES OF VIEWERS

jGRASP data structure viewers fall into two main categories: interface-based and structure-based. Interface-based viewers are not customizable, there is no animation available for these, and currently they are available for only the Java collections framework. Structure-based viewers show the internal structure of a data structure. They are customizable (i.e., the orientation, node shape, width, scale etc. of the viewers can be changed by the user) and animation is also available (which can be turned off if required). For example, an interface-based viewer might show a *HashMap* as a set of keys and values (see Figure 4.9), while a structure-based viewer would show the array of hash slots along with the linked list of key-value pairs at each slot, etc. (see Figure

4.10). Figures 4.2 through 4.7 show viewers for various classes in Java Collection such as ArrayList, LinkedList, TreeMap, HashMap, PriorityQueue and Vector. Clicking on any node of the data structure opens up a sub-viewer for that node. For example in Figure 4.6, clicking on the node at index 10 of the priority queue, open a sub-viewer to display the contents of the element. The element (“monkey”) is an object of the String class, which is in turn displayed as a one-dimensional array. The interface-based view is the same as shown in Figure 4.2 for the following collections classes: Stack, Vector, ArrayList, and LinkedList.

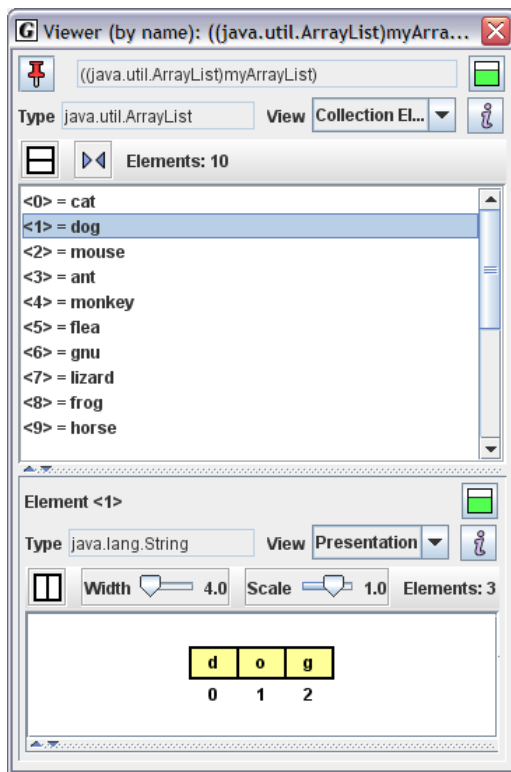


Fig. 4.2: Interface-based viewer for Stack collection class

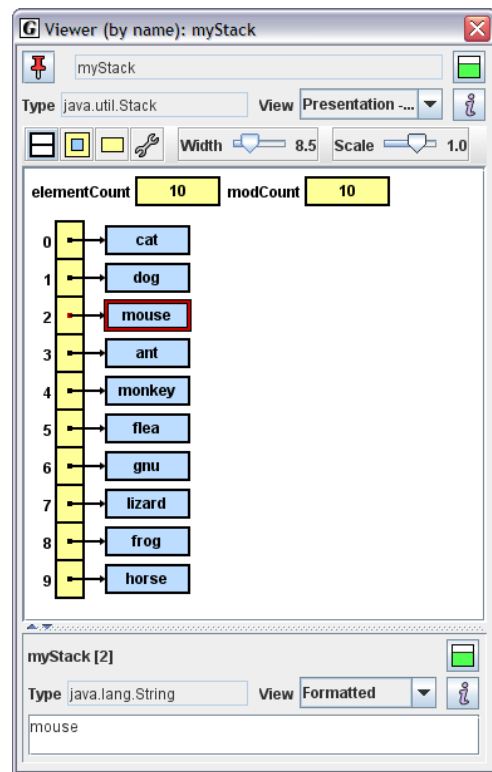


Fig. 4.3: Structure-based viewer for Stack collection class

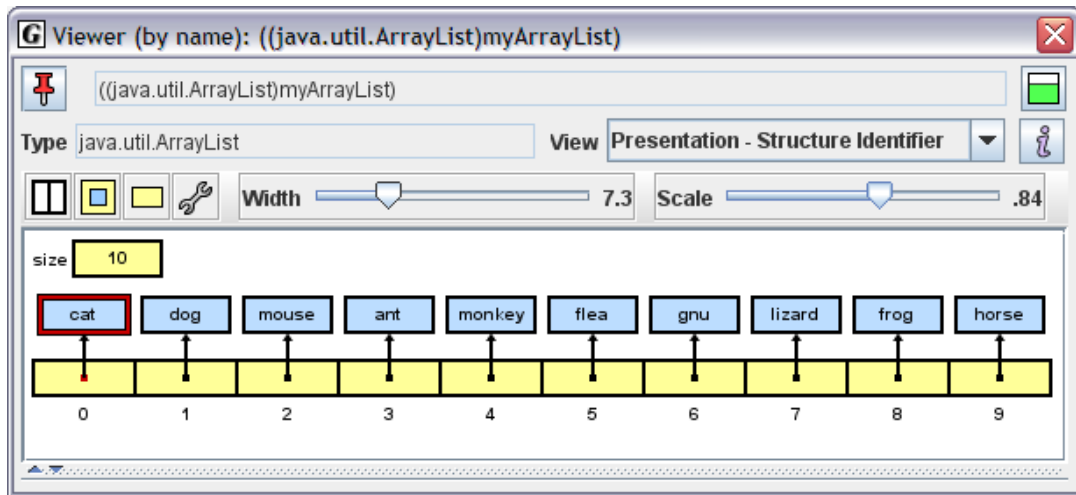


Fig. 4.4: Structure-based viewer for ArrayList collection class

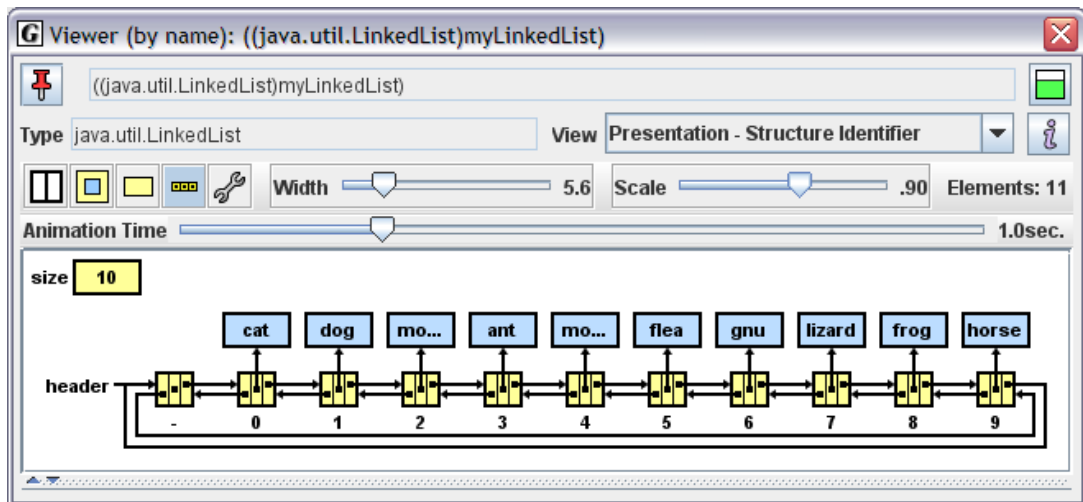


Fig. 4.5: Structure-based viewer for LinkedList collection class

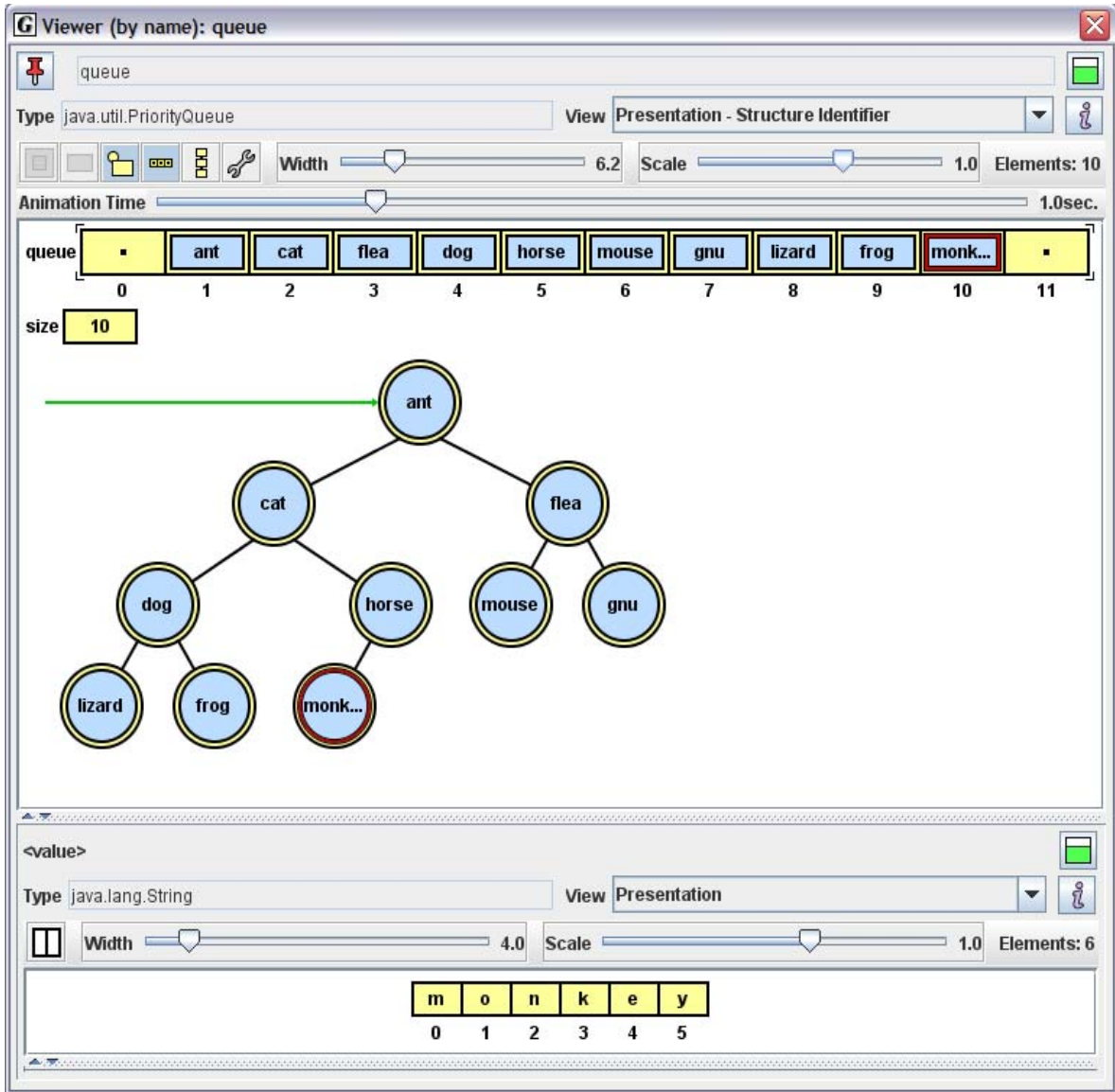


Fig. 4.6: Structure -based viewer for PriorityQueue collection class

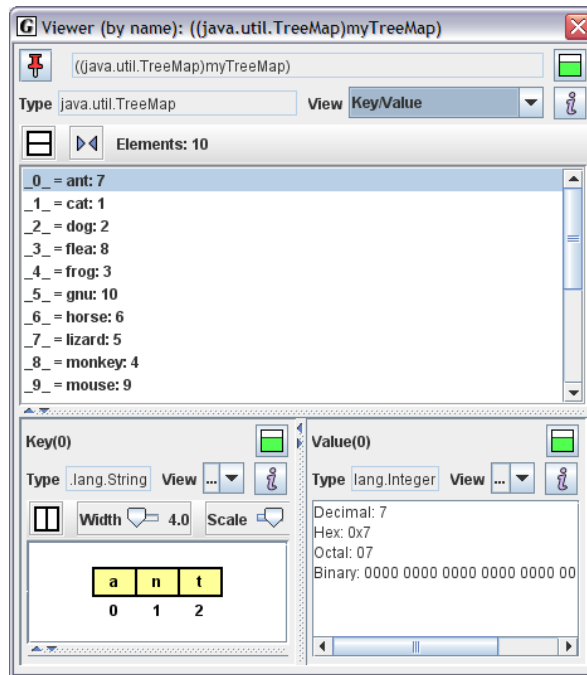


Fig 4.7: Interface-based v viewer for TreeMap collection class

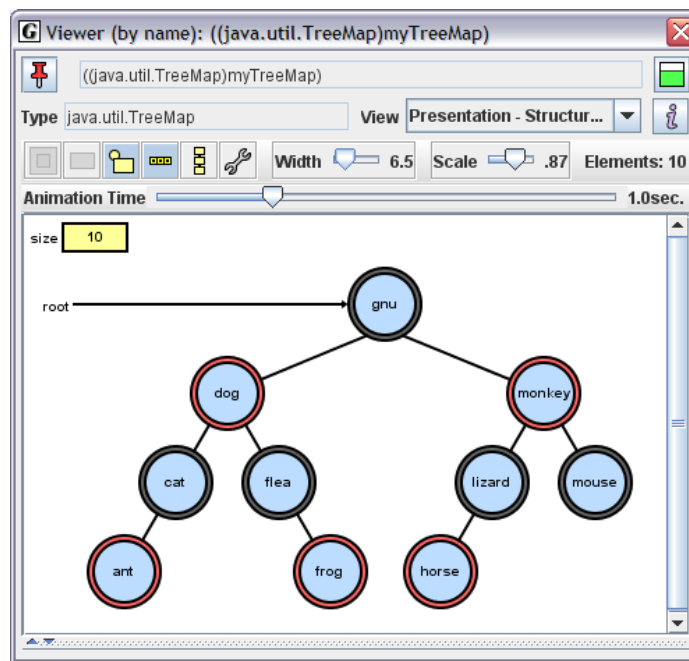


Fig. 4.8: Structure-based viewer for TreeMap collection class

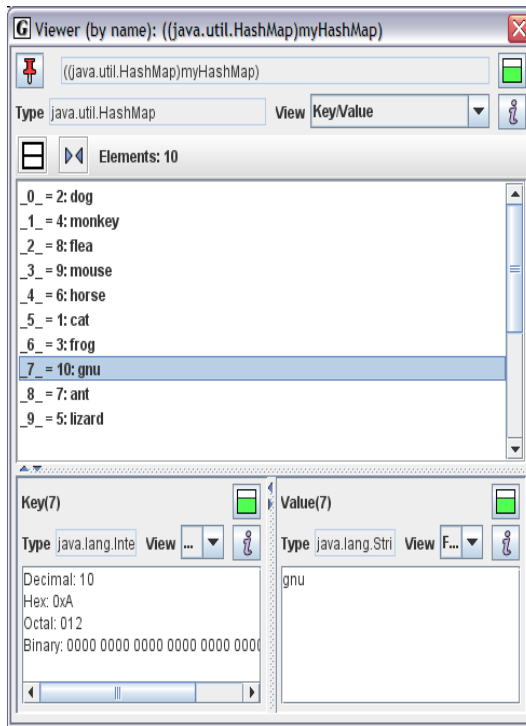


Fig. 4.9: Interface-based viewer for HashMap collection class

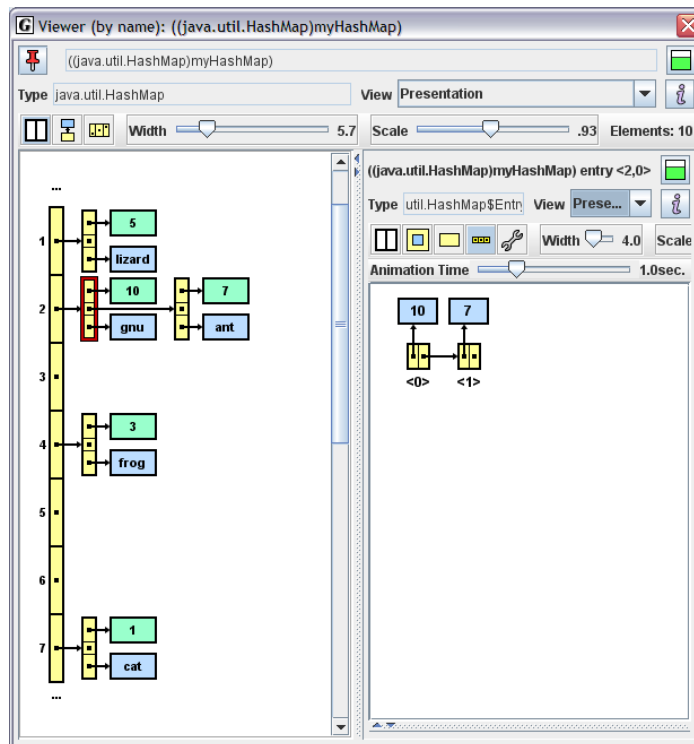


Fig.4.10: Structure-based viewer for HashMap collection class

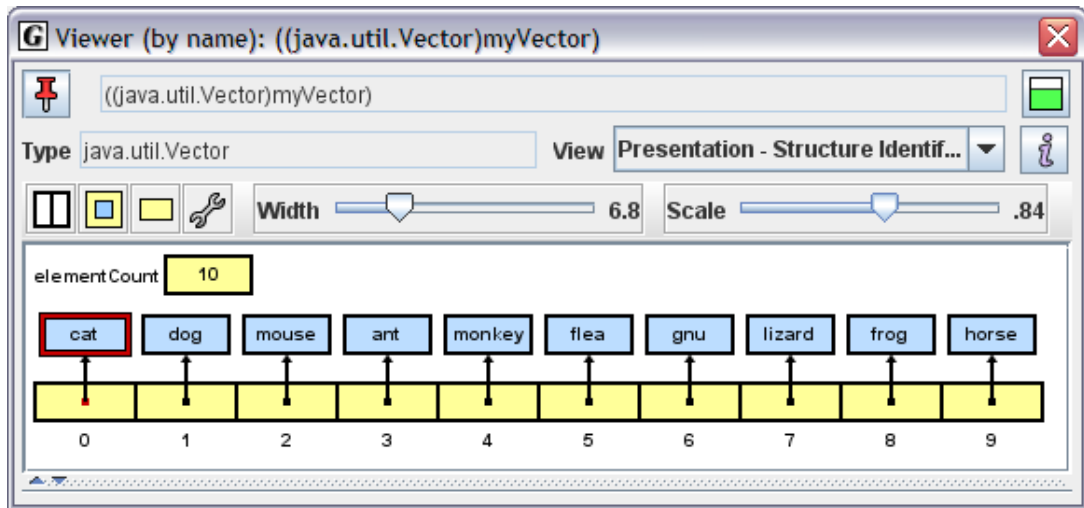


Fig. 4.11: Structure-based viewer for Vector collection class

The structure-based viewers fall into two sub-categories: non-verifying and verifying (all interface-based viewers are non-verifying). Animation can be turned on or off for this category of viewers. 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. This feature allows for the examination of large structures without excessively slowing the debugging process. Non-verifying viewers would generally be used to examine the contents of a structure in the context of an algorithm that uses it, whereas verifying viewers would be more appropriate for examining the workings of the data structure itself.

4.2.1. Animated 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 structure. 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 that occurs when the program is 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, and an "animation update" is presented at interpolated intervals to provide a smooth transition. 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.

During animation, the size allotted to a structure or local variable must be the maximum of its old and new sizes; otherwise, parts of the structure and local variable nodes and structure fragments may overlap. For example, a binary tree may go from 4 to 5 levels deep when a node is added. During animation, the tree would be given space for five levels. To allow the user to adjust to this redistribution of space, the previous node locations are displayed statically for a short time whenever the space has been increased.

4.3. TYPES OF VIEWER GENERATION

4.3.1. API Based

Visualizations for data structures are created in two steps using an API based approach. 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 4.12). The viewer will be automatically updated as the user steps through the code.

4.3.1.1. *An Example*

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 4.12 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; alternately the viewer can provide number of nodes and access to any node by index.

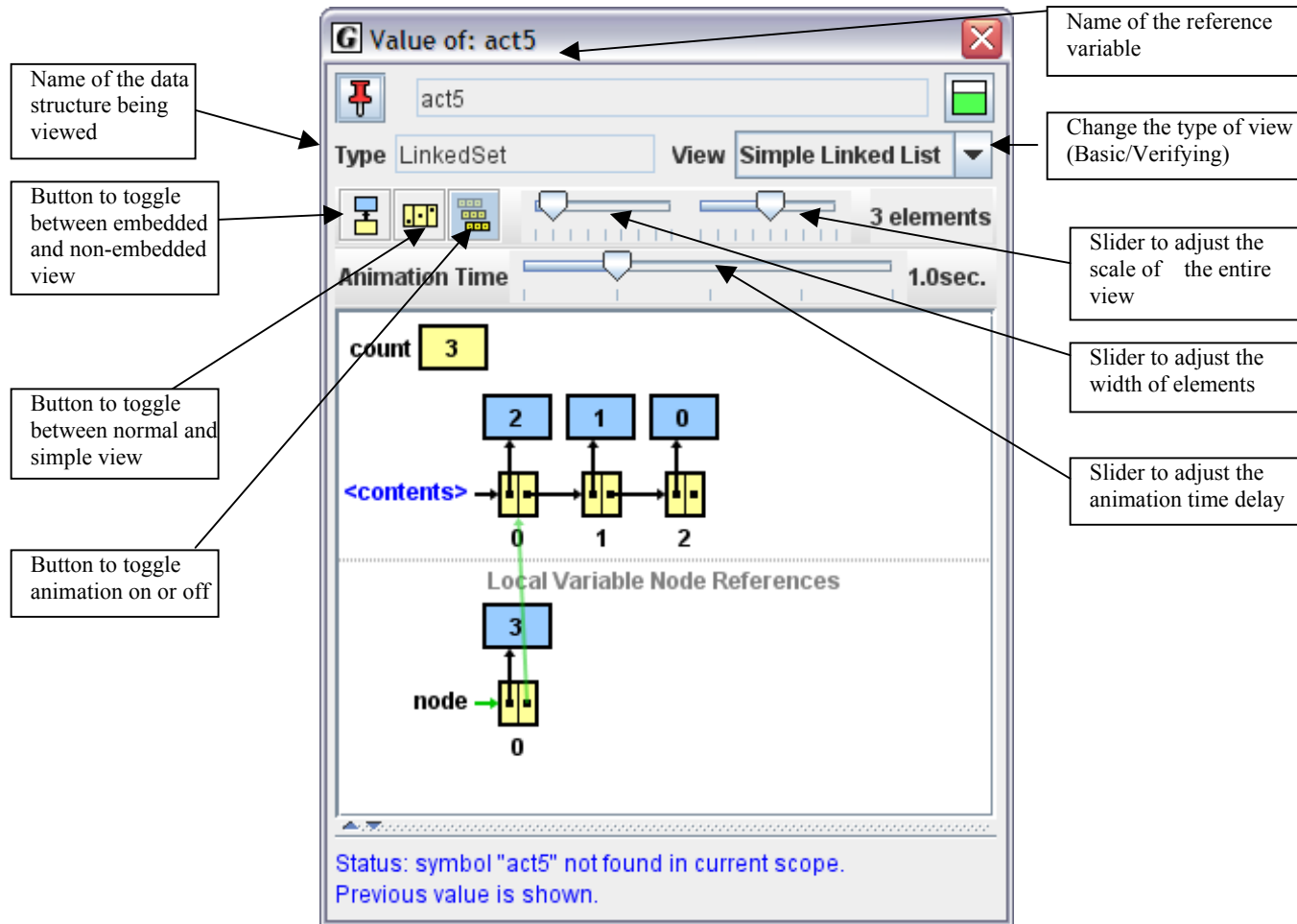


Fig. 4.12: Details of the controls of the viewer window

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

```
class LinkedList
{
    // 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;
}
```

Fig. 4.13: Code fragments of `LinkedList.java` and `LinearNode.java`

In order to create a viewer for `LinkedList.java`, only the instance variables in the following methods need be updated in the template provided with the jGRASP distribution for singly linked list. In effect, only five lines of code need to be modified to create a viewer for `LinkedList.java`.

a) `getDisplayFields()` - indicates the fields of the data structure that are to be displayed in the viewer. In the example provided, the variable `count` (displayed in Figure 4.16) has been passed to the viewer.

b) `getFirstNodeField()` - indicates the pointer (if any) to be displayed at the head of the list. In the example provided, the variable `contents` (displayed in Figure 4.16) has been passed to the viewer.

c) `getNodeType()` - indicates to the viewer the type of the nodes contained in the linked list. In the example provided, the variable `LinearNode` has been passed to the viewer.

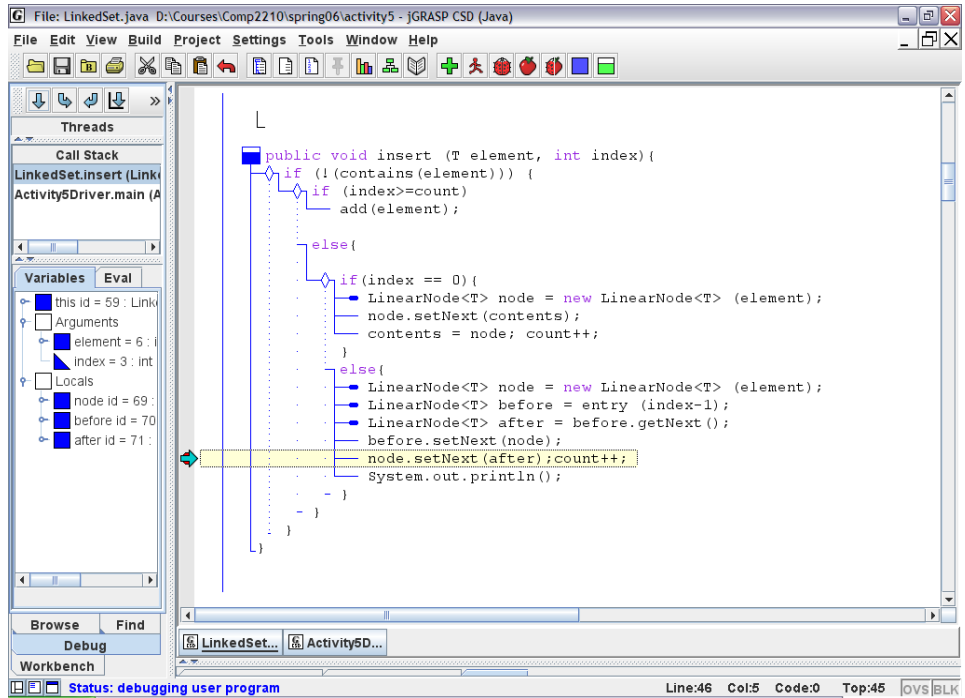


Fig. 4.14: LinkedSet.java - CSD window of jGRASP with the debugger stopped at a break

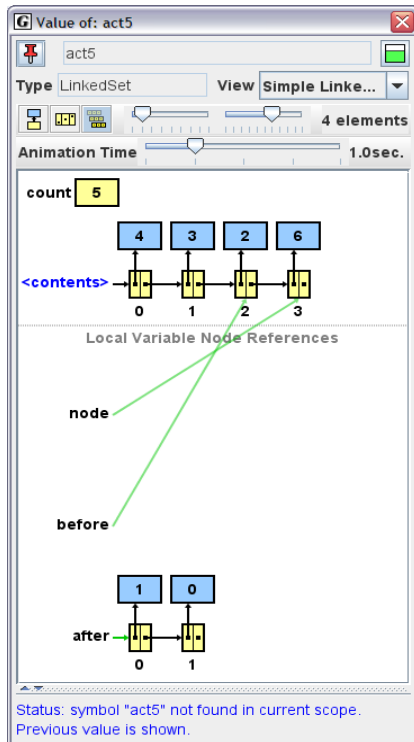


Fig. 4.15: View when the *node* with value 6 has been attached to the previous node *before* in the linked list

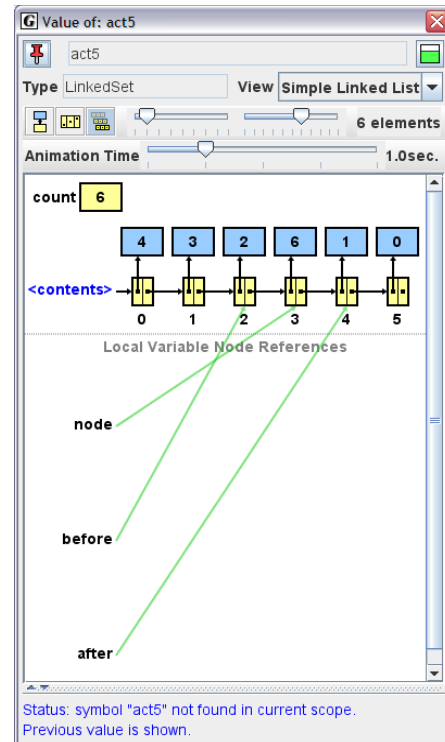


Fig. 4.16: View after the next pointer of the new node is set to point to the rest of the list (pointed to by *after*). The remaining list slides up from the local space to the main structure

d) *getNext()* - indicates to the viewer a path to the next node in the linked list. In the example provided, the variable *next* has been passed to the viewer.

e) *getNodeValue()* - provides the viewer with information about accessing the value of elements in the linked list. In the example provided, the variable *element* (displayed in Figure 4.13) has been passed to the viewer.

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 4.14-4.16, a node with value 6 will be inserted in the index position 3 of the linked list. Figure 4.14 shows the insert method and the breakpoint that has been set in the debugging process. Figure 4.15 depicts the state of the object viewer for singly linked list when the node at index 2 (with an object reference before) points to the value to be inserted (with an object reference node). The rest of the list is pointed to by a reference after. Figure 4.16 shows the state after the line is executed and the next field of node is set to the rest of the list. The local variables *before*, *after*, *node* created in the insert method –can be visualized in the local space of the viewer.

4.3.2. Automatic Identification

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, it was 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, the research direction was focused on 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.

4.3.2.1. *Data Structure Identifier*

For automatic identification of the structure of a class implementing a data structure, the "Data Structure Identifier" is invoked when a viewer on an object is opened. For pointer based implementation of data structures, where the node (which contains a value or element) is an object and a pointer to the next node is an object-reference, automatic identification is done using a two-step process. In step 1, the class structure is examined and name-based heuristics are used to identify the "type" of data structure. For example, consider a class *BinaryTree*, which has a field (depicting the root of the tree) with a class type *BinaryTreeNode*. On examining the class *BinaryTreeNode*, two same-class object references called left and right (pointing to the left and the right sub-trees) are found along with an object called value (depicting the element stored in the respective tree node). This method may lead to multiple possible structures and to multiple possible mappings from a class to a particular structure. Thus, name-based heuristics are used to assign a confidence level to each candidate. For example, the structure of the *BinaryTreeNode* class is very similar to that of a doubly linked node, where left and right object references could be next and previous pointers in 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 step 2, 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.

Link-based heuristics affect the confidence level for non-empty structure instances. Since the viewer may have been opened when the structure was in the process of being modified, a small number of identification errors may occur. However, these will have little effect on the confidence level. An effect of employing this method is that a more accurate identification may be achieved for non-empty instances than for empty ones for some structures. For example, if class A implements a node of the data structure which has two self-references, and for all (or most) of the nodes $A.next.prev = A$, then it is highly likely that class A is a doubly-linked list. In contrast, if all nodes are reachable from the root and there are no cycles, then it is likely that the class is a tree.

If the confidence level of a structure mapping is significantly higher than the confidence level for other potential mappings, then will be automatically used when a viewer is first opened for a particular class. In most cases, only one mapping with a high confidence level will be found, and thus the mechanism of finding the highest confidence level during automatic identification will be transparent to the user. The result is that a suitable structural view will be displayed without any input from the user. 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 during the same jGRASP session.

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.

4.3.2.2. An Example

Consider the following code fragments (Figure 4.17) of two Java programs: a) `LinkedBinarySearchTree.java` that implements a linked binary search tree, and b) `BinaryTreeNode.java`, which is the type of node added to the tree.

```
class LinkedBinarySearchTree {
    int numItems; //number of nodes in the tree
    BinaryTreeNode root;
}
class BinaryTreeNode {
    BinaryTreeNode left; //points to the left sub-tree
    BinaryTreeNode right; //points to the left sub-tree
    Object value; //element stored at the node
}
```

Fig. 4.17: Code fragments of `LinkedBinarySearchTree.java` and `BinaryTreeNode.java`.

Figure 4.18, shows the configuration dialog box that can be accessed by clicking the “configure” icon  on the viewer window. The class structure is examined and the related fields are populated automatically. Referring to Figure 4.17 and Figure 4.18, it can be observed

that 1: is the variable name of the root of the tree, 2: is the class name of the node of the tree (which is stored in a package jgraspvex), 3-5: are the field names of the BinaryTreeNode class and clicking on 6 shows a drop down list of all the identified structural mappings (see Figure 4.19).

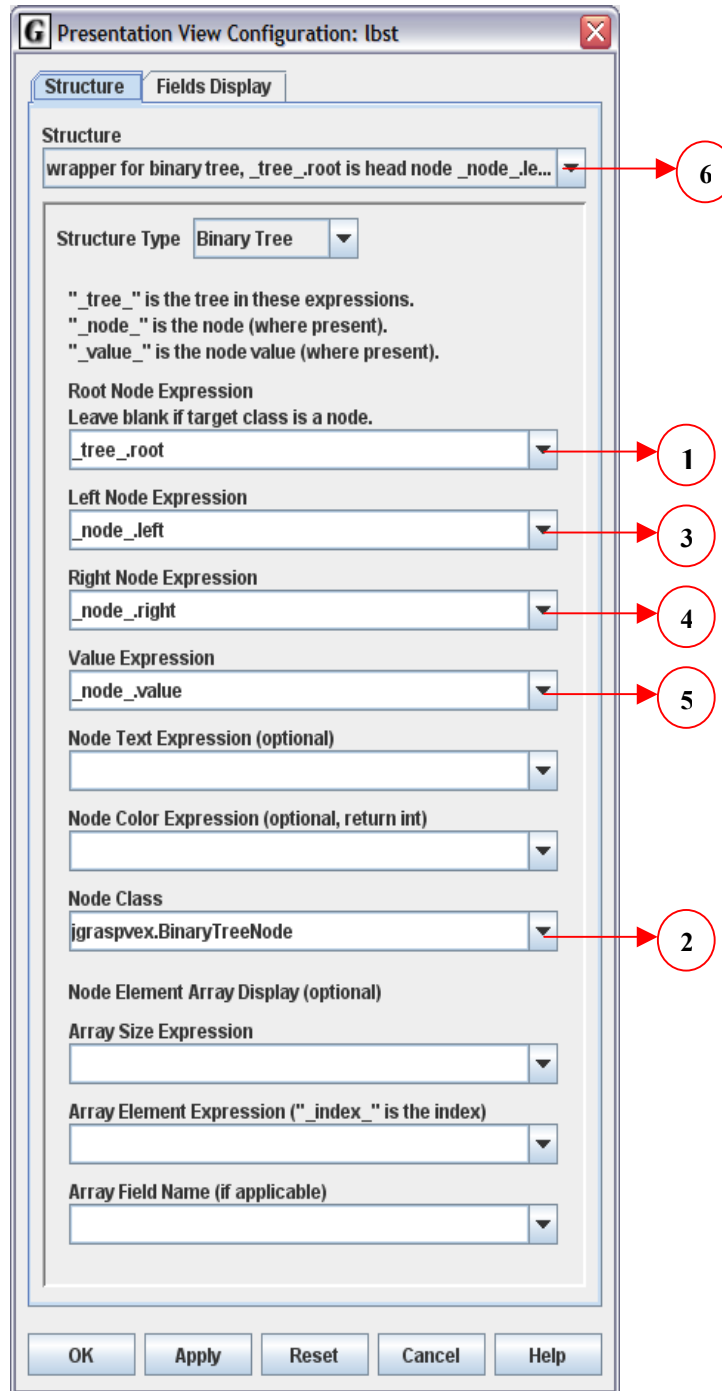


Fig. 4.18: Configuration dialog box for automatic structure identification

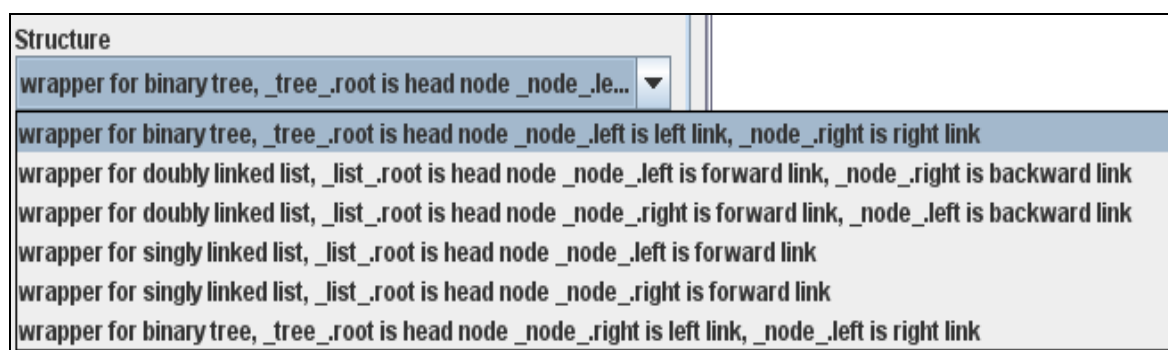


Fig. 4.19: Possible structural mappings identified for the given `LinkedBinarySearchTree` and `BinaryTreeNode` code fragments

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 through the program and into the `insert()` method, the node is added to the data structure and the viewer is updated. Figure 4.20 shows the program while stepping into the `insert()` method. Figure 4.21 shows the instance of `LinkedBinarySearchTree` containing three nodes and the node with value “8” is about to be inserted. Local object reference branch indicates the position in the tree where the new node with the value “8” will be added. When this node is added, the animation provided by the viewer shows the node “sliding” up into the tree. Figure 4.22 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 4.20. 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. In addition, seeing the entire 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.

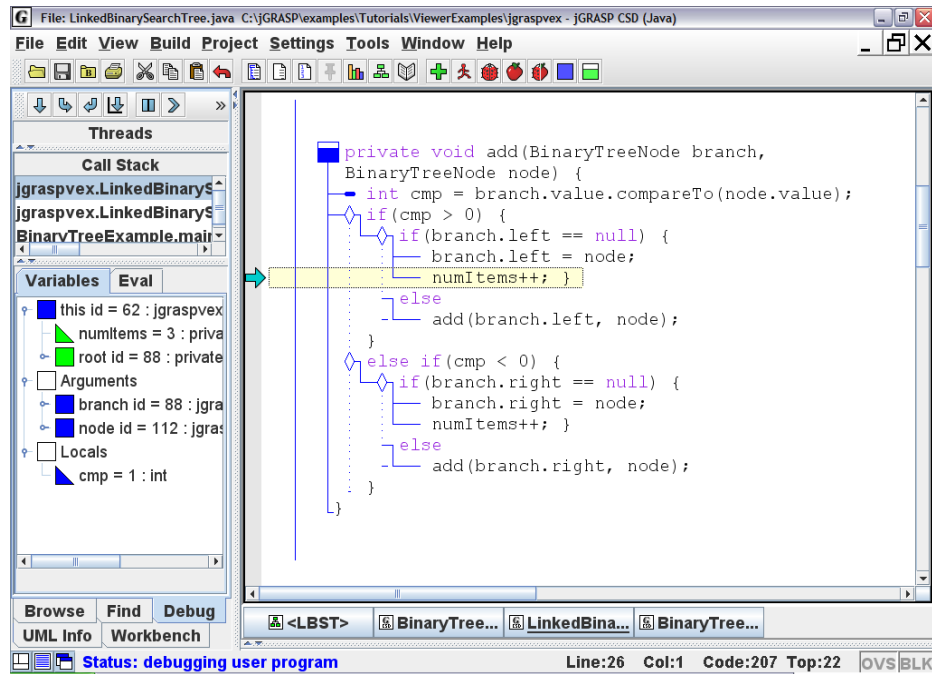


Fig. 4.20: *LinkedListBinarySearchTree* - CSD window of jGRASP with the debugger stopped at a break point

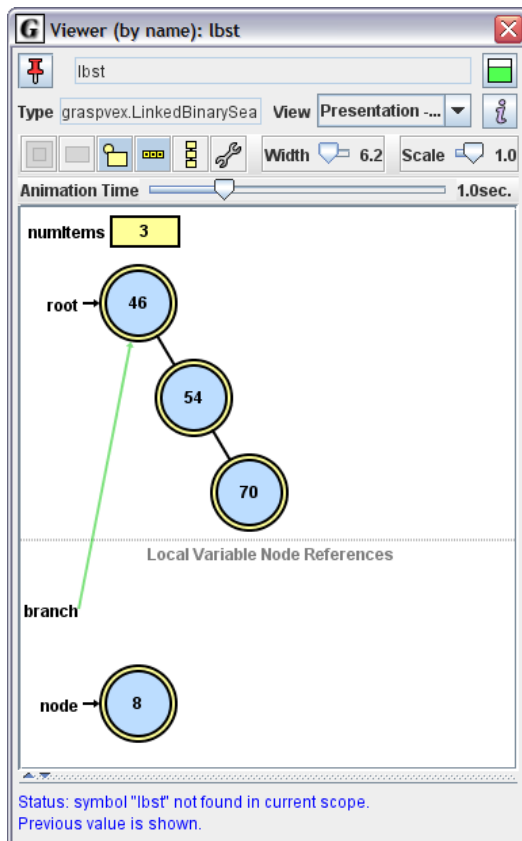


Fig. 4.21: View after local node has been created and is about to be added to the binary tree

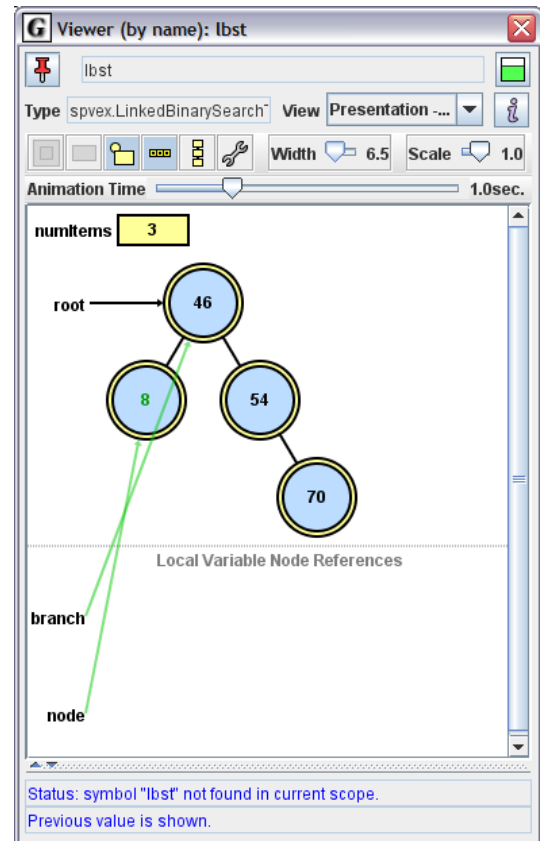


Fig. 4.22: View after the node has “moved” from the local space into the binary tree and prior to *size* being updated

CHAPTER 5

EXPERIMENTAL EVALUATION

Numerous experiments conducted in the field of visualization of data structures and algorithms were considered in the literature review [Jarc and Feldman 1998][Hundhausen et al. 2002] [Kehoe et al. 1999] [Stasko et al. 1993a] [Stasko et al. 1993b]. 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 conceptual level tool. There is yet a requirement for tools that will assist students in their transition from the understanding of concepts to their implementation. jGRASP viewers are designed to address this deficiency, and in this chapter experiments that test the effectiveness of jGRASP viewers are described.

Four controlled experiments were conducted to test the following hypotheses for a relatively easy to understand data structure (*singly linked list* using pointers in Experiments I and II) and a relatively hard to understand data structure (*linked binary search tree* using pointers in Experiments III and IV):

1. Hypothesis 1: Students will be able to code more accurately and in less time using the jGRASP data structure viewers (Experiment I and III).
2. Hypothesis 2: Students will be able to identify and correct more logical errors accurately and faster using jGRASP viewers (Experiment II and IV).

Experiment V (*min-max heap*) was conducted to test if students will be able to transition from concept to implementation faster and more accurately using jGRASP viewers for data structures that are covered only conceptually in lectures.

Experiment VI (*linked priority queue*) was conducted to test if students will be able to apply concepts faster and more accurately using jGRASP viewers for new data structures that were not covered in lectures.

5.1. EXPERIMENTAL DESIGN ISSUES

Two criteria are important when choosing subjects for controlled experiments. First, the subjects must be a close representation of the target population. jGRASP viewers are being developed primarily for students enrolled in an introductory level data structure and algorithms course. Thus 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 in all experimental test groups must be relatively uniform in regard to their programming abilities in order to minimize the variance between groups (see section 5.1.1 for details). Additionally, the following challenges were also considered while designing the repeatable experiments:

1. Integrating experiments seamlessly with the course material.
2. Organizing large subject population.
3. Scheduling experiments such that there are no conflicts with course activities.
4. Controlling hardware and software to ensure all subjects used similar apparatus.
5. Avoiding plagiarism.

Experiments were designed such that they were closely integrated with course requirements and so that they complemented the lab assignments. For example, if the experiments were conducted using singly linked lists, then project assignments were given on doubly linked lists. In Spring 2006, the students completed eight in-lab activities as a part of the COMP 2210

course. The breakdown of the activities was as follows: Activity 1 comprised of two tests which were used to create balanced test groups, Activities 2 through 7 corresponded to Experiments 1 to 6, and Activity 8 was a questionnaire to evaluate the user interface elements of the jGRASP debugger and viewers. 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 or other course procedures.

5.1.1. Subject Selection

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 this research, a selection-bias would imply that factors other than the viewers that were used in the experiments caused different outcomes for the two groups, thus balancing both groups equally was critical for reliable results.

Experiments were designed based on the between-group approach to avoid the transfer of concepts learned in early experiments to a later experiment. Typically, two aspects need to be addressed when using the between-group design. First, groups should be comparable, and second, exactly similar environment must be used to test both groups.

The groups were balanced based on two specific programming skills – the ability to detect and correct logical errors [Test 1 - Appendix C] and the ability to comprehend and trace programs [Test 2 - Appendix D]. For Test 1, common logical errors that are specific to the implementation of data structures were identified [Eisenstadt 1997, Hristova et al. 2003, Metzger 2003, Rubey 1975, Youngs 1974], and problems designed to test for each of the common logical errors (a total of 25) were created. In the second test, eight questions from the twelve in the multi-

national study of reading and tracing skills in novice programmers were chosen [Lister et al. 2004]. Questions on sorting were purposely omitted, since these concepts were not covered in lectures during this time. The following steps were taken to determine group assignments such that the groups are balanced on the basis of programming expertise:

1. Students were sorted in a list in ascending order of their combined scores on Test 1 and Test 2.
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 (using viewers).

Using the steps described above, the programming expertise of all the students were balanced thus having two comparable groups with an equal number of participants. The environment was controlled as well, since both groups had the same course instructor; the experiments were conducted in the same lab using identical machines, and all the experiments were conducted by one person.

Students in Group 1 were familiarized with the jGRASP debugger [Appendix E] and students in Group 2 were familiarized with both the debugger and jGRASP viewers [Appendix F]. It was observed that students in Group 2 took from two to three minutes to learn to open and interact with viewers.

5.1.2. Grading and Compensation

Collection of data was strictly contingent on student consent. In-lab activities were attendance based and comprised of 5% of the course grade. Our scoring of the students' work constituted a grade that was used to calculate up to three extra points on their final numeric average. For each experiment, each group was divided into four quartiles. Quartile 1 (i.e., top 25% of the students)

was awarded the bonus points, quartile 2 was awarded two bonus points, quartile 3 was awarded one bonus point, and quartile 4 (i.e., lowest 25% of the students) was awarded zero points. Using this scheme both groups were rewarded similarly regardless of the experimental treatment they received. Students were eligible for the attendance based 5% of course grade and up to 3 extra bonus points for the in-lab activities even if they decided to opt-out of data collection.

5.1.3. Data Analysis

Hotelling's T^2 statistic was used to analyze the data since the experiments were designed to have two dependent matched groups and more than one response variable. Hotelling's T^2 is a multivariate counterpart of Student's t-test which is typically performed for univariate data [Johnson and Wichern 1998]. In a t-test, differences in the mean response between two populations are studied. T^2 is used when the number of response variables are two or more, although it can be used when there is only one response variable. The null hypothesis is that the group means for all response variables are equal.

The following formula is used to calculate T^2 when it is generalized to p response variables:

$$T_{p, N_1 + N_2 - 2}^2 = \frac{N_1 N_2}{N_1 + N_2} (\bar{y}_1 - \bar{y}_2)' S_{sp}^{-1} (\bar{y}_1 - \bar{y}_2)$$

where y_1 and y_2 are the sample mean vectors of the two groups and S_{sp} is the pooled sample variance-covariance matrix. The diagonal elements of S_{sp} are the variances and the off-diagonal elements are the covariances for the p variables. N_1 is the sample size of Group 1 and N_2 is the sample size of Group 2 for p response variables. For all six experiments, tests were conducted to check the normality of the distribution, and the population was found to be normal in all cases.

SAS code used to calculate the Hotelling's T^2 statistic with two response variables is given in Appendix K and with four response variables is given in Appendix L.

5.2. EXPERIMENT I – LINKED LIST

The hypothesis for this experiment was that students will code faster and with greater accuracy using the jGRASP data structure viewers while implementing a relatively easy to learn data structure.

5.2.1. Method

5.2.1.1. Participants

Sixty-eight students enrolled in COMP2210 participated in the experiment. Participants were given extra credit in their course and were treated in accordance with the “Ethical Principles of Psychologists and Code of Conduct” [American Psychological Association 2002]. See section 5.1.2 for grading details.

5.2.1.2. Materials

The participants for this experiment were separated into two groups. Group 1 was the control group and implemented the tasks without the jGRASP viewers, and Group 2 was the treatment group and implemented the exact same tasks using the jGRASP viewers.

Students were asked to implement four basic operations for singly linked lists. The program `LinkedList.java` from the class textbook was used for this experiment [Lewis and Chase 2004]. The control group implemented all the four methods – *entry()*, *delete()*, *insert()*, and *contains()* using the jGRASP visual debugger. Details of these methods are given in Figure 5.1.

The driver program provided to Group 1 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 four methods using the jGRASP object viewers. The driver program given to Group 2 did not contain the *toString()* method, so the subjects had to use the viewers in order to see the contents of the list, also Group 2 was provided instructions on not to implement the *toString()* method.

5.2.1.3. Design and Procedures

The 68 students were split into two matched groups: 34 in Group 1, the group that used only the jGRASP debugger; and 34 in Group 2, the group that used both the jGRASP debugger and the viewers. In order to minimize the variation between the two groups, the students were matched on two programming skills – the ability to detect and correct logical errors, and the ability to comprehend and trace programs. See section 5.1.1 for details.

Before using the system, 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 machines in the lab were set up with permissions such that only the treatment group had access to the viewers.

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.

Basic Operations for a Singly Linked List

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). If the position is greater than the size of the list, then the element is added to the end of the list.

Example 1) 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) 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) 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 if 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. If the index is greater than the size of the list, then the method does not delete anything.

Example 1) 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) 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 at 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) 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) 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) 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) 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.

Fig. 5.1: Methods used for Experiment I and Experiment II

5.2.2. Results

The normal P-P plots showed that the dependent variables were normally distributed. The null hypothesis was that there is no difference in the accuracy and time taken for both groups. Out of the 34 participants in each group, 31 completed the experiment. For 31 subjects in each group, Hotelling's T^2 statistic was calculated to be 23.732. The critical value for $\alpha = 0.05$, $p=2$ (two response variables), and $n=31$ (sample size) was 4.171. P-value was calculated to be 0.000034. Since the T^2 value is much greater than the critical value, and p-value is much less than the α value, the null hypothesis can be strongly rejected. Thus, there was a statistical significant difference between the two groups.

Post-hoc MANOVA analysis indicated that visualization medium influenced only accuracy, $F(1, 60) = 12.02$, $p=0.0010$. Figure 5.2 shows that the mean time taken by the group with viewers is 109.34 minutes while the mean time taken by the group without viewers is 112.07 minutes.

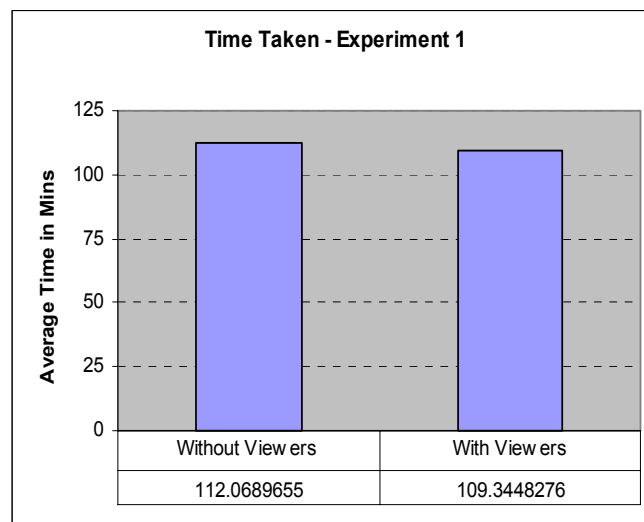


Fig. 5.2: Experiment I - comparison of mean time

Figure 5.3 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.

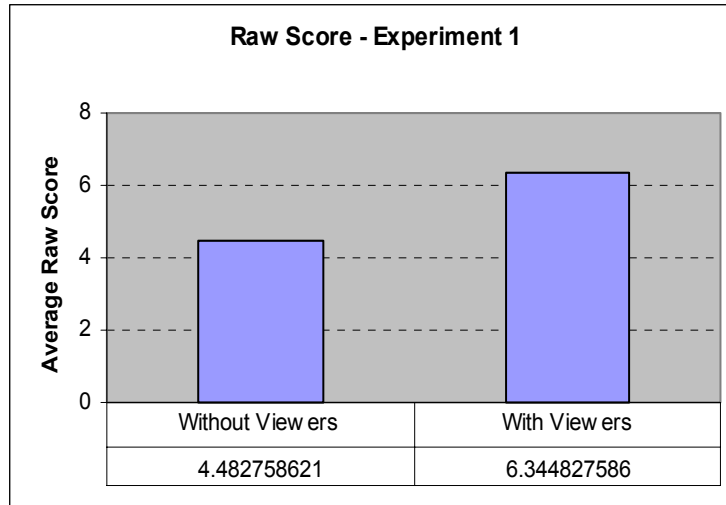


Fig. 5.3: Experiment I - comparison of mean accuracy

Table 5.1: Students that correctly implemented methods 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 5.2: Students that correctly implemented methods 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% |

Tables 5.1 and 5.2 show the breakdown of the number of students in each group that correctly implemented each of the given methods. It was observed that students in the treatment group consistently performed better than the control group for all cases. Thus, it can be concluded

that in 95% of all cases, the use of jGRASP object viewers to write programs to implement data structures resulted in increased accuracy.

5.3. EXPERIMENT II – LINKED LIST

The hypothesis for this experiment was that students will be able to detect a greater number of logical errors and correct them more accurately and in less time using jGRASP viewers while implementing a relatively easy to understand data structure.

5.3.1. Method

5.3.1.1. Participants

Sixty-eight students enrolled in COMP2210 participated in the experiment. All participants were given extra credit in their course and were treated in accordance with the “Ethical Principles of Psychologists and Code of Conduct” [American Psychological Association 2002]. See section 5.1.2 for grading details.

5.3.1.2. Materials

The participants for this experiment were separated into two groups. Group 1 was the control group and implemented the tasks without using the jGRASP viewers, and Group 2 was the treatment group and implemented the exact same tasks using the jGRASP viewers.

A Java program implementing a *singly linked list* with 10 logical errors in four methods *add()*, *insert()*, *delete()* and *contains()* was provided. Descriptions of these methods are given in Figure 5.1. The main program implementing the linked list that was provided to each group can be found in Appendix G. The details of the logical errors are as follows:

(a) *add()*: This method contained one error. The method added the new nodes to the front of the list rather than to the rear of the list as specified in the instructions (see Figure 5.1). Figures 5.4 through 5.6 depict the incorrect *add()* method. The view in Figure 5.5 shows the state of the

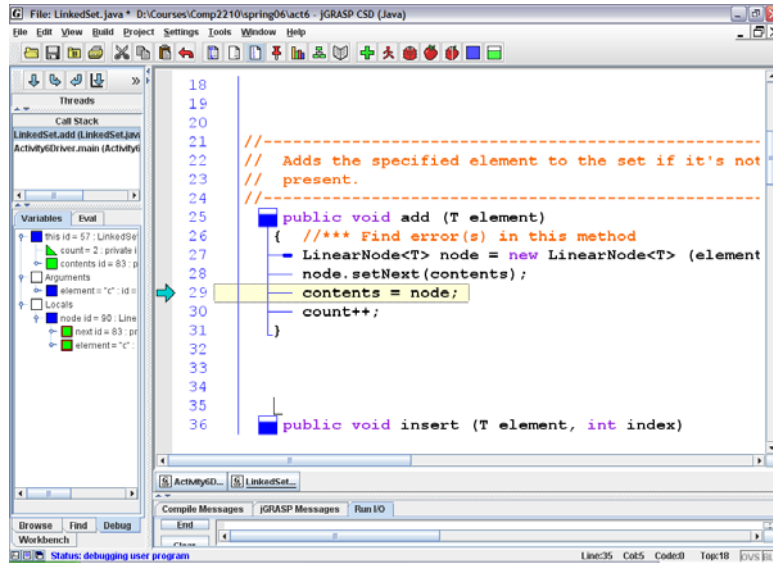


Fig. 5.4: Experiment II - CSD window of jGRASP with debugger stopped at a breakpoint in the *add()* method

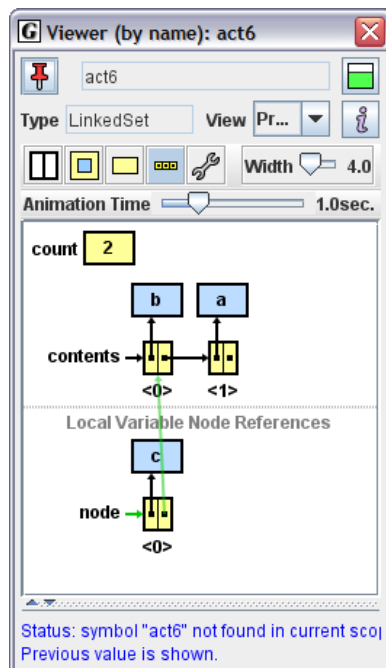


Fig. 5.5: View after two nodes have been added and the next pointer of the third node (to be added) is set to the head node

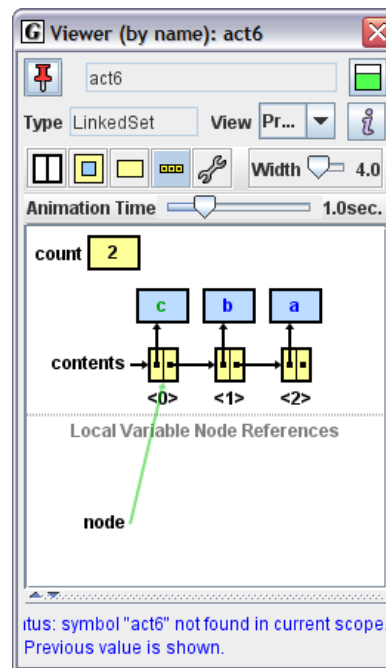


Fig. 5.6: View after head of the list has been set to the third node being added. The node has “moved” from the local space into the linked list and prior to the *count* being incremented

linked list after methods `add("a")` and `add("b")` have been successfully completed, and `add("c")` is in process. Figure 5.6 shows the state of the linked list after node with value "c" has been added to the list but the `count` variable has not been incremented.

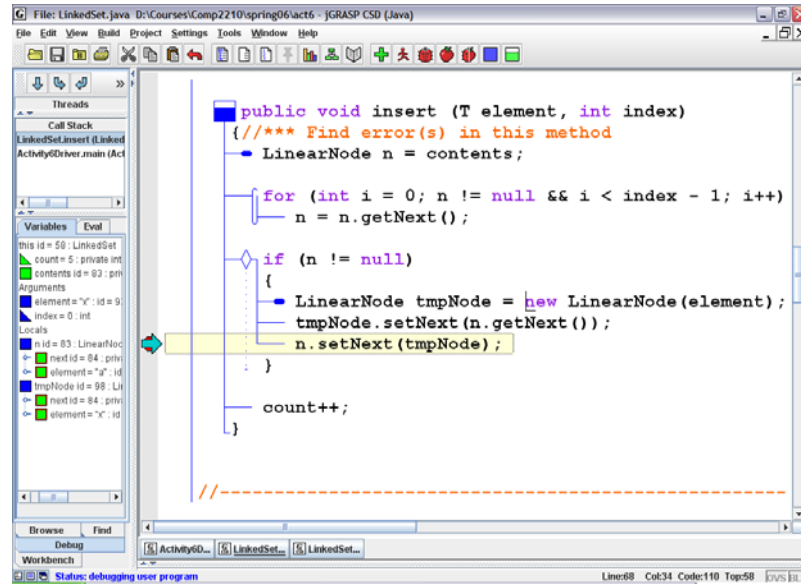


Fig. 5.7: Experiment II - CSD window with the debugger stopped at a breakpoint in the `insert()` method

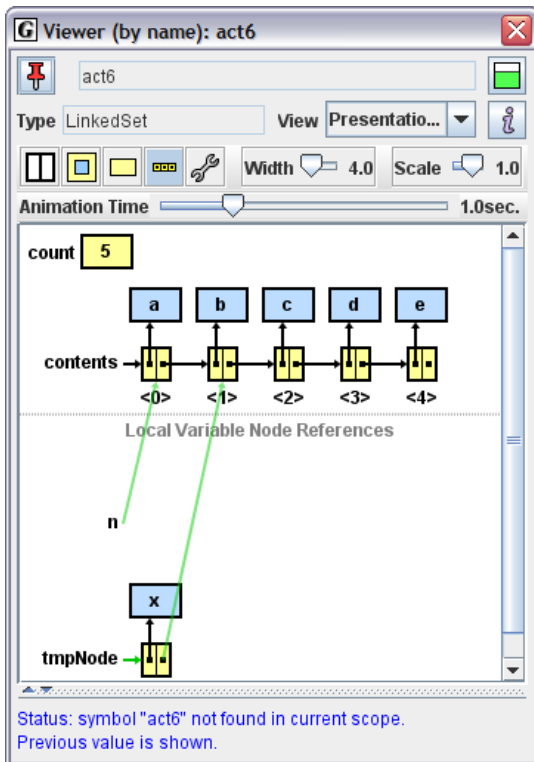


Fig. 5.8: The next pointer of the node to be inserted is set to index 1 instead of index 0.

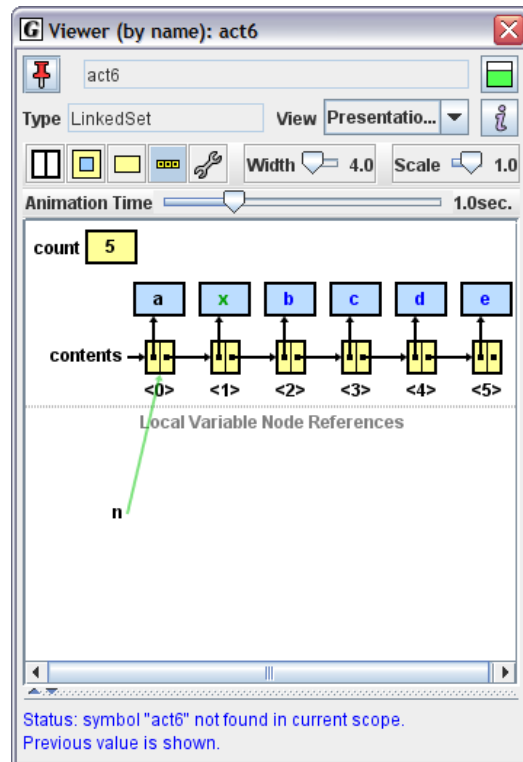


Fig. 5.9: The next pointer of the node at index 0 is set to the node `tmpNode`, and the node slides in from the local space into the linked list prior to `count` being incremented.

(b) *insert()*: This method contained two errors. When inserting a node at index “0”, the method inserted the node after index “0” instead of before it.

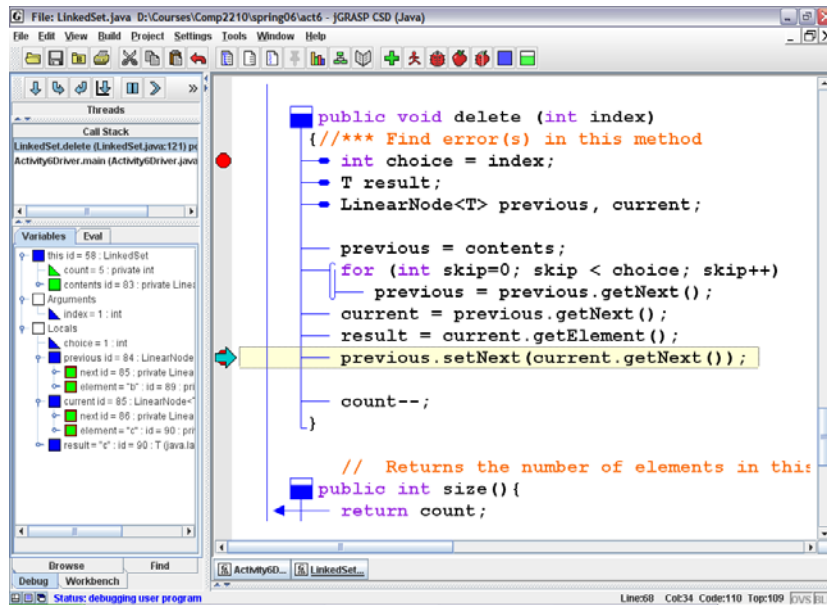


Fig. 5.10: Experiment II - Debugger breakpoint stopped in the delete() method

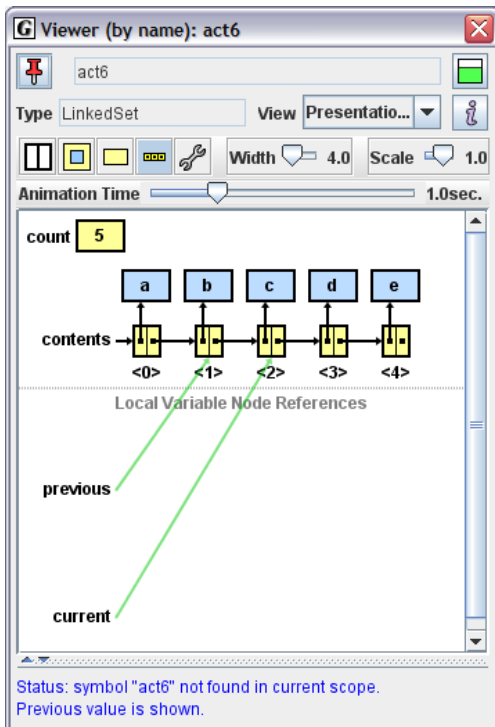


Fig. 5.11: Node at index 1 was supposed to be deleted. *current* points at the node that will be deleted.

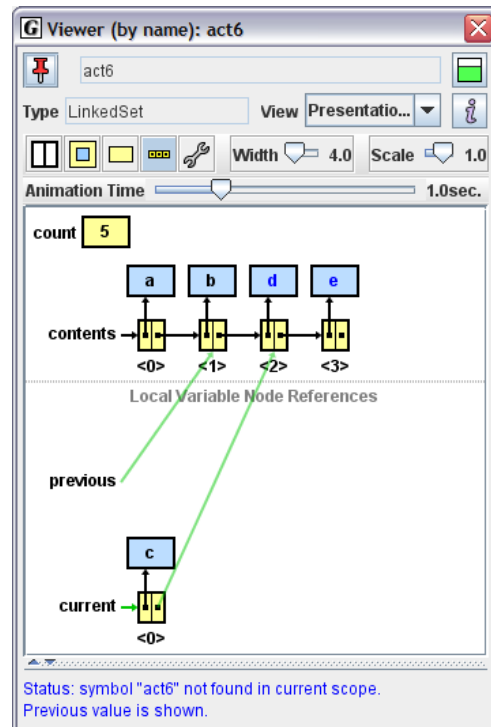


Fig. 5.12: View after the next pointer of *previous* is set to the node pointed by *current*. The node “c” slides down into the local space since it is no longer a part of the linked list. The *count* variable has not been decremented yet.

Figures 5.7 through 5.9 illustrate the scenario when the method *insert ("x", 0)* is invoked (i.e., insert node with value "x" at index 0). However, "x" is inserted at index 1 instead of index 0. If an index number that is greater than the size of the list is passed, then the number of elements in the list is incremented but the node is not added to the list.

(c) *delete()*: This method contained four errors. If the index to be deleted is "0" or the middle of the list, the method incorrectly deleted the node at index+1. If the index to be deleted is the end of the list or the index is much greater than the size of the list, then the method throws a `NullPointerException` since the node being accessed is null. Figure 5.10 to 5.12 illustrate the case where the method *delete(1)* is invoked (i.e., node at index 1 is to be deleted) . Instead of deleting element "b" at index 1, the method incorrectly removes element "c" at index 2 from the linked list.

(d) *contains()*: This method contained three errors. The method was caught in an infinite loop if the element being searched was in the middle or at the end of the loop; or if the element being searched did not exist in the list.

5.3.1.3. Design and Procedures

The 68 students were split into two matched groups: 34 in Group 1, the group that used only the jGRASP debugger; and 34 in Group 2, the group that used both the jGRASP debugger and the viewers. In order to minimize the variation between the two groups, the students were matched on two programming skills – the ability to detect and correct logical errors and the ability to comprehend and trace programs. See section 5.1.1 for details.

Before using the system, 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 machines in the lab were set up with permissions such that only the treatment group had access to the viewers.

The independent variable was the visualization medium (finding errors using jGRASP viewers vs. without viewers). The four dependent variables were: i) number of logical errors found, ii) number of logical errors accurately corrected, iii) number of new bugs introduced in the program while performing the experiment and iv) time taken to complete the experiment.

Both the groups were first required to identify and document errors. In this step, they were required to write the name of the method containing the error, and then describe how the logical error incorrectly affects the state/structure of the data structure. No points were awarded for only pointing out the statement that contained the error. 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.

5.3.2. Results

The normal P-P plots showed that the dependent variables were normally distributed. The null hypothesis was that there is no difference in the number of bugs detected, corrected, introduced, and the time taken for both groups. Out of the 34 participants in each group, 26 completed the experiment. For 26 subjects in each group, Hotelling's T^2 statistic was calculated to be 12.834. 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 0.007. Since the T^2 value is much greater than the critical value, and p-value is much less than the α value, the null hypothesis can be strongly rejected. Thus, there was a statistical difference between the two groups.

Post-hoc MANOVA analysis indicated that visualization medium influenced number of logical errors found, $F(1, 50) = 16.44$, $p=0.0002$; number of logical errors accurately corrected, $F(1, 50) = 7.76$, $p= 0.0075$; and number of new bugs introduced accuracy, $F(1, 50) = 6.41$, $p= 0.0146$. Figure 5.13 shows that the mean time taken by the group using viewers is 88.23 minutes, while the mean time taken by the group without viewers is 87.6 minutes. Figure 5.14 shows that

the group with viewers is able to detect and correct more errors. In addition, this group introduced fewer errors.

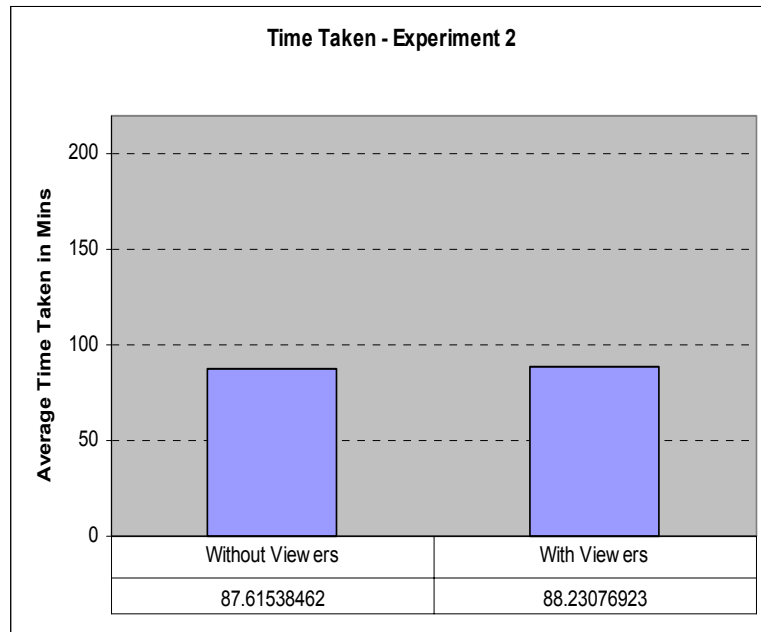


Fig. 5.13: Experiment II - comparison of mean time

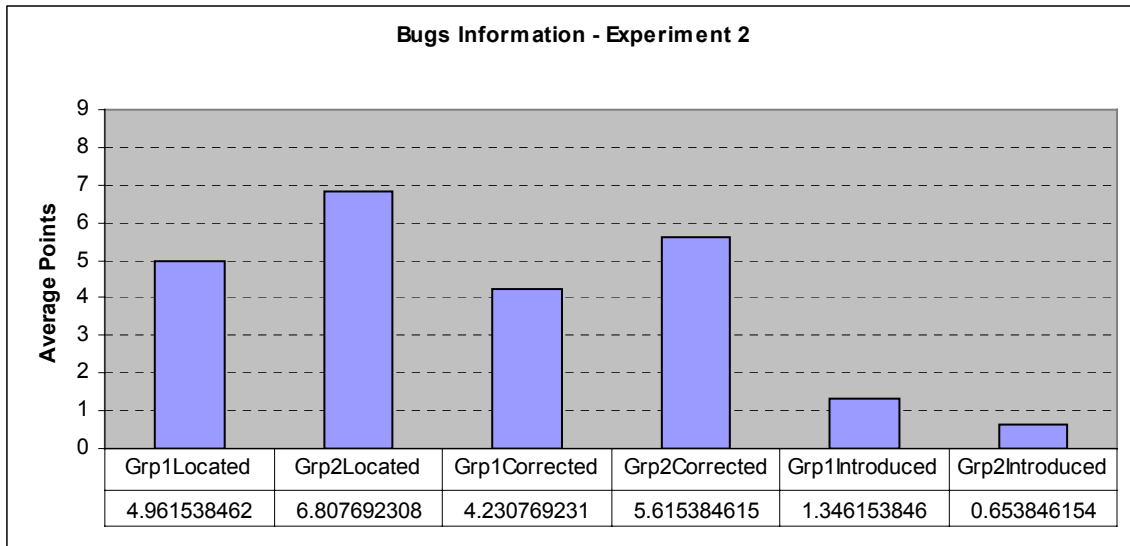


Fig. 5.14: Experiment II - comparison of mean bugs (logical errors) located, corrected and introduced

Table 5.3: Students that correctly completed methods 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 5.4: Students that correctly completed methods 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% |

Tables 5.3 and 5.4 show the breakdown of the number of students in each group that correctly implemented each of the given method. It was observed that students in the treatment group consistently performed better than the control group for all cases. Thus, it can be concluded that in 95% of all cases, the use of jGRASP object viewers to write programs to implement data structures resulted in a greater number of logical errors detected and corrected accurately while introducing fewer errors accuracy.

5.4. EXPERIMENT III – LINKED BINARY TREE

The hypothesis for this experiment was that students will code faster and with greater accuracy using the jGRASP data structure viewers while implementing a relatively hard to learn data structure.

5.4.1. Method

5.4.1.1. Participants

Sixty-eight students enrolled in COMP2210 participated in the experiment. Participants were given extra credit in their course and were treated in accordance with the “Ethical Principles of Psychologists and Code of Conduct” [American Psychological Association 2002]. See section 5.1.2 for grading details.

5.4.1.2. Materials

The participants for this experiment were separated into two groups. Group 1 was the control group and implemented the tasks without using the jGRASP viewers, and Group 2 was the treatment group and implemented the exact same tasks using the jGRASP viewers.

Students were asked to implement a basic traversal operation for linked binary search trees. The program `LinkedBinarySearchTree.java` from the class textbook was used for this experiment [Lewis and Chase 2004]. Details of this method are given in Figure 5.15. 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 the algorithm for `levelOrder()` traversal required three different data structures, with three viewers (for `LinkedBinaryTree`, `LinkedQueue`

and `ArrayUnorderedList`) were provided to the students. 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.

| | |
|---|--|
| <pre>1) ArrayUnorderedList<T> levelorder (BinaryTreeNode<T> root) {}</pre> <p>The following algorithm was provided:</p> <ol style="list-style-type: none"> 1. Create a queue called nodes (using class <code>LinkedList</code>) 2. Create an unordered list called results (using class <code>ArrayUnorderedList</code>) 3. Enqueue the "value" of root onto the nodes queue using method <code>enqueue()</code> 4. While the nodes queue is not empty <ol style="list-style-type: none"> 4a. Dequeue the first element from the queue using method <code>dequeue()</code> 4b. If that element is not null <ol style="list-style-type: none"> 4b1) Add that element to the rear of the results list 4b2) Enqueue the children (if any) of the element on the nodes queue, <ul style="list-style-type: none"> - use the <code>find()</code> method to get the reference of the element - then use the left and right instance variables to get to the children 4c. Else <ol style="list-style-type: none"> 4c1) Add null on the result list 5. Return the results unordered list | |
| <pre> graph TD head --> 5((5)) 5 --- 2((2)) 5 --- 8((8)) 2 --- 3((3)) 8 --- 9((9)) 9 --- 90((90)) 90 --- 13((13)) </pre> | <p>Level order should return the elements in this order -></p> <p>5 2 8 3 9 90 13</p> |

Fig. 5.15: Methods used in Experiment III

5.4.1.3. Design and Procedures

The 68 students were split into two matched groups: 34 in Group 1, the group that used only the jGRASP debugger; and 34 in Group 2, the group that used both the jGRASP debugger

and the viewers. In order to minimize the variation between the two groups, the students were matched on two programming skills – the ability to detect and correct logical errors and the ability to comprehend and trace programs. See section 5.1.1 for details.

Before using the system, 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 machines in the lab were set up with permissions such that only the treatment group had access to the viewers.

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.

5.4.2. Results

The normal P-P plots showed that the dependent variables were normally distributed. 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 (see Figure 5.16). 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 (see Figure 5.16).

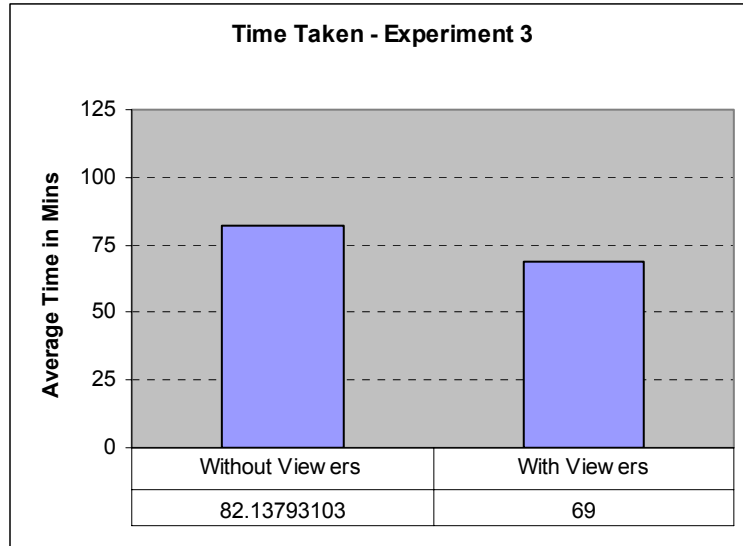


Fig. 5.16: Experiment III - average time taken by the treatment group (with viewers) and the control group (without viewers)

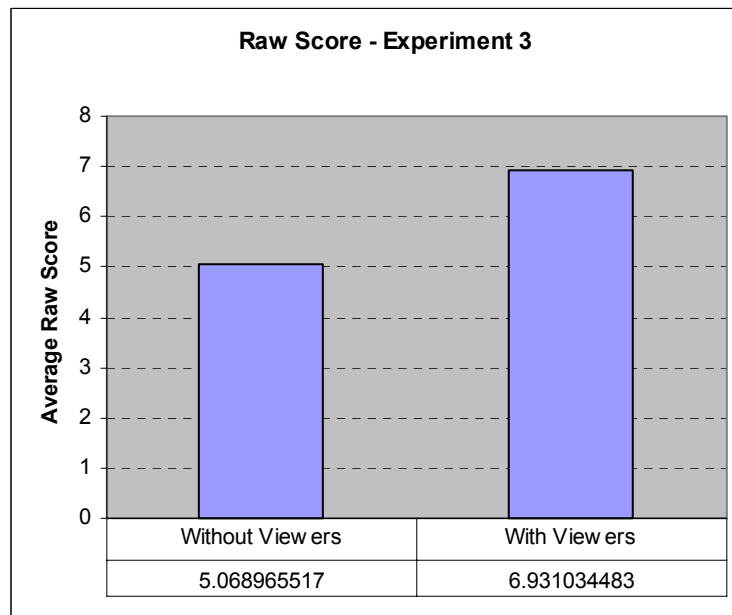


Fig. 5.17: Experiment III - average accuracy of the treatment group (with viewers) and the control group (without viewers)

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 0.0000721. Since the T^2 value is much greater than the critical

value, and p-value is much less than the α value, the null hypothesis can be strongly rejected, thus there was a statistically significant difference between the two groups.

Post-hoc MANOVA analysis indicated that visualization medium influenced both time, $F(1, 60) = 4.71, p=0.0339$ and accuracy $F(1, 60) = 20.33, p= <.0001$. It can be concluded that in 95% of all cases, the use of jGRASP object viewers to write programs to implement data structures resulted in increased accuracy and reduction in time.

5.5. EXPERIMENT IV – LINKED BINARY TREE

The hypothesis for this experiment was that students are able to detect and correct logical bugs more accurately and in less time when using jGRASP viewers while implementing a relatively hard to understand data structure.

5.5.1. Method

5.5.1.1. Participants

Sixty-eight students enrolled in COMP2210 participated in the experiment. Participants were given extra credit in their course and were treated in accordance with the “Ethical Principles of Psychologists and Code of Conduct” [American Psychological Association 2002]. See section 5.1.2 for grading details.

5.5.1.2. Materials

The participants for this experiment were separated into two groups. Group 1 was the control group and implemented the tasks without using the jGRASP viewers, and Group 2 was the treatment group and implemented the exact same tasks using the jGRASP viewers.

A Java program implementing linked binary search tree with five logical errors, one in each of the following methods *addElement()*, *findAgain()*, *removeElement()*, *inOrder()* and *postOrder()* was provided. The descriptions of these methods are given in Figure 5.18. The main program *LinkedBinarySearchTree.java* provided to each group can be found in Appendix H. The details of the logical errors are as follows:

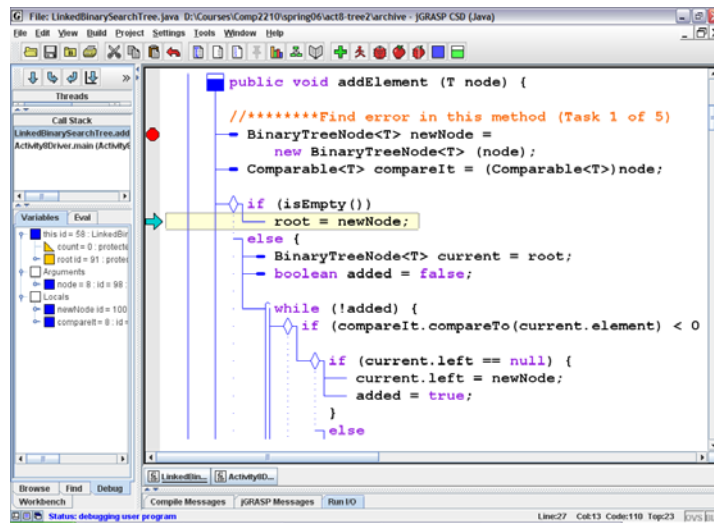


Fig. 5.19: Experiment IV: Debugger stopped at a breakpoint in the `addElement()` method

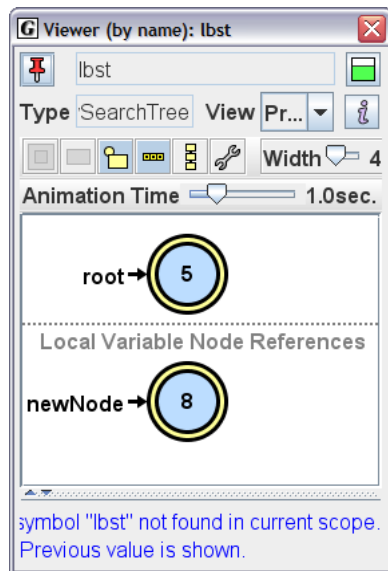


Fig. 5.20: View after local node with value 8 has been created but not yet added to the tree

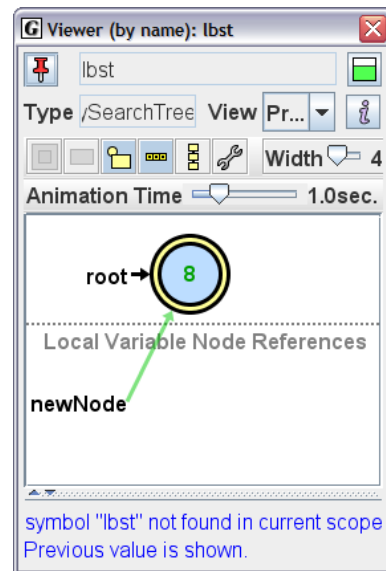


Fig. 5.21: View after `newNode` is added to the tree, and is incorrectly set as `root`

a) *addElement()*: The first node was added correctly, there on when a node was added, it simply replaced the root node and all the previous nodes were lost, thus the left and the right sub-trees of the root did not grow.

The problem was that the count variable remained zero and was not incremented appropriately, thus the *isEmpty()* method always returned true. Figures 5.19 through 5.21 illustrate this error. In Figure 5.29, node with value “5” has already been added and is set as the root of the tree. A temporary node with value “8” is created in the local space and is pointed to by the reference *newNode*. Since the *count* variable is not being incremented, *isEmpty()* method returned true, thus it is seen in Figure 5.19 that the *if* statement was executed as true. In Figure 5.21 the *newNode* is set as the root of the tree and the previous node reference is lost.

b) *findAgain()*: This method kept searching down the right sub-tree, so if the node to be searched had an ancestor which belonged to the left sub-tree or if the target node itself was a left child, then the method would not be able to find it. The method worked partially: if the node belonged to the right sub-tree then the method would return a reference to it.

c) *removeElement()*: This method did not delete a node even if it existed in the tree. The problem was that the algorithm for traversing the tree was incorrectly implemented if the target node was lower than the second level of the tree.

d) *inOrder()*: The in order traversal should travel down the left sub-tree, visit the node, and then travel down the right sub-tree. The method was implemented incorrectly since it was traversing the right sub-tree, the node and then the left sub-tree.

e) *postOrder()*: The post order traversal should travel down the left sub-tree, then travel down the right sub-tree, and then visit the node. The method was implemented incorrectly since it was traversing the right sub-tree, the left sub-tree, and then the node.


```
1) T removeElement (T targetElement) {}
   Removes the first element that matches the specified target element from the binary
   search tree and returns a reference to it.

2) void addElement (T element) {}
   Adds the specified object to the binary search tree in the appropriate position
   according to its key value. Note that equal elements are added to the right.

3) BinaryTreeNode<T> find (T targetElement) {}
   Returns a reference to the specified target element if it is found in the binary tree.

4) void inOrder (BinaryTreeNode<T> node,
   ArrayUnorderedList<T> templist) {}
   Performs a recursive inorder traversal.

5) void postOrder (BinaryTreeNode<T> node,
   ArrayUnorderedList<T> templist) {}
   Performs a recursive postorder traversal.
```

Fig. 5.18: Methods used in Experiment IV

5.5.1.3. Design and Procedures

The 68 students were split into two matched groups: 34 in Group 1, the group that used only the jGRASP debugger; and 34 in Group 2, the group that used both the jGRASP debugger and the viewers. In order to minimize the variation between the two groups, the students were matched on two programming skills – the ability to detect and correct logical errors and the ability to comprehend and trace programs. See section 5.1.1 for details.

Before using the system, 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 machines in the lab were set up with permissions such that only the treatment group had access to the viewers.

The independent variable was the visualization medium (finding errors using jGRASP viewers vs. without viewers). The four dependent variables were: i) number of logical errors

found, ii) number of logical errors accurately corrected, iii) number of new bugs introduced in the program while performing the experiment and iv) time taken to complete the experiment.

Both the groups were first required to identify and document errors. In this step, they were required to write the name of the method containing the error, and then describe how the logical error incorrectly affects the state/structure of the data structure. No points were awarded for only pointing out the statement that contained the logical error. 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.

5.5.2. Results

The normal P-P plots showed that the dependent variables were normally distributed. 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 (Figure 5.22). 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 (Figure 5.23).

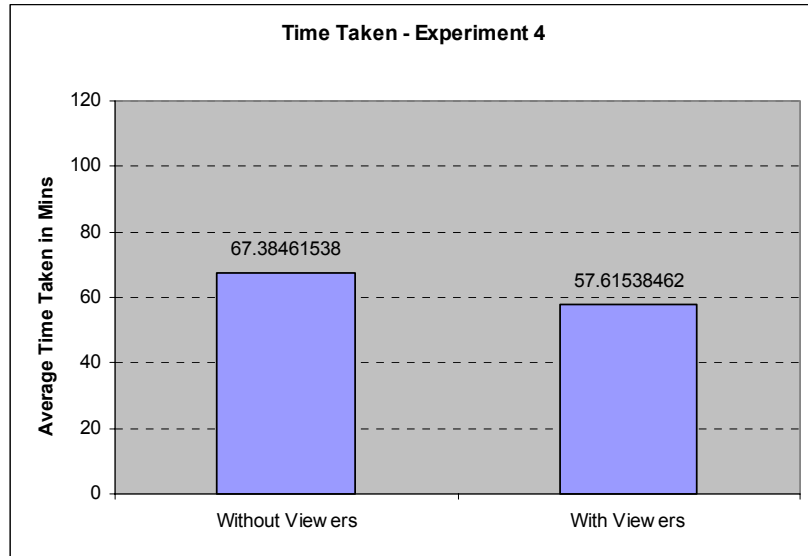


Fig. 5.22: Experiment IV - average time taken by the treatment group (with viewers) and the control group (without viewers)

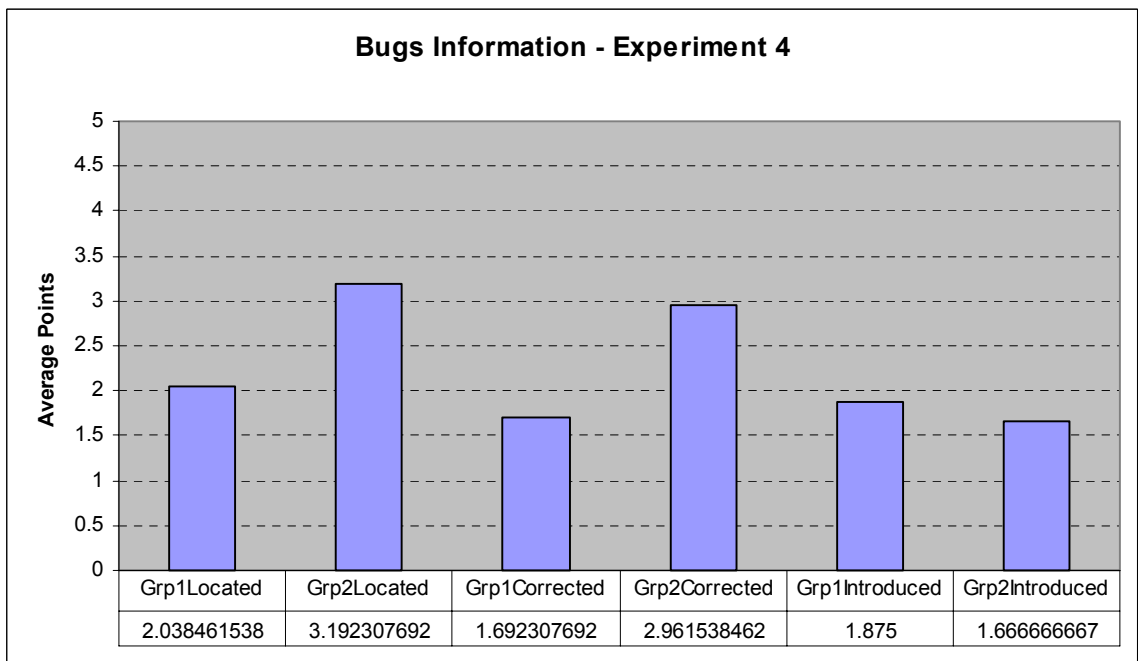


Fig. 5.23: Experiment IV - average accuracy of the treatment group (with viewers) and the control group (without viewers)

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 $n=34$ (sample size) was 7.0891.

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 alpha value, the null hypothesis can be strongly rejected, thus there was a statistical difference between the two groups. It was observed that students in the treatment group consistently performed better than the control group for all cases.

Post-hoc MANOVA analysis indicated that visualization medium influenced number of logical errors found, $F(1, 66) = 13.52$, $p = 0.0005$; number of logical errors accurately corrected, $F(1, 66) = 8.91$, $p = 0.0040$; number of new bugs introduced accuracy, $F(1, 66) = 5.08$, $p = 0.0275$ and time, $F(1, 66) = 4.56$, $p = 0.0365$. Thus, it can be concluded that in 95% of all cases, the use of jGRASP object viewers to write programs to implement data structures resulted in a greater number of logical errors detected and corrected accurately while introducing fewer errors accuracy and in less time.

5.6. EXPERIMENT V – MIN-MAX HEAP

The hypothesis for this experiment was that students will be able to transition from concept to implementation for data structures that are covered conceptually in lectures faster and more accurately using jGRASP viewers.

5.6.1. Method

5.6.1.1. Participants

Sixty-eight students enrolled in COMP2210 participated in the experiment. Participants were given extra credit in their course and were treated in accordance with the “Ethical Principles of Psychologists and Code of Conduct” [American Psychological Association 2002]. See section 5.1.2 for grading details. As required by the experiment, the participants had no experience with the implementation of *min-max heaps*.

5.6.1.2. Materials

The participants for this experiment were separated into two groups. Group 1 was the control group and implemented the tasks without the jGRASP viewers, and Group 2 was the treatment group and implemented the exact same tasks using the jGRASP viewers.

The min heap and max heap data structures were covered only conceptually during lectures by the instructor; no programming code was discussed in class; and no lab assignments were given on these either. For the experiment, students were provided with detailed conceptual explanation for the various operations of the max heap data structure, and a Java code implementation of the min heap data structure. The goal was to understand the code for min heap and then convert the given data structure to a max heap, and also implement the *addElement()*, *removeMax()*, *findMax()* methods for a max heap.

The program Heap.java from the class textbook was used in this experiment [Lewis and Chase 2004]. The control group implemented the *addElement()*, *removeMax()*, *findMax()* methods for a max heap using the jGRASP visual debugger. Details of the materials provided are shown in Figure 5.24. 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 three 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 main program (Heap.java) provided to each group can be found in Appendix I.

5.6.1.3. Design and Procedures

The 68 students were split into two matched groups: 34 in Group 1, the group that used only the jGRASP debugger; and 34 in Group 2, the group that used both the jGRASP debugger and the viewers. In order to minimize the variation between the two groups, the students were

matched on two programming skills – the ability to detect and correct logical errors and the ability to comprehend and trace programs. See section 5.1.1 for details.

Before using the system, 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 machines in the lab were set up with permissions such that only the treatment group had access to the viewers.

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.

5.6.2. Results

The normal P-P plots showed that the dependent variables were normally distributed. The null hypothesis was that there is no difference in the accuracy and time taken for both groups. For 34 samples in each group, Hotelling's T^2 statistic was calculated to be 10.813. 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 0.0024. Since the T^2 value is much greater than the critical value, and p-value is much less than the alpha value, the null hypothesis can be strongly rejected. Thus, there was a statistical significant difference between the two groups.

Post-hoc MANOVA analysis indicated that visualization medium influenced both time, $F(1, 66) = 5.18$, $p = 0.0261$; and accuracy $F(1, 66) = 5.60$, $p = 0.0209$. Figure 5.25 shows that the mean time taken by the group with viewers is 41.48 minutes while the mean time taken by the group without viewers is 51.24 minutes.

Max Heap

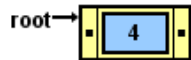
A heap is a specialized tree-based data structure. If A and B be nodes of a heap, such that B is a child of A . The heap must then satisfy the following condition (*heap property*):

$$\text{Value}(A) \geq \text{Value}(B)$$

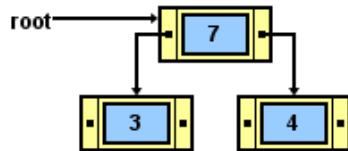
In this form it implies that the greatest element is always in the root node, and such a heap is called a *max heap*.

For example:

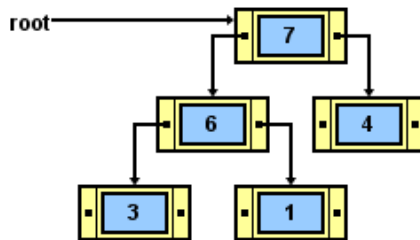
a) The first node to be added to the heap is a node with value “4”. The heap looks like this:



b) Next we add a node with a value “3” and then a node with a value “7”. Heap now looks like this:



c) Next we add a node with a value “6” and then a node with a value “1”. Heap now looks like this:



d) After we call `removeMax()`, the root should be removed since it contains the node with the maximum value. The Heap looks like this after `removeMax()` is called. Node with value “7” is removed.

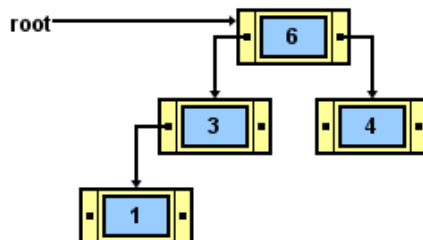


Fig. 5.24: Details of max heap used in Experiment V

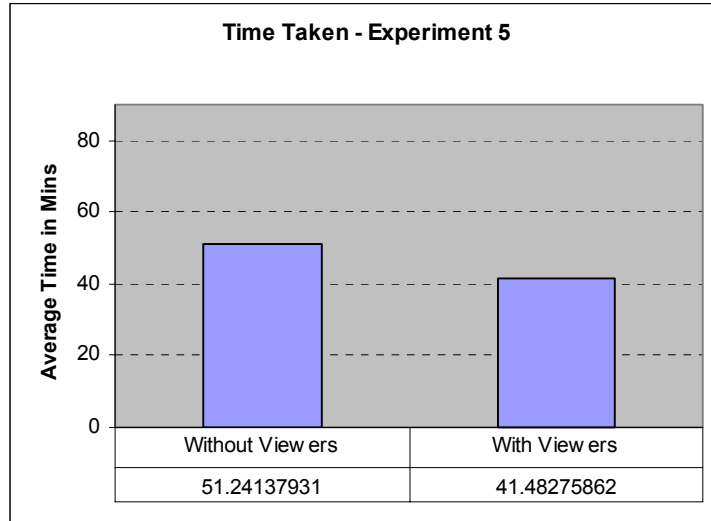


Fig. 5.25: Experiment V - comparison of mean time

Figure 5.26 shows that the mean accuracy of the treatment group with viewers is 2.86 points, while the mean accuracy of the control group without viewers is 4.03 points.

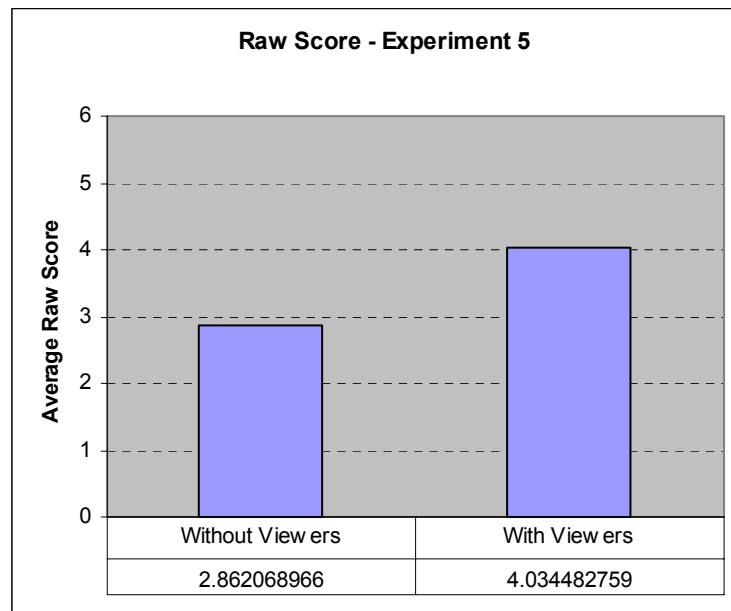


Fig. 5.26: Experiment V - comparison of mean accuracy

It was observed that students in the treatment group consistently performed better than the control group for all cases. Thus, it can be concluded that in 95% of all cases, using jGRASP

object viewers, students will be able to transition from concept to implementation for data structures that are covered conceptually in lectures more accurately and in less time.

5.7. EXPERIMENT VI – LINKED PRIORITY QUEUE

The hypothesis for this experiment was that students will be able to apply concepts for new data structures faster and more accurately using jGRASP viewers.

5.7.1. Method

5.7.1.1. Participants

Sixty-eight students enrolled in COMP2210 participated in the experiment. All participants were given extra credit in their course and were treated in accordance with the “Ethical Principles of Psychologists and Code of Conduct” [American Psychological Association 2002]. See section 5.1.2 for grading details. As required by the experiment, the participants had no conceptual or implementation knowledge of priority queues.

5.7.1.2. Materials

The participants for this experiment were separated into two groups. Group 1 was the control group and implemented the tasks without the jGRASP viewers, and Group 2 was the treatment group and implemented the exact same tasks using the jGRASP viewers.

Students were provided with detailed conceptual explanation for the *add()* method of the priority queue data structure (see table 5.5). All the students were seeing this data structure for the first time (i.e., it was not covered in the lectures and they had not read about it). The goal for this experiment was to understand the conceptual working of the basic add operation for linked priority queues and then to implement it. The program `PriorityQueueLinked.java` from the class

textbook was used in this experiment [Lewis and Chase 2004]. The control group implemented the *add()* method using the jGRASP visual debugger. Details of the materials are shown in Figure 5.27. 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 *add()* method 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 main program (PriorityQueueLinked.java) provided to each group can be found in Appendix J.

Table 5.5: Method implemented for Experiment VI

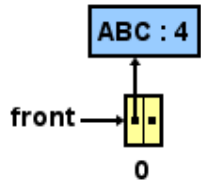
| PROGRAM | TASKS |
|--------------------------|--|
| PriorityQueueLinked.java | <pre data-bbox="721 1035 1386 1066">public void add(E value, int priority)</pre> <p data-bbox="721 1100 1360 1194">Fully implement and test this method. Your method should add nodes to the queue as shown in Step 1. As usual there will be no time limit for this activity.</p> |

Priority Queue

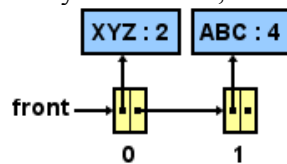
Priority queues behave exactly like queues but only differ in the add operation. In a regular queue nodes are always added towards the end/rear/tail of the queue, but in a priority queue nodes are added based on their priority. Our convention is that lower the number higher the priority of the node.

For example:

a) The first node to be added to the queue is a node with value “ABC” and priority 4. Priority queue looks like this:



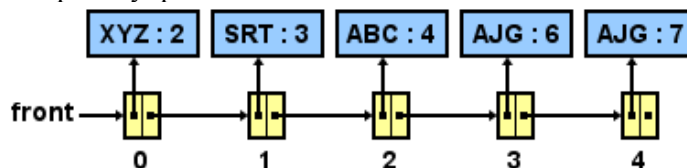
b) Next we add a node with “XYZ” and priority 2. Since this has a GREATER priority than the previously added node, it is added to the front. Priority queue looks like this:



c) Next, let us add 3 more nodes given below.

- 1) a node with value “SRT” and priority 3 followed by
- 2) a node with value “AJG” and priority 6 followed by
- 3) a node with value “AJG” and priority 7

The priority queue looks like this:



d) The last node to be added has a value BCD with a priority 3. This should be inserted between nodes at index 1 and 2. This is added after node (SRT, 3) since it arrived after this node. So given two nodes with the same priority, the one which arrive before is given HIGHER priority.

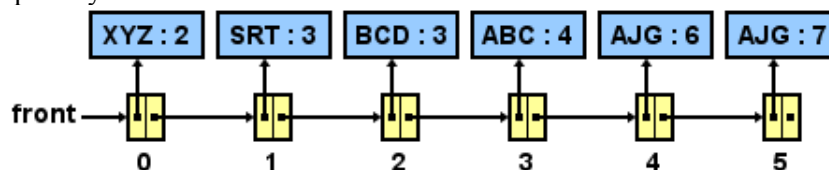


Fig. 5.27: Details of priority queue used in Experiment VI

5.7.1.3. Design and Procedures

The 68 students were split into two matched groups: 34 in Group 1, the group that used only the jGRASP debugger; and 34 in Group 2, the group that used both the jGRASP debugger and the viewers. In order to minimize the variation between the two groups, the students were matched on two programming skills – the ability to detect and correct logical errors and the ability to comprehend and trace programs. See section 5.1.1 for details.

Before using the system, 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 machines in the lab were set up with permissions such that only the treatment group had access to the viewers.

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.

5.7.2. Results

The normal P-P plots showed that the dependent variables were normally distributed. The null hypothesis was that there is no difference in the accuracy and time taken for both groups. For 34 samples in each group, Hotelling's T^2 statistic was calculated to be 19.756. 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 0.00009. Since the T^2 value is much greater than the critical value, and p-value is much less than the alpha value, the null hypothesis can be strongly rejected. Thus, there was a statistically significant difference between the two groups.

Post-hoc MANOVA analysis indicated that visualization medium influenced both time, $F(1, 66) = 7.13$, $p = 0.0095$; and accuracy $F(1, 66) = 6.44$, $p = 0.0135$. Figure 5.28 shows that the

mean time taken by the group with viewers is 50.79 minutes while the mean time taken by the group without viewers is 58.93 minutes.

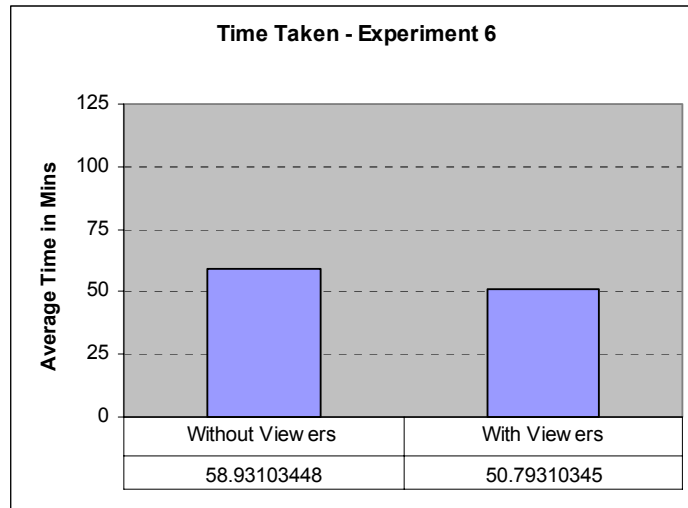


Fig. 5.28: Experiment VI - comparison of mean time

Figure 5.29 shows that the mean accuracy of the treatment group with viewers is 2.58 points, while the mean accuracy of the control group without viewers is 2.10 points.

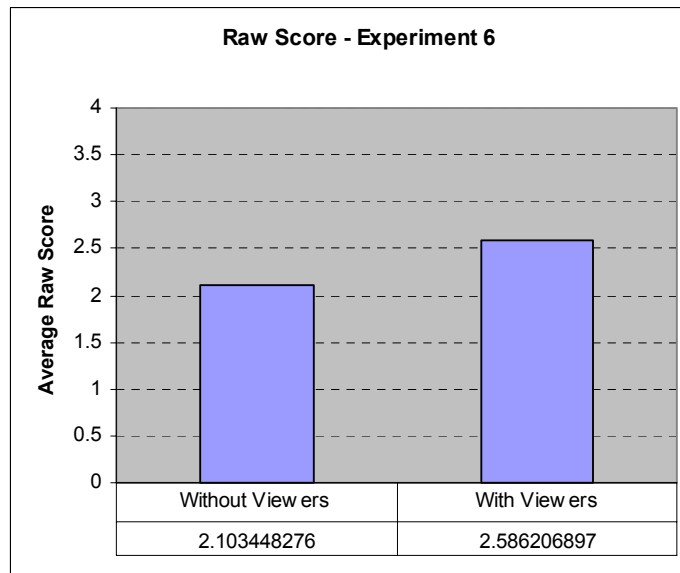


Fig. 5.29: Experiment VI - comparison of mean accuracy

It was observed that students in the treatment group consistently performed better than the control group for all cases. Thus, it can be concluded that in 95% of all cases, using jGRASP object viewers, students will be able to apply concepts for new data structures (that are not covered in the lectures) faster and more accurately.

5.8. SAMPLE SIZE ANALYSIS

To calculate power the non-centrality parameter for this distribution is required. This non-centrality parameter is defined as follows:

$$\lambda = \frac{N_1 N_2}{N_1 + N_2} (\mu_1 - \mu_2)' \Sigma^{-1} (\mu_1 - \mu_2)$$

$$\lambda = \frac{N_1 N_2}{N_1 + N_2} \Delta^2$$

where $\Delta = \sqrt{(\mu_1 - \mu_2)' \Sigma^{-1} (\mu_1 - \mu_2)}$ and Σ is the common variance covariance matrix. The formula above is defined as effect size because it provides an expression for the magnitude of the standardized difference between the null and alternative means. Using this non-centrality parameter, the power of the Hotelling's T^2 may be calculated for any value of the means and standard deviations. Since there is a simple relationship between the non-central T^2 and the non-central F, calculations are actually based on the non-central F using the formula

$$\beta = \Pr(F' < F'_{\alpha, df_1, df_2, \lambda})$$

where

$$df_1 = p$$

$$df_2 = N_1 + N_2 - p - 1$$

The power was calculated for each experiment as shown in the following output where:

- **Power** is the probability of rejecting a false null hypothesis. Note that $\text{Power} = 1 - \text{Beta}$.
- **N1** and **N2** are the sample sizes of the two groups.
- **Alpha** is the probability of rejecting a true null hypothesis (Was set to 0.05).
- **Beta** is the probability of accepting a false null hypothesis. Note that $\text{Beta} = 1 - \text{Power}$
- **Effect Size** is a standardized version of T^2 under the alternative hypothesis.
- **DF1** is the first degrees of freedom of T^2 . It is the number of response variables.
- **DF2** is the second degrees of freedom of T^2 .

Experiment I

| Power | N1 | N2 | By (K) | Multiply Means Beta | Effect Size | Number of Y's (DF1) | DF2 |
|--------------|-----------|-----------|---------------|----------------------------|--------------------|----------------------------|------------|
| 0.7735 | 20 | 20 | 1.0000 | 0.2265 | 0.87 | 2 | 37 |
| 0.8584 | 25 | 25 | 1.0000 | 0.1416 | 0.87 | 2 | 47 |
| 0.9139 | 30 | 30 | 1.0000 | 0.0861 | 0.87 | 2 | 57 |
| 0.9488 | 35 | 35 | 1.0000 | 0.0512 | 0.87 | 2 | 67 |
| 0.9702 | 40 | 40 | 1.0000 | 0.0298 | 0.87 | 2 | 77 |

Experiment II

| Power | N1 | N2 | By (K) | Multiply Means Beta | Effect Size | Number of Y's (DF1) | DF2 |
|--------------|-----------|-----------|---------------|----------------------------|--------------------|----------------------------|------------|
| 0.6172 | 20 | 20 | 1.0000 | 0.3828 | 0.78 | 3 | 36 |
| 0.7181 | 25 | 25 | 1.0000 | 0.2819 | 0.78 | 3 | 46 |
| 0.7970 | 30 | 30 | 1.0000 | 0.2030 | 0.78 | 3 | 56 |
| 0.8566 | 35 | 35 | 1.0000 | 0.1434 | 0.78 | 3 | 66 |
| 0.9004 | 40 | 40 | 1.0000 | 0.0996 | 0.78 | 3 | 76 |

Experiment III

| Power | N1 | N2 | By (K) | Multiply Means Beta | Effect Size | Number of Y's (DF1) | DF2 |
|--------------|-----------|-----------|---------------|----------------------------|--------------------|----------------------------|------------|
| 0.6797 | 20 | 20 | 1.0000 | 0.3203 | 0.78 | 2 | 37 |
| 0.7738 | 25 | 25 | 1.0000 | 0.2262 | 0.78 | 2 | 47 |
| 0.8436 | 30 | 30 | 1.0000 | 0.1564 | 0.78 | 2 | 57 |
| 0.8937 | 35 | 35 | 1.0000 | 0.1063 | 0.78 | 2 | 67 |
| 0.9289 | 40 | 40 | 1.0000 | 0.0711 | 0.78 | 2 | 77 |

Experiment IV

| Power | N1 | N2 | By (K) | Multiply Means Beta | Effect Size | Number of Y's (DF1) | DF2 |
|--------|----|----|--------|---------------------|-------------|---------------------|-----|
| 0.7520 | 20 | 20 | 1.0000 | 0.2480 | 0.92 | 3 | 36 |
| 0.8451 | 25 | 25 | 1.0000 | 0.1549 | 0.92 | 3 | 46 |
| 0.9063 | 30 | 30 | 1.0000 | 0.0937 | 0.92 | 3 | 56 |
| 0.9449 | 35 | 35 | 1.0000 | 0.0551 | 0.92 | 3 | 66 |
| 0.9684 | 40 | 40 | 1.0000 | 0.0316 | 0.92 | 3 | 76 |

Experiment V

| Power | N1 | N2 | By (K) | Multiply Means Beta | Effect Size | Effect (DF1) | Number of Y's DF2 |
|--------|----|----|--------|---------------------|-------------|--------------|-------------------|
| 0.6395 | 20 | 20 | 1.0000 | 0.3605 | 0.74 | 2 | 37 |
| 0.7347 | 25 | 25 | 1.0000 | 0.2653 | 0.74 | 2 | 47 |
| 0.8082 | 30 | 30 | 1.0000 | 0.1918 | 0.74 | 2 | 57 |
| 0.8635 | 35 | 35 | 1.0000 | 0.1365 | 0.74 | 2 | 67 |
| 0.9042 | 40 | 40 | 1.0000 | 0.0958 | 0.74 | 2 | 77 |

Experiment IV

| Power | N1 | N2 | By (K) | Multiply Means Beta | Effect Size | Number of Y's (DF1) | DF2 |
|--------|----|----|--------|---------------------|-------------|---------------------|-----|
| 0.6635 | 20 | 20 | 1.0000 | 0.3365 | 0.76 | 2 | 37 |
| 0.7583 | 25 | 25 | 1.0000 | 0.2417 | 0.76 | 2 | 47 |
| 0.8297 | 30 | 30 | 1.0000 | 0.1703 | 0.76 | 2 | 57 |
| 0.8820 | 35 | 35 | 1.0000 | 0.1180 | 0.76 | 2 | 67 |
| 0.9195 | 40 | 40 | 1.0000 | 0.0805 | 0.76 | 2 | 77 |

For experiments 1 and 3, average sample sizes of 25 in each group results in a power of 80-85%, and for the other experiments average sample sizes of 30-35 will result in a power of 90%. The two-sample Hotelling's T^2 test statistic was used with a significance level of 0.05.

5.9 RETENTION OF CONCEPTS

Early indicators suggest that jGRASP viewers help with retention of concepts as well. Table 5.6 shows the average scores of students in Group 1 and Group 2 for Quizzes, Exam 1, Exam 2, Final Exam and Overall Grade for the course COMP 2210. In all five cases the performance of Group 2 was much better than Group 1. Exam 1 tested the following topics: sets, linked structures, stacks, queues, lists and recursions. Experiments conducted before Exam 1 covered linked structures, stacks, queues. Exam 2 tested the following topics: trees, binary search trees, multi-way search trees, heaps and hashing. Experiments conducted before Exam 2 covered binary search trees, min-max heaps and priority queues. Final Exam was comprehensive covering all topics, and the Overall Grade was an average of exam scores, in-lab paper-based quizzes, and in-lab programming assignments.

Table 5.6: Comparison of average scores of Group 1 and Group 2 in the COMP 2210 course

| Averages | Group 1 | Group 2 |
|-----------------|----------------|----------------|
| Quizzes | 76.09 % | 80.12 % |
| Exam 1 | 69.15 % | 73.06 % |
| Exam 2 | 56.52 % | 61.67 % |
| Final Exam | 70 % | 77.38 % |
| Overall Grade | 66.23 % | 72.44 % |

CHAPTER 6

QUESTIONNAIRE TO EVALUATE THE USER INTERFACE ASPECTS OF jGRASP VIEWERS AND DEBUGGER

A questionnaire to evaluate the user interface aspects of the jGRASP debugger and the viewers was conducted. Group 1 (control group) was given a set of questions to determine if they understood the functionality of the debugger features and the icons used to represent the features. Group 2 (treatment group) was given a set of questions to determine the same for the viewer features.

6.1. DEBUGGER QUESTIONNAIRE

Debug tab pane is divided into three sub panes or sections – Threads, Call Stack, Variables/Eval. The *Threads* section lists all the active threads running in the program. The *Call Stack* section shows the current call stack and allows the user to switch from one level to another in the call stack. When this occurs, the CSD window that contains the source code associated with that particular call is brought to the top of the desktop, and the associated variables are updated in the *Variables* pane. The *Variables/Eval* section shows the details of the current state of the program in the *Variables* tab, and provides an easy way to evaluate expressions involving these variables in the *Eval* tab.

6.1.1 The Debug Buttons

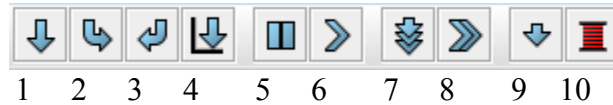


Fig. 6.1: The debug button panel in jGRASP

1 – *Step*: Clicking this button will single step to the next statement. E.g., If the statement contains a method, the entire method is executed and the control is moved to the next statement in the program file being debugged.

2 – *Step in*: Clicking this button for a statement with a method call (that is a part of the user’s source code) will step into the method implementation in the file containing the method definition. The top entry in the Call Stack indicates where the user is in the program.

3 – *Step out*: Clicking this button will return control to the statement from where the user previously “stepped in”.

4 – *Run to cursor*: Clicking this button will execute the program until the statement with the cursor (L) is reached. If the cursor is not on a statement along the control path, the program will stop at the next breakpoint. The “Run to cursor” is convenient since placing the cursor on a statement is like setting a “temporary” breakpoint.

5 – *Pause*: Clicking this button pauses the thread wherever it happens to be while using the “Auto step” or “Auto resume” functionality. It also pauses execution of the thread which is in an infinite loop or waiting for user input.

6 – *Resume*: Clicking this button resumes the thread to the next breakpoint in the program. If the breakpoint is set in another file, and this breakpoint is on the control path, then the other source file will be given focus when the breakpoint is reached.

7 – *Auto step*: This button is used to toggle on and off a mode which allows the user to automatically step repeatedly after the Step button (#1) is clicked once. The program can be paused by clicking the Pause button (#5), and auto stepping can be restarted by clicking the Step button (#1).

8 – *Auto resume*: This button is used to toggle on and off a mode which allows the user to automatically resume repeatedly (stopping briefly at breakpoints) after the Resume button (#6) is clicked once. The program can be paused by clicking the Pause button (#5), and auto resuming can be restarted by clicking the Resume button (#6).

9 – *Use byte code size steps*: This button is used to toggle on and off a mode which allows the user to step through the program in the smallest increments possible.

10 – *Suspend new threads*: This button is used to toggle on and off the mode that will immediately suspend any new threads.

The questionnaire used for Group 1 is given in Appendix M. Question 1 asked the participants to answer the question: “After you start the debugging procedure, how usefulness are the following features?” The four features were Threads, Call Stack, Variables, and Eval tabs. A four point Likert scale (1: Useful; 2: Somewhat Useful; 3: Somewhat Useless; 4: Useless) was used to rate Question 1.

Question 2 asked them “How often did you use the following features?” The features are listed in Table 1. A four point Likert scale (1: For most of the activities; 2: For at least half of the activities; 3: For 1 or 2 activities; 4: Never needed to use this feature) was used to rate Question 2 as well.

Question 3 asked them “Is this icon a good representation or depiction of the following features?” These features were the same as the ones used in Question 2. A five point Likert scale (1: Yes – I was immediately able to recognize the feature; 2: Yes – I was able to recognize after I read what it does; 3: No – I had to repeatedly look up what it does; 4: No – change the icon since it is not a good representation of the feature; 0: N/A I never used this feature) was used to answer Question 3.

6.1.2. Results and Discussions

The *Variables* section is the most useful and most frequently used feature, and the *Eval* section is the least useful and least frequently used feature [see Figure 6.2 and 6.2, Table 6.1]. The *Call stack* was rated as a “useful” to “somewhat useful” feature by approximately 85% of students, but was frequently used by only 50% of the students. In this scenario, students switched between the main/driver program stack and data structure call stack during the step-in process. The ability to switch between threads was not relevant to the programs implementing data structures since these were all single threads. Thus as expected, most students did not use this feature.

Based on Figure 6.4, it was observed that the three most used debug features were step-over, step-in and step-out and the three least used features suspend new threads, use byte size steps and auto resume. Based on this information multiple recommendations for the layout of the

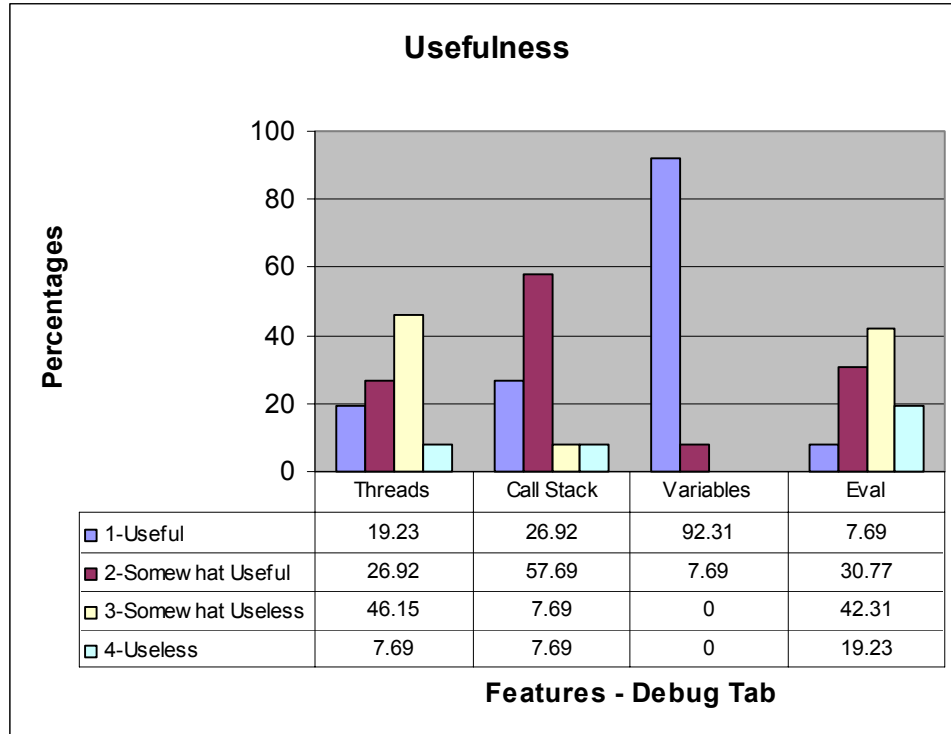


Fig. 6.2: Usefulness of the *Debug Tab* features

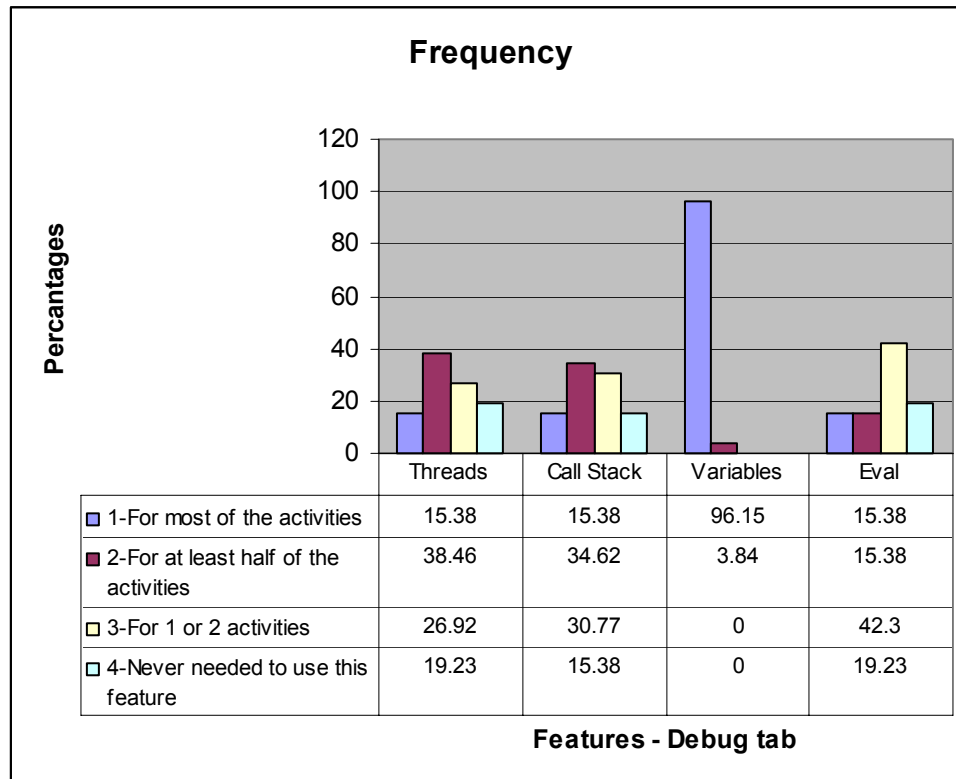


Fig. 6.3: Frequency of use of the *Debug Tab* features

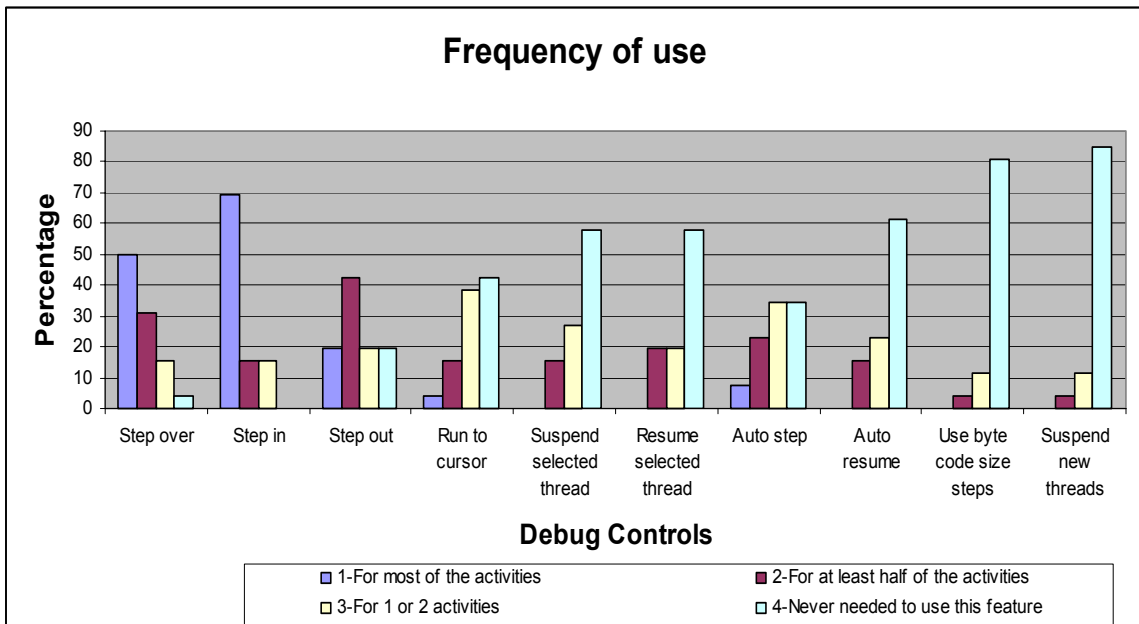


Fig. 6.4: Frequency of use of Debug controls

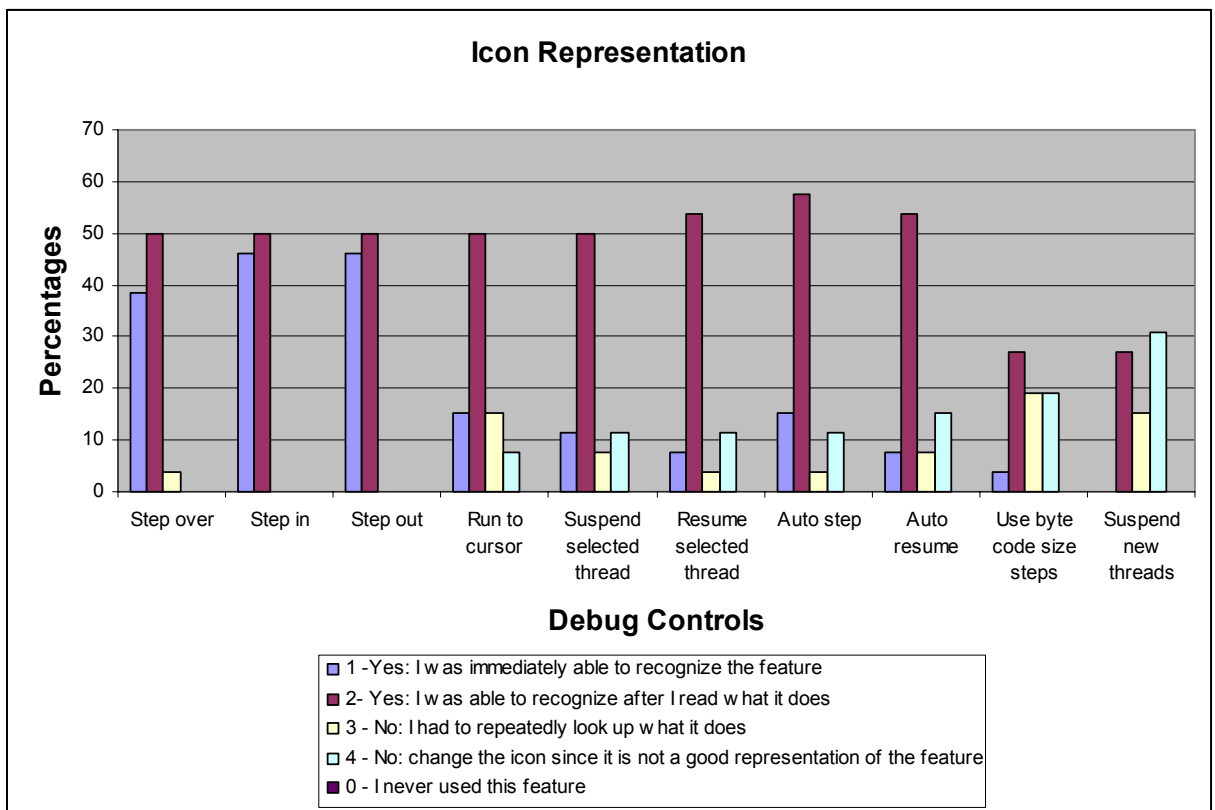






















Fig. 6.5: Icon representation of Debug controls

user interface of the debug panel will be presented in the next section. Overall the icons used to represent debug features were relatively easily recognizable. The three icons that students had to repeatedly look up were for suspend new threads, use byte size steps, and auto resume (see Figure 6.5).

Table 6.1: Results of jGRASP debugger questionnaire for Group 1

| | |
|--|---|
| 1. After you start the debugging procedure how useful are the following features? | Median score (Mode)¹ n = 26 |
| a. Threads | 2 (2) |
| b. Call Stack | 2 (2) |
| c. Variables | 1 (1) |
| d. Eval tab (next to Variables) | 3 (3) |
| 2. How often did you use the following features: | Median score (Mode)² n = 26 |
| a. Threads | 3 (3) |
| b. Call Stack | 2 (2) |
| c. Variables | 1 (1) |
| d. Eval tab (next to Variables) | 3 (3) |
|  Step over | 1.5 (1) |
|  Step in | 1 (1) |
|  Step out | 2 (2) |
|  Run to cursor | 3 (4) |
|  Suspend selected thread | 4 (4) |
|  Resume selected thread | 4 (4) |
|  Auto step | 3 (bimodal 3 and 4) |
|  Auto resume | 4 (4) |
|  Use byte code size steps | 4 (4) |
|  Suspend new threads | 4 (4) |
| 3. Is this icon a good representation or depiction of the feature? | Median score (Mode)³ n = 26 |
|  Step over | 2 (2) |
|  Step in | 2 (2) |

| 3. Is this icon a good representation or depiction of the feature? | Median score (Mode) ³ n = 26 |
|--|--|
|  Step out | 2 (2) |
|  Run to cursor | 2 (2) |
|  Suspend selected thread | 2 (2) |
|  Resume selected thread | 2 (2) |
|  Auto step | 2 (2) |
|  Auto resume | 2 (2) |
|  Use byte code size steps | 3 (2) |
|  Suspend new threads | 3 (bimodal 2 and 4) |

In Table 6.1, responses for the first question is based on a Likert scale ranging from 1 to 4 on which 1 = Useful, 2 = Somewhat Useful, 3 = Somewhat Useless, and 4 = Useless. The responses for the second question is based on a Likert scale ranging from 1 to 4 on which 1 = For most of the activities, 2 = For at least half of the activities, 3 = For 1 or 2 activities, and 4 = Never needed to use this feature. The responses for the third question is based on a Likert scale ranging from 0 to 4 on which 1 = Yes – I was immediately able to recognize the feature, 2 = Yes – I was able to recognize after I read what it does, 3 = No – I had to repeatedly look up what it does, 4 = No – change the icon since it is not a good representation of the feature, 0 = N/A I never used this feature.

6.1.3. Interface Layout Recommendations

The “Thread” feature is useful for visualizing aspects of multithreaded applications. But since multithreading is not taught in CS1 and CS2, features related to threading are not used. The thread section on the “Debug” tab, and the “Suspend new thread” button related to multithreading can be removed from the default view and shifted under “Settings” as an advanced feature which

can be turned on/off (e.g., Settings-> Debugger ->Multithreading). This will simplify the debug tab. Most students did not use this feature, and a majority did not think the icon used for this button was a good representation of the feature. This feature can be moved to the advanced settings, and the icon needs to be redone.

The new suggested layouts of the debug buttons are as follows:

Layout 1:



Fig. 6.6: Layout recommendation 1 for debug panel

- a) The buttons related to stepping – one statement at a time; from one statement to a temporary breakpoint; and from one statement to the next breakpoint should be grouped together since they have similar functionality
- b) The buttons related to step in and step out should be grouped together
- c) The auto step/auto resume buttons should be grouped together separately, where the pause button is visible only if the auto step and/or the auto resume buttons are toggled to an ON position or when stepping is activated.

Layout 2:

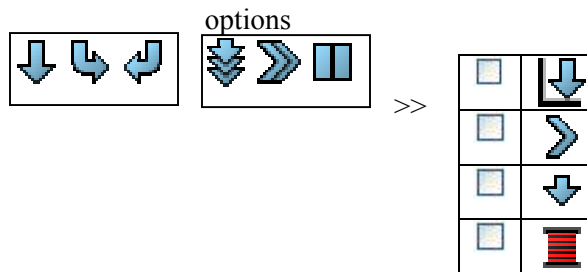


Fig. 6.7: Layout recommendation 2 for debug panel

- a) The buttons which are used frequently are displayed on the main toolbar, the other features are optional and can be turned on by using the checkboxes
- b) The toggle buttons must be grouped together, since their functionality differs from the other regular buttons. Additionally, a descriptive heading (e.g., options) can be provided.

The look and feel of the toggle buttons must differ from regular buttons:

- i. When a toggle button is selected, the button can be highlighted using a border, or the button can be designed to look “pressed”.
- ii. The toggle buttons must be placed next to each other with no space between them, otherwise they might be mistaken for regular buttons.

6.2. JGRASP VIEWERS QUESTIONNAIRE

The questionnaire used for Group 2 is given in Appendix N. Question 1 asked the participants to answer the question: “On the viewer window, how useful are the following features?” The eight features used for this question are listed in Table 6.2. A four point Likert scale (1: Useful; 2: Somewhat Useful; 3: Somewhat Useless; 4: Useless) was used to rate Question 1.

Question 2 asked them “How often did you use the following features?” The nine features used for this question are listed in Table 6.2. A four point Likert scale (1: For most of the activities; 2: For at least half of the activities; 3: For 1 or 2 activities; 4: Never needed to use this feature) was used to rate Question 2 as well.

Question 3 asked them “Is this icon a good representation or depiction of the following features?” A five point Likert scale (1: Yes – I was immediately able to recognize the feature; 2: Yes – I was able to recognize after I read what it does; 3: No – I had to repeatedly look up what it does; 4: No – change the icon since it is not a good representation of the feature; 0: N/A I never used this feature) was used to answer Question 3.

Group 2 then answered some open-ended questions where the goal was to understand what other feature if other features could be added to improve that performance such as: 1) Will the ability to customize the color of the nodes be useful? 2) Will stepping back during the debugging process so that the before and after states of a data structure can be compared be useful? 3) Will the ability to control the orientation of the data structure (switching between vertical and horizontal) be useful? 4) Will the ability to add more variables to the viewer be useful? For example: if the method is using some local variables (which are currently being shown in the Debug tab) which are not a part of the main data structure, but do interact with the data structure during the step-in process, then would it be useful to have a canvas view where the user can drag and drop *any* global or local variable and the viewer would automatically show how the variables interact with the main data structure.

6.2.1. Results and Discussions

The animation related features (on/off and time adjuster) and adjusting the scale and width of the visualization were the top four most used features (see Figure 6.8 and 6.9 and table 6.2). The other viewer features were rated as useful, but were not frequently used by the students (see Figure 6.8 and 6.9). The reason is that once the viewer has been adjusted to match the mental or the textbook model, no further adjustments are deemed necessary.

The only adjustments used by students during debugging are the ones that adjust the size (width/scale) as the visualization grows, and the animation time adjuster.

The icon representation of almost all icons was rated well. The two icons that got the lowest ratings are animation-on and animation-off (Figure 6.10).

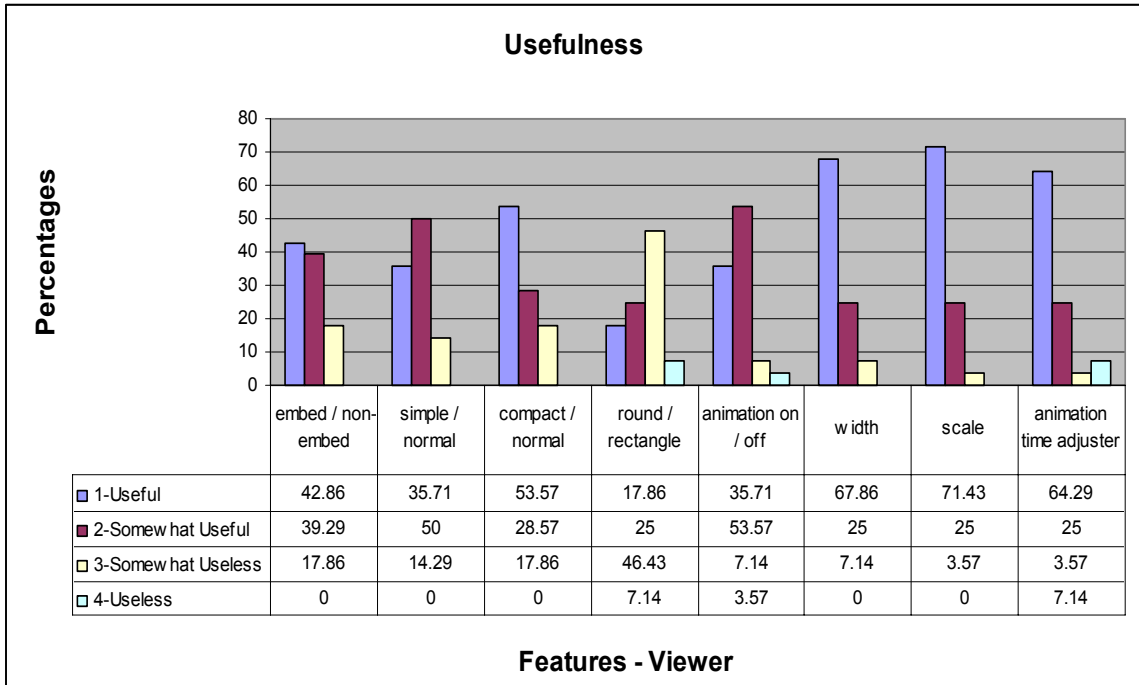


Fig. 6.8: Usefulness of viewer features

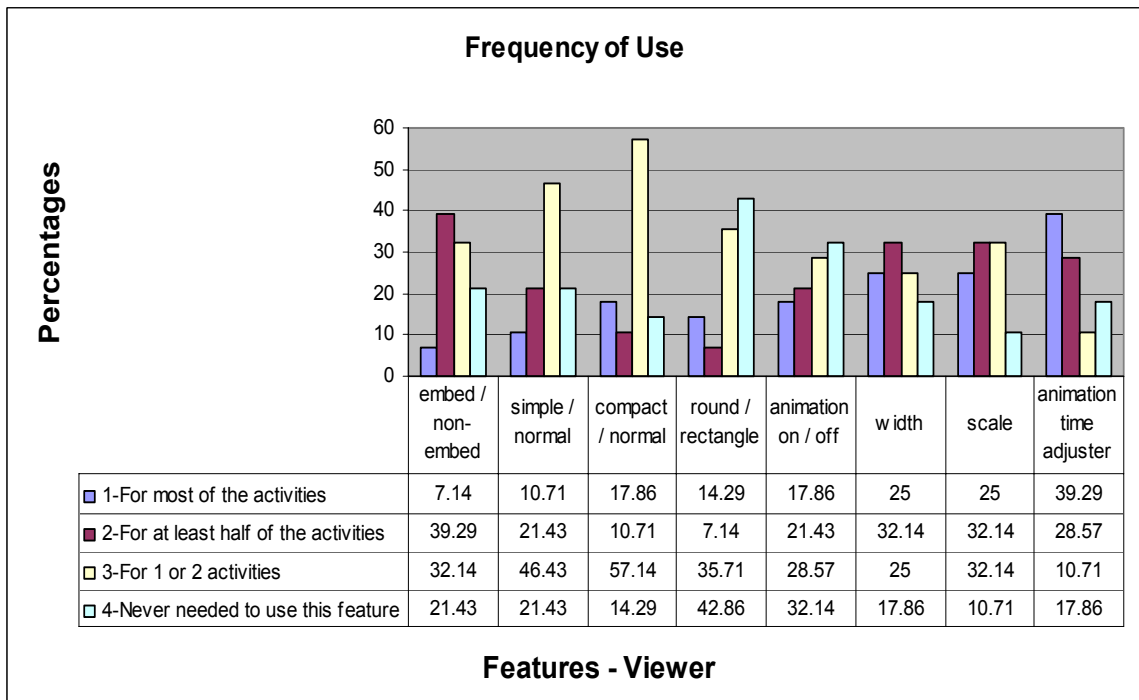


Fig. 6.9: Frequency of use of viewer features

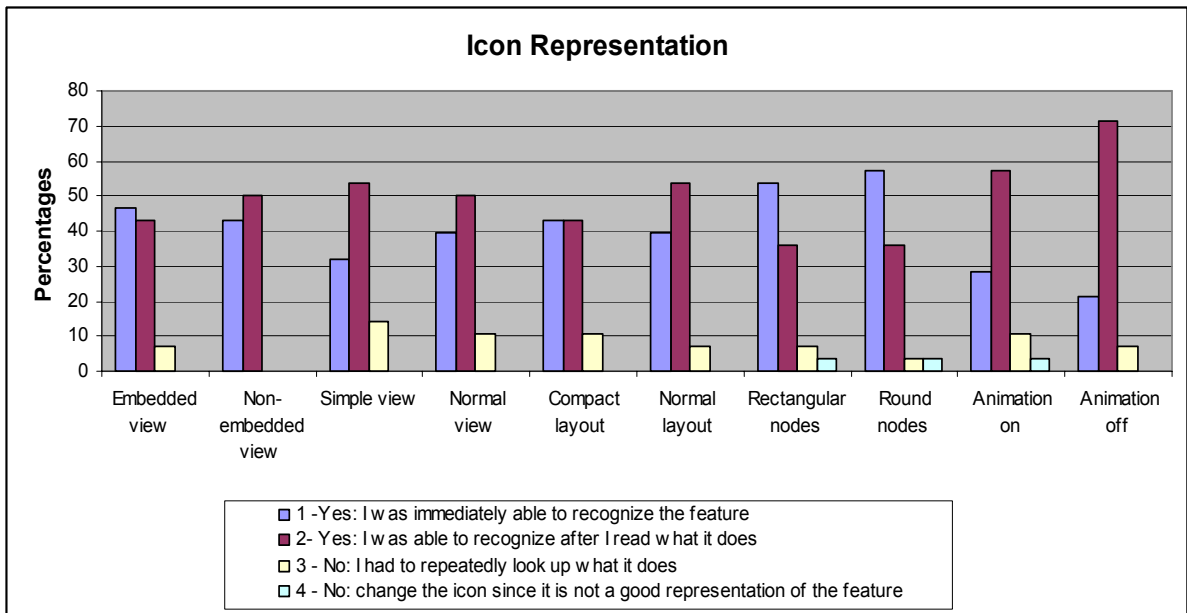
































Fig. 6.10: Icon representation of viewer features

Table 6.2: Results of jGRASP viewers questionnaire for Group 2

| 1: How useful are the following features (on the viewer window)? | Median score (Mode)¹ n = 28 |
|---|---|
| The feature to toggle between embedded  to non-embedded  view is: | 2 (1) |
| The feature to toggle between simple  and normal  view is: | 2 (2) |
| The feature to toggle between compact  and normal  layout is: | 1 (1) |
| The feature to toggle between rectangular  and round nodes  is: (Tree viewer) | 3 (3) |
| The feature to toggle between animation on  and off  is: | 2 (2) |
| The slide to adjust width of elements: | 1 (1) |
| The slide to adjust scale of the entire view: | 1 (1) |
| Increase or decrease animation time: | 1 (1) |

| | |
|--|---|
| 2: How often did you use the following features: | Median score (Mode)² N = 28 |
| Toggle between embedded  to non-embedded  view is: | 3 (2) |
| 2: How often did you use the following features: | Median score (Mode)² N = 28 |
| Toggle between simple  and normal  view | 3 (3) |
| Toggle between compact  and normal  layout | 3 (3) |
| Toggle between rectangular  and round nodes  (Tree viewer) | 3 (4) |
| Turn animation OFF  | 3 (4) |
| Turn animation ON  | 3 (bimodal 3 and 4) |
| The slide to adjust width of elements: | 2 (2) |
| The slide to adjust scale of the entire view: | 2 (bimodal 3 and 4) |
| Increase or decrease animation time: | 2 (1) |
| 3: Is this icon a good representation or depiction of the feature? | Median score (Mode)³ n = 28 |
| Embedded view  | 1.5 (1) |
| Non-embedded view  | 2 (2) |
| Simple view  | 2 (2) |
| Normal view  | 2 (2) |
| Compact layout  | 2 (bimodal 1 and 2) |
| Normal layout  | 2 (2) |
| Rectangular nodes  | 1 (1) |
| Round nodes  | 1 (1) |
| Animation on  | 2 (2) |
| Animation off  | 2 (2) |

In Table 6.2 the responses for the first question is based on a Likert scale ranging from 1 to 4 on which 1 = Useful, 2 = Somewhat Useful, 3 = Somewhat Useless, 4 = Useless. The responses for the second question is based on a Likert scale ranging from 1 to 4 on which 1 = For

most of the activities, 2 = For at least half of the activities, 3 = For 1 or 2 activities, 4 = Never needed to use this feature. The responses for the third question is based on a Likert scale ranging from 0 to 4 on which 1 = Yes – I was immediately able to recognize the feature, 2 = Yes – I was able to recognize after I read what it does, 3 = No – I had to repeatedly look up what it does, 4 = No – change the icon since it is not a good representation of the feature, 0 = N/A I never used this feature.

6.2.2. Interface Layout Recommendations

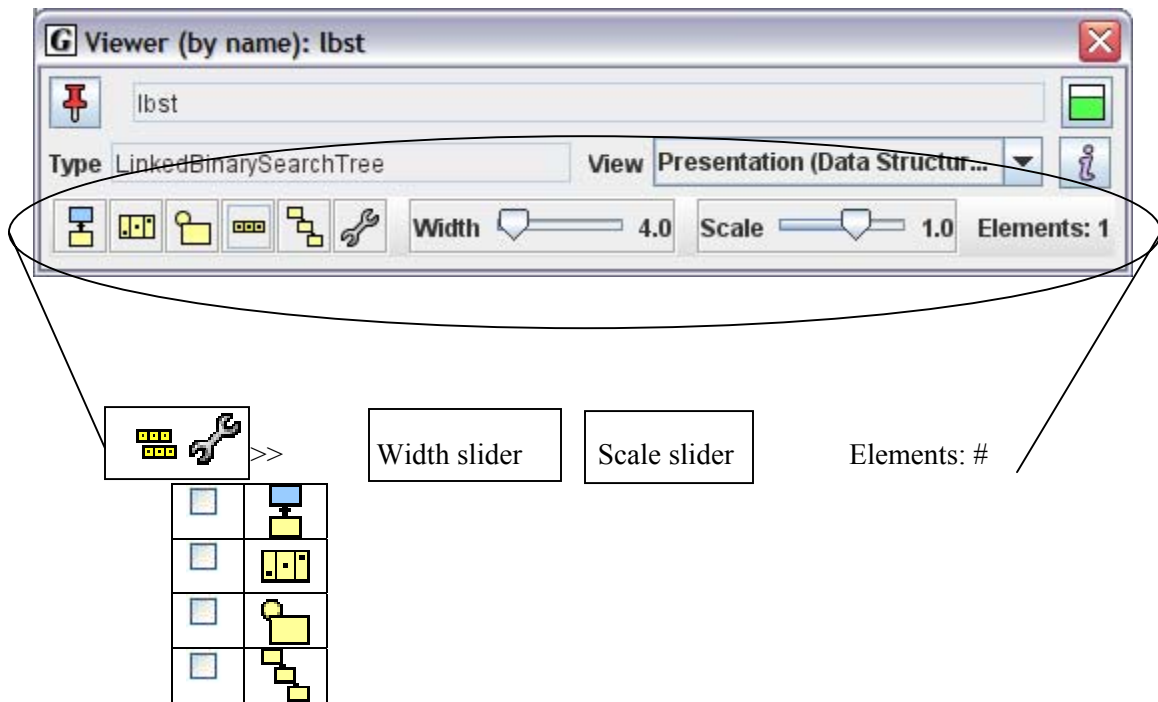


Fig. 6.11: Layout recommendation for the viewer controls based on usefulness and frequency of use

Students were also asked to rate four other features as shown in Table 6.3. 90% of the students rated the ability to step backwards as very useful. Approximately, 68% of the students rated the ability to add other variables as useful. Other features to change the look and feel of the

viewers, such as changing node colors and orientation were rated as useless by approximately 75% of the students.

Table 6.3: Results of open ended questions for jGRASP viewers for Group 2

| Is there any other feature that you think would be useful to the viewer? | Useful % n = 28 | Useless % n = 28 |
|---|----------------------------|-----------------------------|
| 1) Changing the color of the nodes in the viewer. | 21.43 | 78.57 |
| 2) Stepping back during the debugging process so that you can compare states | 89.29 | 10.71 |
| 3) Changing the orientation of the data structure (switching between vertical and horizontal) | 28.57 | 71.43 |
| 4) Ability to add more variables to the viewer (For example: if the method is using local integer and String variables, it would be great if those would be shown on the viewer as well. Right now you can see those in the Debug tab on the left hand side.) | 67.86 | 32.14 |

CHAPTER 7

SUMMARY AND CONCLUSIONS

Data structures and algorithms are abstract concepts, and the understanding of this topic and the material covered in class can be divided into three levels: conceptual, implementation, and application. Over the course of the past few years a consistent decline in enrollment in the Computer Science department has been observed. This trend is most noticeable during the COMP 2210 course when a majority of students decide to drop this required course. Paper-based surveys and multiple interviews were conducted in Fall 2004 and Spring 2005 to understand the aspects of the COMP 2210 that students find most difficult. It was found that the main problem was transitioning from static abstract concepts to dynamic program implementation of data structures.

The jGRASP IDE 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. The purpose of these viewers is to provide fine-grained support for understanding instances of classes representing data structures. When a class has more than one view associated with it, the user 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.

The purpose of the 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 data structure. 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, structural changes are animated in the viewers.

Initially, jGRASP viewers could only be generated using an API based approach. 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, it was soon discovered that it was unrealistic to expect students who are in the process of learning about data structures to be also able to modify a separate viewer class in order to see an instance of their own data structure. Research efforts were thus directed towards building a mechanism that could determine if an instance was a linked list or binary tree based on a class structures and a set of heuristics, and then automatically generate an appropriate view.

Six controlled experiments were conducted to test various hypotheses. Experiments I and II were conducted using singly linked lists, Experiments III and IV were conducted using linked binary trees, Experiment V was conducted using min and max heaps, and finally Experiment VI was conducted using linked priority queues. Since for each experiment more than one response variable was measured, Hotelling's T^2 statistical method was used for data analysis.

The goal of Experiments I and III was to determine if students would be able to code more accurately and in less time using the jGRASP data structure viewers for a relatively easy (singly linked list) and a relatively hard (linked binary tree) to understand data structure. Students were asked to implement basic operations for each data structure. The group that performed the tasks using the jGRASP viewers performed significantly better than the other group which did not use the viewers. This means that students should be able to transition from conceptual knowledge to implementation easily for both relatively easy and hard to understand data structures that are taught in details during lectures.

The goal of Experiments II and IV was to determine if students would be able to find and correct more logical errors accurately and faster using jGRASP viewers for a relatively easy (singly linked list) and a relatively hard (linked binary tree) to understand data structure. Students were provided code implementation with multiple logical errors. Their tasks consisted of locating and documenting the errors on paper, and then correcting the errors using jGRASP. It was observed that the group using viewers not only detected and corrected more errors in less time, but they also introduced fewer logical errors in the process. It was noticed that for Experiments I and II, there was not a statistically significant improvement in the time taken to complete the tasks. In later experiments there is a clear improvement in the time taken by the group that uses viewer. A likely reason for the initial results is that students were inexperienced in pointer implementation.

The results of the on the paper-based surveys (described in chapter 3), indicated that students rated data structures covered abstractly in class as “difficult to understand” even though historically these are not that difficult. Experiment V was conducted using min-max heap to test if students would be able to transition from concept to implementation faster and more accurately using jGRASP viewers for data structures that are covered only conceptually in lectures. Students were given a min heap implementation, which they were asked to understand the code and convert into a max heap implementation and additionally implement other related operations. It was found that the group using viewers was able to complete the tasks more accurately and in less time. Thus viewers can be used outside of classroom to understand concepts and transition to implementation.

Experiment VI was conducted using linked priority queue to test if students would be able to apply concepts for data structures that were not covered in lectures faster and more accurately using jGRASP viewers. Students were provided with detailed conceptual explanation of the priority queue data structure. All the students were introduced to this data structure for the

first time. It was found that the group using the jGRASP viewers was able to complete the tasks more accurately and in less time.

Finally, a questionnaire was conducted to evaluate the user interface aspects of the jGRASP debugger and the viewer window. It was found that students who knew how to use the debugger, only needed approximately two to three minutes to learn to use the viewers. Minor interface redesign is currently in progress. Students also reported that stepping back during the debugging process would be very useful as that would allow different states of the data structure to be compared. Due to technical issues in Java 1.5, this feature will be considered after Java version 1.6 is released.

Initial comparison of average scores in exams and quizzes of students in the two experimental groups (i.e., the treatment group using viewers and control group that did not use viewers) in COMP2210 shows that the treatment group outperformed the control group. A set of follow-on experiments to test if jGRASP viewers help with retention of concepts are recommended. It would also be useful to measure the amount of time spent in different activities while debugging (such as reading and editing source code and interacting with viewers) to determine of these for which activities jGRASP viewers are most helpful.

Data structure implementations from widely adopted CS2 textbooks are being tested for compatibility with the current jGRASP viewers, and the results so far are very positive. For the five textbooks tested, approximately 70% of data structures were recognized correctly by the structure identifier in jGRASP version 1.8.5 Beta 2. For the 30% that were not recognized the structure identifier could be manually configured to recognize and render the viewers. The main reason for lack of recognition was limited heuristics. As the heuristics become more comprehensive in the successive beta versions of jGRASP 1.8.5, the viewers should be able to automatically recognize over 95% of the textbook implementations for data structures in Java where the class name and fields are commonly used English identifiers.

Many software visualization tools have been developed that target low-level program comprehension, development and debugging, and high-level algorithm animation. Although many of these tools have been demonstrated to be pedagogically effective, no one single tool was found that would satisfy all of the following requirements: (1) serves the dual purpose of classroom demonstration and development environment, (2) provides automatic generation of dynamic views, including multiple and synchronized views, and (3) supports a seamless transition from concept to implementation of data structures. jGRASP viewers address all of these deficiencies and the experimental results of this research clearly indicate the potential for data structure viewers to significantly improve the teaching and learning for CS2 students.

BIBLIOGRAPHY

- [Agresti 1996] AGRESTI, A. 1996. An Introduction to Categorical Data Analysis, *Wiley-Interscience*. ISBN: 0471113387.
- [Akingbade 2003] AKINGBADE, A., FINLEY T., JACKSON D., PATEL P., AND RODGER S. H. 2003. JAWAA: easy web-based animation from CS 0 to advanced CS courses. In *Proceedings of the 34th technical symposium on Computer science education (SIGCSE)*, February 19-23, 2003, Reno, Nevada, USA, pp. 162 – 166.
- [American Psychological Association 2002] Ethical Principles of Psychologists and Code of Conduct. Available at <http://www.apa.org/ethics/code2002.html>
- [Baker et al. 1999] BAKER, R.S., BOILEN, M., GOODRICH, M., TAMASSIA, R., AND STIBEL, B.A. 1999. Testers and visualizers for teaching data structures. In *Proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, New Orleans, Louisiana, United States, pp. 261-265.
- [Ben-Ari et al. 2002] BEN-ARI, M., MYLLER, N., SUTINEN, E., AND TARHIO, J. 2002. Perspectives on program animation with Jeliot. In *Software Visualization, State-of-the-Art Survey*, Lecture Notes in Computer Science 2269, Springer, 2002, pp. 31-45.
- [Bergin et al. 1996] BERGIN, J., BRODLIE, K., GOLDWEBER, M., JIMÉNEZ-PERIS, R., KHURI, S., PATIÑO-MARTÍNEZ, M., MCNALLY, M., NAPS, T., RODGER, AND WILSON, J. 1996. An overview of visualization: its use and design. In *Proceedings of the ACM SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*, 1996, pp. 192-200.
- [Boies and Gould 1974] BOIES, S.F., AND GOULD, J.D. 1974. Syntactic errors in computer programming. *Human Factor*, volume 16, pp. 253-257.
- [Brown 1988] BROWN, M.H. 1988. Perspectives on algorithm animation. In *Proceedings of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems*, Washington D.C., May 1988, pp. 33-38.
- [Byrne et al. 1996] BYRNE, M. D., CATRAMBONE, R., AND STASKO, J.T. 1996. Do Algorithm Animations Aid Learning? Technical Report, *Georgia Institute of Technology*, 1996.

- [Byrne et al. 1999] BYRNE, M. D., CATRAMBONE, R., AND STASKO, J.T. 1999. Evaluating animations as student aids in learning computer algorithms. *Comput. Educ., Elsevier Science Ltd.*, 1999, vol. 33, pp. 253-278.
- [Campbell et al. 2003] CAMPBELL, A. E. R., CATTO, G.L., AND HANSEN, E. E. 2003. Language-independent interactive data visualization. In *Proceedings of the 34th technical symposium on Computer science education (SIGCSE)*, February 19-23, 2003, Reno, Nevada, USA, pp. 215 – 219.
- [Cattaneo et al. 2002] CATTANEO, G., ITALIANO, G. F., AND FERRARO-PETRILLO, U. 2002. CATAI: Concurrent Algorithms and Data Types Animation over the Internet. *Journal of Visual Languages & Computing*, vol. 13, no. 4, Aug 2002, pp. 391-419.
- [Chen et al. 2003] CHEN, T., SOBH, T., AND TIBREWAL, A. 2003. A tool for data structure visualization and user-defined algorithm animation. *Journal of STEM Education Innovations and Research*, vol. 4, July – December 2003.
- [Colaso et al. 2002] COLASO, V., KAMAL, A., SARAIYA, P., NORTH, C., MCCRICKARD, S., AND SHAFFER, C.A. 2002. Learning and Retention in Data Structures: A Comparison of Visualization, Text, and Combined Methods. In *Proceedings of ED-MEDIA 2002*, Denver CO, June 2002.
- [Costigan et al. 2002] COSTIGAN, J., WILHITE, B. , AND NORTH, C. 2002. Data structure visualization with visual debugger: A tool for automatic visualization of run-time data structures. Online. Internet. September 2002. Available WWW: <http://infovis.cs.vt.edu/datastruct/>.
- [CS curriculum 2005] CS CURRICULUM. 2005. Bachelor of Science in Computer Science Curriculum. Available At <http://eng.auburn.edu/files/file320.pdf>
- [Davis 1971] DAVIS, JAMES ALLAN. 1971. *Elementary Survey Analysis*, Prentice-Hall. ISBN: 0132605473.
- [Dershem et al. 2002] DERSHEM, H.L., MCFALL, R.L., AND UTI, N. 2002. Animation of Java linked lists. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, February 27-March 03, 2002, Cincinnati, Kentucky, pp. 53-57.
- [Douglas et al. 1996] DOUGLAS S.A., HUNDHAUSEN, C.D. & MCKEOWN, D. 1996. Exploring Human Visualization of Computer Algorithms. *Graphics Interface Proceedings*, Toronto, Canada: Canadian Human-Computer Communications Society, pp. 9-16.
- [Felder 1993] FELDER, R.M. 1993. Reaching the Second Tier: Learning and Teaching Styles in College Science Education. *Journal of College Science Teaching*, 1993, vol. 23, pp. 286-290.

- [Felder and Silverman 1988] FELDER, R.M., AND SILVERMAN, L.K. 1988. Learning and Teaching Styles in Engineering Education. *Engr. Education*, 1988, vol. 78, pp. 674-681.
- [Gloor 1998a] GLOOR, P. 1998. Animated Algorithms. In *Software Visualization: Programming as a Multimedia Experience*, the MIT Press, 1998, pp. 409-416.
- [Gloor 1998b] GLOOR, P. 1998. User Interface Issues for Algorithm Animation. In *Software Visualization: Programming as a Multimedia Experience*, the MIT Press, 1998, pp. 145-152.
- [Grissom et al. 2003] GRISSOM, S., MCNALLY, M. F., AND NAPS, T. 2003. Algorithm visualization in CS education: comparing levels of student engagement. *ACM Press*, 2003, pp. 87-94.
- [Gurka and Citrin 1996] GURKA, J. S., AND CITRIN, W. 1996. Testing effectiveness of algorithm animation. In *Proceedings of the IEEE Symposium on Visual Language*, Boulder, CO: IEEE, pp. 182-189.
- [Hamer 2004a] HAMER, J. 2004. Visualising Java data structures as graphs. In *Proceedings of the Sixth Conference on Australasian Computing Education*, volume 30 (Dunedin, New Zealand), pp. 125-129.
- [Hamer 2004b] HAMER, J. 2004. A Lightweight Visualizer for Java. In *Third Program Visualization Workshop*, University of Warwick, July 1-2, 2004.
- [Hamilton-Taylor and Kraemer 2002] HAMILTON-TAYLOR, A. G., AND KRAEMER, E. 2002. SKA: Supporting algorithm and data structure discussion. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, Feb 2002, volume 34, issue 1, pp 58-63.
- [Hanciles et al. 1997] HANCILES, B., SHANKARARAMAN, V., AND MUNOZ, J. 1997. Multiple representation for understanding data structures. *Computers and Education*, August 1997, vol. 29, issue 1, pp. 1-11.
- [Hansen et al. 2000] HANSEN, S. R., NARAYANAN, N. H., AND SCHRIMPSHER, D. 2000. Helping learners visualize and comprehend algorithms. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, vol. 2, issue 1, May 2000, Association for the Advancement of Computing in Education.
- [Hansen et al. 2002] HANSEN, S. R., NARAYANAN, N. H., AND HEGARTY, M. 2002. Designing Educationally Effective Algorithm Visualizations: Embedding Analogies and Animations in Hypermedia. *Journal of Visual Languages and Computing*, 2002, vol. 13, pp. 291-317.

- [Hendrix et al. 2004] 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 (SIGCSE) 2004*, pp.387-391.
- [Hundhausen 1998] HUNDHAUSEN, C.D. 1998. Toward Effective Algorithm Visualization Artifacts: Designing for Participation and Negotiation in an Undergraduate Algorithms Course. In *CHI 98 Summary*, New York: ACM Press, pp. 54-55.
- [Hundhausen 2002] HUNDHAUSEN, C.D. 2002. Integrating Algorithm Visualization Technology into an Undergraduate Algorithms Course: Ethnographic Studies of a Social Constructivist Approach. *Computers & Education*, vol. 39, issue3, pp. 237-260.
- [Hundhausen and Douglas 2000] HUNDHAUSEN, C.D., AND DOUGLAS, S.A. 2000. Using Visualizations to Learn Algorithms: Should Students Construct Their Own, or View an Expert's? In *2000 IEEE Symposium on Visual Languages*, Los Alamitos, CA, IEEE Computer Society Press, pp. 21-28.
- [Hundhausen et al. 2002] HUNDHAUSEN C., DOUGLAS S., STASKO J. T. 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 2002, vol. 13, pp. 259-290.
- [Jain et al. 2005a] JAIN, J., CROSS, J., AND HENDRIX, D. 2005. Qualitative Comparison of Systems Facilitating Data Structure Visualization. *Proceedings of the 43rd Southeast ACM Conference*. Kennesaw, GA.
- [Jain et al. 2005b] 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 et al. 2006] JAIN, J., CROSS, J., HENDRIX, D., AND BAROWSKI, L. (2006. Experimental Evaluation of Animated-Verifying Object Viewers for Java. *ACM Symposium on Software Visualization (SoftVis)*, September 4-5, Brighton, UK, 2006.
- [Jarc and Feldman 1998] JARC, D. J., AND FELDMAN, M. B. 1998. An empirical study of web-based algorithm animation courseware in an ada data structure course. In *Proceedings of the 1998 annual ACM SIGAda international conference on Ada*, Washington, D.C., pp-68-74.
- [Jarc et al. 2000] JARC, D. J., FELDMAN, M.B., AND HELLER, R. S. 2000. Assessing the benefits of interactive prediction using Web-based algorithm animation courseware. *SIGCSE Bulletin*, ACM Press, 2000, vol. 32, issue 1, pp. 377-381.
- [Jive 2002] JIVE. Available at <http://jive.dia.unisa.it/>
- [Jsave 2003] JSAVE. Available at <http://www.cs.hope.edu/jsave/>

- [Kann et al. 1997] KANN, C., LINDEMAN, R. W., AND HELLER, R. 1997. Integrating algorithm animation into a learning environment. *Computer Education*, Elsevier Science Ltd., 1997, vol. 28, pp. 223-228.
- [Karavirta et al. 2002] KARAVIRTA, V., KORHONEN, A., NIKANDER, J., AND TENHUNEN, P. 2002. Effortless Creation of Algorithm Visualization. *Proceedings of the Second Annual Finnish / Baltic Sea Conference on Computer Science Education*. pp. 52–56.
- [Karavirta et al. 2004a] KARAVIRTA, V., KORHONEN, A., MALMI, L., AND STÅLNACKE, K. 2004. MatrixPro - A Tool for Demonstrating Data Structures and Algorithms Ex Tempore. *Proceedings of the 4th IEEE International Conference on Advanced Learning Technologies*, Joensuu, Finland, pp. 892–893.
- [Karavirta et al. 2004b] KARAVIRTA, V., KORHONEN, A., MALMI, L., AND STÅLNACKE, K. 2004. MatrixPro - A Tool for On-The-Fly Demonstration of Data Structures and Algorithms. *Proceedings of the Third Program Visualization Workshop*, Department of Computer Science, University of Warwick, UK, The University of Warwick, UK, pp. 26–33.
- [Kehoe et al. 1999] KEHOE, C., STASKO, J., AND TAYLOR, A. 1999. Rethinking the evaluation of algorithm animations as learning aids: an observational study. *International Journal of Human-Computer Studies*, vol. 54, no. 2, February 2001, pp. 265-284.
- [Khuri 2001] KHURI, S.A. 2001. User-Centered Approach for Designing Algorithm Visualizations. *Informatik / Informatique, Special Issue on Visualization of Software*, Apr. 2001, pp. 12-16.
- [Korhonen et al. 2004] KORHONEN, A., MALMI, L., SILVASTI, P., KARAVIRTA, V., LÖNNBERG, J., NIKANDER, J., STÅLNACKE, K. AND IHANTOLA, P. 2004. Matrix - A Framework for Interactive Software Visualization. *Research Report*, Laboratory of Information Processing Science, Department of Computer Science and Engineering, Helsinki University of Technology. TKO-B 154/04.
- [Korn and Appel 1998] KORN, J., AND APPEL, A. 1998. Traversal-based Visualization of Data Structures. *IEEE Information Visualization 1998*, October 1998, pp. 11-18.
- [Kraemer and Stasko 1993] KRAEMER, E., AND STASKO, J. T. 1993. The visualization of parallel systems: an overview. *Journal of Parallel and Distributed Computing*, vol. 18, no. 2, June 1993, pp. 105-117.
- [Lattu et al. 2000] LATTU, M., TARHIO, J., AND MEISALO, V. 2000. How a Visualization Tool Can Be Used - Evaluating a Tool in a Research & Development Project. *12th Workshop of the Psychology of Programming Interest Group*, pp. 19–32.

- [Lawal 2003] LAWAL, BAYO. 2003. Categorical Data Analysis with SAS and SPSS Applications, *Lawrence Erlbaum Associates*. ISBN: 0805846050.
- [Lawrence 1993] LAWRENCE, A. W. 1993. Empirical studies of the value of algorithm animation in algorithm understanding. Ph.D. Thesis, Georgia Institute of Technology, 1993.
- [Lawrence 1994] LAWRENCE, A.W., BADRE, A.M., AND STASKO, J.T. 1994. Empirically Evaluating the Use of Animations to Teach Algorithms. In *Proceedings of 1994 IEEE Symposium on Visual Languages*, St. Louis, Missouri, Oct. 1994, pp. 48-54.
- [Levy et al. 2003] LEVY, R. B., BEN-ARI, M., AND URONEN, P. A. 2003. The Jeliot 2000 program animation system. *Comput. Educ., Elsevier Science Ltd.*, 2003, vol. 40, pp. 1-15.
- [Likert 1932] LIKERT, RENSIS. 1932. A Technique For The Measurement Of Attitudes, *Archives of Psychology*, 140, 5-53.
- [Lönnberg et al. 2004] LÖNNBERG, J., KORHONEN, A., AND MALMI, L. 2004. MVT - A system for visual testing of software. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI'04)*, May 25-28, Gallipoli, Italy, ACM, 2004, pp. 385-388.
- [Lucas et al. 2003] LUCAS, J., NAPS, T. L. AND RÖBLING, G. 2003. VisualGraph - A Graph Class Designed for Both Undergraduate Students and Educator. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, Reno, Nevada, USA, pp. 167 – 171.
- [Moreno and Niko 2003] MORENO, A. AND MYLLER, N. 2003. Producing an Educationally Effective and Usable Tool for Learning, The Case of the Jeliot Family. In *Proceedings of International Conference on Networked e-learning for European Universities* (Granada, Spain).
- [Moreno et al. 2004] MORENO, A., MYLLER, N., SUTINEN, E., AND BEN-ARI, M. 2004. Visualizing Programs with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces AVI 2004*, Gallipoli (Lecce), Italy, 25-28 May, 2004.
- [Morrison et al. 2000] MORRISON, J. B., TVERSKY, B., AND BÉTRANCOURT, M. 2000. Animation: Does it facilitate learning? In *Proceedings of the AAAI 2000 Spring Symposium Smart Graphics*, 20-22 March 2000, Stanford, CA, USA, pp 53-60.
- [Mukherjea and Stasko 1993] MUKHERJEA, S., AND STASKO, J.T. 1994. Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding. In *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, MD, May 1993, pp. 456-465.

- [Mukherjea and Stasko 1994] MUKHERJEA, S., AND STASKO, J.T. 1994. Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source Level Debugger. *ACM Transactions on Computer-Human Interaction*, volume 1, issue 3, September 1994, pp. 215-244.
- [Murphy and Myors 2004] MURPHY, K.R. AND MYORS, B. 2004. Statistical Power Analysis: A Simple and General Model for Traditional and Modern Hypothesis Tests, 2nd Edition. *Lawrence Erlbaum Associates, Inc.*
- [Myers 1983] MYERS, B.A. 1983. INCENSE: A system for displaying data structures. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, July 25-29, 1983, Detroit, Michigan, United States, pp. 115-125.
- [Myers 1986] MYERS, B.A. 1986. Visual programming, programming by example, and program visualization: a taxonomy. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, Boston, Massachusetts, pp. 59 – 66.
- [Myers 1990] MYERS, B.A. 1990. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, vol. 1, no. 1, 1990, pp. 97-123.
- [Naps and Bressler 1998] NAPS, T. L. AND BRESSLER, E. 1998. A multi-windowed environment for simultaneous visualization of related algorithms on the World Wide Web. In *Proceedings of the SIGCSE Session*, ACM Meetings, Atlanta, Georgia, February, 1998.
- [Naps et al. 2000] NAPS, T.L., EAGAN, J.R., AND NORTON, L.L. 2000. JHAVÉ—an environment to actively engage students in Web-based algorithm visualizations. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, March 07-12, 2000, Austin, Texas, United States, pp.109-113.
- [Naps et al. 2003a] NAPS, T.L., RÖBLING, G., ALMSTRUM, V., DANN, W., FLEISCHER, R., HUNDHAUSEN, C., KORHONEN, A., MALMI, L., MCNALLY, M., RODGER, S.H., AND VELAZQUEZ-ITURBIDE, J.A. 2003. Exploring the Role of Visualization and Engagement in Computer Science Education. *Inroads - Paving the Way Towards Excellence in Computing Education*, ACM Press, New York, 2003, pp. 131-152.
- [Naps et al. 2003b] NAPS, T., COOPER, S., KOLDEHOFE, B., LESKA, C., RÖBLING, G., DANN, W., KORHONEN, A., MALMI, L., RANTAKOKKO, J., ROSS, R. J., ANDERSON, J., FLEISCHER, R., KUITTINEN, M., AND MCNALLY, M. 2003. Evaluating the educational impact of visualization. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, ACM Press, 2003, pp. 124-136.

- [Pierson and Rodger 1998] PIERSON, W. C., AND RODGER, S.H. 1998. Web-based animation of data structures using JAWAA. *ACM SIGCSE Bulletin*, vol. 30, issue 1, ACM Press, March 1998, pp. 267-271.
- [Price et al. 1993] PRICE, B.A., BAECKER, R.M., AND SMALL, I.S. 1993. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, vol. 4, no. 3, 1993, pp. 211-266.
- [Rea 1997] REA, LOUIS M. 1997. Designing and Conducting Survey Research: A Comprehensive Guide, *Jossey-Bass*. ISBN: 078790810x.
- [Rodger 2002] RODGER, S.H. 2002. Introducing computer science through animation and virtual worlds. *ACM SIGCSE Bulletin*, vol.34, no.1, March 2002.
- [Roman and Cox 1993] ROMAN, G. C., AND COX, K. C. 1993. A taxonomy of program visualization systems. *IEEE Computer*, vol. 26, no. 12, December 1993, pp. 11-24.
- [Röbbling 2003] RÖBLING, G. 2003. Key Decisions in Adopting Algorithm Animations for Teaching. In *Informatics and the Digital Society*, Kluwer Academic Publishers, Boston / Dordrecht / London, 2003, pp. 149-156.
- [Röbbling and Freisleben 2000a] RÖBLING, G., AND FREISLEBEN, B. 2000. Approaches for Generating Animations In Lectures. In *Proceedings of the 11th International Society for Information Technology and Teacher Education (SITE 2000) Conference*, Association for the Advancement of Computers in Education (ACE), Charlottesville, VA, 2000, pp. 809-814.
- [Röbbling and Freisleben 2000b] RÖBLING, G., AND FREISLEBEN, B. 2000. Experiences in Using Animations in Introductory Computer Science Lectures. In *Proceedings of the ACM 31st SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000) Conference*, ACM Press, New York, 2000, pp. 134-138.
- [Röbbling and Freisleben 2002] RÖBLING, G., AND FREISLEBEN, B. 2002. ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, vol. 13, no. 3, Elsevier, Amsterdam, The Netherlands, 2002, pp. 341-354.
- [Röbbling and Naps 2002] RÖBLING, G. AND NAPS, T.L. 2002. A Testbed for Pedagogical Requirements in Algorithm Visualizations. In *Proceedings of the 7th Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2002)*, Aarhus, Denmark, ACM Press, 2002, pp. 96-100.
- [Röbbling and Naps 2002a] RÖBLING, G. AND NAPS, T.L. 2002. Towards Improved Individual Support in Algorithm Visualizations. In *Proceedings of the Second Program Visualization Workshop*, Aarhus, Denmark, pp. 125-130.

- [Rößling and Naps 2002b] RÖBLING, G., AND NAPS, T.L. 2002. Towards Improved Individual Support in Algorithm Visualizations. In *Proceedings of the Second Program Visualization Workshop*, Aarhus, Denmark, Department of Computer Science, University of Aarhus, Denmark, 2002, pp. 125-130.
- [Saraiya et al. 2004] SARAIYA, P., SHAFFER, C. A., MCCRICKARD, D. S., AND NORTH, C. 2002. Effective features of algorithm visualizations. In *Proceedings of the 35th SIGCSE technical symposium on Computer Science Education*, March 2004, pp. 382 – 386.
- [Scanlan 1987] SCANLAN, D. 1987. Data-structures students may prefer to learn algorithms using graphical methods. In *Proceedings of the eighteenth SIGCSE technical symposium on Computer science education*, St. Louis, Missouri, United States, pp. 302 – 307.
- [Shaffer et al. 1996] 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.
- [Shimomura and Isoda 1990] SHIMOMURA, T.; ISODA, S. 1990. VIPS: a visual debugger for list structures. In *Proceedings of the Fourteenth Annual International Computer Software and Applications Conference (COMPSAC '90)* 31 Oct.-2 Nov. 1990, pp. 530 – 537.
- [Shimomura and Isoda 1991] SHIMOMURA, T.; ISODA, S. 1991. Linked-list visualization for debugging. *IEEE Software*, volume 8, issue 3, May 1991, pp. 44 – 51.
- [Shu 1988] SHU N. C. 1988. *Visual programming*. Van Nostrand Reinhold Co., New York, NY, 1988.
- [Singh and Chignell 1992] SINGH, G., AND CHIGNELL M.H. 1992. Components of the visual computer: a review of relevant technologies. *Visual Computer* vol. 9, issue 3, November 1992, pp. 115-142.
- [Stasko 1990] STASKO, J.T. 1990. Tango: A Framework and System for Algorithm Animation. *Computer*, v.23 n.9, September 1990, pp. 27-39.
- [Stasko 1997] STASKO, J. T. 1997. Using student-built algorithm animations as learning aids. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, 1997, pp. 25-29.
- [Stasko 1998] STASKO, J.T. 1998. JSamba, described online at <http://www.cc.gatech.edu/gvu/softviz/algoanim/jsamba/>.
- [Stasko and Patterson 1992] STASKO, J.T., AND PATTERSON, C. 1992. Understanding and Characterizing Software Visualization Systems. In *Proceedings of the 1992 IEEE International Workshop on Visual Languages*, September 1992, pp. 3-10.

- [Stasko et al. 1993a] STASKO, J. T., BADRE, A., AND LEWIS, C. 1993. Do Algorithm Animations Assist Learning? An Empirical Study and Analysis. In *Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems*, Amsterdam, Netherlands, April 1993.
- [Stasko et al. 1993b] STASKO, J.T., BADRE, A., AND LEWIS, C. 1993. Do algorithm animations assist learning?: an empirical study and analysis. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, 1993, pp. 61-66.
- [Stasko et al. 1997] STASKO, J.T., BROWN, M.H., AND PRICE, B.A. 1997. *Software Visualization*. MIT Press, Cambridge, MA, 1997.
- [Stokes 2001] STOKES, MAURA ELLEN. 2001. Categorical Data Analysis Using the SAS System, *Wiley-Sas*. ISBN: 0471224243.
- [Tudoreanu 2003] TUDOREANU, M.E. 2003. Designing effective program visualization tools for reducing user's cognitive effort. In *Proceedings of the 2003 ACM symposium on Software visualization*, San Diego, California, pp. 105-114.
- [Upton 1978] UPTON, G. J. G. 1978. The Analysis of Cross-Tabulated Data, *John Wiley*. ISBN: 0471996599.
- [Whale 1994] WHALE, G. 1994. DRUIDS: Tools for understanding data structures and algorithms, In *Proceedings of the First IEEE Int. Conf on Multi-Media Engineering Education*, Melbourne, pp. 403-407.
- [Zeller 2001] ZELLER, A. 2001. Visual debugging with DDD. *Dr. Dobbs' Journal*, March 2001.
- [Zeller and Lütkehaus 1996] ZELLER, A., AND LÜTKEHAUS, D. 1996. DDD—a free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, volume 31, issue 1, January 1996, pp. 22 – 27.

APPENDICES

Appendix A - Survey for data structure understanding (Fall 2004/Spring 2005)

**COMP 2210 Fundamentals of Computer Science II
DATA STRUCTURE UNDERSTANDING SURVEY**

We are in the process of developing data structure viewers for jGRASP such that students can use it in two ways:

1. For learning the basic concepts of how a data structure is built and modified (without writing any code)
2. For synchronously visualizing data structures and the code being used to implement it.

What is the purpose of this survey?

In this survey we are trying to understand/gauge three things – which data structures are:

- a) difficult to understand – at an abstract level?
- b) difficult to code?
- c) difficult to re-use in an application?

After we identify the data structures which are most difficult to understand, we will then conduct another survey to identify exactly which operations are most problematic.

Contact Jhilmil Jain (jainjhi@auburn.edu) if you are interested in the results of this survey.

1) Please circle the degree and year that you are enrolled in.

1) Undergraduate degree

- i. 1 year ii. 2 year iii. 3 year iv. 4 year v. 4+ year

2) Masters degree

- i. 1 year ii. 2 year iii. 3 year iv. 4 year v. 4+ year

3) Doctoral degree

- i. 1 year ii. 2 year iii. 3 year iv. 4 year v. 4+ year

4) Other: _____

2) What is your major? (please do not use abbreviations)

3) Was your level of Java experience appropriate for this class?

- a) Yes
- b) No

Comments: _____

- 4) Rate the *ease of understanding* of data structures on a 1 to 5 scale, where 1 means that the data structure was **very hard** to understand and 5 means that it was **very easy** to understand. 0 means that the data structure was not covered in class.

Legend:

| 0 | 1 | 2 | 3 | 4 | 5 |
|----------------------|-------------------------|--------------------|---|--------------------|-------------------------|
| not covered in class | very hard to understand | hard to understand | not too hard yet not easy to understand | easy to understand | very easy to understand |

| Data Structure | Easy to understand conceptually | Easy to code or write a program | Ease to create applications using it |
|---|---------------------------------|---------------------------------|--------------------------------------|
| 1) List – array implementation | | | |
| 2) List – pointer or linked implementation | | | |
| 3) Stack - array implementation | | | |
| 4) Stack - pointer or linked implementation | | | |
| 5) Queue - array implementation | | | |
| 6) Queue - pointer or linked implementation | | | |
| 7) Dictionary | | | |
| 8) Hash Table | | | |
| 9) Tree | | | |
| a) Binary Search Tree | | | |
| b) Expression Tree | | | |
| c) Decision Tree | | | |
| d) Parse Tree | | | |
| e) Game Tree | | | |
| f) Balanced Search Tree | | | |
| 10) Heap | | | |
| 11) Graph | | | |
| a) Adjacency Matrix | | | |
| b) Linked Adjacency Lists | | | |
| c) Array Adjacency Lists | | | |
| d) Spanning Tree | | | |
| e) Minimum Cost Spanning Tree | | | |

- 5) Comments/Suggestions

Appendix B - Interview results for data structure understanding (Fall 2004/Spring 2005)

Hardware issues

- Lab machines are too slow
- Learning how to use a new operating system (Solaris) was okay, but the browser (Mozilla) takes 3-4 minutes to load
- Netscape does not support the engineering website or Webct

Software implementation issues

1. Transition from concept to implementation is hard; data structures are not difficult to understand conceptually, but implementation is tough
2. Examples of code during lectures are helpful
3. Debugger's user interface needs to be worked on:
 - a. The button on the debugger interface are not very not intuitive
 - b. Sometimes the debugger does not do what I want; or it is difficult for me to reproduce what I had done in the past.
 - c. When I debug the basic view shows me so many variables that it is difficult to find what I am looking for. For example, if I am developing a linked list, the basic view throws out so many fields. All I care about is the data value that the node holds and the pointer to the next node.
 - d. Viewers must match the ones that we see in the textbook
 - e. Adding breakpoints is hard
 - f. The Threads tab is confusing
4. Big jump from Java 1 to Java 2 – difficult to handle such large and complex programs
5. The feedback from all students was that being able to visualize data structures during debugging would be a great plus. A lot of students also wanted to see the transition step by step of how an operation is performed on a data structure. This would be helpful as they are learning about a new data structure. Should be easy to turn the feature on and off easily.

jGRASP usability issues

1. How to make the API more visible – add hot keys, useful to open up API in the default browser
2. CSD should be more intelligent, instead of not being generated it should give some indication of what the error might be
3. Matching braces should be highlighted – some suggested adding a closing brace for every open brace typed.

Appendix C - Test 1: Questions to test error detection and correction

Code Number: _____

Section: 1 2 3 4

Note: Your answers to these questions WILL NOT affect your grade in any way. The goal of this test is to help us survey your expertise in being able to detect a logical or syntax error and being able to correct it. Your combined performance in Test 1 and Test 2 (will be given out next week) will be used to assign you to a particular group that will be maintained throughout the semester and will be used for in-lab activities.

1. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```
1   public class Q1 {
2       public static void main(String[] args) {
3           double radius;
4           double area = radius * radius * Math.PI;
5           System.out.println("Area is " + area);
6       }
7   }
```

- a.
- b.
- c.

2. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```
1   public class Q2 {
2       int x;
3
4       public Q2(String t) {
5           System.out.println("Test");
6       }
7
8       public static void main(String[] args) {
9           Q2 test = null;
10          System.out.println(test.x);
11      }
12  }
```

- a.
- b.
- c.

3. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```
1   class Q3{
2       void method (int a, int b, int c){}
3   }
4
```

```

5     class Driver{
6         public static void main(String[] args) {
7             Q3 a = new Q3();
8             a.method (5, 10);
9         }
10    }

```

- a.
- b.
- c.

4. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```

1     class Q4{
2         void method (int a, String b, double c){}
3     }
4
5     class Driver{
6         public static void main(String[] args) {
7             Q4 a = new Q4();
8             a.method ("cat", 10, 35.56);
9         }
10    }

```

- a.
- b.
- c.

5. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```

1     class Q5 {
2         void method (int a){}
3     }
4
5     class Driver{
6         public static void main(String[] args) {
7             Q5 a = new Q5();
8             a.method (5, 10);
9         }
10    }

```

- a.
- b.
- c.

6. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```

1     class Q6 {
2         int method (){
3         }
4     }

```

```

5
6     class Driver{
7         public static void main(String[] args) {
8             Q6 a = new Q6();
9             a.method ();
10        }
11    }

```

- a.
- b.
- c.

7. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```

1     class Q7{
2         int method (){
3             return true;
4         }
5     }
6
7     class Driver{
8         public static void main(String[] args) {
9             Q7 a = new Q7();
10            a.method ();
11        }
12    }

```

- a.
- b.
- c.

8. The following program determines the area and circumference of a circle. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```

1     class Circle10{
2         double radius;
3
4         Circle10(double r){
5             radius = r;
6         }
7
8         double area(){
9             return (Math.PI * radius * radius);
10        }
11
12        double circumference (){
13            return (2 * Math.PI * radius);
14        }
15
16        public static void main(String[] args) {
17            Circle10 a = new Circle10 (5);
18            System.out.println ("Area = " + a.circumference ());
19            System.out.println ("Circumference = " + a.area());

```

```
20     }
21 }
```

- a.
- b.
- c.

9. The following program determines the area and circumference of a circle. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```
1     class Circle11{
2         private double radius, area, cir;
3
4         Circle11(double r){
5             radius = r;
6         }
7
8         void area(){
9             cir = Math.PI * radius * radius;
10        }
11
12        void circumference (){
13            area = 2 * Math.PI * radius;
14        }
15
16        double getArea(){
17            return area;
18        }
19
20        double getCircumference (){
21            return cir;
22        }
23
24        public static void main(String[] args) {
25            Circle11 a = new Circle11 (5);
26            System.out.println ("Area = " + a.getArea());
27            System.out.println ("Circumference = " +
a.getCircumference());
28        }
29    }
```

- a.
- b.
- c.

10. The following program determines the area and circumference of a circle. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```
1    class Circle12{
2        private double radius, area, cir;
3
4        Circle12(double r){
5            radius = r;
6            area();
7            circumference();
8        }
9
10       void area(){
11           area = Math.PI * radius * radius;
12       }
13
14       void circumference (){
15           cir = 2 * Math.PI * radius;
16       }
17
18       double getArea(){
19           return cir;
20       }
21
22       double getCircumference (){
23           return area;
24       }
25
26       public static void main(String[] args) {
27           Circle12 a = new Circle12 (5);
28           System.out.println ("Area = " + a.getArea());
29           System.out.println ("Circumference = " +
30 a.getCircumference());
31       }
```

- a.
- b.
- c.

11. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```
1     class Q11{
2         public static void main (String args[]){
3             int res = MyMath.add ("A", "B", "C");
4         }
5     }
6
7     class MyMath{
8         static int add (int x, int y, int z){
9             return (x+y+z);
10        }
11
12        static String add (String x, String y, String z){
13            return (x+y+z);
14        }
15    }
```

- a.
- b.
- c.

12. Is there an error in this program? If yes, a) specify the type of error (compiler / run-time / logical), and b) how would you correct the error?

```
1     interface Q12 {
2         int add (int a, int b);
3
4         int subtract (int a, int b);
5     }
6
7     class Temp implements Q12{
8
9         public int add (int a, int b){
10            return (a + b);
11        }
12    }
```

- a.
- b.

13. Does the following program print out this pattern? If not, a) specify the type of error (compiler or runtime or logical), b) specify the line number(s) and c) how would you correct the error so that the following pattern is printed?

```
*****
*****
***
```

```
1     class Q13{
2         public static void main (String args[]){
3
4             int count = 5;
5             for (int i=0; i<3; i++){
6                 for (int j = count; j>1; j--)
7                     System.out.print ("*");
8                 count--;
9                 System.out.println();
10            }
11        }
12    }
```

- a.
- b.
- c.

14. Does the following program print out this pattern? If not, a) specify the type of error (compiler or runtime or logical), b) specify the line number(s) and c) how would you correct the error so that the following pattern is printed?

```
*****
*****
***
```

```
1     class Q14{
2         public static void main (String args[]){
3
4             int count = 5;
5             for (int i=1; i<3; i++){
6                 for (int j = count; j>0; j--)
7                     System.out.print ("*");
8                 count--;
9                 System.out.println();
10            }
11        }
12    }
```

- a.
- b.
- c.

15. Does the following program print out this pattern? If not, a) specify the type of error (compiler or runtime or logical), b) specify the line number(s) and c) how would you correct the error so that the following pattern is printed?

```
*****
*****
***
```

```
1      class Q15{
2          public static void main (String args[]){
3
4              int count = 5;
5              for (int i=0; i<=3; i++){
6                  for (int j = count; j>0; j--)
7                      System.out.print ("*");
8                  count--;
9                  System.out.println();
10             }
11         }
12     }
```

- a.
- b.
- c.

16. Does the following program print out this pattern? If not, a) specify the type of error (compiler or runtime or logical), b) specify the line number(s) and c) how would you correct the error so that the following pattern is printed?

```
*****
*****
***
```

```
1      class Q16{
2          public static void main (String args[]){
3
4              int count = 5;
5              for (int i=0; i<3; i++){
6                  for (int j = count; j>0; j++)
7                      System.out.print ("*");
8                  count--;
9                  System.out.println();
10             }
11         }
12     }
```

- a.
- b.
- c.

17. The following program is supposed to count vowels. Does this program work as expected? If not, a) specify the type of error (compiler or run-time or logical), b) specify the line number(s) and c) how would you correct the error?

```
1   import java.util.Scanner;
2
3   class Q17 {
4       static int a,e,i,o,u;
5
6       public static void main (String args[]){
7
8           Scanner scan = new Scanner (System.in);
9           System.out.println ("Enter String: ");
10          String line = scan.next();
11
12          for (int x=0; x<line.length(); x++){
13
14              switch (line.charAt(i)){
15                  case 'a': case 'A': a++;
16                      break;
17
18                  case 'e': case 'E': e++;
19                      break;
20
21                  case 'i': case 'I': i++;
22                      break;
23
24                  case 'o': case 'O': o++;
25                      break;
26
27                  case 'u': case 'U': u++;
28                      break;
29              }
30          }
31      }
32  }
```

- a.
- b.
- c.

18. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```
1   class Q18{
2       public static void main (String args[]){
3
4           int x = 5;
5           double y = 4.0;
6
7           x = y/4;
8
9       }
10  }
```

- a.

b.

c.

19. The following program extracts the last digit of the number. Is there an error in this program? If yes, a) specify the type of error (compiler or run-time or logical), b) locate the line number(s) where it occurs, and c) how would you correct the error?

```
1    class Q19{
2        public static void main (String args[]){
3
4            int number = 1234;
5
6            int last = number/10;
7            System.out.println ("Last digit = "+ last);
8        }
9    }
```

a.

b.

c.

20. Is there an error in this program? If yes, a) specify the type of error (compiler / run-time / logical), b) locate the line number(s) and c) how would you correct the error?

```
1    clas Q20{
2        public static void main (String args[]){
3
4            System.out.println ("Hello World")
5        }
6    }
```

a.

b.

c.

Appendix D - Test 2: Questions to test program understanding and tracing

Code Number: _____

Section: 1 2 3 4

Note: Your answers to these questions WILL NOT affect your grade in any way. The goal of this test is to help us survey your expertise in program tracing. Your combined performance in Test 1 (was given last week) and Test 2 will be used to assign you to a particular group that will be maintained throughout the semester and will be used for in-lab activities.

Question 1: Consider the following code fragment:

```
int[] x = {2, 1, 4, 5, 7};
int limit = 3;
int i = 0;
int sum = 0;
while ( (sum<limit) && (i<x.length)){
    ++i;
    sum += x[i];
}
```

What value is in the variable “i” after this code is executed?

- a) 0 b) 1 c) 2 d) 3

Question 2: Consider the following code fragment:

```
int[] x1 = {1, 2, 4, 7};
int[] x2 = {1, 2, 5, 7};
int i1 = x1.length-1;
int i2 = x2.length-1;
int count = 0;
while ((i1 > 0 ) && (i2 > 0 ))
{
    if ( x1[i1] == x2[i2] )
    {
        ++count;
        --i1;
        --i2;
    }
    else if (x1[i1] < x2[i2])
    {
        --i2;
    }
    else
    { // x1[i1] > x2[i2]
        --i1;
    }
}
```

After the above while loop finishes, “count” contains what value?

- a) 3 b) 2 c) 1 d) 0

Question 3: Consider the following code fragment:

```
int [] x = {1, 2, 3, 3, 3};
boolean b[] = new boolean[x.length];
```

```

for ( int i = 0; i < b.length; ++i )
    b[i] = false;
for ( int i = 0; i < x.length; ++i )
    b[ x[i] ] = true;
int count = 0;
for (int i = 0; i < b.length; ++i )
{
    if ( b[i] == true ) ++count;
}

```

After this code is executed, "count" contains:

- a) 1 b) 2 c) 3 d) 4 e) 5

Question 4: Consider the following code fragment:

```

int[ ] x1 = {0, 1, 2, 3};
int[ ] x2 = {1, 2, 2, 3};
int i1 = 0;
int i2 = 0;
int count = 0;
while ( (i1 < x1.length) &&
(i2 < x2.length))
{
    if ( x1[i1] == x2[i2] )
    {
        ++count;
        ++i2;
    }
    else if (x1[i1] < x2[i2])
    {
        ++i1;
    }
    else
    { // x1[i1] > x2[i2]
        ++i2;
    }
}

```

After this code is executed, "count" contains:

- a) 0 b) 1 c) 2 d) 3 e) 4

Question 5: Consider the following code fragment:

```
int[] x = {0, 1, 2, 3};
int temp;
int i = 0;
int j = x.length-1;
while (i < j)
{
    temp = x[i];
    x[i] = x[j];
    x[j] = 2*temp;
    i++;
    j--;
}
```

After this code is executed, array “x” contains the values:

- a) {3, 2, 2, 0} b) {0, 1, 2, 3} c) {3, 2, 1, 0} d) {0, 2, 4, 6} e) {6, 4, 2, 0}

Question 6: Consider the following code fragment:

```
int[] x = {2, 1, 4, 5, 7};
int limit = 7;
int i = 0;
int sum = 0;
while ( (sum<limit) && (i<x.length) )
{
    sum += x[i];
    ++i;
}
```

What value is in the variable “i” after this code is executed?

- a) 0 b) 1 c) 2 d) 3 e) 4

Question 7: Consider the following code fragment:

```
int[] array1 = {2, 4, 1, 3};
int[] array2 = {0, 0, 0, 0};
int a2 = 0;
for (int a1=1; a1<array1.length; ++a1)
{
    if ( array1[a1] >= 2 )
    {
        array2[a2] = array1[a1];
        ++a2;
    }
}
```

After this code is executed, the array “array2” contains what values?

- a) {4, 3, 0, 0} b) {4, 1, 3, 0} c) {2, 4, 3, 0} d) {2, 4, 1, 3}

Question 8: The skeleton code below is intended to copy into an array of integers called “array2” any numbers in another integer array “array1” that are even numbers. For example, if “array1” contained the numbers:
array1: 4 5 6 2 1 3

then after the copying process, “array2” should contain in its first three places:
array2: 4 6 2

The following code assumes that “array2” is big enough to hold all the even numbers from “array1”:

```
int a2 = 0;
for ( int a1=0 ; xxx1xxx ; ++a1 )
{
    // if array1[a1] is even
    if ( array1[a1] % 2 == 0 )
    {
        // array1[a1] is even,
        // so copy it
        xxx2xxx;
        xxx3xxx;
    }
}
```

The missing pieces of code “xxx1xxx”, “xxx2xxx” and “xxx3xxx” in the above code should be replaced respectively by:

a) a1<array1.length
++a2
array2[a2] = array1[a1]

b) a1<array1.length
array2[a2] = array1[a1]
++a2

c) a1<=array1.length
array2[a2] = array1[a1]
++a2

d) a1<=array1.length
++a2
array2[a2] = array1[a1]

Hint: in all four options above, the second and third parts are the same, just reversed.

Appendix E - Activity to familiarize students with jGRASP debugger

Group 1 – Without Object Viewers

Today's goal is to learn how to use the jGRASP visual debugger. This is required for future Lab Activities.

Step 1: Right click anywhere on the desktop -> Right click on Shells -> Left click on Shell Tool. A shelltool will open.



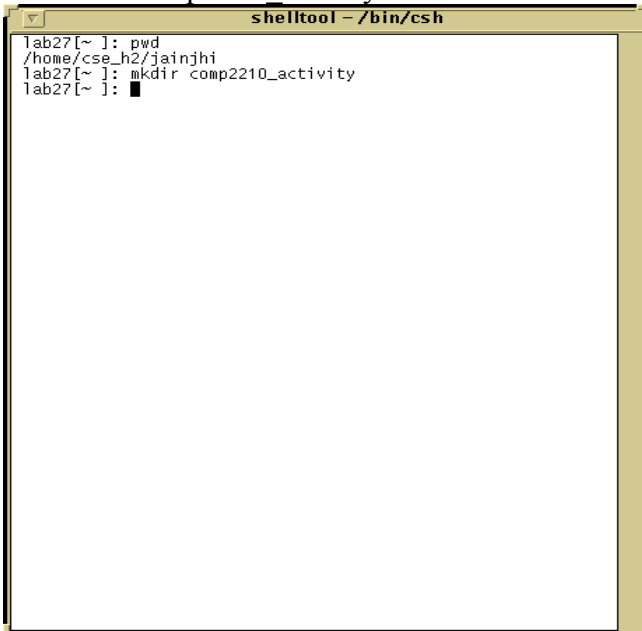
Step 2: Type the following command `pwd` and click the “Return” key to confirm that you are in your home directory

>> `pwd`

This should return your home path -> `/home/u2/yourusername`

Step 3: Type the following command and click the “Return” key to create the directory `comp2210_activity` in your home directory.

>> `mkdir comp2210_activity`



Step 4: Type the following command (cd – change directory) and click the “Return” key to enter the new directory

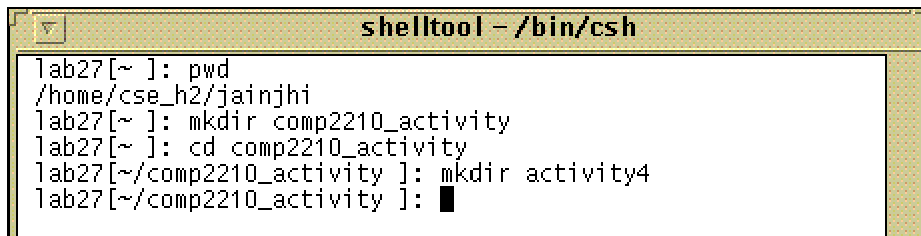
```
>> cd comp2210_activity
```



```
shelltool - /bin/csh
lab27[~ ]: pwd
/home/cse_h2/jainjhi
lab27[~ ]: mkdir comp2210_activity
lab27[~ ]: cd comp2210_activity
lab27[~/comp2210_activity ]: █
```

Step 5: Now type the following command and click the “Return” key to create a sub-directory for today’s assignment.

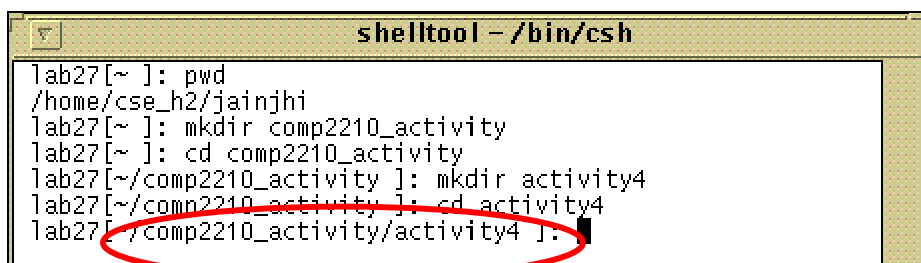
```
>> mkdir activity4
```



```
shelltool - /bin/csh
lab27[~ ]: pwd
/home/cse_h2/jainjhi
lab27[~ ]: mkdir comp2210_activity
lab27[~ ]: cd comp2210_activity
lab27[~/comp2210_activity ]: mkdir activity4
lab27[~/comp2210_activity ]: █
```

Step 6: Type the following command and click the “Return” key to confirm that the directory activity4 was created.

```
>> cd activity4
```



```
shelltool - /bin/csh
lab27[~ ]: pwd
/home/cse_h2/jainjhi
lab27[~ ]: mkdir comp2210_activity
lab27[~ ]: cd comp2210_activity
lab27[~/comp2210_activity ]: mkdir activity4
lab27[~/comp2210_activity ]: cd activity4
lab27[~/comp2210_activity/activity4 ]: █
```

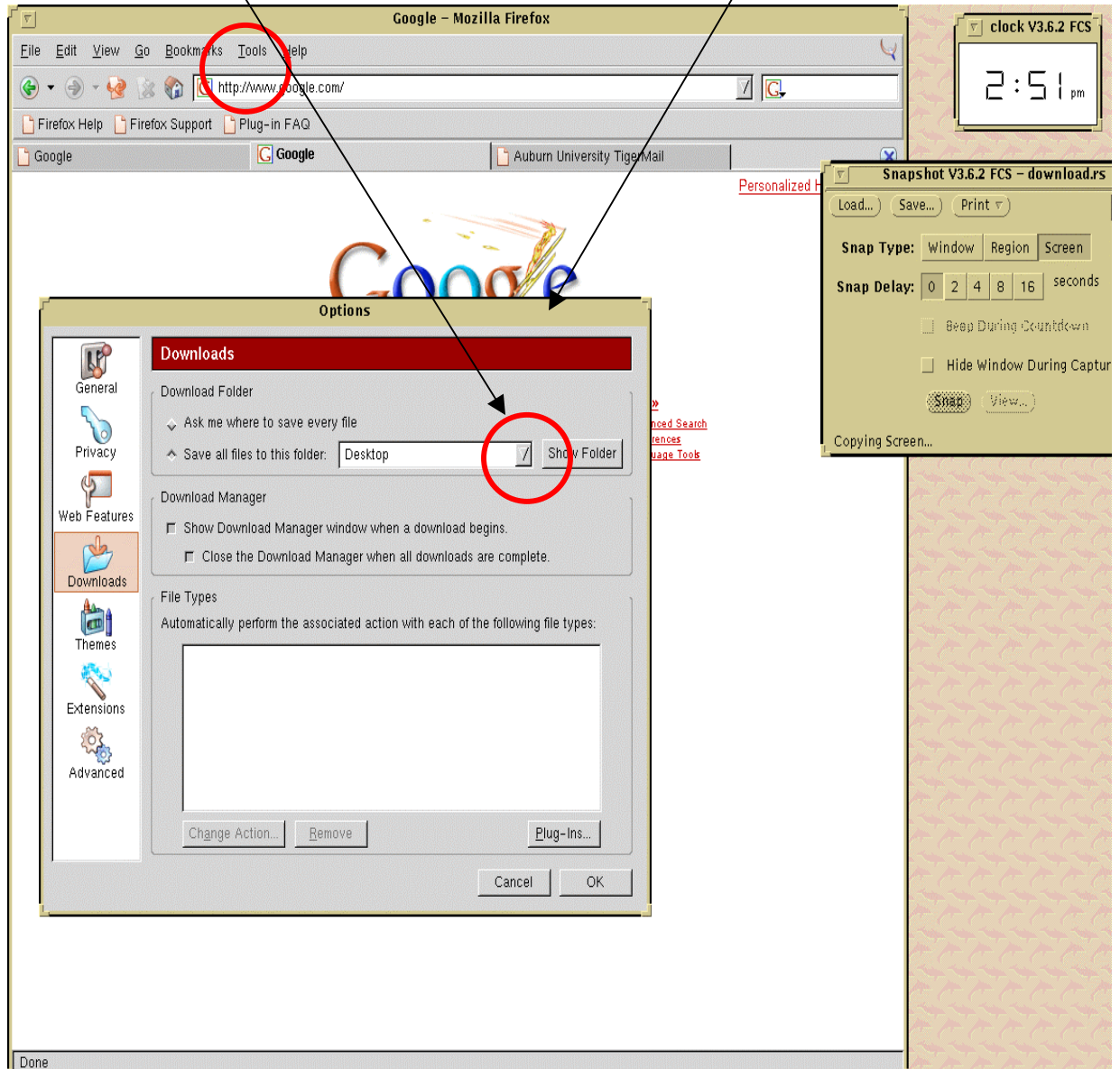
Confirm that the circled path is visible in your shelltool.

This is where you must save the programs for today. **DO NOT CLOSE THIS SHELL TOOL**

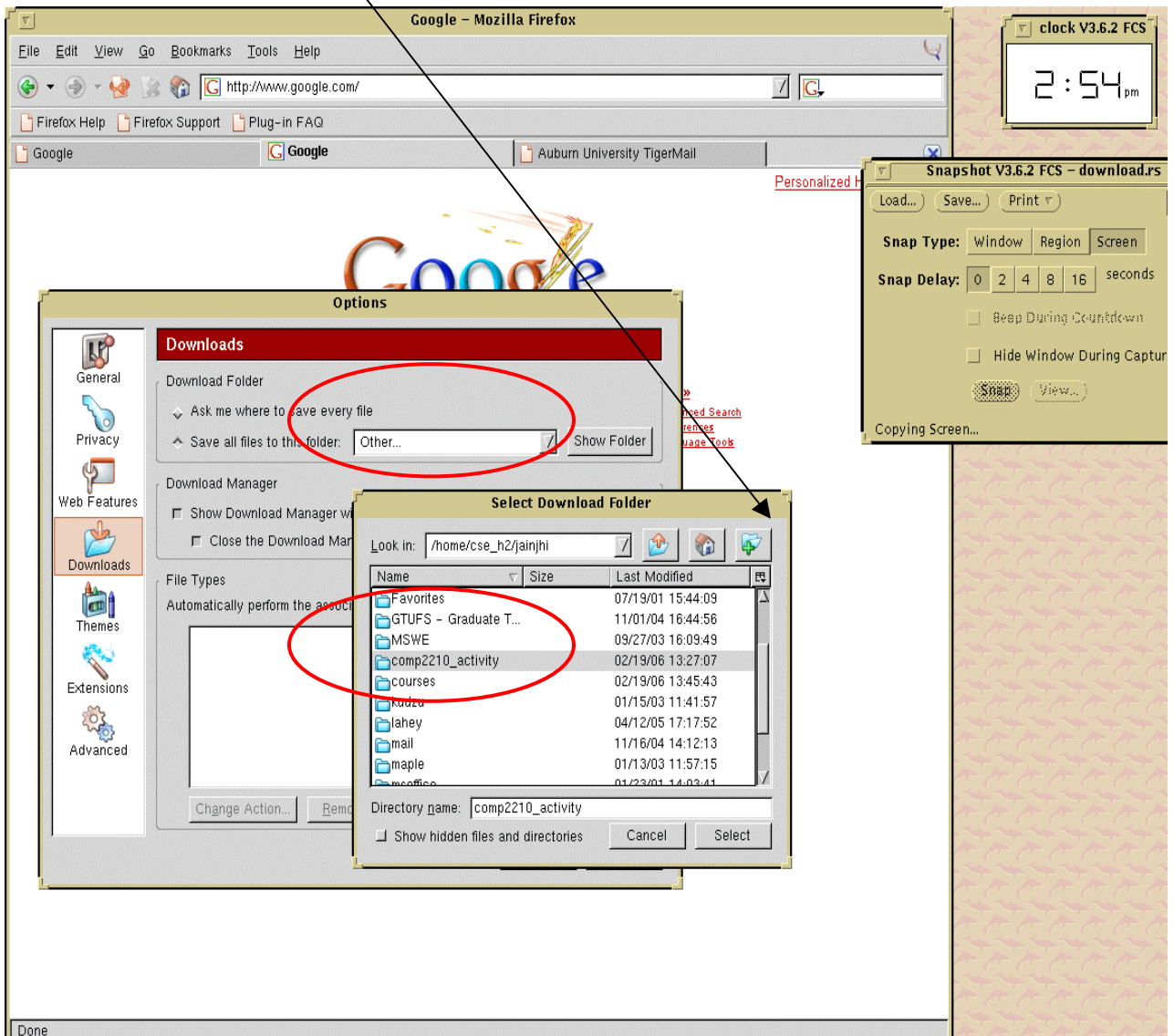
Step 7: If firefox is already open. Left click on File, and left click on “New Tab” to create a new tab.

[If firefox is NOT open, right Click on desktop, right click on “Information”, and left click on “Firefox” to start the Firefox browser.]

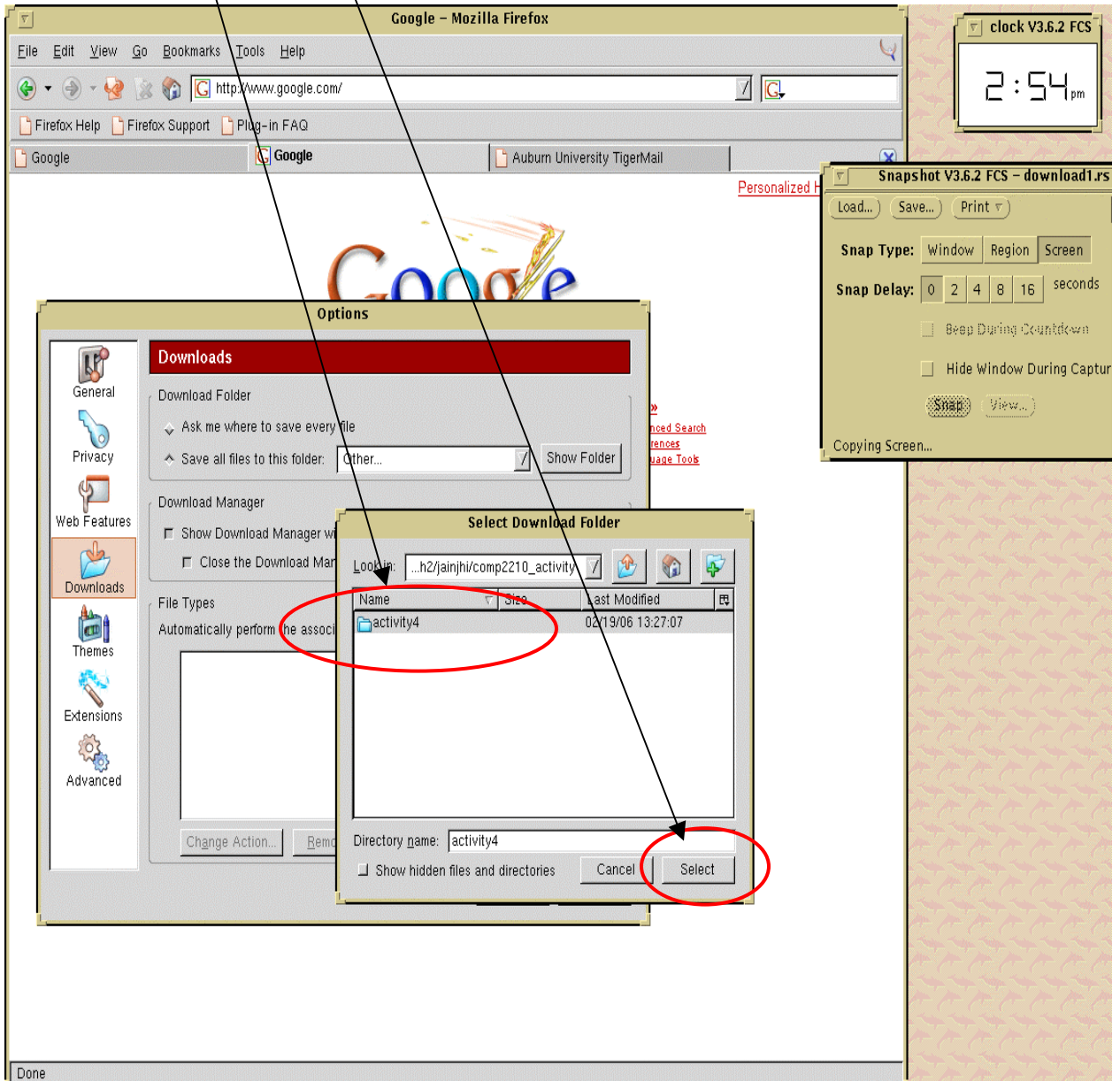
Go to the “Tools” menu and choose “Options”. The following window will open. Now click on the drop down box, near “Desktop” and click on “Other...”.



Step 8: The following window will open. Now search and double click on the directory “comp2210_activity”



Step 9: Next single click on the directory “activity4” (Please do not double click on this directory). Now click the select button.



Step 10: You will notice that your path has been selected where you should download today’s activity files. Now click on “Ok” button.

Step 11: Go to <http://www.eng.auburn.edu/~jainjhi/comp2210/> and download the zipped file Activity4.zip from here. The zipped file will be downloaded into the correct location.

Step 12: Now switch to your shell tool and type in the following command to unzip the files.

```
>> unzip Activity4.zip
```

Step 13: Congratulations! Now you are ready to run jGRASP☺☺ **IF JGRASP IS RUNNING AT THIS POINT MAKE SURE TO KILL IT BEFORE PROCEEDING.** Type in the following command in the shelltool and press the enter/return key to run the latest version of jGRASP (please wait for a minute or so for jGRASP to load)

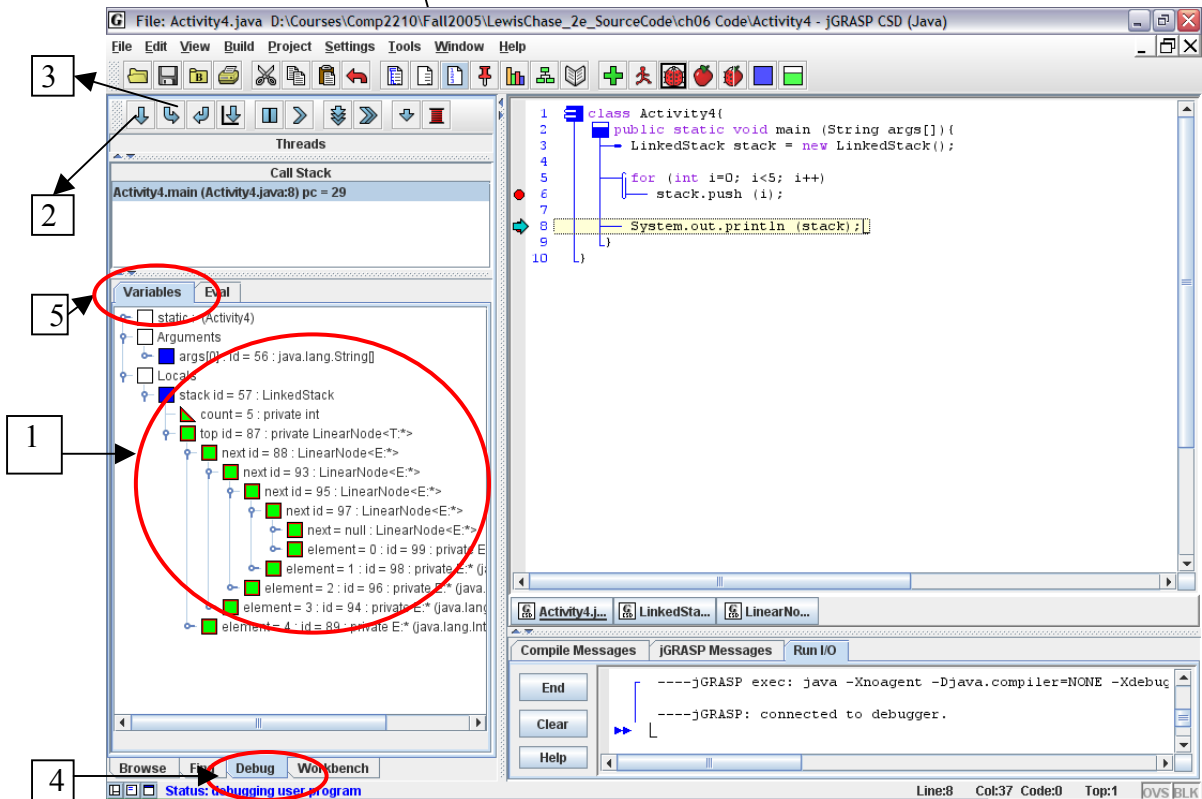
```
>>jgrasp_test
```

Step 14: After jGRASP has started open the program Activity4.java. If you are not familiar with the jGRASP debugger (setting breakpoints, stepping through the program, viewing value of variables etc), then **go through the jGRASP tutorial on Debugging.** This pdf file is available at

http://www.eng.auburn.edu/grasp/tutorials18/07_Debugger.pdf

Please don't skip going through this tutorial because you must demo the steps below in order to be marked "present" for the activity and future activities build on the visual debugger concepts.

Step 15: Turn on the line numbers for Activity4.java. Add breakpoints at line 6 and line 8 of Activity4.java. Now start the debugger and step through the program. Step in to the methods “push” and “toString” in order to visualize how the methods are implemented.



- 1: Examine the value of reference variable top after the “for” loop is finished.
- 2: This button is used to “Step over” a method during debugging
- 3: This button is used to “Step in” a method during debugging
- 4: Debug tab
- 5: Variables section

Step 16: When you feel you are comfortable using the debugger please raise your hand to demo the following:

1. Add breakpoints in methods push() and toString() [LinkedList.java].
2. Start the debugger
3. Step through the driver program.
4. Point out the “variables section” in the debug window.
5. Point out the “call stack” in the debug window.
6. Demonstrate that you can differentiate between “Stepping In” and “Stepping Over” the toString() method.
7. Remove breakpoints added in Step 1.

Appendix F - Activity to familiarize students with jGRASP debugger and viewers

Group 2 – With Object Viewers

Today's goal is to learn how to use the jGRASP visual debugger and object viewers. This is required for future Lab Activities.

Step 1: Right click anywhere on the desktop -> Right click on Shells -> Left click on Shell Tool. A shelltool will open.



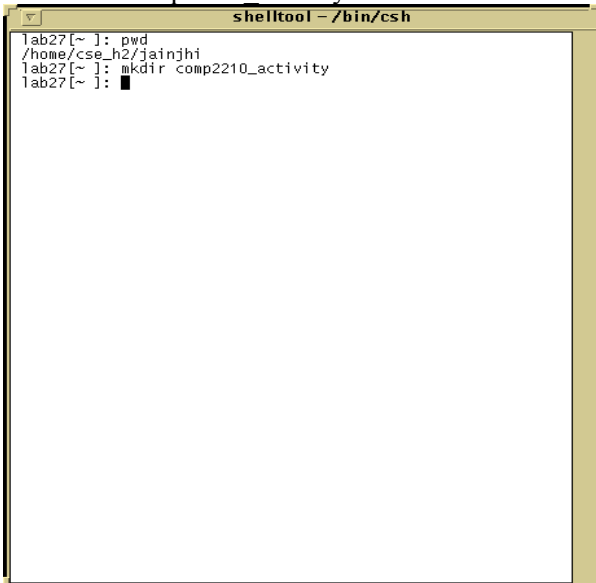
Step 2: Type the following command `pwd` and click the "Return" key to confirm that you are in your home directory

```
>> pwd
```

This should return your home path -> `/home/u2/yourusername`

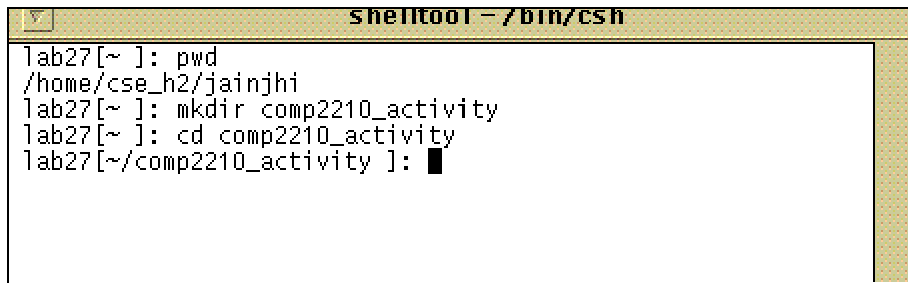
Step 3: Type the following command and click the "Return" key to create the directory `comp2210_activity` in your home directory.

```
>> mkdir comp2210_activity
```



Step 4: Type the following command (cd – change directory) and click the “Return” key to enter the new directory

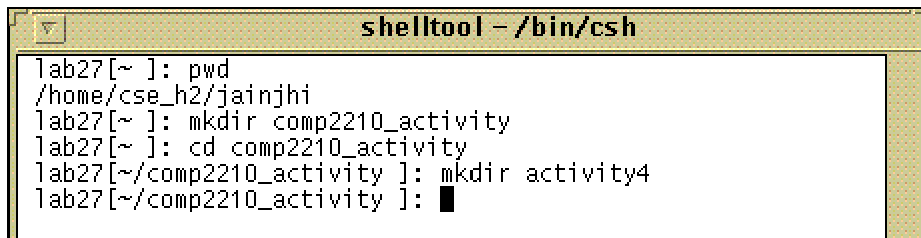
```
>> cd comp2210_activity
```



```
shelltool - /bin/csh
lab27[~ ]: pwd
/home/cse_h2/jainjhi
lab27[~ ]: mkdir comp2210_activity
lab27[~ ]: cd comp2210_activity
lab27[~/comp2210_activity ]: █
```

Step 5: Now type the following command and click the “Return” key to create a sub-directory for today’s assignment.

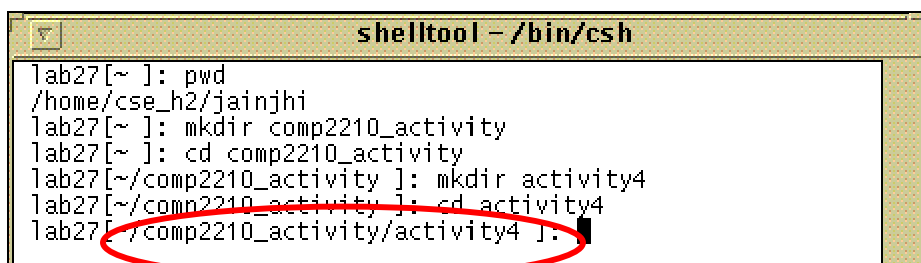
```
>> mkdir activity4
```



```
shelltool - /bin/csh
lab27[~ ]: pwd
/home/cse_h2/jainjhi
lab27[~ ]: mkdir comp2210_activity
lab27[~ ]: cd comp2210_activity
lab27[~/comp2210_activity ]: mkdir activity4
lab27[~/comp2210_activity ]: █
```

Step 6: Type the following command to and click the “Return” key confirm that the directory activity4 was created.

```
>> cd activity4
```



```
shelltool - /bin/csh
lab27[~ ]: pwd
/home/cse_h2/jainjhi
lab27[~ ]: mkdir comp2210_activity
lab27[~ ]: cd comp2210_activity
lab27[~/comp2210_activity ]: mkdir activity4
lab27[~/comp2210_activity ]: cd activity4
lab27[~/comp2210_activity/activity4 ]: █
```

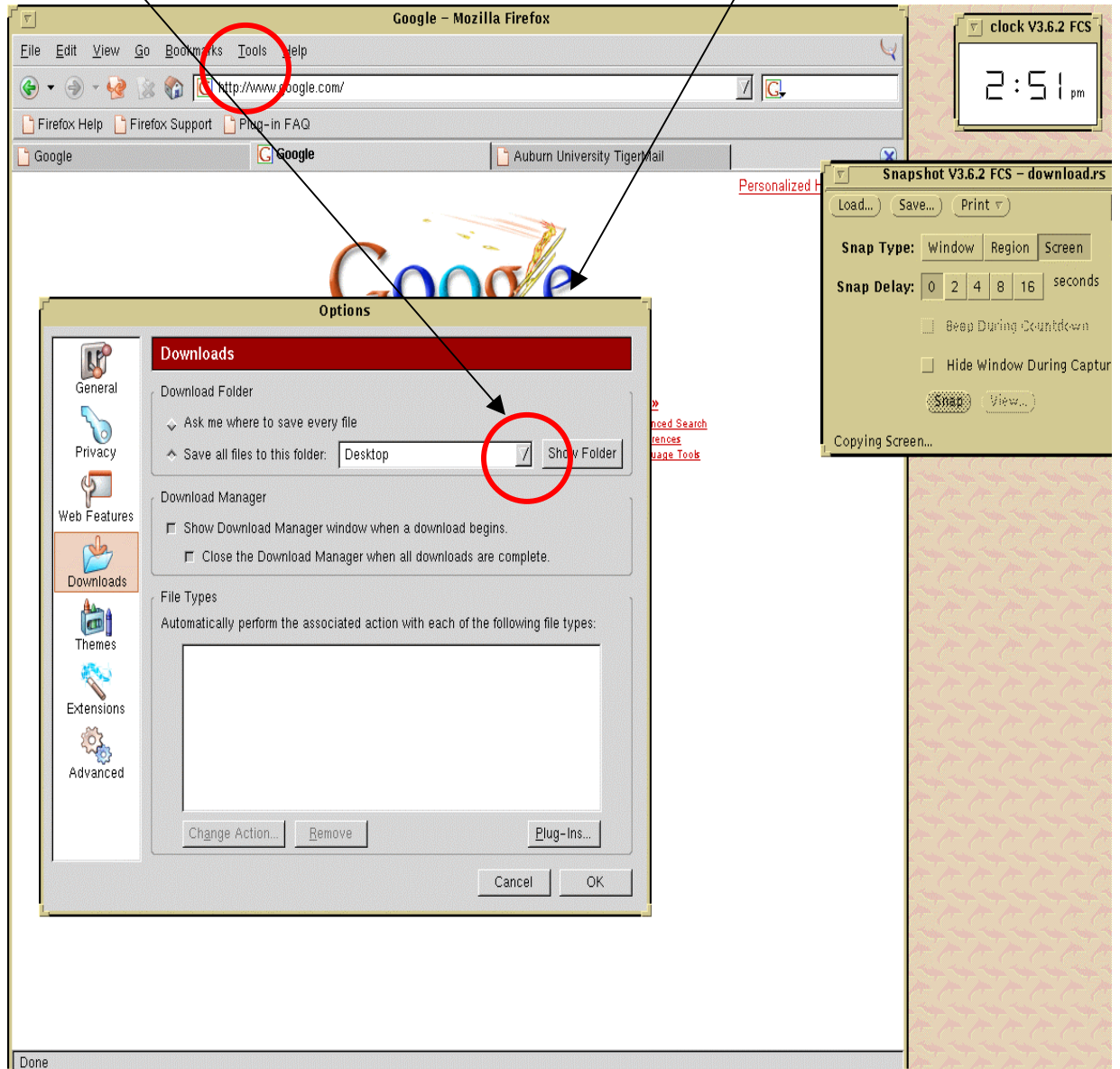
Confirm that the circled path is visible in your shelltool.

This is where you must save the programs for today. **DO NOT CLOSE THIS SHELL TOOL**

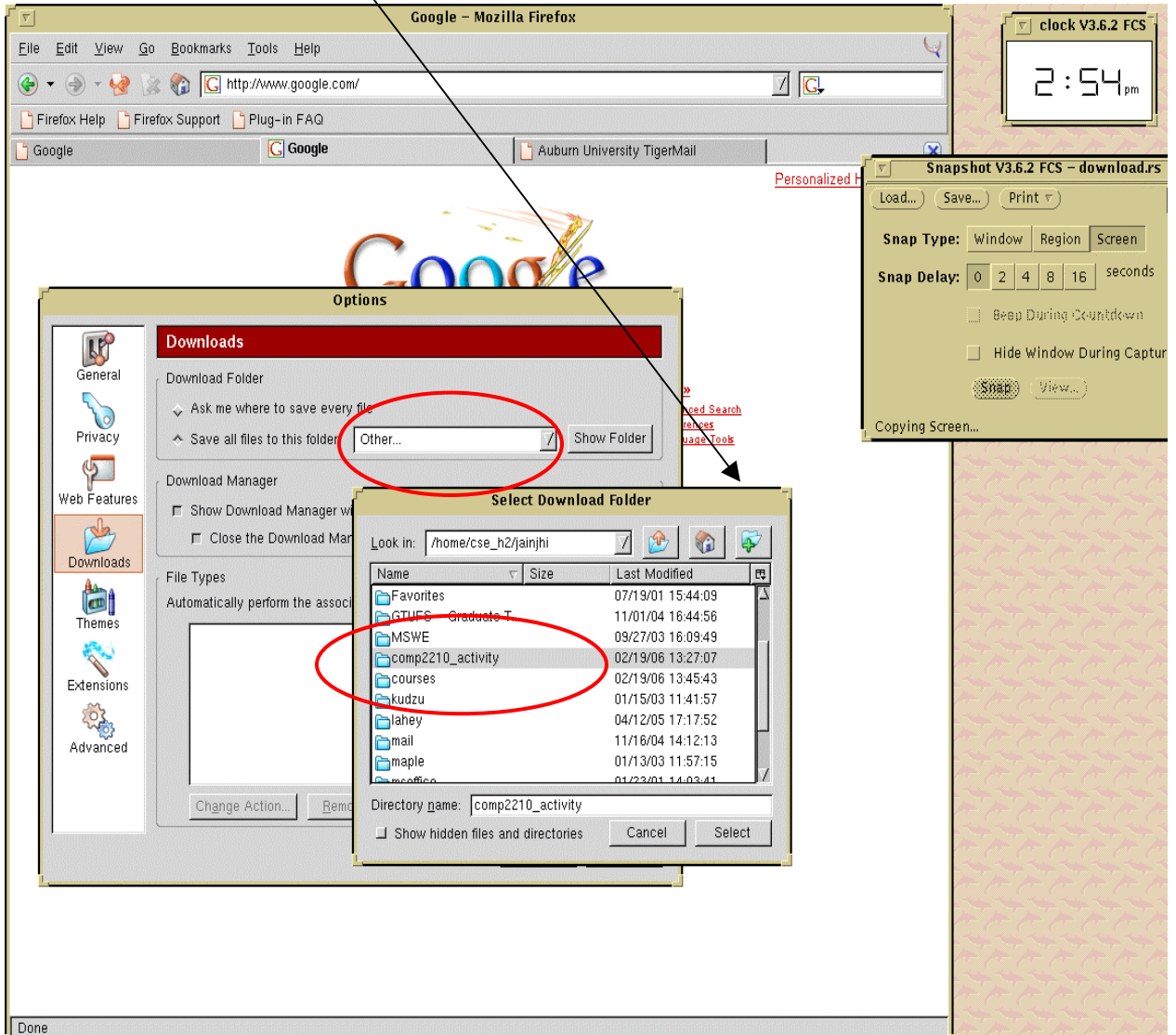
Step 7: If firefox is already open. Left click on File, and left click on “New Tab” to create a new tab.

[If firefox is NOT open, right Click on desktop, right click on “Information”, and left click on “Firefox” to start the Firefox browser.]

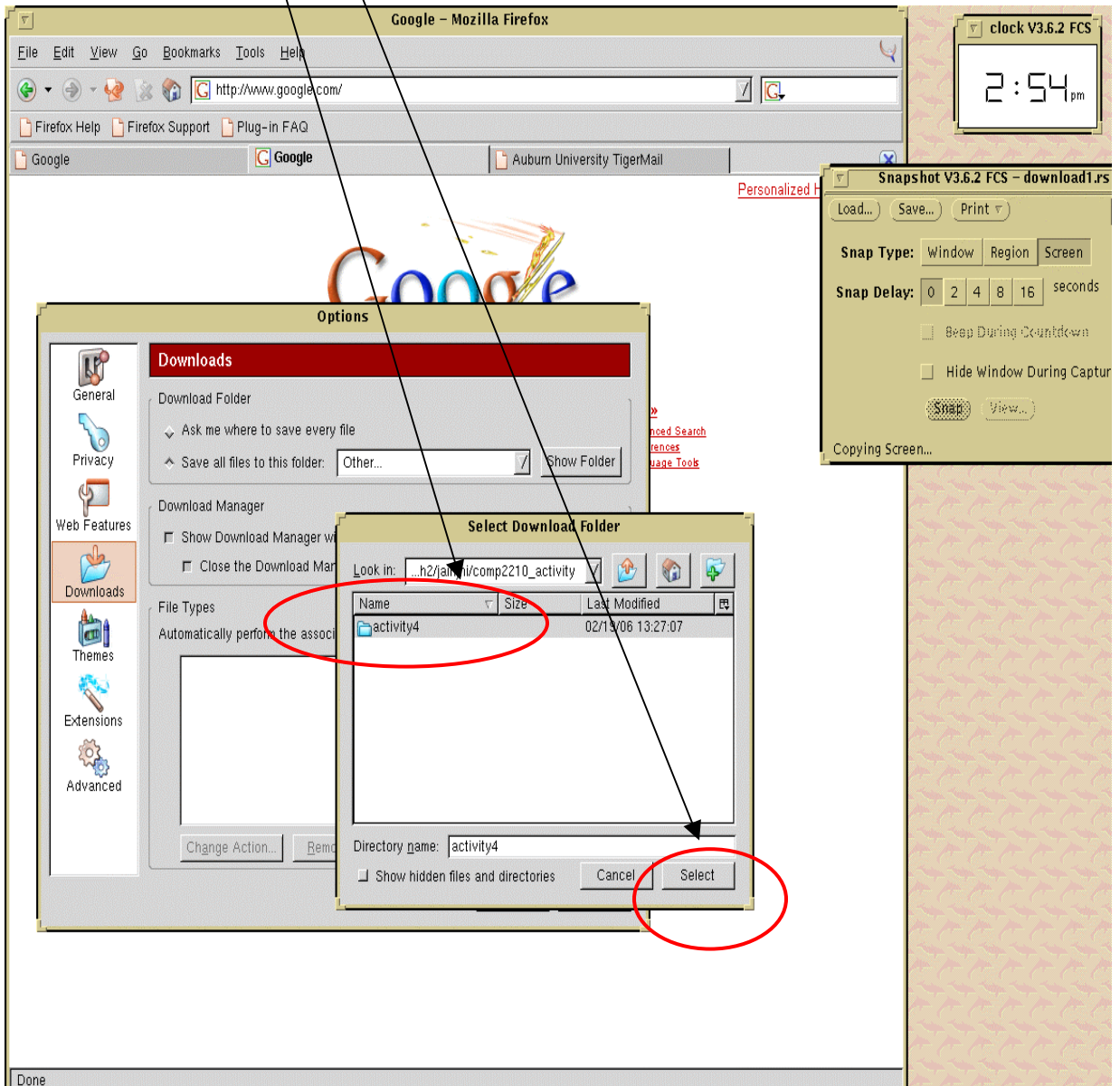
Go to the “Tools” menu and choose “Options”. The following window will open. Now click on the drop down box, near “Desktop” and click on “Other/..”.



Step 8: The following window will open. Now search and double click on the directory “comp2210_activity”



Step 9: Next single click on the directory “activity4” (Please do not double click on this directory). Now click the select button.



Step 10: You will notice that your path has been selected where you should download today’s activity files. Now click on “Ok” button.

Step 11: Go to <http://www.eng.auburn.edu/~jainjhi/comp2210/> and download the zipped file Activity4.zip from here. The zipped file will be downloaded into the correct location.

Step 12: Now switch to your shell tool and type in the following command to unzip the files.

```
>> unzip Activity4.zip
```

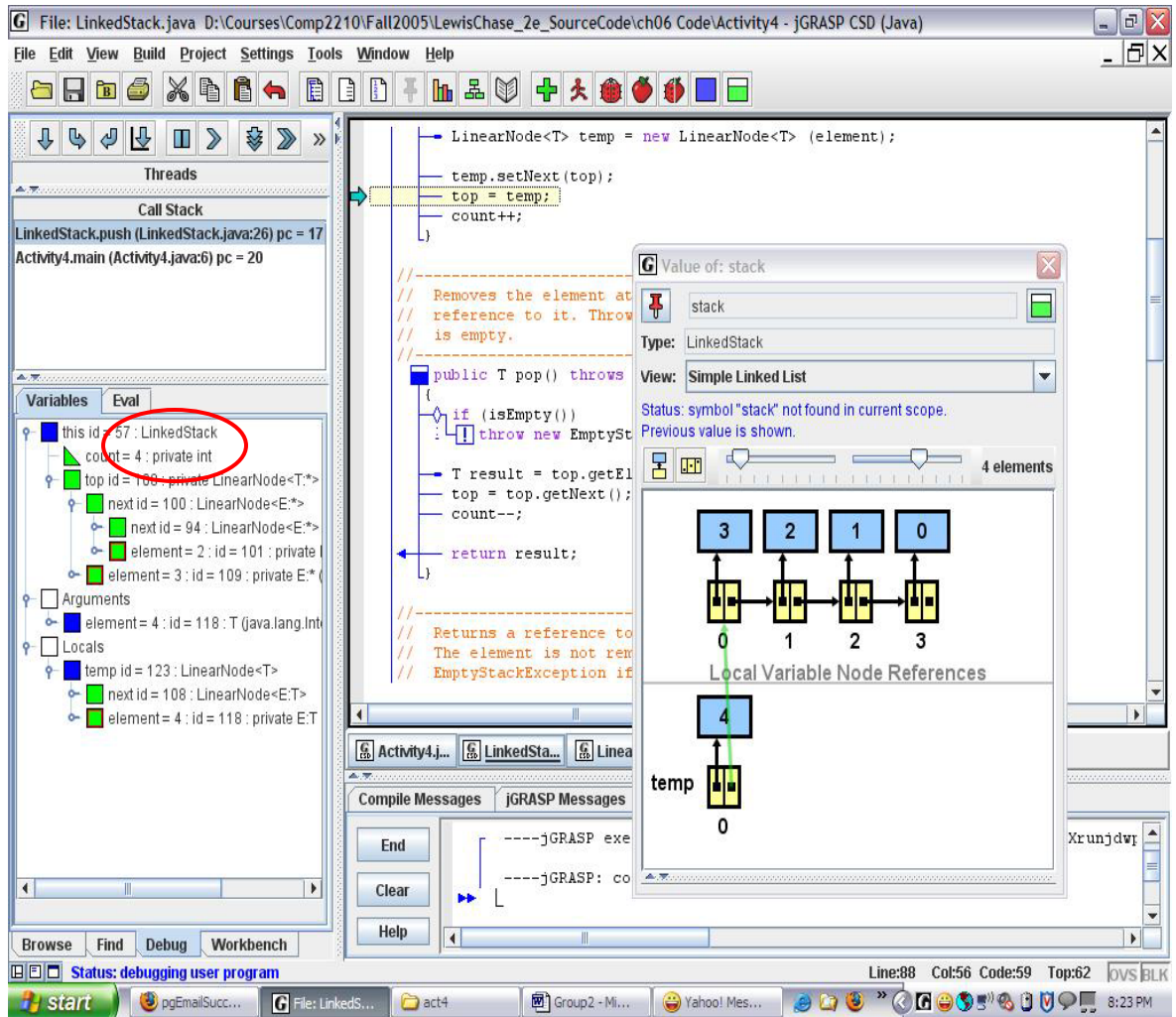
Step 13: Congratulations! Now you are ready to run jGRASP© **IF JGRASP IS RUNNING AT THIS POINT MAKE SURE TO KILL IT BEFORE PROCEEDING.** Type in the following command in the shelltool and press the enter/return key to run the latest version of jGRASP (please wait for a minute or so for jGRASP to load)

```
>>jgrasp_test
```

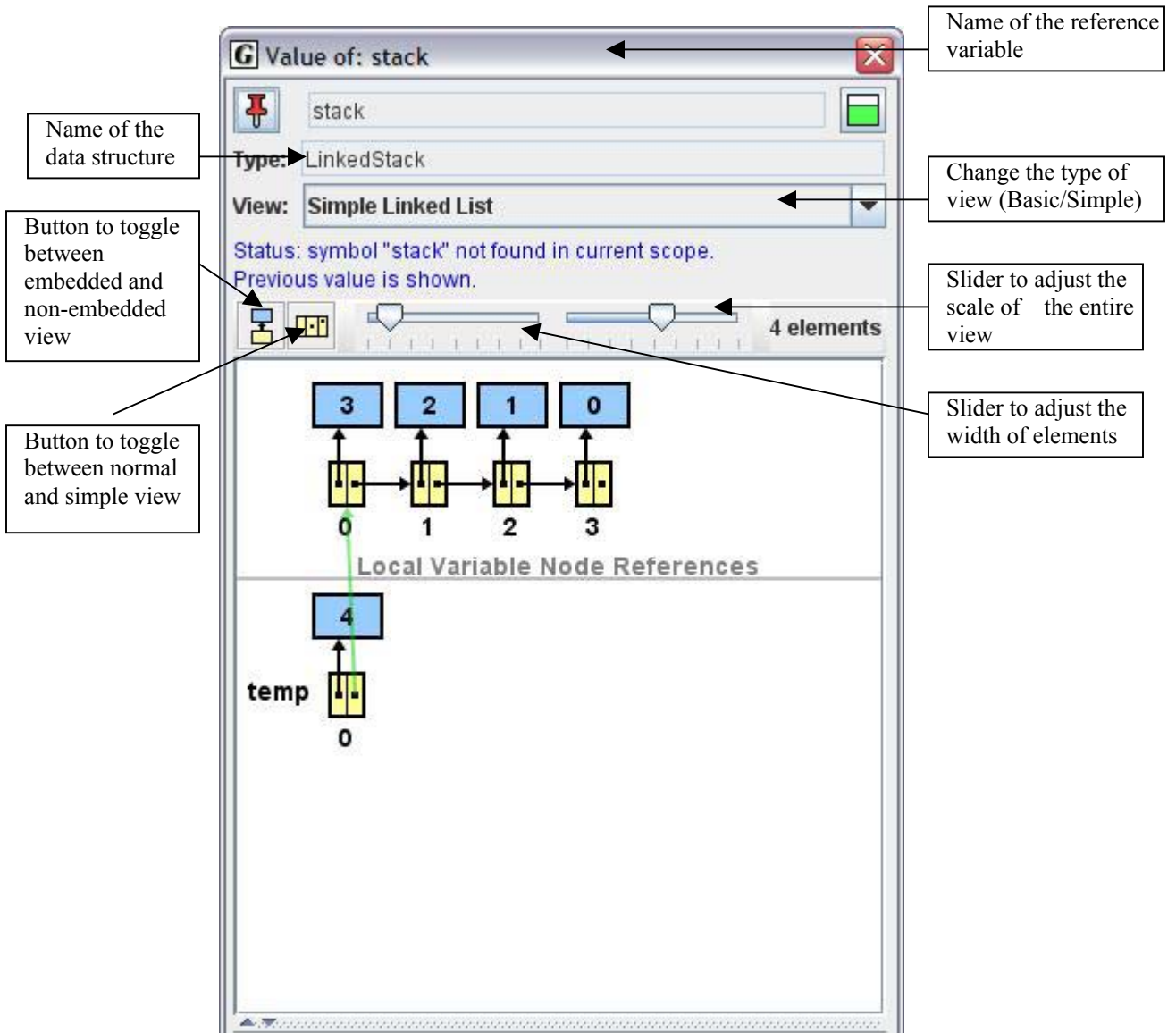
Step 14: After jGRASP has started, open the program Activity4.java. If you are not familiar with the jGRASP debugger (setting breakpoints, stepping through the program, viewing value of variables etc), then **go through the jGRASP tutorial on Debugging.** This pdf file is available at http://www.eng.auburn.edu/grasp/tutorials18/07_Debugger.pdf **Please don't skip going through this tutorial because you must demo the steps below in order to be marked "present" for the activity and future activities build on the visual debugger concepts.**

Step 15: Turn on the line numbers for the program Activity4.java. Add breakpoints at line 6 and line 8 of Activity4.java. Now start the debugger and step through the program.

Step 16: All objects have a basic view as shown in the debug tab. We will open a separate viewer window for the reference variable “stack” which is displayed in the *Variables* section of the Debug tab. Left click on the “stack” variable in the debug tab and drag it to open a verifying view of the LinkedStack data structure. Now when you “step into” the methods “push” and “toString” you can visualize how the methods are implemented.



Step 17: Description of the viewer window features.



Step 18: When you feel you are comfortable using the debugger please raise your hand to demo the following:

1. Add breakpoints in methods push() and toString() [LinkedList.java], and start the debugger.
2. At the first breakpoint, open a viewer for the reference variable "stack".
3. Point out the "variables section" in the debug window.
4. Demonstrate that you can differentiate between "Stepping In" and "Stepping Over" the toString() method using the object viewer.
5. Switch between Basic and SimpleView.
6. Switch from non-embedded to embedded view, and switch from normal to simple view.
7. Scale the size of data structure to make it larger
8. Make the width of elements larger

Appendix G - Program LinkedSet.java provided for Experiment II

```
1  import java.util.*;
2
3  public class LinkedSet<T> implements SetADT<T>
4  {
5      private int count;
6
7      private LinearNode<T> contents;
8
9      //-----
10     // Creates an empty set.
11     //-----
12     public LinkedSet()
13     {
14         count = 0;
15         contents = null;
16     }
17
18     //-----
19     // Adds the specified element to the set if it's not already
20     // present.
21     //-----
22     public void add (T element)
23     { /*** Find error(s) in this method
24         LinearNode<T> node = new LinearNode<T> (element);
25         node.setNext(contents);
26         contents = node;
27         count++;
28     }
29
30     public void insert (T element, int index)
31     {/*** Find error(s) in this method
32         LinearNode n = contents;
33
34         for (int i = 0; n != null && i < index - 1; i++)
35             n = n.getNext();
36
37         if (n != null)
38         {
39             LinearNode tmpNode = new LinearNode(element);
40             tmpNode.setNext(n.getNext());
41             n.setNext(tmpNode);
42         }
43
44         count++;
45     }
```

```

46
47
48 //-----
49 // Returns true if this set contains the specified target
50 // element.
51 //-----
52     public boolean contains (T target)
53     {/** Find error(s) in this method
54         LinearNode<T> tempNode = contents;
55         while ((tempNode != null) &&
56 (!tempNode.getElement().equals(target)))
57         {
58             tempNode = contents.getNext();
59         }
60         if (tempNode == null)
61             return false;
62         return true;
63     }
64
65 //-----
66 // Returns a string representation of this set.
67 //-----
68     public String toString()
69     {
70         String result="";
71         LinearNode<T> temp = contents;
72
73         for (int i=0; i<count; i++){
74             result += temp.getElement() + " ";
75             temp = temp.getNext();
76         }
77         return result;
78     }
79
80
81     public void delete (int index)
82     {/** Find error(s) in this method
83         int choice = index;
84         T result;
85         LinearNode<T> previous, current;
86
87         previous = contents;
88         for (int skip=0; skip < choice; skip++)
89             previous = previous.getNext();
90         current = previous.getNext();
91         result = current.getElement();
92         previous.setNext(current.getNext());

```

```
93
94     count--;
95 }
96
97     // Returns the number of elements in this set
98     public int size(){
99         return count;
100    }
101 }
```

Appendix H - Program LinkedBinarySearchTree.java provided for Experiment IV

```
1  import java.util.Iterator;
2
3  public class LinkedBinarySearchTree<T> {
4      int count;
5      BinaryTreeNode<T> root;
6
7      public LinkedBinarySearchTree() {
8          count = 0;
9          root = null;
10     }
11
12     public LinkedBinarySearchTree (T element) {
13         count = 1;
14         root = new BinaryTreeNode<T> (element);
15     }
16
17     public void addElement (T node) {
18         /******Find error in this method (Task 1 of 5)*/
19         BinaryTreeNode<T> t = new BinaryTreeNode<T> (node);
20         Comparable<T> compareIt = (Comparable<T>)node;
21
22         if (isEmpty())
23             root = t;
24         else {
25             BinaryTreeNode<T> tmp = root;
26             boolean added = false;
27
28             while (!added) {
29                 if (compareIt.compareTo(tmp.element) < 0)
30
31                     if (tmp.left == null) {
32                         tmp.left = t;
33                         added = true;
34                     }
35                     else
36                         tmp = tmp.left;
37                 else
38                     if (tmp.right == null) {
39                         tmp.right = t;
40                         added = true;
41                     }
42                     else
43                         tmp = tmp.right;
44             }
45         }
```

```

46     count ++;
47 }
48
49     public boolean isEmpty() {
50         if (count == 0)
51             return true;
52         else
53             return false;
54     }
55
56     public BinaryTreeNode<T> find (T targetElement){
57         BinaryTreeNode<T> current=findagain(targetElement, root );
58         if( current == null )
59             System.out.println("element not found");
60         return (current);
61     }
62 }
63
64 //=====
65 // Returns a reference to the specified target element if it is
66 // found in the binary tree.
67 //=====
68     public BinaryTreeNode<T> findagain (T elm,
69         BinaryTreeNode<T> root){
70
71         //*****Find error in this method (Task 2 of 5)
72         if (root == null) {
73             return null;
74         }
75         if (root.element.equals(elm)) {
76             return root;
77         }
78         BinaryTreeNode<T> x = findagain(elm, root.right);
79         if (x == null) {
80             x = findagain(elm, root.right);
81         }
82         return x;
83     }
84
85 //=====
86 // Removes the first element that matches the specified target
87 // element from the tree and returns a reference to
88 // it. Throws a ElementNotFoundException if the target
89 // element is not found in the binary search tree.
90 //=====
91     public T removeElement (T targetElement) throws
92         ElementNotFoundException {
93         //*****Find error in this method (Task 3 of 5)

```

```

94
95     T result = null;
96
97     if (!isEmpty())
98
99         if(((Comparable)targetElement).equals(root.element)) {
100             result = root.element;
101             root = replacement (root);
102             count--;
103         } //if
104         else {
105             BinaryTreeNode<T> tmp, parent = root;
106             boolean found = false;
107
108             if
109 ((Comparable)targetElement).compareTo(root.element)<0)
110                 tmp = root.left;
111             else
112                 tmp = root.right;
113
114             while (tmp != null && !found) {
115                 if (targetElement.equals(tmp.element)) {
116                     found = true;
117                     count--;
118                     result = tmp.element;
119
120                     if (tmp == parent.left)
121                     {
122                         parent.left = replacement (tmp);
123                     }
124                     else
125                     {
126                         parent.right = replacement (tmp);
127                     }
128                 } //if
129                 else
130                 {
131                     parent = tmp;
132
133                     if
134 ((Comparable)targetElement).compareTo(tmp.element) > 0)
135                         tmp = tmp.left;
136                     else
137                         tmp = tmp.right;
138                 } //else
139             } //while
140         if (!found)
141             System.out.println (targetElement +

```

```

140         " was not found in binary tree");
141     }
142
143     return result;
144
145 }
146
147 //=====
148 // Returns a reference to a node that will replace the one
149 // specified for removal. In the case where the removed
150 // node has two children, the inorder successor is used
151 // as its replacement.
152 //=====
153 protected BinaryTreeNode<T> replacement
154 (BinaryTreeNode<T> node) {
155     BinaryTreeNode<T> result = null;
156
157     if ((node.left == null)&&(node.right==null))
158         result = null;
159     else if ((node.left != null)&&(node.right==null))
160         result = node.left;
161     else if ((node.left == null)&&(node.right != null))
162         result = node.right;
163     else
164     {
165         BinaryTreeNode<T> current = node.right;
166         BinaryTreeNode<T> parent = node;
167
168         while (current.left != null)
169         {
170             parent = current;
171             current = current.left;
172         }//while
173
174         if (node.right == current)
175             current.left = node.left;
176         else
177         {
178             parent.left = current.right;
179             current.right = node.right;
180             current.left = node.left;
181         }
182         result = current;
183     }//else
184     return result;
185
186
187 } // method replacement

```



```

188
189
190     public String toString()
191     {
192         String result = "";
193
194         Iterator<T> it = iteratorPostOrder();
195         while(it.hasNext()){
196             result += it.next().toString() + " ";
197         }
198         return result;
199     } // method toString
200
201     public Iterator<T> iteratorInOrder()
202     {
203         ArrayUnorderedList<T> templist = new
ArrayUnorderedList<T>();
204         inorder (root, templist);
205         return templist.iterator();
206     }
207
208     //=====
209     // Performs a recursive inorder traversal.
210     //=====
211     protected void inorder (BinaryTreeNode<T> x,
212         ArrayUnorderedList<T> list)
213     {
214
215         //*****Find error in this method (Task 4 of 5)
216         if (x != null)
217         {
218             inorder (x.right, list);
219             list.addToRear(x.element);
220             inorder (x.left, list);
221         }
222
223     }
224
225     //=====
226     // Performs an postorder traversal on the tree by calling
227     // an overloaded, recursive postorder method that starts
228     // with the root.
229     //=====
230     public Iterator<T> iteratorPostOrder()
231     {
232         ArrayUnorderedList<T> templist = new
ArrayUnorderedList<T>();
233         postorder (root, templist);

```

```

234         return templist.iterator();
235     }
236
237     //=====
238     // Performs a recursive postorder traversal.
239     //=====
240     protected void postorder (BinaryTreeNode<T> x,
241         ArrayUnorderedList<T> list)
242     {
243
244         //*****Find error in this method (Task 5 of 5)
245
246         if (x != null)
247         {
248             postorder (x.right, list);
249             postorder (x.left, list);
250             list.addToRear(x.element);
251         }
252     }
253 } // class BinarySearchTree

```

Appendix I - Program Heap.java provided for Experiment V

```
1  public class Heap<T> extends LinkedBinaryTree<T>
2  {
3      public HeapNode<T> lastNode;
4
5      public Heap() {
6          super();
7      } // constructor Heap
8
9      //=====
10     // Adds the specified element to the heap in the appropriate
11     // position according to its key value. Note that equal
12     // elements are added to the right.
13     //=====
14     public void addElement (T obj) {
15
16         //modify do that it works as a MAX HEAP
17         HeapNode<T> node = new HeapNode<T>(obj);
18
19         if (root == null)
20             root=node;
21         else
22         {
23             HeapNode<T> next_parent = getNextParentAdd();
24             if (next_parent.left == null)
25                 next_parent.left = node;
26             else
27                 next_parent.right = node;
28             node.parent = next_parent;
29         }
30         lastNode = node;
31         count++;
32         if (count>1)
33             heapifyAdd();
34     } //method addElement
35
36     //=====
37     // Returns the node that will be the parent of the new node
38     //=====
39
40     private HeapNode<T> getNextParentAdd(){
41         HeapNode<T> result = lastNode;
42         while ((result != root) && (result.parent.left != result))
43             result = result.parent;
44
45         if (result != root)
```

```

46         if (result.parent.right == null)
47             result = result.parent;
48         else
49             {
50                 result = (HeapNode<T>)result.parent.right;
51                 while (result.left != null)
52                     result = (HeapNode<T>)result.left;
53             }
54         else
55             while (result.left != null)
56                 result = (HeapNode<T>)result.left;
57
58         return result;
59     } //method getNextParentAdd
60
61
62     //=====
63     // Reorders the heap after adding a node
64     //=====
65     private void heapifyAdd(){
66         T temp;
67
68         HeapNode<T> next = lastNode;
69         while ((next != root) &&
70             ((Comparable)next.element).compareTo(next.parent.element) < 0)
71             {
72                 temp = next.element;
73                 next.element = next.parent.element;
74                 next.parent.element = temp;
75                 next = next.parent;
76             }
77     } //method heapifyAdd
78
79     //=====
80     // Remove the element with the lowest value in the heap and
81     // returns a reference to it. Throws an
82     // EmptyCollectionException if the heap is empty.
83     //=====
84     public T removeMin() throws EmptyCollectionException {
85
86         if (isEmpty())
87             throw new EmptyCollectionException ("Empty Heap");
88
89         T minElement = root.element;
90
91         if (count == 1){
92             root = null;

```

```

93         lastNode = null;
94     }
95     else{
96         HeapNode<T> next_last = getNewLastNode();
97         if (lastNode.parent.left == lastNode)
98             lastNode.parent.left = null;
99         else
100             lastNode.parent.right = null;
101
102         root.element = lastNode.element;
103         lastNode = next_last;
104         heapifyRemove();
105     }
106
107     count--;
108     return minElement;
109
110 } // method removeMin
111
112
113 //=====
114 // Reorders the heap after removing the root element
115 //=====
116
117 private void heapifyRemove(){
118     T temp;
119     HeapNode<T> node = (HeapNode<T>)root;
120     HeapNode<T> left = (HeapNode<T>)node.left;
121     HeapNode<T> right = (HeapNode<T>)node.right;
122     HeapNode<T> next;
123
124     if ((left == null) && (right == null))
125         next = null;
126     else if (left == null)
127         next = right;
128     else if (right == null)
129         next = left;
130     else if
131 ((Comparable)left.element).compareTo(right.element) < 0)
132         next = left;
133     else
134         next = right;
135
136     while ((next != null) &&
137 ((Comparable)next.element).compareTo(node.element) < 0))
138     {
139         temp = node.element;
140         node.element = next.element;

```

```

140         next.element = temp;
141         node = next;
142         left = (HeapNode<T>)node.left;
143         right = (HeapNode<T>)node.right;
144         if ((left == null) && (right == null))
145             next = null;
146         else if (left == null)
147             next = right;
148         else if (right == null)
149             next = left;
150         else if
151             (((Comparable)left.element).compareTo(right.element) < 0)
152             next = left;
153         else
154             next = right;
155     } //method heapifyRemove
156
157     //=====
158     // Returns the node that will be the new last node a remove
159     //=====
160
161     private HeapNode<T> getNewLastNode() {
162         HeapNode<T> result = lastNode;
163
164         while ((result != root) && (result.parent.left == result))
165             result = result.parent;
166         if (result != root)
167             result = (HeapNode<T>)result.parent.left;
168
169         while (result.right != null)
170             result = (HeapNode<T>)result.right;
171
172         return result;
173     } //method getNewLastNode
174
175
176     //=====
177     // Returns the element with the highest value in the heap.
178     // Throws an EmptyCollectionException if the heap is empty.
179     //=====
180
181     public T findMax () throws EmptyCollectionException {
182         //fill this in
183         return null;
184     }
185
186

```

```
187 //=====
188 // Remove the element with the highest value in the heap and
189 // returns a reference to it. Throws an
190 // EmptyCollectionException if the heap is empty.
191 //=====
192     public T removeMax() throws EmptyCollectionException
193     {
194         //fill this in
195         return null;
196     }
197 } // class Heap
```

Appendix J - Program PriorityQueueLinked.java provided for Experiment VI

```
1  class PriorityQueueLinked<E> implements PriorityQueueADT<E>{
2      int count = 0;
3      Node<E> front;
4
5      public void add(E value, int priority){
6          // complete this method
7      }
8
9      public E remove(){
10         Node<E> temp = front;
11         front = front.getNext();
12         count --;
13         return temp.getElement();
14     }
15
16     public E peek(){
17         return front.getElement();
18     }
19
20     public boolean isEmpty(){
21         if (count == 0)
22             return true;
23         else
24             return false;
25     }
26
27     public int size(){
28         return count;
29     }
30
31     public String toString(){
32         String res = "";
33         Node<E> tmp = front;
34         for (int i=0; i<count; i++){
35             res += tmp.getElement() + " ";
36             tmp = tmp.getNext();
37         }
38         return res;
39     }
40 }
```


Appendix K - SAS code for Experiment I, III, V and VI: 2 response variables

```
data TwoResponseVariables;
proc import datafile="C:\ExperimentDataA.xls" out=Exp1 replace;
run;

proc print data=Exp1;
run;
data diffs;
  set Exp1;
  Timediff = Grp2Time - Grp1Time;
  Rawdiff = Grp2Raw - Grp1Raw;
run;

proc corr data = diffs cov outp = corrout;
  var Timediff Rawdiff;
run;

proc iml;
  use corrout;
  read all var {Timediff Rawdiff} where (_type_='COV') into
  S;
  read all var {Timediff Rawdiff} where (_type_='MEAN') into
  dbartran;

  dbar = dbartran`;

  n = 34; /*sample size was adjusted depending on the
  experiment*/
  p = 2; /*number of response variables */
  q = p;
  alpha = 0.05; /*always set to 0.05*/

  T2= n *dbar`*inv(S)*dbar;
  Fcrit=finv(1-alpha,q-1,n-q+1)*(n-1)*(q-1)/(n-q+1);
  scaledT2 = T2*(n-q+1)/((q-1)*(n-1));
  pval=1-probf(scaledT2,q-1,n-q+1);

  print T2 Fcrit pval;
run;
```

Appendix L - SAS code for Experiment II and IV: 4 response variables

```
data FourResponseVariables;
proc import datafile="C:\ExperimentDataB.xls" out=Exp2 replace;
run;

proc print data=Exp2;
run;

data diffs;
  set Exp2;
  TimeDiff = Grp2Time - Grp1Time;
  LocatedDiff = Grp2BugsLocated - Grp1BugsLocated;
  CorrectedDiff = Grp2BugsCorrected - Grp1BugsCorrected;
  IntroducedDiff = Grp2BugsIntroduced - Grp1BugsIntroduced;
run;

proc corr data = diffs cov outp = corrout;
  var TimeDiff LocatedDiff CorrectedDiff;
run;

proc iml;
  use corrout;
  read all var {TimeDiff LocatedDiff CorrectedDiff
  IntroducedDiff} where (_type_='COV') into S;

  read all var {TimeDiff LocatedDiff CorrectedDiff
  IntroducedDiff} where (_type_='MEAN') into dbartran;

  print S;

  n = 34; /*sample size was adjusted depending on the
  experiment*/
  p = 4; /*number of response variables */
  q = p;
  alpha = 0.05; /*always set to 0.05*/

  dbar=dbartran`;







  T2= n *dbar`*inv(S)*dbar;
  Fcrit=finv(1-alpha,q-1,n-q+1)*(n-1)*(q-1)/(n-q+1);















  scaledT2 = T2*(n-q+1)/((q-1)*(n-1));
  pval=1-probf(scaledT2,q-1,n-q+1);

  print T2 Fcrit pval;
run;
```

Appendix M - Questionnaire: Group 1 that used only the jGRASP Debugger

This is an anonymous survey. After turning in this sheet please remember to sign the attendance sheet. Your feedback is critical to this project. On a scale of 1-4 please rate the following. **PUT A CHECK MARK IN THE BOX THAT YOU WANT TO CHOOSE.**

| Task | Scale | | | |
|---|---|--|------------------------------------|---|
| After you start the debugging procedure how usefulness are the following features? | 1 Useful | 2 Somewhat Useful | 3 Somewhat Useless | 4 Useless |
| Threads | | | | |
| Call Stack | | | | |
| Variables | | | | |
| Eval tab (next to Variables) | | | | |
| | | | | |
| How often did you use the following features: | 1 For most of the activities | 2 For at least half of the activities | 3 For 1 or 2 activities | 4 Never needed to use this feature |
| Threads | | | | |
| Call Stack | | | | |
| Variables | | | | |
| Eval tab (next to Variables) | | | | |
|  Step over | | | | |
|  Step in | | | | |
|  Step out | | | | |
|  Run to cursor | | | | |
|  Suspend selected thread | | | | |
|  Resume selected thread | | | | |











| | | | | | |
|--|--|--|--|--|--------------------------------------|
|  Auto step | | | | | |
|  Auto resume | | | | | |
|  Use byte code size steps | | | | | |
|  Suspend new threads | | | | | |
| | | | | | |
| | 1 | 2 | 3 | 4 | 0 |
| Is this icon a good representation or depiction of the feature? | Yes – I was immediately able to recognize the feature | Yes – I was able to recognize after I read what it does | No – I had to repeatedly look up what it does | No – change the icon since it is not a good representation of the feature | N/A I never used this feature |
|  Step over | | | | | |
|  Step in | | | | | |
|  Step out | | | | | |
|  Run to cursor | | | | | |
|  Suspend selected thread | | | | | |
|  Resume selected thread | | | | | |
|  Auto step | | | | | |
|  Auto resume | | | | | |
|  Use byte code size steps | | | | | |
|  Suspend new threads | | | | | |















Are there any other features that you think is missing from the debugger?









Other jGRASP related comments:

Appendix N - Questionnaire: Group 2 that used the jGRASP viewers

This is an anonymous survey. After turning in this sheet please remember to sign the attendance sheet. Your feedback is critical to this project. On a scale of 1-4 please rate the following. **PUT A CHECK MARK IN THE BOX THAT YOU WANT TO CHOOSE.**

| Task | Scale | | | |
|---|---|--|--|---|
| | 1 Useful | 2 Somewhat Useful | 3 Somewhat Useless | 4 Useless |
| The feature to toggle between embedded  to non-embedded  view is: | | | | |
| The feature to toggle between simple  and normal  view is: | | | | |
| The feature to toggle between compact  and normal  layout is: | | | | |
| The feature to toggle between rectangular  and round nodes  is: (Tree viewer) | | | | |
| The feature to toggle between animation on  and off  is: | | | | |
| The slide to adjust width of elements: | | | | |
| The slide to adjust scale of the entire view: | | | | |
| Increase or decrease animation time: | | | | |
| | | | | |
| How often did you use the following features: | 1 For most of the activities | 2 For at least half of the activities | 3 For 1 or 2 activities | 4 Never needed to use this feature |

| | | | | | |
|--|--|--|--|--|--------------------------------------|
| Toggle between embedded  to non-embedded  view is: | | | | | |
| Toggle between simple  and normal  view | | | | | |
| Toggle between compact  and normal  layout | | | | | |
| Toggle between rectangular  and round nodes  (Tree viewer) | | | | | |
| Turn animation OFF  | | | | | |
| Turn animation ON  | | | | | |
| The slide to adjust width of elements: | | | | | |
| The slide to adjust scale of the entire view: | | | | | |
| Increase or decrease animation time: | | | | | |
| Toggle between embedded  to non-embedded  view is: | | | | | |
| | | | | | |
| | 1 | 2 | 3 | 4 | 0 |
| Is this icon a good representation or depiction of the feature? | Yes – I was immediately able to recognize the feature | Yes – I was able to recognize after I read what it does | No – I had to repeatedly look up what it does | No – change the icon since it is not a good representation of the feature | N/A I never used this feature |
| Embedded view  | | | | | |
| Non-embedded view  | | | | | |

| | | | | | |
|--|--|--|--|--|--|
| Simple view  | | | | | |
| Normal view  | | | | | |
| Compact layout  | | | | | |
| Normal layout  | | | | | |
| Rectangular nodes  | | | | | |
| Round nodes  | | | | | |
| Animation on  | | | | | |
| Animation off  | | | | | |

Is there any other feature that you think would be useful to the viewer?

For example:

- 1) Changing the color of the nodes in the viewer

- 2) Stepping back during the debugging process so that you can compare the before and after state of a data structure

- 3) Changing the orientation of the data structure (switching between vertical and horizontal)

- 4) Ability to add more variables to the viewer (For example: if the method is using local integer and String variables, it would be great if those would be shown on the viewer as well. Right now you can see those in the Debug tab on the left hand side.)

Other jGRASP and viewer related comments: