

Air Force Institute of Technology

AFIT Scholar

Theses and Dissertations

Student Graduate Works

3-2004

Visual Debugging of Object-Oriented Systems with the Unified Modeling Language

Wendell E. Fox

Follow this and additional works at: <https://scholar.afit.edu/etd>

 Part of the [Computer Sciences Commons](#)

Recommended Citation

Fox, Wendell E., "Visual Debugging of Object-Oriented Systems with the Unified Modeling Language" (2004). *Theses and Dissertations*. 3987.

<https://scholar.afit.edu/etd/3987>

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact richard.mansfield@afit.edu.



**VISUAL DEBUGGING OF OBJECT-ORIENTED
SYSTEMS WITH THE UNIFIED MODELING
LANGUAGE**

THESIS

Wendell E. Fox, Flight Lieutenant, RAAF

AFIT/GCS/ENG/04-07

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense or the United States Government.

AFIT/GCS/ENG/04-07

VISUAL DEBUGGING OF OBJECT-ORIENTED SYSTEMS WITH THE UNIFIED MODELING LANGUAGE

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Wendell E. Fox, B.Eng (Aero Av)

Flight Lieutenant, RAAF

March 2004

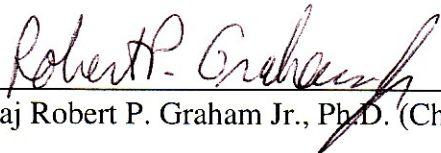
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

VISUAL DEBUGGING OF OBJECT-ORIENTED SYSTEMS
WITH THE UNIFIED MODELING LANGUAGE


Wendell E. Fox, B.Eng (Aero Av)

Flight Lieutenant, RAAF

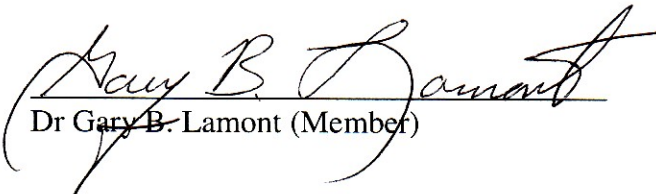
Approved:


Maj Robert P. Graham Jr., Ph.D. (Chairman)

12 Mar 2004
date


LtCol Brian G. Hermann, Ph.D. (Member)

12 Mar 2004
date


Dr Gary B. Lamont (Member)

12 MARCH '04
date

Acknowledgements

I would like to express my appreciation to my advisor, Major Robert Graham, firstly for adopting me after I became an AFIT thesis orphan, and more importantly for his guidance and for keeping me honest in this long, often frustrating but nevertheless rewarding endeavor. Thanks must also go to my original thesis advisor, Tim Jacobs, for introducing me to the world of Software Visualization. Thank you, Ben Musial, my predecessor to this research for your help in clarifying concerns and issues along the way. We can now share the pain.

I will never forget my time in the United States. I have met many fantastic individuals who have made my experience a memorable and enjoyable one at AFIT. Thank you to all my classmates, instructors, fellow Australian officers and the AFIT International Affairs staff for your friendship and camaraderie throughout the course of this 20 month adventure.

I would like to thank my parents, brothers and sisters for their undying support and encouragement. Finally and most of all, my sincere gratitude goes to my beautiful wife. No words can really express how grateful I am to have you by my side through all of this. Thank you for looking after me and always being there for me. You are a blessing, an inspiration and the reason I get back up when I'm down.

Wendell Fox

Table Of Contents

	Page
Acknowledgements	iv
List of Figures	viii
List of Tables	x
Abstract	xi
1. Research Introduction	1
1.1 Problem Statement	1
1.2 Background	2
1.3 Research Focus	3
1.3.1 Objectives	3
1.3.2 Approach	5
1.4 Research Contributions	7
1.5 Thesis Overview	8
2 Background	9
2.1 Introduction	9
2.2 Software Visualization	9
2.2.1 Program Visualization	10
2.2.1.1 Program Visualization Taxonomies	12
2.2.1.2 The Unified Modeling Language (UML)	15
2.2.2 Spatial Visualization	17
2.2.2.1 Overview+Detail	18
2.2.2.2 Focus+Context	19
2.3 Graph Layouts	22
2.3.1 Hierarchical Approach	24
2.3.2 Topology Shape Metrics Approach	25
2.4 User Interfaces	27
2.5 Debugging Issues	28
2.5.1 Bugs	28
2.5.2 Debugging	30
2.5.2.1 Scientific Method	32
2.5.2.2 Intuition	32

	Page
2.5.2.3	Leap Of Faith 32
2.5.2.4	Diagnostics..... 33
2.5.3	Debugging Tools..... 33
2.6	Data Extraction 35
2.6.1	The Java Platform Debug Architecture..... 35
2.6.2	The Java Virtual Machine Profiler Interface (JVMPPI) 37
2.7	Testing the Effectiveness of the Application 37
2.8	Summary 39
3	Methodology 41
3.1	Introduction..... 41
3.2	Objectives 41
3.2.1	Visual Objectives 42
3.2.2	System Design Objectives 44
3.3	System Architecture..... 45
3.3.1	ArgoUML 49
3.3.2	Debugger Interface..... 50
3.4	Experimental and Testing Techniques..... 51
3.5	Metrics 53
3.6	Summary 54
4	Detailed Design and Implementation..... 55
4.1	Introduction..... 55
4.2	Visualization Elements 55
4.2.1	Visual Modifications – Focus + Context 56
4.2.2	Visual Modifications - Graph Layout..... 59
4.2.2.1	Algorithm Details..... 59
4.2.2.2	Graph Layout Example..... 61
4.2.3	Implementation 64
4.3	Debugger Design 65
4.3.1	Data Extraction 65
4.3.2	Reverse Engineering and the Debugger Subsystem 66
4.4	Experimental Design..... 72
4.4.1	Experimental Objectives..... 72
4.4.2	Criteria 73

	Page
4.4.2.1 Software Visualization.....	73
4.4.2.2 Debugging.....	74
4.4.2.3 Usability.....	75
4.4.3 Approach.....	76
4.5 Summary.....	77
5 Results and Analysis.....	79
5.1 Introduction.....	79
5.1.1 Presentation of Results.....	79
5.2 Testing Preparation and Procedure.....	80
5.3 Phase I – Regression Test.....	80
5.4 Phase II – General System Display Analysis.....	82
5.5 Phase III – Visual Debugging Analysis.....	84
5.5.1 Small System Testing.....	84
5.5.2 Medium System Testing.....	86
5.5.3 Large System Testing.....	89
5.6 Phase IV - Usability Tests.....	93
5.6.1 Software Visualization.....	93
5.6.2 Debugging.....	96
5.6.3 Usability.....	97
5.7 Summary.....	100
6 Conclusion and Future Work.....	102
6.1 Introduction.....	102
6.2 Motivation and Objectives.....	102
6.2.1 Background.....	102
6.2.2 Research Impact.....	103
6.3 Future Developments.....	107
6.4 Summary.....	108
Appendix: Usability Survey Responses.....	110
Bibliography.....	125
Vita.....	129

List Of Figures

	Page
Figure 1 - Overview + detail technique, with intermediate view [CS99].	19
Figure 2 - Flat view of article.	20
Figure 3 - Furnas fisheye view of text from Figure 2	21
Figure 4 - Example of graph with layered layout.	25
Figure 5 - Screenshot of JarInspector which uses topology-shape-metrics approach.	27
Figure 6 - JPDA system overview.	36
Figure 7 - High-level system architecture.	48
Figure 8 - system data flow model.	48
Figure 9 - Pseudo code for data flow	49
Figure 10 - Display figure hierarchy for ArgoUML.	50
Figure 11 - NovoSoft UML hierarchy.	51
Figure 12 - Initial Class diagram.	58
Figure 13 - Class diagram following application of Focus+Context techniques.	58
Figure 14 – Reversed version of Figure 13	58
Figure 15 – Initial layout from original ArgoUML.	62
Figure 16 – Class diagram layout with focus+context applied.	62
Figure 17 – Class diagram illustrating hierarchical layout schema	64
Figure 18 - Connector options.	66
Figure 19 – Pseudo code for Reverse Engineering algorithm	67
Figure 20 – Debugging screen layout	68
Figure 21 – Accessing object information in ArgoUML	68
Figure 22 – Jtree representation of object information in JSwat	69
Figure 23 – Breakpoint setting in ArgoUML	70
Figure 24 – The ArgoUML Debug Control buttons	71
Figure 25 – Adapter Pattern used in the visual debugger system	72
Figure 26 – Diagram Panel after two OO agents are connected to ArgoUML	81
Figure 27 – ArgoUML with two agents loaded and focus point on the second agent	82
Figure 28 – 9 nodes linked up with no visualization techniques applied	83
Figure 29 – Nodes with degree of interest display and hierarchical layout applied	83
Figure 30 – Typical display of a small system in an IDE	85
Figure 31 – Initial display on ArgoUML after loading AFITShape	85
Figure 32 – Package Diagram representation of JUnit in TogetherControlCenter	87
Figure 33 – Initial view of JUnit	87
Figure 34 – The JUnit display after application of filtering techniques	89
Figure 35 – Initial graph display for ArgoUML	91

	Page
Figure 36 – ArgoUML later in its execution cycle	92
Figure 37 – ArgoUML layout after filtering is applied	93
Figure 38 – Software Visualization Criteria Survey Results	95
Figure 39 – Debugging Criteria Survey Results	97
Figure 40 – Screen Criteria Survey Results	98
Figure 41 – Learning Criteria Survey Results	98
Figure 42 – System Capability Survey Results.....	99

List Of Tables

	Page
Table 1 – Survey: Software Visualization Criteria	73
Table 2 – Survey: Debugging Criteria	75
Table 3 – Survey: Usability Criteria-Screen.....	76
Table 4 – Survey: Usability Criteria-Learning	76
Table 5 – Survey: Usability Criteria-System Capabilities.....	76
Table 6 – Regression Test Procedure.....	81
Table 7 – Survey Results showing average scores for each criteria.....	94

Abstract

The Department of Defense (DoD) is developing a Joint Battlespace Infosphere, linking a large number of data sources and user applications. Debugging and analysis tools are required to aid in this process. Debugging of large object-oriented systems is a difficult cognitive process that requires understanding of both the overall and detailed behavior of the application. In addition, many such applications linked through a distributed system add to this complexity. Standard debuggers do not utilize visualization techniques, focusing mainly on information extracted directly from the source code. To overcome this deficiency, this research designs and implements a methodology that enables developers to analyze, troubleshoot and evaluate object-oriented systems using visualization techniques. It uses the standard UML class diagram coupled with visualization features such as focus+context, animation, graph layout, color encoding and filtering techniques to organize and present information in a manner that facilitates greater program and system comprehension. Multiple levels of abstraction, from low-level details such as source code and variable information to high-level structural detail in the form of a UML class diagram are accessible along with views of the program's control flow. The methods applied provide a considerable improvement (up to 1110%) in the number of classes that can be displayed in a set display area while still preserving user context and the semantics of UML, thus maintaining system understanding. Usability tests validated the application in terms of three criteria – software visualization, debugging, and general system usability.

VISUAL DEBUGGING OF OBJECT-ORIENTED SYSTEMS WITH THE UNIFIED MODELING LANGUAGE

1. Research Introduction

1.1 Problem Statement

Information systems have become a critical aspect of successful military operations. The Joint Battlespace Infosphere (JBI) proposes “a combat information management system that provides individual users with the specific information required for their functional responsibilities during crisis or conflict” [USA00]. Such a system is inherently complex with distributed communications among numerous and diverse data sources. Any system of such complexity is difficult to design, analyze, test and debug. To aid in this activity, this research develops interactive program visualization techniques to facilitate debugging of object-oriented systems both as stand alone and connected in a distributed system.

Debugging is a crucial activity in any software development process. In an environment such as the JBI, where a large number of user applications and databases are linked, debugging is further complicated. Further to this, the introduction of object-oriented programming has presented challenges in the traditional way of debugging software. Visualization techniques exist that aim to aid this debugging process, however whether these techniques actually help is yet to be determined [BDM97]. It is well known that visualization greatly increases human-computer data bandwidth and since debugging of software generally involves large amounts of data, it is appropriate to incorporate visualization in the debugging of software. This thesis effort designs and implements a methodology that enables developers to analyze, troubleshoot and evaluate

object-oriented systems using visualization techniques. It then validates the effectiveness of the developed tool through controlled user experiments.

1.2 Background

Prior work by AFIT in the visualization field established a Java based system primarily concerned with the analysis of distributed systems based on inter-agent communication [KIL02]. Further work by AFIT produced another Java based system that employs UML to analyze distributed systems [MUS03]. The features of Java ensure easy platform independence in distributed monitoring systems.

Debugging software can result in enormous amounts of data. Hence the use of visualization techniques can aid in this process by increasing the amount of data that can be taken in by the user. The traditional debugging method of character reading can be significantly improved by the use of visualization as it increases the human-computer data bandwidth. Furthermore, pre-cognitive processing of the data is more efficient thus requiring less effort from the user [WAR00].

Telles states that one of the most important aspects of debugging is the ability to take the “big picture” into account [TH01]. The user should base their hypothesis of the cause and location of the bug on the overall system behavior. As such, the inclusion of UML modeling to represent the system is considered advantageous. Although a sequence diagram representation provides assistance in debugging in many existing tools, they are unable to provide an overview of the total system. Class diagrams are able to describe the system adequately in most cases. The software engineering and design industry is very familiar with class diagrams and uses them frequently during system design and they are in fact the most commonly used type of UML diagram [CB99]. The system

developed at AFIT based on version 0.10.1 of ArgoUML – an open source CASE tool – is modified to enhance visualization features with the aim of improving space efficiency and its performance to make it useful for the purposes of real time debugging. This thesis investigation shows that the use of dynamic UML class diagrams coupled with visualization techniques such as focus + context, hierarchical graph layout algorithm, and filtering assists the user during the debugging process.

1.3 Research Focus

This research focuses on developing more effective methods for applying visualization to the debugging of object-oriented systems. These methods are implemented in an existing Java based system. Similar to other software visualization tools, it is necessary to view the system being debugged from various levels of abstraction to allow the user to analyze and correct the modules that are most likely causing errors. The use of multiple levels of abstraction allows the user to gain a global view of the system while having access to detailed information essential to effective debugging.

The domain of the solution is limited to Java based applications to simplify the data extraction process. The Java Platform Debug Architecture (JPDA) is used as the debugging framework while ArgoUML executes the visualization component of the system.

1.3.1 Objectives

The primary objective of this thesis effort is to *develop visualization methodologies that allow more effective debugging of object-oriented systems*. This research must meet several requirements in order to achieve this primary goal. Initially, the debugging

process is analyzed to determine key techniques and requirements. The major requirement in the software debugging process is to gain an understanding of system behavior, which includes both static and dynamic information [TH01]. Wide varieties of views are necessary in order to present users with as wide a scope of information as they require without being too overwhelming.

Any system with this goal should *automate the method of data extraction*, as large amounts of data are required for this debugging process. Displaying the program structure aids in the presentation of data. However users may not require access to all the information for many classes at any given time; hence the display of the system should accommodate for this case.

It is essential that users performing the debugging *have access to multiple levels of abstraction* to aid in program understanding. Users should have views ranging from source code to high-level architectural views showing where the current code segment fits in with the rest of the system and the relationships it maintains with the modules that surround it.

A display of the program structure is an important part of the visualization created for the debugging process. The ability to see the *source of the debug data in a well-known layout representation* assists the user in detecting program errors. It is crucial that the layout be in a well-known form to take advantage of inherent user knowledge of program layout – showing inheritance relationships vertically for instance. The program layout information should be rapidly reverse-engineered without the need for the debuggee's source code which may not be available or current.

As with all visualization systems, the application must address several key issues such as the use of color, patterns, animation, and space efficiency. *Effective use of these visualization techniques are required to maximize the benefits of the system.* To enhance visual processing by the user, other *visualization techniques are incorporated to allow a global view with detail for areas of interest.*

Another major objective of this thesis effort is to *evaluate the effectiveness of the resulting tool through user experiments.* A combination of the three most common forms of usability testing – testing, inspection and inquiry [USA] – are used to achieve this task.

1.3.2 Approach

In order to extract run-time data from the system for debugging, the JPDA library is used. JPDA allows for distributed debugging to take place on any platform running the Java Virtual Machine (JVM). A Graphical User Interface (GUI) is responsible for the display and collection of input from the user. This comprehensive GUI provides various levels of abstraction from system structure right down to source code level. It allows the user to select any variable for expression evaluation and trace through code to identify observed errors.

The system obtains the program structure to be represented through the reverse engineering of the extracted data from the JPDA. The main visualization is presented as a UML class diagram and is supported by focus+context and information filtering techniques. The system displays diagrams that follow UML conventions in accordance with UML version 1.4 [OMG00].

The programs that are handled by the developed application may be extremely large, which implies the structural models are very large. Focus+context visualization

techniques and fisheye views allow a much larger portion of the program structure to be displayed on the screen while allowing the user access to detailed information for an area of interest. These have already been employed in existing systems but are enhanced in this research to provide a more efficient display.

The visual system includes a graph layout algorithm to further increase the space efficiency of the visualization. The layered graph layout algorithm represents the hierarchical nature of UML class diagrams by using layering based on inheritance relationships within the program being debugged. The visual transformation system applies the algorithm in two phases to preserve context of the system where it already exists. More specifically, smooth animation is applied in the visual transformation of the changing graph so that ‘context’ or ‘understanding’ of the visual representation is maintained by the user.

The debugger system highlights the execution path of programs on the structural view. This allows the user to see what the program is doing, rather than having to comprehend trace data.

As well as the highlighting of the execution path, a variety of visualization tools are included to increase the effectiveness of the system. This research selects color, layouts, filtering and animation based on the foundations of program visualization to achieve maximum effect [WAR00]. Focus + context is also included to allow a more complete view of the UML object diagrams presented by the system.

This research implements a debugging system with the described features to evaluate the effectiveness of the proposed visualization techniques. Testing takes place to measure the effectiveness of these techniques in various environments. These tests are conducted

as an individual effort and validated through user experiments, involving subjects at AFIT. Testing includes the use of questionnaires and interviews to correctly determine the effectiveness of the resultant application.

1.4 Research Contributions

The effectiveness of the application is verified through a usability test and an analysis of it in terms of a chosen set of criteria and performance metrics. The effectiveness was measured in terms of software visualization effectiveness, debugging effectiveness and general system usability. User responses to both the software visualization and debugging effectiveness of ArgoUML determined that the method of providing a UML diagram assisted by the visualization techniques provided in this research was indeed effective for the purposes of debugging object oriented systems. Some usability aspects of the application need further refinement, primarily in the area of system performance, however the general usability criteria scores from the conducted tests still gained positive results. The presentation of UML class diagrams with a focus+context visualization feature and a hierarchical graph layout algorithm provides a considerable improvement (up to 1110%) in the number of classes that can be displayed in a set display area while still preserving the semantics of UML and thus maintaining system understanding. This great improvement in space efficiency allows users to debug large systems more easily based on the amount of data that is made manageable in a set information space. Chapter 6 provides a comprehensive summary of how each of the goals of this research was met.

1.5 Thesis Overview

Chapter 2 provides a literature review discussing topics such as software visualization, debugging, and methods for testing user interfaces. Chapter 3 outlines the methodology for obtaining the goals outlined in section 1.3.1. Chapter 4 presents design considerations and related details of implementing a UML based visual debugger as well as the method used to determine its effectiveness. This chapter is decomposed into three sections – that of Visual Design, Debugger Design and Experimental Design. Chapter 5 describes the application of the developed system and discusses results from analyzing its effectiveness as a debugger. This includes explanation of the evaluation techniques used prior to presenting results. Chapter 6 presents conclusions and potential avenues for future work.

2 Background

2.1 Introduction

This chapter discusses different ways in which Software Visualization (SV) has been used in the field of debugging and visualization of Object-oriented Software. Debugging is crucial to the success of any software development process, regardless of whether it is for a commercial, an enterprise or a personal application. Techniques are being developed to reduce the time and effort spent on debugging, including the use of visual representations of program behavior [BDM97]. These techniques have major relevance to this research. The topics covered in this chapter include SV and more specifically the domain of program visualization. This is followed by a description of visualization methods as well as graphical concepts such as layout algorithms. It then discusses issues in debugging, including methods in detecting bugs and how SV can aid in this process. It also covers visualization applications and visual representations of software, both for its static and dynamic characteristics and methods in determining the effectiveness of SV for debugging. Since the research involves evaluating the effectiveness of a visualization application, different approaches in conducting this are also discussed. The following sections provide a description of each topic along with its relevance to the research.

2.2 Software Visualization

Price *et al.* formally define Software Visualization to be “the systematic and imaginative use of the technology of interactive computer graphics and the disciplines of graphic design, typography, color, cinematography, animation, and sound design to

enhance the comprehension of algorithms and computer programs” [PBS93]. As can be deduced from this definition, SV comprises two main fields – algorithm visualization (AV) and program visualization (PV). The main area of SV that is relevant to this research is that PV, which encompasses debugging as well as program analysis and software engineering. Algorithm visualization is also commonly known as algorithm animation and is mainly used for educational purposes.

Visualization has greatly contributed to the human ability to solve problems. According to Card *et al.* [CS99], information visualization amplifies cognition in the following ways:

- Increased Resources
- Reduced Search
- Enhanced Recognition of Patterns
- Perceptual Inference
- Perceptual Monitoring
- Manipulable Medium

In other words, visualization greatly increases the bandwidth at which humans can accept data and make sense out of what otherwise would be a complex representation.

2.2.1 Program Visualization

This research evaluates techniques for visualizing program execution. According to Roman and Cox, program visualization may be viewed as a mapping from programs to graphical representations [RC93]. This concept is crucial to this research as it provides the basis from which software can be transformed into a graphical representation to enable debugging.

Stasko affirms that an effective SV environment builds on the text based techniques that are generally used, with techniques such as pretty-printing of code to make better use of the human computer interface [STA98]. However, to improve on these methods and for ease of understanding, the represented visualization must allow the user to filter the visualization. That is, the user must be able to control the visualization such that certain information that is not required at a certain time can be omitted.

It is difficult for the user to have knowledge of erroneous behavior from the observed system in advance. Therefore, the system must provide enough flexibility to capture this dynamic data. Runtime data is more difficult to display than static data as it must be displayed dynamically. Taking this into account, Stasko developed the following principles for dynamic software display [STA98].

- Animation – animation allows for the display of temporal relationships. It can be constructed by modifying the size, shape, position or the appearance of an object.
- Metaphors – this refers to symbols that represent objects in a way that makes the object more easily understood. The idea behind using metaphors is to minimize cognitive load on the user.
- Interconnection – this refers to the use of various techniques to represent relationships between components and their patterns of behavior
- Interaction – this refers to the user’s ability to interact with and manipulate the data that is visually presented.

Animation plays an important part in this research as it provides the means by which the user can maintain context in a changing display. Context is an important concept in Human-Computer Interaction (HCI) and in essence refers to what determines the meaning of the display [SBG99]. The application of smooth animation ensures that

updates to the display do not result in confusing transformations caused by ‘sudden’ changes. The final application minimizes the complexity of the animation by limiting the number of behaviors that are animated and selecting simple animation types.

2.2.1.1 Program Visualization Taxonomies

Visualization of program structure, control flow and data has long been a part of software development. Examples include flow charts, class diagrams, state-charts, pretty printing of code, and algorithm animation. To evaluate PV techniques, Roman and Cox propose a taxonomy consisting of five criteria – scope, abstraction, specification method, interface and presentation [RC93].

Scope is concerned with which aspects of a program are being visualized. The domain of an individual visualization is a program. Formally, a program can be characterized by its code, data state, control state, and execution behavior. Visualization systems often limit their scope to a subset of these program aspects; however, a broader scope is generally preferred in order to provide more options to the viewer. Scope is an important criterion as it determines exactly what the application intends to visualize.

Abstraction refers to the degree to which the visual representation is removed from the actual code. The purpose of abstraction is to decrease complexity by aggregating details into entities, without losing the descriptive qualities of the system being visualized. Roman and Cox define five levels of abstraction [RC93] as follows:

- Direct representation.
- Structural representation
- Synthesized representation
- Analytical representation

- Explanatory representation

With *direct* representation, some aspect of the program is mapped directly to a visual representation. Some examples of direct representations include setting gauges to indicate the values of variables, or two-dimensional representation of binary trees, or color encoding of values stored in an array. UML operates on a similar principle but leads more towards the *structural* representation, which encapsulates multiple sub-components into a single object to conceal information that is not currently of interest. For example, diagrams and graphs typically used to depict program structures (like UML) and network connectivity structurally represent what otherwise would be a complex object, and reduce it to subcomponents treated as a single simple object, with its internal structure hidden. The representation simply conveys the information in a more economical way by suppressing aspects that are not relevant to the viewer. *Synthesized* representations go even further, deriving and presenting information that is not directly in the program. The other levels of abstraction for PV that have been formally defined – *analytical* and *explanatory* representations – are more sophisticated and go beyond presenting simple representations of the program state. They use a variety of visual techniques to illustrate program behavior and are added for the sake of improving the aesthetic quality of the presentation to communicate the implications of certain events, and in order to focus the viewer's attention. An example of an analytical representation would be highlighting an area that has already been computed in a dataset that is being evaluated by a certain sorting algorithm, to give insight as to the direction and behavior of the algorithm with respect to its input. An example of an explanatory representation, on the other hand, would be to apply an animation to a sorting algorithm that may

smoothly swap the position of two objects to represent the exchange of two array elements. In this case, the intermediate positions of the objects during the animation do not represent any actual states of the computation. Abstraction is useful for compacting large amounts of program information for presentation in a display of limited resolution. Multiple levels of abstraction are preferred for any visualization system.

Designers of PV applications must be able to identify the specific program aspects that are to be extracted and *how* they are displayed. *Specification methods* may be predefined by the visualization system or may require annotation of the code. Alternatively, visualization designers may use a declarative language or manipulation to specify the mapping between program aspects and visual objects. Specification methods should require minimal effort from the viewer. The ease and efficiency with which these operations can be done is a major factor in the utility of a visualization system; however, systems should also provide sufficient flexibility to tailor the information for specific viewer needs.

The *interface* consists of graphical objects presented to the viewer and interaction with the display using buttons, menus, and other controls or through direct manipulation of the graphical objects. The interface should be intuitive and easy to understand and use. In general, direct manipulation interfaces tend to be more intuitive than interaction through controls.

Presentation refers to the semantics of the graphical objects that are presented to the viewer. The presentation is that aspect of the visualization that facilitates interpretation and understanding of the graphics. This covers issues concerning human cognition and effective visual communication such as the use of color, size, spatial relationships and

other visual concepts to depict additional meanings. Presentation semantics must be sufficiently abstract and powerful to reduce the cognitive load on the viewer. It is especially important to capture the complex interrelationships between various program aspects.

Debuggers such as those in Borland's TogetherControlCenter illustrate the low end of PV techniques [BOR02]. This environment provides little abstraction and minimalist interface capabilities, that is, debugging takes place with a GUI but the data itself has not undergone any transformation. It is typical in these environments for users to be able to interact with textual representations of code and data state using simple text, menu, or button commands. However, users have very limited capability to specify the desired information and visual presentation. Presentation semantics are no more than those provided by the underlying code. This research attempts to come up with a more effective means of visualizing software by satisfying all of the visualization criteria discussed in the taxonomy above. According to Ungar *et al.*, bringing the programmer closer to the program promotes immediacy and helps the programmer understand, change and ultimately debug [ULF97]. The vision of this research is to attain a reasonable level of immediacy in debugging through the dynamic visualization of program structure.

2.2.1.2 The Unified Modeling Language (UML)

An appropriate candidate for the visual representation of software is UML. UML provides diagrams to model static system information and different aspects of dynamic system behavior [OMG00]. It is the standard used by many developers in the software industry to model and represent software characteristics. UML comprises a variety of diagrams relevant to software engineering, including class, use case, sequence, state,

collaboration and deployment diagrams. This research investigates the use of UML as a visual representation for debugging.

The most common UML diagram that is used is the Class Diagram [CB99]. A class diagram comprises a group of classes and interfaces that reflect the important entities of the system being modeled, and the relationships between those classes and interfaces. Classes in a class diagram are interconnected by association relationships, and in a hierarchical fashion, by generalization and specification relationships consisting of a set of or an individual parent class and related sub-classes under those parent classes. An important thing to remember is that a class diagram is a static view of a system. It is used to represent the structure of the system being modeled. However, prior work by Jacobs and Musial [JM03], has seen the use of animated class diagrams to represent some dynamic behavior of software. This was achieved by drawing the class diagram node of each entity of the system being analyzed as it is called. Object diagrams, which model instances of classes are also defined by UML. This type of diagram is used to describe the system at a particular point in time and for this research, may be more appropriate for debugging purposes than class diagrams. From a notation standpoint, object diagrams are very similar to class diagrams.

A useful UML representation for debugging may be that of a *state* diagram. A state diagram captures an object's dynamically changing set of attributes called states. However, a state diagram representation is usually dependent on 'interesting' events that the system possesses. This presents challenges in how to parse such events for a debugging application. Specification methods presented in current SV literature are unsuitable for such a task since the information required is not explicit at the source code

level. Therefore the specification method used for such an application would likely require a great amount of user intervention, which as discussed in section 2.2.1.1, is undesirable from a specification viewpoint. Possible debug platforms that are considered for this research are covered in further sections of this chapter.

2.2.2 Spatial Visualization

There are a number of complications associated with designing a system to visualize object-oriented Programs. Software systems can consist of hundreds or thousands of unique classes with complex inheritance and containment hierarchies and diverse patterns of dynamic interaction [DHKV93]. Analysis and comprehension of such a software system requires both a high-level overview of the system structure (consisting of numerous classes and the relationships among them) and a detailed examination of the characteristics of individual classes or small subsets of classes. SV techniques should provide access to both high-level and detailed views.

Systems that consist of several hundreds of thousands of lines of code and comprise thousands of classes pose challenges in terms of how to visually represent the big picture of the system. One of the biggest problems in the science of visualization is the small window area available from which to view large information spaces. Visualization methods exist that can make these large amounts of data much more manageable to the user. This section focuses on those techniques that show a complete view of the data while providing detail for an area of interest. This is known as spatial visualization and is concerned with maintaining a view of the whole information space while pursuing detailed analysis of a portion of it. There are two common spatial visualization

techniques. Focus+context shows detail within the existing display, and overview + detail maintains separate areas for displaying the different levels of detail.

2.2.2.1 Overview+Detail

As discussed, browsing large information spaces like software can pose a big problem to the human visual system, which only has a limited bandwidth for the intake of information. Card states that it is essential to maintain an overview [CS99]. Overview reduces search, allows easier detection of patterns, and helps the user to choose the next move. The display of the detail on the other hand allows the user access to meaningful data rapidly. This combination of the two views allows the user to track a region of interest in the global display while having a detailed view for further work.

This method usually incorporates some highlighting of the global view to indicate the current region of detailed display. The detailed information is then presented elsewhere (e.g., in the background). The user then easily tracks updates to the position in the global context. Figure 1 shows an example of this technique applied to code in a software evolution application called SeeSoft; the highlighting is shown on each higher-level view [CS99]. Shneiderman claims that this zoomed detail view should have an effective zoom factor of between 3 and 30 [SHN98].

Card stipulates that the detailed view can be presented either in a different location on the screen at the same time (space multiplexing) or one at a time (time multiplexing) [CS99]. There are obviously careful tradeoffs in the use of space that need to be made in the design of overview+detail applications.

The detailed view is generally a scaled view of the overview. However it may also use a different representation to present more clear and detailed information. User

control of the views becomes an important issue in the design of overview + detail visualizations. Zoom-and-replace is a technique used with overview + detail where a mouse-click by the user on a location in the global display results in the display of the selected location in detail.

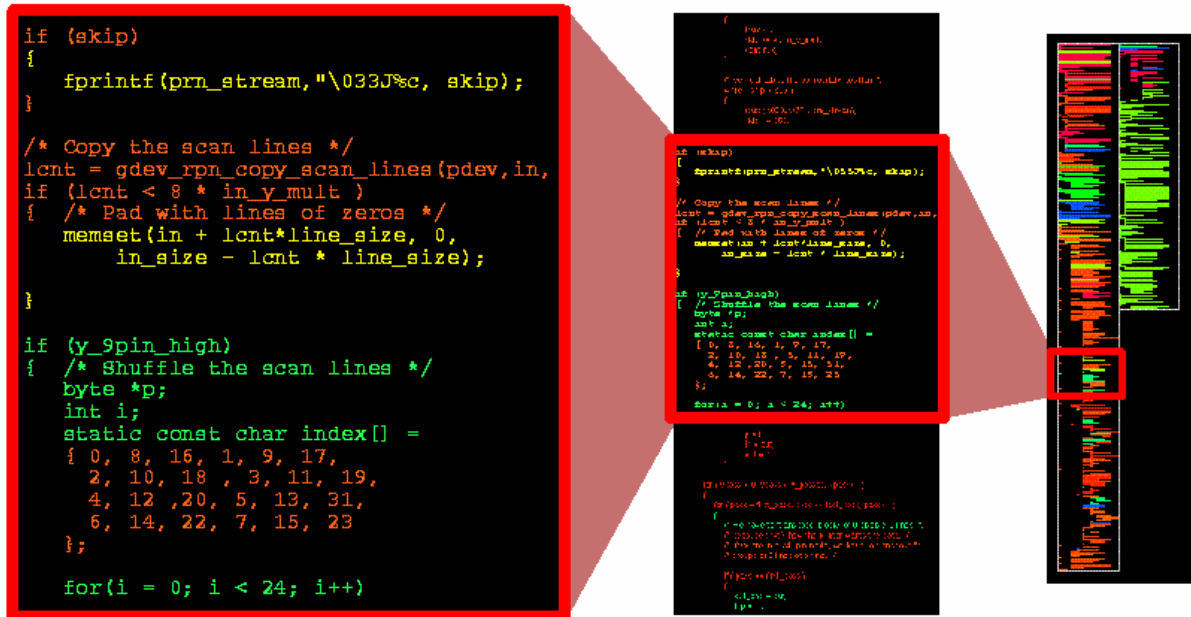


Figure 1 - Overview + detail technique, with intermediate view [CS99].

2.2.2.2 Focus+Context

Card also reviews the focus+context technique. The concept behind it is based on three premises [CS99]. The first is that the user requires both the overview (context) and a detail view (focus) simultaneously. Secondly, there may be different requirements for information in the detail view than in the overall display. Third, the visualization combines this information into a single dynamic view, similar to human vision.

Work by Furnas with Fisheye views showed that it is beneficial to combine the two views as user attention drops off away from the areas of detail [FUR81]. The Fisheye view is a Focus+Context technique that is analogous to a fisheye lens whereby objects

nearby are magnified and distant objects are shrunk. It illustrates how the space-time efficiency for the user can be greatly improved as it takes advantage of a larger amount of useful information per unit of area and reduced amount of time to find that useful information [CS99]. Figure 2 shows a standard flat view of text from a journal paper displaying sequential headings. Figure 3 shows a fisheye view of the same text with focus+context applied, showing some localized headings with context provided by major headings for the entire chapter.

70	i i. logarithmic compression, under user control
71	i i i. branching factor is critical
72	c. Iso-DOI contours are ellipses
73	e. The dangling tree
74	Figure 2: shows the dangline DOI contours
75	f. Changing focii --lowest common ancestor
76	B. Examples of Fisheye for Tree Structured Files
77	1. Indent Structured Files: Structured Programs, Outlines, etc.
78	a. Examples: Programs, Outlines, etc.
79	b. Usually ordered -fisheye is compatible
80	c. Specific example 1: paper outline
81	<i>Figures 3,4,5: outline, regular and fisheye views</i>
82	i. some adjacent info missing
83	ii. traded for global information
84	d. Comment: standard window view = degenerate fisheye
85	e. Specific example 2: C program code
86	Figures 4: C-program, regular and fisheye views
87	i. What is shown
88	ii. What is traded for what
89	f. Other indent structures: bioI. taxon. , org. hierarch. ..
90	2. Count-Until: A Simple Generalization of Indent Structure
91	a. Other similar structures
92	i. in addition to indent

Figure 2 - Flat view of article.

1	The FISHEYE view: a new look at structured files
2	I. ABSTRACT
3	II. INTRODUCTION
...23	III. GENERAL FORMULATION
...51	IV. A FISHEYE DEFINED FOR TREE STRUCTURES
52	A. The Underlying Fisheye Construction and its Properties
...76	B. Examples of Fisheye for Tree Structured Files
77	1. Indent Structured Files: Structured Programs, Outlines, etc.
78	a. Examples: Programs, Outlines, etc.
79	b. Usually ordered -fisheye is compatible
80	c. Specific example 1: paper outline
81	<i>Figure 3: outline, regular and fish views</i>
82	i. some adjacent info missing
83	ii. traded for global information
84	d. Comment: standard window view = degenerate fisheye
85	e. Specific example 2: C program code
...89	f. Other indent structures : biol. taxon. , org. hierarch. ..
90	2. Count-Until: A Simple Generalization of Indent Structure
...100	3. Examples of the Tree Fisheye: Other Hierarchical Structures
...106	V. FISHEYE VIEWS FOR OTHER TYPES OF STRUCTURES
...117	VI. A FEW COMMENTS ON ALGORITHMS
...140	VII. OTHER ISSUES
...162	VIII. CONCLUDING REMARKS AND SUMMARY

Figure 3 - Furnas fisheye view of text from Figure 2

In the case of class diagrams, not only is focus+context applicable to the whole diagram but for each node in the diagram as well. Large systems (and in some cases, even small applications) can contain classes with many attributes and methods. Showing all these attributes in the screen not only compromises the efficiency of the display but can also cause confusion. For this research, it is therefore reasonable to consider applying focus+context to individual nodes, as necessary to increase the efficiency of the displayed information.

There are a variety of methods that systems can implement to provide selective reduction of the presented information. These include information filtering, selective aggregation of related information, micro-macro readings, highlighting, and distortion [CS99]. For class diagrams, the technique of information filtering can be useful, especially when nodes in the diagram become too large due to the many attributes and methods a class can have. Prior work by Jacobs and Musial on ArgoUML made use of

selective aggregation [JM03]. This technique hides aggregate elements within other components. It achieves a focus+context view by collapsing away hierarchical structures from the user's focus. When the user then brings the context into focus, the hierarchy expands to reveal the aggregate relationships contained within. Filtering by way of information hiding is also useful when it comes to UML diagrams. Since UML diagrams are composed of different compartments, allowing the user to enable or disable the display of compartments would be useful in increasing the display's space efficiency. DIAGEN is another application in which focus+context has been applied [KM01]. This application provides an adjustable detail level for editing large diagrams. It uses selective aggregation for UML diagrams to hide the contents of packages and other components to reduce the amount of detailed information displayed.

2.3 Graph Layouts

Since this research proposes the use of UML representations for the visualization of object-oriented systems, a major factor that comes into play is the graph layout algorithms that are used. Efficient use of the small information window available is a key to SV and follows from Tufte's principles of graphical excellence, one of which preaches to present many numbers (or information) in a small space [TUF01].

Aesthetics is an important consideration in graph drawing as it describes the qualities that define a 'good' and 'visually pleasing' graph. From statistical analysis, several principles that influence the aesthetics of a particular graph were determined as follows by Batini *et al.* [BFN85]:

- Minimize line crossing – keep the number of times that lines cross to a minimum.

- Minimize area – keep the area that the graph takes up to a minimum by producing a compact graph.
- Minimize the sum of the edge lengths.
- Obtain a uniform edge length – try to keep each of the edges at the same lengths.
- Minimize bends – keep the number of times there is a bend to a minimum.
- Permit easy reversal of actions – this allows the user to try something without the fear of errors destroying their work.
- Support the internal locus of control – make the user the initiator of any system response so they feel they are in control.
- Reduce short-term memory load – keep displays simple with online help to information.

Drawing constraints are constraints placed on the drawing algorithm in order to create the final drawing. For example, a diagram representing a sequential model may show a number of nodes in a left-right sequence. Constraints control the layout in order to meet the expectations of users (conventions) or standards of a particular system. The constraints that can be placed on a graph drawing algorithm are as follows [DETT99]:

- Center – place a given vertex in the center.
- External – place a given vertex on the boundary.
- Cluster – place a group of vertices together.
- Pre-arrange a path.
- Draw a sub-graph with a certain shape.

There is no ultimate ‘magic’ solution for any graph due to conflicting aesthetics. Even if the chosen aesthetics do not conflict, it is computationally expensive to optimize

them all. Therefore, precedence in aesthetics is often a necessary compromise for any graph layout algorithm [DETT99].

2.3.1 Hierarchical Approach

The algorithm to layout a UML class or object diagram should preserve the hierarchical nature of the diagram. A variety of tools for displaying class diagrams present generalization hierarchies down the vertical axis [RAT02], [AUB02]. The best algorithm for this purpose is a layered drawing approach [DETT99]. A layered approach allows for a hierarchical presentation with vertices arranged in horizontal layers as shown in the example in Figure 4. One can apply this approach to any directed graph. By considering the UML diagram to be undirected for the purpose of layout, the complexity of the algorithm can be reduced. The layered approach comprises the following steps:

1. *Layer Assignment*: Assign vertices to horizontal layers, this determines their y-coordinate. If there is a gap in between layers, such as when there is an edge from L_1 to L_3 , a dummy vertex is placed at L_2 . For UML, a layer difference exists across each generalization relationship, with the top most vertex being the most generalized class. In the simple case where each vertex is the same size, the y-coordinate of each layer is determined by adding a suitable gap to the last layer.
2. *Crossing reduction*: Order the vertices within each layer to minimize edge crossings. Here, the number of times that edges cross over is reduced to create a more aesthetically pleasing drawing.
3. *Horizontal coordinate assignment*: determine the x-coordinate for each vertex. By this step, all the points are assigned a position, the dummy vertices are removed,

and the edges are all drawn. In this stage many different aesthetics can be stressed such as minimizing bends or minimizing area.

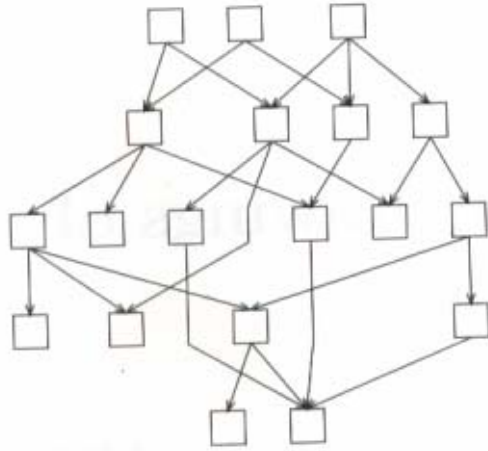


Figure 4 - Example of graph with layered layout.

The first step requires a process to determine the layer for each vertex. The standard layered approach requires that the directed graph be compact in both dimensions and that the gaps between vertices are fixed.

For the second step, Di Battista describes the insertion of dummy nodes to ensure that a relation does not directly cross more than one layer. This ensures minimization of edge crossing [DETT99]. When the vertices are of variable size however, the insertion of dummy nodes will not prevent edge crossings in the graph.

The third step requires the positioning of each vertex on a layer. The choice of algorithm for this step is dependent on user requirements.

2.3.2 Topology Shape Metrics Approach

Another approach used in Graph drawing is the Topology-Shape-Metrics (TSM) approach. This approach results in an orthogonal drawing and is commonly used in electrical engineering schematics. Orthogonal drawings only assign right angles between

connecting links. It stresses the minimization of crossings in the drawing as it applies planarization to the original graph [DETT99]. Like the hierarchical approach, the topology-shape-metrics consists of several steps [GJKKLM03]:

1. *Planarization*: This step determines the topology of the drawing. It ensures that graphs are planar by adding dummy nodes that represent crossings in the original graph.
2. *Orthogonalization*: This step determines the angles and bends in the drawing. Only multiples of 90° are assigned as angles. Generally, algorithms minimize the number of bends in this phase.
3. *Compaction*: this step assigns final coordinates to the nodes and to the edge bends. The main goal of this final step is to minimize the sum of the length of all edges and/or the area of the drawing.

Eiglsperger *et al.* discovered an effective automatic layout algorithm based on the topology-shape-metrics approach [EKS03]. Their algorithm was used in an application implemented in Java called JarInspector which showed an effective and efficient use of this approach (Figure 5).

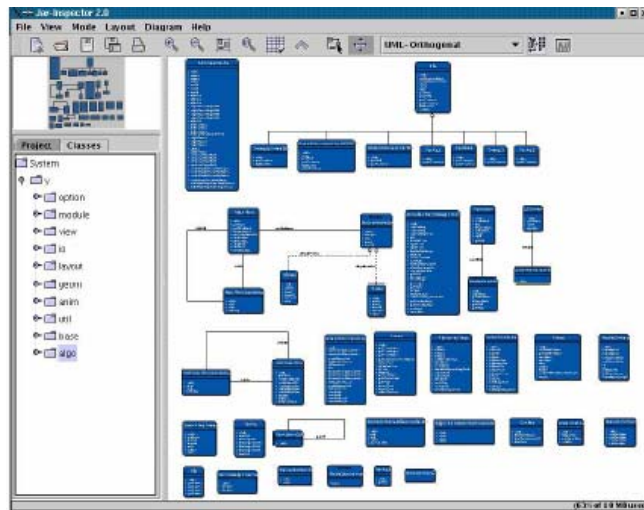


Figure 5 - Screenshot of JarInspector which uses topology-shape-metrics approach.

2.4 User Interfaces

User interactivity and control is an important subject to consider in software design. The user interface has a large effect on the usability factor of a program. As such, the system implemented in this research should strive to meet the goals required for an effective interface. With this in mind, Shneiderman discusses some goals for effective GUI design [SHN98]. The following are known as the Golden Rules for GUI Design:

- Strive for consistency – use consistent sequences of actions as well as identical terminology for prompts.
- Enable frequent users to use shortcuts – Hotkeys reduce the time taken to initiate an action for frequent users.
- Offer informative feedback – System responses to user requests should be rapid and helpful to the user.
- Design dialogs to yield closure – Group sequences of actions with an informative and meaningful end to that group.
- Offer error prevention and simple error handling – design the system so the user cannot make a simple error and offer simple instructions to recover.

- Permit easy reversal of actions – this allows the user to try something without the fear of errors destroying their work.
- Support the internal locus of control – make the user the initiator of any system response so they feel they are in control.
- Reduce short-term memory load – keep displays simple with online help to information.

2.5 Debugging Issues

Software systems are becoming increasingly large and complicated as technology progresses and user requirements in different domains expand. This makes debugging difficult and is compounded by the fact that many developers are not familiar with the subject of the system they are creating. According to Kolawa, it is not uncommon for the debugging phase to take 60-70% of the overall software development time, and that it is responsible for 80% of software project overruns [KOL96]. This is obviously brought about by the fact that programming is a human activity and is always prone to error. The following reviews some of the issues involved in the process of debugging and the essential tools for a debugger.

2.5.1 Bugs

Telles determined the human-centric definition for a bug to be “behaviors of the system that the software development team (developers, testers, and project managers) and customers have agreed as undesirable” [TH01]. A bug may originate in the requirements, architecture, design, or implementation of the system. Work-arounds may be implemented for many of the typical and expected errors, however the system should still meet the user requirements. Telles explains that software defects have a potential to become bugs, and on average 25 defects are contained in every 1000 lines of code. In

this sense, the difference between bugs and defects is that a defect is a logic error that produces unexpected results without causing the abnormal end (abend) or the undesirable behavior defined to be a bug. Therefore all bugs are defects, but defects are not necessarily bugs.

Defects can be introduced at every stage of the Software Life Cycle, but are generally caused by the designer's lack of understanding of the domain. Kan found a correlation between the number of changes and enhancements made to a system and the number of defects [KAN95] to be approximately 0.628 defects for each change/enhancement.

Telles provides a taxonomy for bugs with the following classifications [TH01]:

- Requirement Bugs
- Design Bugs
- Implementation bugs
- Process Bugs
- Build Bugs
- Deployment Bugs
- Future Planning Bugs

The types of bugs above are classified with the use of Telles' taxonomy which presents the following characteristics of bugs:

- Name
- Description
- Most common environment
- Symptoms
- Example

As the bugs can be introduced at almost any stage, their effects may also vary greatly. Telles classifies the effects as one of the following [TH01]:

- Memory or Resource Leaks
- Logic Errors
- Coding Errors

- Memory Overruns
- Loop Errors
- Conditional Errors
- Pointer Errors
- Allocation/De-allocation errors
- Multi-threaded errors
- Timing errors
- Distributed Application errors
- Storage
- Integration
- Conversion
- Hard-Coded Lengths/Sizes
- Versioning Bugs
- Reuse
- Boolean

As illustrated, bugs can be classified into many different areas and by understanding these classifications, it is easier to predict the types of problems that may arise from these different types. Each have varying magnitudes of negative consequences.

2.5.2 Debugging

Telles defines debugging as “the process of understanding the behavior of a system to facilitate the removal of bugs” [TH01]. Fixing the symptoms alone does not solve the problem, and in fact may create new ones. Anecdotal evidence suggests that the probability of introducing a new error while attempting to fix another is between 15 and 50 percent [TH01]. Holzmann describes commonalities between software bugs and real-life bugs, pointing out that both find a way to adapt to their environment [HOL02]. For example, software bugs can invade the test environment, adjust to the level of experience of the programmer, and quietly replace any one that is detected immediately to spawn one or more others.

The general process of debugging is stipulated by Telles to consist of four steps [TH01]. The first is *problem identification*, which entails determining exactly what needs to be fixed and whether or not the problem is really a bug, an enhancement (or a ‘nice to have’ fix), a document modification or simply a misunderstanding between the developer and user. Once the problem has been established and it is determined that the problem is indeed a bug, the next step is *information gathering* which involves determining how the problem is occurring, what the symptoms of the problem is, and what users expect to occur when they follow prescribed steps. After enough information has been gathered to make an educated guess as to what the root cause of the problem might be, *hypothesis formation* can be conducted. The root cause may be directly or indirectly related to the symptoms. The hypothesis should explain the symptoms and observations made in the application. The final step is *hypothesis testing* which involves matching up the hypothesis against available observations to see if it fits all evidence. If the hypothesis is indeed correct, a fix for the problem may be implemented.

The link between PV and debugging is fairly obvious. Since PV is used to aid in program understanding, and generally bugs are caused by a lack of understanding, it makes sense that the former should aid in addressing the latter. Any tool that can increase understanding goes a long way towards reducing the number of bugs in a system.

There are two factors affecting understanding in the debugging process. Firstly, one must understand how the system should operate based on customer requirements. Secondly, the implementation of the system must also be understood in order to recognize the differences between expected and actual behavior.

There is not a straightforward process to determine the location of a bug and correct it. Techniques are discussed that may or may not work depending on the situation – each has its own strengths and weaknesses. The following subsections highlight these methods.

2.5.2.1 Scientific Method

Once a failure or undesirable event is observed in debugging, using the scientific method involves forming a hypothesis about the cause of the failure. This hypothesis needs to be consistent with the observations and necessary conditions. This hypothesis can then be used to make predictions. Further modifications to the hypothesis are then conducted through testing via experiments or further observations. This whole process continues until the actual cause is found. This method works effectively when the problem is easily reproducible, for example, a particular input value results in an undesirable event [TH01]

2.5.2.2 Intuition

Intuition is the most commonly used debugging method (especially by debuggers that are very familiar with the code) [TH01]. In order to use this technique effectively, the analyst requires a thorough understanding of the system and be must able to narrow down the portion of code that is likely to be causing the bug.

2.5.2.3 Leap Of Faith

In effect, a leap of faith is simply an educated guess. The developer examines some of the symptoms and jumps to conclusion without truly examining the behavior of

the system. A leap of faith is more likely to lead one in the wrong direction than using other debugging methods.

2.5.2.4 Diagnostics

Diagnostic debugging, sometimes called *advance strike debugging*, attempts to predict in advance the problems that occur in the application, and to log these problems so that reoccurrences are easily fixed in the future. It serves a useful purpose as it is based on statistics and events that were gathered at the time the bug eventuated. However, as effective as they may seem, pure diagnostic debugging is not ideal because it is highly improbable that every error condition can be identified.

2.5.3 Debugging Tools

Since this research proposes the implementation of a debugging application, this section briefly highlights some issues in debugging tools that have been implemented.

There are many existing tools that can be used to aid in debugging. These tools provide users with a great amount of the debugging information they require. Tools that transform the data to provide more meaning rather than just providing raw data are highly preferred, hence the attempt of this research to incorporate visualization. Tools such as testing environments are effective tools for debugging and aid in limiting the scope of the tests to a likely region. Various debugging tools are able to narrow down the region of code under test using techniques such as logging and tracing. However, although the process of logging or tracing bugs in the system provides an effective way to narrow the problem area, they both suffer in terms of the amount of information they output. Telles recommends that although these techniques provide a rough estimate of where the problem might be, it is important to whittle down the large amount of output at any given

time [TH01]. Without disabling the output, the volume of data provided may actually confuse the issue.

Once the likely cause of bug has been identified and ‘problem areas’ have been highlighted, the problem can be examined in more detail with the use of mid-level debugging tools. These tools assume that the user has examined the symptoms of the problem, found the likely areas causing the symptoms, and provided an acceptable hypothesis for the cause of the bug. Examples of mid-level debugging tools include memory leak detection tools and cross-indexing and usage tools. The latter is used to see where various symbols such as global variables, methods or functions are used within the application [TH01]. Typical IDE debuggers such as Borland’s TogetherControlCenter and JBuilder have these capabilities built in.

The most direct and obvious debugging tool available is the Debugger. A debugger is a programming tool used to execute a program, test its states, update its environments, and set breakpoints [CPHEB97]. Debuggers allow programmers to stop the system in its execution at any point and examine the values of variables and in some cases step backwards over code to see what executed prior. Telles states that although Debuggers are an essential tool, they nevertheless pose problems such as giving false sense of security to the developer and modifying the environment in which the application is run, and in turn modifying the normal behavior of that application [TH01]. Thus it is recommended that debuggers be restricted to single-user products where the environment the program is running under is not the cause of the problem.

2.6 Data Extraction

This section discusses two Java debugging tools that are considered in this research. These tools are used to extract static and dynamic program data for debugging purposes.

2.6.1 The Java Platform Debug Architecture

The JPDA is a recent addition to the Java Software Development Kit (SDK) providing the capability for remote debugging of applications in the JVM as described in [SUN]. It provides portability since debugging occurs at the JVM and not at the application itself. Debugging with the JPDA can be performed by almost any two systems, a debugger and a debugee, with the JVM installed, regardless of the operating systems or configuration. The goals of JPDA are to:

- Provide standard interfaces to simplify the creation and use of Java debugging tools, regardless of platform
- Describe the complete architecture for these interfaces allowing remote debugging
- Have a modular design

The JPDA is comprised of two interfaces and a protocol. Figure 6 shows the integration of these components and these components are discussed in the following paragraphs:

- Java Debug Interface (JDI),
- Java Virtual Machine Debugger Interface (JVMDI), and
- Java Debug Wire Protocol (JDWP).

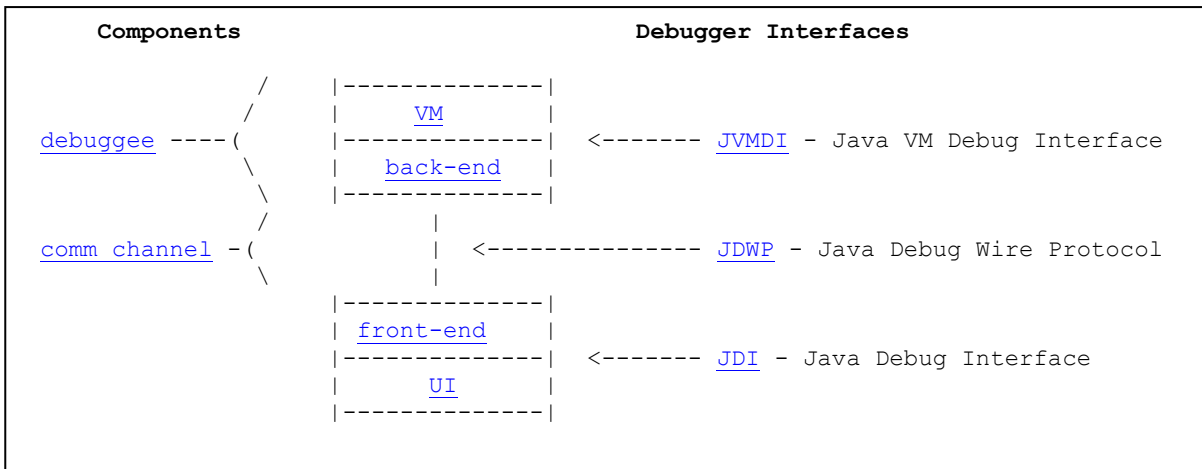


Figure 6 - JPDA system overview.

JDI provides an interface to a remote view of the debugging process occurring in a JVM that may be located in another system. JDI is the most commonly used layer for access as it is the highest level and easiest to use.

JVMDI defines the interface for the JVM to allow debugging by debugger programs running in other JVMs. This is the source of all debugger specific information. It includes requests for information from the JVM, actions such as setting and removing breakpoints, and notification when the program counter reaches a breakpoint.

JDWP is the protocol that defines how two-way transfer of information should occur between the debuggee process and the debugger front-end. It does not provide low-level detail of the actual communication mechanisms; rather it defines the format of the data transfer between the debugger and debuggee.

The JPDA provides the capability to extract data in a platform-independent way from distributed systems for debugging. Use of the JPDA Application Programming Interface (API) means implementation of debugger connections and data extraction are

relatively simple. The JPDA also reduces the effects of some of the problems caused by other data extraction techniques as described.

2.6.2 The Java Virtual Machine Profiler Interface (JVMPi)

Jacobs and Musial incorporated the JPDA into ArgoUML [JM03]. However, the results obtained were less than suitable for debugging purposes due to the slow nature of the application. This is typical of many applications that aim to present dynamic visualization of software, as they require the programmer to run the program in an environment that produces the appropriate trace data, thereby slowing program execution significantly. In the case of ArgoUML, the cause of the lag is yet to be determined. A successful visualization platform for Java programs that alleviated this problem to some extent was developed by Reiss [REI03]. His application made use of the JVMPi and byte code patching. The JVMPi is a two-way function call interface between the Java virtual machine and an in-process profiler agent. On one hand, the virtual machine notifies the profiler agent of various events, corresponding to, for example, heap allocation, thread start, etc. On the other hand, the profiler agent issues controls and requests for more information through the JVMPi. For example, the profiler agent can turn on or off a specific event notification, based on the needs of the profiler front-end [JVM03]. In short, the JVMPi is able to invoke user routines whenever profilable events such as method entry or exit, monitor waits, or garbage collection occurs.

2.7 Testing the Effectiveness of the Application

This research included experiments to confirm whether or not the final application succeeds in providing a UML based visualization that aids in debugging. Several approaches are available in achieving this validation. Only a very small number of

empirical studies have been conducted in the past to determine whether certain visualization aids help in the task of debugging [STA98]. For this research, the most appropriate method to validate the effectiveness of the final application is to use the same methods used in Human-Computer-Interaction (HCI) Evaluation. For HCI, there are four main approaches in testing [RAU96]:

1. The *interaction-oriented view* – this approach measures usability quality in terms of how the user interacts with the product. This is the most common method.
2. The *user-oriented view* – this approach measures usability quality in terms of the mental effort and attitude of the user (via questionnaires, surveys, interviews, etc.).
3. The *product-oriented view* – this approach measures usability quality in terms of the ergonomic attributes of the product itself.
4. The *formal view* – usability is formalized and simulated in terms of mental models.

The first three approaches listed above are the most relevant to this research. In visualization applications, it is necessary that user interaction is evaluated to determine the effectiveness of that particular visualization. Questionnaires, interviews and surveys are also relevant and will provide supplementary results. Although the goal of the final application is to visually debug software, the third point above is still an important criteria as the effectiveness of a particular visualization representation is negated if poor ergonomics are applied to the application. Another measure that the tests need to capture

is the debugging metrics such as the speed at which bugs are detected by the user and the application's accuracy in suggesting problem areas in the program.

A study conducted by Baecker *et al.* to evaluate a particular SV tool for debugging used an observational ethnographic study to observe how programmers could make use of the subject visualization tool [BDM97]. In usability engineering, the traditional magic number for the number of participants needed in a test is 5 (plus or minus 2), however this theory has been challenged by many recently [BAR03]. This study only used three subjects and hence the results obtained, although interesting and supported the effectiveness of the application, was nevertheless only suggestive rather than definitive. Such an experiment needs to be supplemented by other methods to ensure that the final results are more convincing. In this research, this usability heuristic of 5 plus or minus 2 participants are followed. It is considered acceptable since a significant amount of system tests are also conducted in addition to this test.

2.8 Summary

SV is an interesting area of research for developing debugging tools for object-oriented systems. However, little research has been conducted into the effectiveness of using visual representations to aid in debugging. This is mainly due to the way software developers have performed debugging in the past, preferring the traditional way of stepping through program execution until the bug eventuates, and formulating and testing a hypothesis to rectify the cause of the problem. Furthermore, until recently, dynamic PV has been computationally expensive, and the software industry is only just beginning to use new technologies that allow implementation of dynamic PV in an affordable way.

This chapter discusses the relevant and important background information on the visualization and debugging of object-oriented systems. Software visualization and more specifically, Program visualization were defined in terms of taxonomies used to evaluate applications, and the techniques associated with this domain. These techniques are crucial in the efficient use of the limited space available in PV. Graphical principles, such as graph layout algorithms and use of color were also discussed. Aesthetics that these principles add to the application determine to a large degree, the effectiveness of the displayed information. A discussion on debugging was also provided in this chapter to highlight the nature of software bugs, the different ways in which to debug them and the debugging tools that are relevant to this research. Finally, testing for the effectiveness of visualizing software for debugging was covered to illustrate the various considerations to be taken into account in the testing phase of this research.

3 Methodology

3.1 Introduction

The literature review summarized the advantages of using visualization techniques to aid in the presentation of large amounts of information. Visual techniques can allow quicker and easier cognitive access to more relevant data in large information spaces like software. The objective is to provide a visual presentation that facilitates system evaluation using high-level design representations rather than using the low-level traditional way of stepping through lines of code. The derived hypothesis from this objective is that *a visually enhanced debugging system is more effective than a standard debugger for object-oriented systems*. To determine the success of this hypothesis, a prototype system is developed through modification and enhancement of an existing platform. Tests are then conducted to prove the hypothesis. Section 3.2 provide, firstly, an initial discussion of the considerations in the methodology of the research then presents the visual and system design objectives. Section 3.3 then explains the system architecture of the developed prototype. Section 3.4 and 3.5 discusses the experimental techniques and metrics considered in testing the effectiveness of the system.

3.2 Objectives

The use of UML is a common way to communicate software design and analysis models, but once these models are implemented, the documentation containing all the UML representation is seldom used by software developers. In recent times, researchers have begun exploring UML to analyze the execution of software. For example, Mehner extends UML to incorporate execution semantics of concurrent programs [MW00]. The

divide between design and implementation is further impacted by the lack of automated support for testing and debugging with UML. Debugging any system requires structural knowledge of the software and detailed information about components (down to the source code level). Debugging of object-oriented systems is more challenging as it introduces concepts such as inheritance and polymorphism, making program understanding even more difficult. Traditional debugging involves the user creating a mental image of the structure and execution path based on source code. According to Miller, the 7 ± 2 rule makes it very difficult for humans to construct large mental models as the human short-term memory span is generally limited by this constraint [MIL56]. To alleviate this problem, this research aims to enhance an existing visual execution model for object-oriented systems. This model is implemented in a Java based application and uses UML to visually represent the execution of the object-oriented system for the purposes of debugging. Following is a detailed discussion on the visual and design objectives of this research.

3.2.1 Visual Objectives

The main intention of the visualization generated for this research is to enhance the debugging process. For effective debugging, it is essential for users to have access to various levels of abstraction. The system should present both high-level views and detailed information to allow for more precise debugging.

Regardless of the visual language used in any SV tool, large software systems still result in models that are extremely large and complicated. In the case of UML, a typical class diagram for a moderately sized system would cover many pages and be too large for efficient navigation of the whole system. Common solutions to this problem include

multi-page printouts and incorporation of an overview+detail capability. Multi-page printouts or multiple windows as used in IDEs such as JBuilder and TogetherControlCenter can show overall system structure and individual component details. However, this technique has several shortcomings such as increased visual scanning leading to increased cognitive load, difficult production and management and lack of support for interaction and dynamic editing. Furthermore, with the multiple windows method, the ‘big picture’ can be lost. Wong *et al.* states that when analyzing static information of large software systems, it is preferable to obtain an understanding of the overall, high-level structure of the software before proceeding to lower level details [WTMS95]. On the other hand, use of the overview+detail technique can provide access to overall system structure and individual component details; however, with this technique, the information is not provided in one single view. Therefore, it also suffers from drawbacks such as increased cognitive load and thus reduces the overall effectiveness of the visualization.

In their presentation of principles for a Modeling Language Design, Paige *et al.* identify nine factors to consider when designing and evaluating modeling languages — simplicity, uniqueness, consistency, seamlessness, reversibility, scalability, supportability, reliability, and space economy [POB00]. This research only considers space economy. The extent to which UML satisfies the remaining criteria will vary depending on the user’s experience with modeling languages, as most of them requires subjective judgment. Space economy requires that “models should take up as little space on the printed page as possible” since smaller models have less to understand and less work is required for modelers and tools to perform in order to maintain the models

[POB00]. UML does not effectively address space economy, leaving a lot of unused space within and between components. Graph layout algorithms are required to make the display of such graphs more space efficient. UML's inefficient use of space results in models that cover many pages for large systems. Navigating through many pages significantly increases the time to access relevant components. The problem is further compounded by the resulting inability to simultaneously view high-level system structure and individual component details. One goal of this research is to maintain the symbology and semantics of UML while enhancing its space efficiency by the use of visualization and graph layout techniques to accommodate rapid access to both high-level and detailed system information.

The visual objectives of this research are summarized as follows:

- Present information through multiple levels of abstraction.
- Present the visualization model in a familiar manner,
- Preserve context to aid in better program understanding,
- Present a dynamic visualization display such that it is suitable for real time debugging and match the changing runtime nature of programs, and
- Improve space efficiency to reduce search while presenting the user with detail for a region of interest.

3.2.2 System Design Objectives

The prototype system is based on a Java CASE tool, ArgoUML, which is described in more detail. This section addresses the functional, performance and user requirements for this system to be effective. First, the system must be user-friendly and functional. Software engineering principles, including the use of design patterns, are taken into account to make the tool more adaptable to future requirements. As many of the actions

and their associated algorithms are computationally intensive, there is also consideration of efficient processing and data storage. Objectives for the modified ArgoUML as part of this research are summarized below. It shall:

- have the ability to monitor processes for debugging,
- have the ability to automatically reverse engineer object-oriented systems,
- have the ability to display multiple systems over socket connections for debugging,
- have no effect on the functional system behavior of monitored programs,
- have minimal effect on system performance to enable real time debugging,
- provide standard debugging controls,
- provide a well designed GUI with consideration of Shneiderman's principles as stated in Chapter 2 [SHN98],
- have a low CPU load and memory usage for the monitoring system, and
- have a design adhering to software engineering principles wherever possible to ensure it adapts to future requirements.

3.3 System Architecture

This section analyzes the system architecture of the developed prototype. To simplify the development process, modifications are made to existing code where possible. This re-used code is based on the ArgoUML project [MUS03]. Figure 7 shows a high-level architectural view of the overall system.

This research adds a series of visual modifications, primarily an improved focus+context capability and animation techniques, to the ArgoUML framework. Chapter 4 provides a more detailed description of these modifications. The visual modification process retrieves input data from a NovoSoft UML Model, an open-source library for storing UML data. The type of UML diagram that was chosen to be the visualization

presentation for ArgoUML was the class diagram. Although the Novosoft UML library stores object diagram data, little documentation exists that would help in implementing the use of object diagrams within ArgoUML. Classes already existed in the original version of ArgoUML that makes use of the Novosoft UML library to draw class diagrams. These classes are discussed further in this chapter. The drawing of object diagrams for ArgoUML is further complicated by data parsing issues between the JVM and the debugger interface. Drawing object diagrams obviously would require a much larger volume of data since there are many more objects than classes. The complexity is further increased due to the way in which object information is accessed, in that the execution of the program needs to be suspended before accessing the appropriate stack frame and getting all the object information required at that particular time. Therefore, it is a very challenging task to come up with a reverse engineering procedure (or a specification method) that would automatically create an object diagram. Several attempts to efficiently extract the JPDA information into object diagram form failed. Accessing object information is still a requirement of this research even though its visualization as an object diagram is not provided.

Figure 7 also shows the integration of the debuggee JVMs and the connections from them into the rest of the system. The JPDA forms the basis for the debuggee connection system. Two types of connections to the JVM are provided: shared memory and sockets. The JVMPI was also discussed in Chapter 2. Although profiling information can be useful, especially in providing different aspects of object information, the JPDA was chosen since it provides all the debugging capability required of a standard debugger. The JVMPI is useful in providing a log of profilable events but is not appropriate for the

purposes of debugging systems as it has no actual debugging capability. That is, typical debugging functionality like breakpoint specification and stepping are not available. The option of adding the JVMPI to the ArgoUML framework was also waived as it would over-burden the system performance, since two separate threads would need to be created to communicate with each of the different interfaces.

As well as functionality, this research added a numerous amount of bug fixes to the system. These bugs varied from debugger deficiencies to display deficiencies.

The overall system architecture is primarily event-based. Modifications to the UML models are detected by observer threads for each JVM. These observers then trigger visual modifications and perform updates to the display.

Figure 8 shows a high-level data flow model for the entire system. The observer searches for new classes and class information in the debuggee JVM. The UML model is constructed and updated through a reverse-engineering process as information becomes available from the connected JVMs. The UML model is stored in a NovoSoft UML library. The `ExecutionManager` classes control the connections and pass the data to observer threads. These threads add to the UML model and call visual modification processes when required. The display of the model updates when the threads call the visual modification processes. The user is able to see the resultant view of the software along with source code and debuggee process input and output. The user may also update the display by retrieving further information from the NovoSoft model.

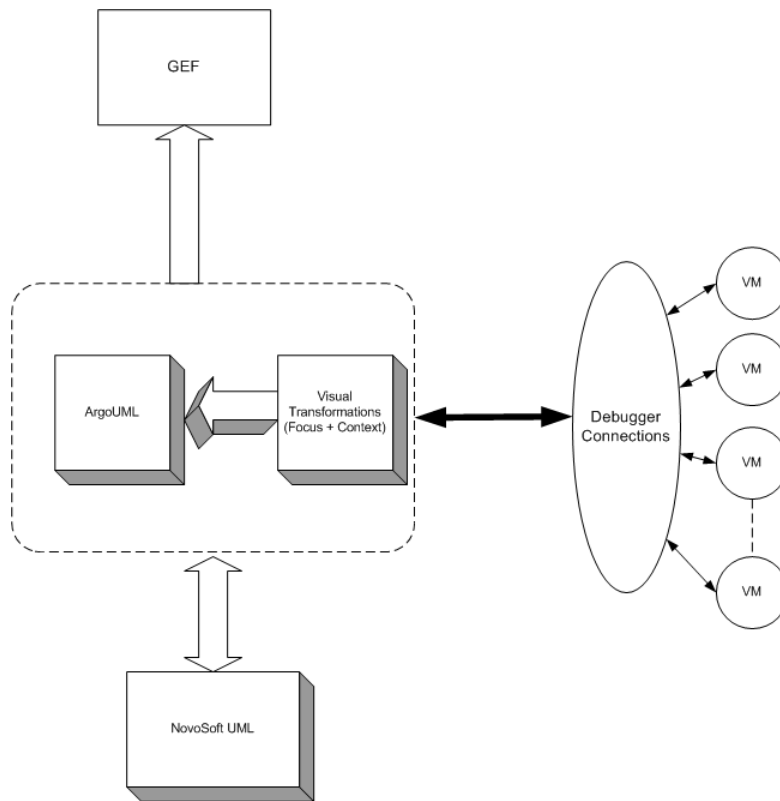


Figure 7 - High-level system architecture.

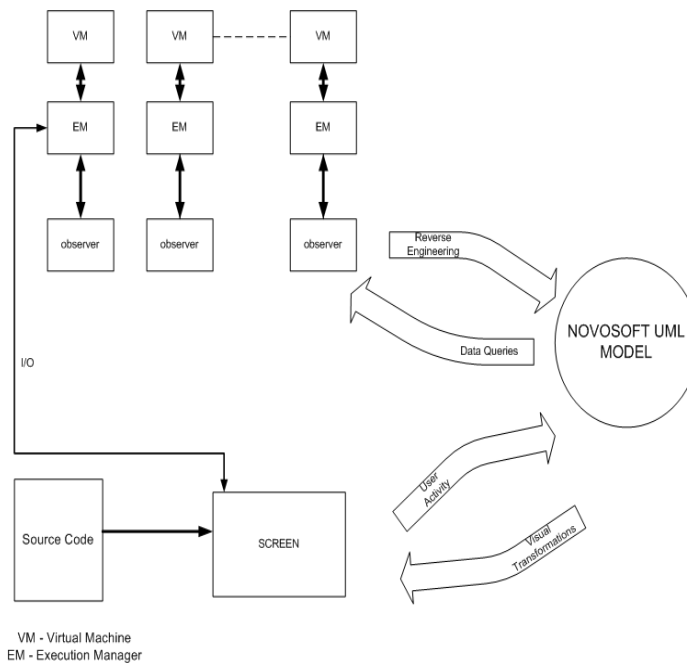


Figure 8 - system data flow model

Figure 9 is a pseudo-code representation of the processing involved for each debuggee process.

```
Connect to JVM
While JVM is running
  Do
    Perform reverse engineering (extract data out of JPDA and
                                convert to UML model info)
  Listen for changes to model
  If model changes
    Update model
    Update diagram on display panel
    For each segment
      Set focal points
      Apply Visualization changes
      Apply Graph layout
      Reposition nodes with smooth animation
```

Figure 9 - Pseudo code for data flow

3.3.1 ArgoUML

ArgoUML is derived from the Graph Editing Framework (GEF) which is a Java Class library that provides basic capabilities to display a variety of simple shapes and connect them via links [GEF]. GEF itself is not a drawing program, rather, it is a library supporting the construction of custom drawing programs for particular domains. ArgoUML was developed by a separate research project [ARG02] and it provides additional functionality to form a CASE tool. The most important parts of ArgoUML for this research are the components responsible for drawing class diagrams. This section explores these components further.

Figure 10 shows the inheritance hierarchy in the figure elements contained in both GEF and ArgoUML. The basic class used to represent a graphical component is `Fig`. `Figs` (short for Figures) are basic drawing elements such as lines and rectangles. Other classes extend `Fig` to provide the functionality required for each UML component.

Developers of GEF used a unique coding style so in most cases, the functionality can be determined from the name of the class. For example, `FigEdgePoly` is a class that extends `FigEdge` which represents a start and end point for a line. The `FigEdgePoly` class is used to draw the line between these points as a series of connected lines in order to avoid line crossings.

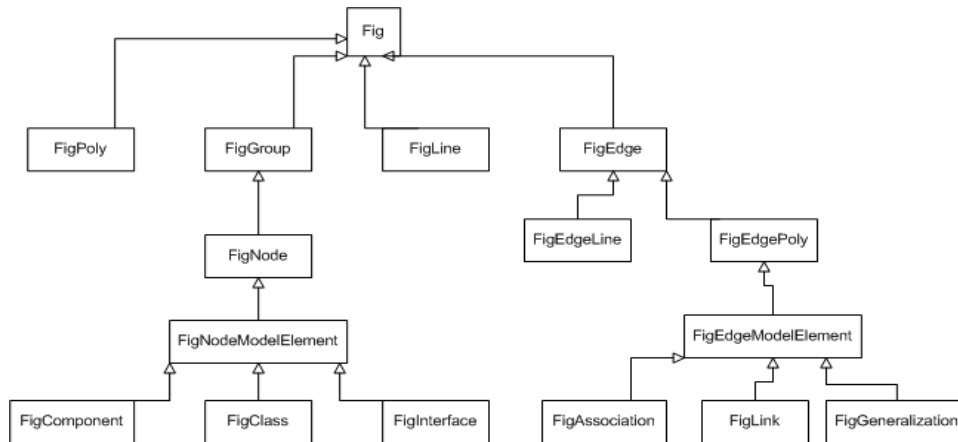


Figure 10 - Display figure hierarchy for ArgoUML.

The inheritance hierarchy shown in Figure 11 describes the relationships between classes used to represent UML components. These classes are all contained in the NovoSoft UML library used by ArgoUML.

3.3.2 Debugger Interface

The debugger component in the system is based on JPDA demo code provided by the Sun JDK version 1.3 and higher [JPD02]. This component establishes a connection to the debuggee JVM which is stored in an execution manager. The JPDA defines an API for accessing data from the JVM. The execution manager maintains methods to extract data from the JVM by using the JPDA API. The user should have a simple means of extracting data from the debuggee program but the execution manager is still too low-level

for this function. To improve data access, the system includes a `CommandInterpreter` class. The user or a user program can retrieve data from the JVM with simple commands, for example “classes” which returns a list of the classes currently loaded in the JVM. The `CommandInterpreter` operates as an adapter pattern. Several commands were included in the debugger as part of this research to allow users access to other information available from the JPDA that are not visually presented by ArgoUML.

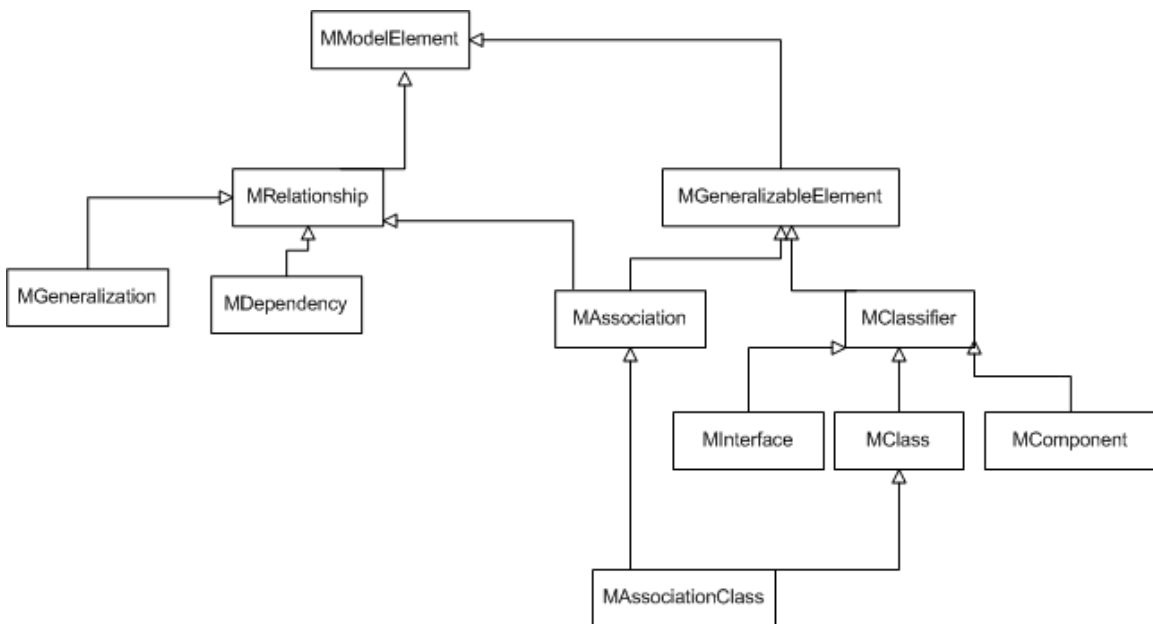


Figure 11 - NovoSoft UML hierarchy.

3.4 Experimental and Testing Techniques

To test the hypothesis that a visual debugger with the capabilities described is more effective than a standard debugger for object-oriented systems requires that a prototype system be developed to enable this comparison.

Most objective methods require months of test design, special facilities, and user trials on many subjects to provide quantitative results [USA02]. Examples of these

methods include the *performance measurement* technique and *retrospective testing*. Alternatively, users or expert analysts may compare the system with a previous system in a more subjective manner and then support the findings with a simple usability test. In this research, the debugger prototype is compared against debuggers currently available in an integrated development environment (IDE) such as Borland's JBuilder or TogetherControlCenter. A mixture of quantitative estimates (wherever possible) and empirical results is used to support the hypothesis of this research. The research evaluates the system against the PV criteria established in Chapter 2. To further support testing and to be able to more appropriately determine the effectiveness of the system, test subjects at AFIT will be involved in testing the system. The *coaching method* of usability testing is deemed the most appropriate for this purpose [USA02]. This technique has test participants ask any system-related questions of an expert coach (in this case, the tester), who will then answer to the best of his/her capability. At the same time, the tester can determine not only inherent problems in the system due to the questions asked, but also the level at which the debugging process is enhanced. This is achieved by coupling the coaching method with interview questions. Nielsen lists guidance for developing an effective interview [NEI93]. In essence there are two ways to interview for the purpose of usability testing – unstructured and structured interviewing. For this research, a set of specific, predetermined agenda with specific questions to guide and direct the interview exists and therefore the structured approach is more appropriate.

Since the design and implementation of the prototype is conducted in a modular manner, testing of the system takes place in the same way. Initial analysis takes place on the effectiveness of the visualization techniques. The modified UML displays are

compared to those in the original ArgoUML as well as other typical IDEs that provide UML representation of software. The overall goal of this part of the research is to maintain the symbology and semantics of UML while improving its space efficiency to accommodate speedy access to both high and low-level details. The research analyzes the effectiveness of the techniques with respect to this goal.

Quantitative results are provided based on the number of classes displayable in a set display area. Empirical discussion is also generated to support the hypothesis that these visualization techniques more effectively display object-oriented systems than would standard debuggers. Overall, the evaluation of the effectiveness of the debugging system is empirically based.

3.5 Metrics

As with most human-computer interactive activities, it is difficult to quantify the effectiveness of this system under test, in this case, the visual debugger. This section discusses the quantitative and empirical techniques used to gather as much objective evidence where possible and subjective evidence where it is not.

This research considers four types of metrics. The main one is the number of viewable classes displayed in a unit area. This metric is easy to measure and gives a very accurate and appropriate indication of the application's space economy. The empirical evidence gathered in support of the hypothesis as discussed in section 3.3 is another set of metrics and the final two are memory usage and CPU utilization. The empirical evidence includes discussion on the effectiveness of debugging with these techniques including the effects on access time, cognition and the size of portion of the model able to be displayed on a screen. This empirical analysis is supported by findings from a standard usability

test that involves human subjects. The CPU utilization metric is presented as a percentage of that available and memory usage is the amount of memory (in KB) consumed by the debugger system. Profiling tools are used to aid in these activities.

3.6 Summary

This chapter presents the methodology behind the system developed for this thesis research based on the objectives for the visual display and the design objectives for the prototype debugger that is required for testing. This chapter introduces the experimental techniques that are used to validate the system. Chapter 4 discusses the experimental design in more detail. This chapter also introduced the metrics that are used in the validation process of this research. In addition, this chapter explores the overall architecture of the prototype system including the chosen visualization platform – ArgoUML; and the debug platform – the JPDA.

4 Detailed Design and Implementation

4.1 Introduction

This chapter discusses the design and implementation details related to this research. Section 4.2 discusses the visualization elements the visualization techniques implemented into ArgoUML. Section 4.3 discusses the design details related to the debugger platform that interfaces with the visualization platform. Finally section 4.4 discusses the design of the experiment used to validate the application in terms of its effectiveness as a visual debugger.

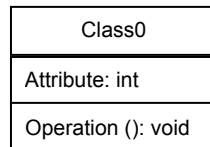
4.2 Visualization Elements

The visualization elements incorporated into ArgoUML to facilitate debugging form the basis of this thesis effort. This research implements a focus+context methodology to a UML diagram-editing tool – ArgoUML – which is described in Chapter 3. Alternatives to focus+context such as overview+detail and use of multiple windows through a drill-down capability are available. However as section 3.2.1 discusses, these alternatives had several disadvantages, primarily concerning increased cognitive load to the user. The prototype focus+context system presents an appropriate level of detail for each component using a degree of interest function that is based on distance from a focal point (either user specified or automatically selected) and frequency of access. It then uses smooth animation to reposition diagram elements to emphasize hierarchical relationships and maximize space economy while maintaining user context. The final space-efficient layout is achieved using a hierarchical graph layout algorithm. These visual modifications are described in the following sub-sections.

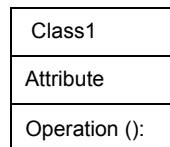
4.2.1 Visual Modifications – Focus + Context

Selective filtering and modification to graphical elements is employed to create multiple level-of-detail (LOD) representations for UML classes determined by distance between nodes (the number of links that separate them) and frequency of access. There are five levels of detail used in the application:

- LOD 4 contains the highest amount of detail. For a UML class representation, this includes a full size graphical representation that includes textual labels for all attributes and methods along with type information.



- LOD 3 hides attribute and return types while minimizing margin size.



- LOD 2 only displays the name of the class at a reduced font size.



- LOD 1 removes all textual information and only indicates the presence of a component.



- LOD 0 contains no textual information and indicates the presence of a component at a reduced size.



The technique used in ArgoUML selectively filters the textual labels at lower levels of detail according to the perceived relevance of the information. It displays a class at a particular level of detail using a degree of interest (DOI) function based on the frequency of access to a particular class and its distance from the current class in focus as given by the following equation:

$$DOI \propto \lg(freq) + (max_doi - dist) \quad (1)$$

In Equation 1, *freq* refers to the number of times the node in question has been accessed by the user since it was generated in the current debugger session; *dist* refers to the graphical distance from the node to the focal point (defaults to one if being considered is the focal point); and *max_doi* refers to the maximum DOI designed into the system (in this case 4).

The degree of interest is recalculated when the user accesses a node. The display then updates to reflect the appropriate LOD for each node.

As an example, consider the class diagram in Figure 12. If the user selects Class A as the node of focus, the representations of all remaining classes are modified as shown in Figure 13. Here Class A is the node of focus; hence it is displayed with no change at LOD 4. The degree of interest for Class B is one lower than that for Class A so Class B is displayed at LOD 3. As each subsequent class in this diagram is one additional link away from the node of focus, Class A, each of those classes is displayed at the next lower LOD. Beyond Class E, all classes in this path would be displayed at LOD 0. Similarly, the reverse effect would occur if Class E were to be selected as the node of focus initially as shown in Figure 14.

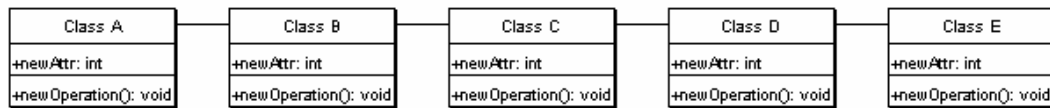


Figure 12 - Initial Class diagram.

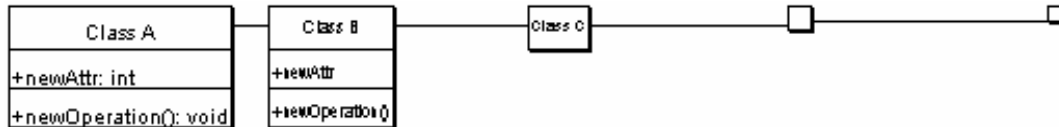


Figure 13 - Class diagram following application of Focus+Context techniques.

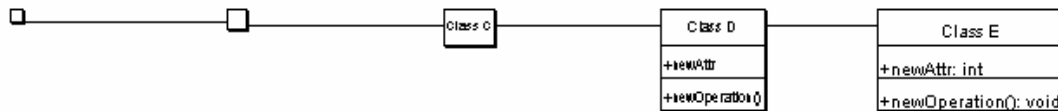


Figure 14 – Reversed version of Figure 13

This algorithm is further explained in the following text. In Figure 12, once all diagrams have been drawn and, each node has been accessed once. When Class A was made the node of focus, it is automatically set at the highest LOD, that is, LOD 4. Since each of the other nodes have only been accessed once, equation 1 gives the following LOD values for each class:

- Class B: $\lg(1) + (4-1) = 0 + 3 = 3$
- Class C: $\lg(1) + (4-2) = 0 + 2 = 2$
- Class D: $\lg(1) + (4-3) = 0 + 1 = 1$
- Class E: $\lg(1) + (4-4) = 0 + 0 = 0$

It is clear from above that any distance further than 4 from the node of focus will yield an negative LOD. However, notice that equation one is not an equivalence equation; rather it is a proportional equation. For this system, the lowest LOD that can be

assigned to a node is 0 and the highest LOD is 4, regardless of what the equation may give.

4.2.2 Visual Modifications - Graph Layout

To maximize the space efficiency of the focus+context techniques for UML, the graph layout algorithm used in the prototype application modifies the positions of elements in the model. Chapter 2 briefly presented the findings from the literature survey of common graph layout algorithms. A common method is to present generalization hierarchies down the vertical axis and as such is used in this application (hierarchical approach). The layout algorithm also considers the size of the nodes in order to maximize space efficiency and preserves context of the system (if any exists) by minimizing the relative reassignment of node positions. Chapter 2 also mentioned prior work on ArgoUML that employed a selective aggregation function as a display feature [JM03]. This was removed as part of this research as it introduced a great amount of confusion to the display. As much as possible, this research intends to follow a familiar schematic – UML – and the addition of a selective aggregation function breaches this schematic as it replaces nodes with lines, arrows or diamonds. Furthermore, the selective aggregation function that was employed on ArgoUML forcibly hid certain nodes that, although far away from a given focal point, may nevertheless be required by the user. The layout algorithm applied by this research satisfies all the display goals and preserves context. Results of the layout algorithm are discussed in Chapter 5.

4.2.2.1 Algorithm Details

The main algorithm requirement is to present the UML Class diagram hierarchically, that is, the algorithm should position generalization relationships vertically

on the display. Layered layout algorithms, as discussed in Chapter 2, meet this requirement. This approach was preferred to the TSM approach since it is simpler and provides an easier way to incorporate space efficiency due to less restrictions. The TSM approach would be more useful if graph aesthetics were the priority. The hierarchical approach is simpler to implement and flexible enough to incorporate some level of graph aesthetics. These algorithms allow the hierarchical presentation of diagrams with vertices arranged in horizontal layers. The standard method consists of the following steps as discussed in Chapter 2:

1. *Layer Assignment*
2. *Crossing reduction*
3. *Horizontal coordinate assignment*

Layer assignment requires a process to determine which layer each vertex belongs to. With the ArgoUML class diagram, a layer difference is considered to exist across each generalization relationship, with the top most vertex being the most generalized class. The algorithm places a node that is a direct descendant of an inheritance relationship at a lower level. All other nodes remain at their original level determined through association relationships. In the simple case where each vertex is the same size, the y-coordinate of each layer is determined by adding a suitable gap to the last layer.

Due to the great variation in vertex sizes, this research will not apply the crossing reduction process. With variable vertex sizes, the edge crossing minimization process would not apply as it relies on uniform row sizes to reduce crossings.

The last step deals with the positioning of each vertex on a layer. The priority for the algorithm for ArgoUML is to preserve context rather than minimize edge crossings, so

the algorithm allocates the position of each vertex on each row based on its position prior to ordering. The idea behind this is to allow the user to see objects in the same relative position as they were prior to the modification.

Application of the layout algorithm results in the repositioning of nodes and links in the class diagram. If done instantaneously or rapidly, this movement may cause the user to lose context. This problem is avoided by using smooth animation when moving graphical elements between their original and adjusted positions. To further preserve context, the technique moves elements in the class diagram in two stages. The first stage positions elements in the appropriate level of the inheritance hierarchy. The second stage positions leaf nodes to optimize space efficiency. The system employs the graph layout algorithm after all degree of interest functions are applied.

The layout algorithm also involves segments. A segment is a group of connected nodes. Each segment is treated separately in the layout algorithm and placed next to each other across a row. The following example illustrates the effect of the graph layout algorithm used in this research.

4.2.2.2 Graph Layout Example

Consider the graph shown in Figure 15. This graph was drawn using the original version of ArgoUML. The modified version resulting from this research automatically applies a hierarchical layout to any graph drawn and therefore such a graph could not be displayed in the modified version. It is provided below to illustrate the graph layout algorithm.

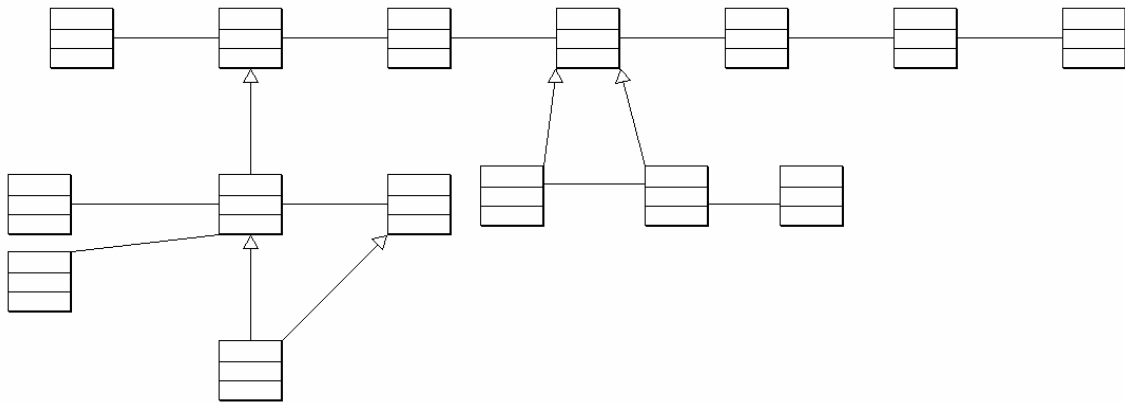


Figure 15 – Initial layout from original ArgoUML.

If the same connected set of nodes were to be drawn using the modified application, the resulting display would be as shown in Figure 16. Here, degree of interest display is enabled as well as the focus+context feature with the focal point being set at the upper left node.

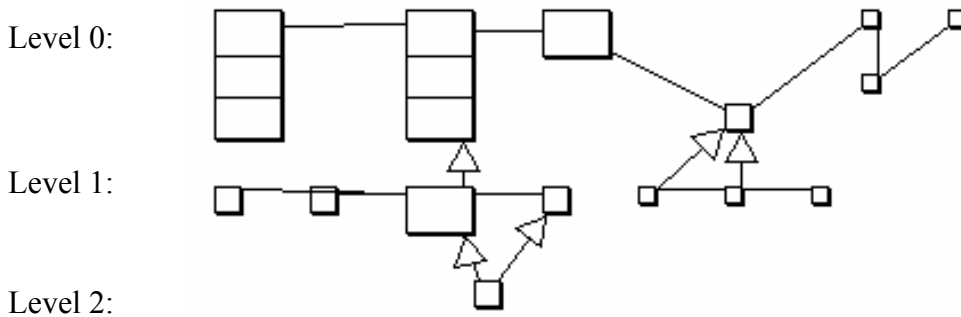


Figure 16 – Class diagram layout with focus+context applied.

Prior to the application of the hierarchical layout algorithm, the level of each visible node is determined. Superclasses are separated from their subclasses by 1 level and nodes connected via an association or aggregation relationship are placed in the same level.

From here, the first stage of the layout algorithm will examine each row, starting from the top and searches for inheritance links within that row. The horizontal (x)

position of any node with an inheritance link is equivalent to the average x position of its parents in the row above it. Figure 16 illustrates this point with the bottom inheritance leaf residing in a horizontal position in between the horizontal positions of its parent nodes. On the other hand, when one node has more than one child in an inheritance relationship, that node is simply shifted right according to the number of child node it has. Figure 16 also shows this type of relationship with the fourth node in level 0 being shifted to the right once since it has one extra child node. This was done for computational efficiency. Shifting parent nodes to the right was just as space efficient for reasonably large systems than finding a central x position above all the children nodes.

When room exists to fit more than one node in a level, then those nodes are stacked on top of each other within the same level so long as no inheritance relationship exists between those nodes. Figure 16 shows where each level commences. As evident in this illustration, although only three hierarchical levels exists, we can distinguish 4 distinct rows. This is because the nodes far from the focal point that are not connected by an inheritance relationship are stacked on top of each other within the same level, according to the height of the largest node in that level. Here only two LOD 0 nodes can fit within level 0, however more would fit if the upper left node was greater in size. Figure 17 shows the same configuration with the upper left node slightly enlarged. Here we see that the outer leaf node from Figure 16 is now placed below its association node.

Each segment (or group of connected nodes) added to the display is treated separately and the process explained above is repeated for each one of them. They are each placed in order, with the first segment on the upper left side of the display then continues across the display. Hence for this research, when using ArgoUML to debug

more than one program, each program is placed on the display starting from the upper left side of the display then is laid out next to each other, separated by a gap. This is illustrated later in Figures 25 and 26, where the display contains four segments (see chapter 5).

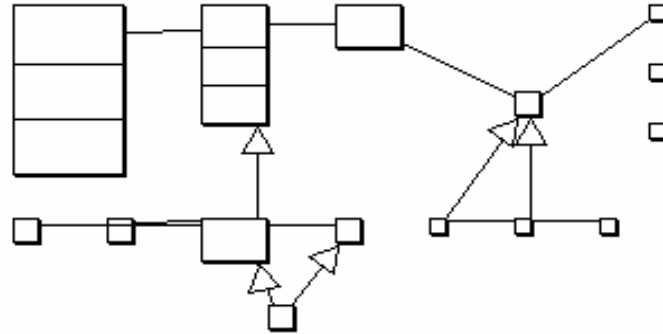


Figure 17 – Class diagram illustrating hierarchical layout schema

4.2.3 Implementation

This section further explains the design choices involved in integrating the SV features into the ArgoUML. To determine the most effective design for integrating these features, the processes that must occur are considered.

From a visualization standpoint, the modifications applied to ArgoUML are largely to do with presentation of the graph nodes and links. As such, large parts of the changes occur in the objects that represent the graphical elements for classes and the various UML link types. The classes representing the display of these objects include `FigClass`, `FigEdgeModelElement`, `FigAssociation`, `FigGeneralization`. No changes are necessary to the NovoSoft UML library that stores the model information.

The `ClassDiagramGraphModel` class maintains a list of the figure elements to display for an object diagram, making it a suitable location for the modification methods.

The `ClassDiagramGraphModel` class controls most of the fish-eye view modifications. It calculates the hierarchical level of each `FigClass` (the class that represents the image of a class), the DOI and LOD for each, and finally calls a method to apply the visual modifications. The `ClassDiagramGraphModel` class also determines how each link should be drawn, that is either fully displayed, only the arrow, a partial line, or not at all. If a `FigClass` has hierarchical or aggregate relationships, the class also determines whether the branches for these relationships should collapse.

Mouse and debugger events in `FigClass` trigger the alteration process. These two events largely control the visualization process. Therefore, this module of the overall system is event driven. The “change” method within this class paints the object based on its LOD value.

4.3 Debugger Design

The debugger system consists of several key features to extract and transform the data, and provide the standard debugging features required by users. The following sections discuss these topics with additional information provided on implementation.

4.3.1 Data Extraction

Chapter 2 discusses the need to minimize the effects caused from observing a system. Failure to do this may cause the monitor to obtain inaccurate results. The JPDA allows system data extraction directly from the observed JVM requiring no modifications to the original code. Thus, some reduction in processing speed is expected to occur, however synchronization points and all data values remain unchanged from an unmonitored system. This research modifies a GUI demo application available in the JPDA to interface with ArgoUML. The system gives the user two options to connect to a

JVM: via specified arguments or attach to an existing JVM and communicate via socket as shown in Figure 18.

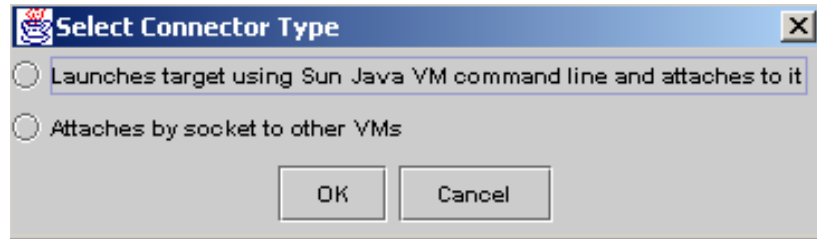


Figure 18 - Connector options.

4.3.2 Reverse Engineering and the Debugger Subsystem

Once connected via the JPDA, raw data is available from each monitored JVM. The basic premise behind this research, however, states that the user can debug more effectively with the data presented in a visual format in the form of a UML class diagram. To produce this diagram, the system applies a reverse-engineering process. The following discusses some of the issues involved in this process.

Once ArgoUML connects to the debuggee process, the JVM sends a signal notifying the debugger that it is running. A separate observer thread is created within ArgoUML. The observer queries the JVM and suspends and resumes the execution of the debuggee process at set intervals. This solution also improves system performance. If the debuggee process executes without interruption, then the JVMs will provide too much data to the debugger and the display will not be able to update in time.

When the observer thread detects new classes in the JVM, the reverse-engineering process examines them for operations and fields, where a field contains the name and type of an attribute. The observer constructs a NovoSoft UML model from each of the added classes. The observer also adds association, aggregation and inheritance

relationships to the model as it detects these relationships. Figure 19 describes the algorithm for the reverse engineering process.

```
for each class detected
  if class not yet in model
    Get fields
    Get Methods
    for each field
      Add attribute information to class
      If attribute type is contained in the model
        Build an association from the class to the attribute type
    for each method
      Add method information to the class
      If the method parameter type is contained in the model
        Build an association from the class to the parameter type
for each class in the model
  If there is an inheritance relationship to an existing type in the model
    Add the inheritance link
```

Figure 19 – Pseudo code for Reverse Engineering algorithm

Figure 20 shows a typical screen shot of a program connected to ArgoUML for debugging. As mentioned in chapter 3, the debugger sub-system is largely based upon example code provided by Sun with version 1.3 (and later) of the JDK. Operation of this code is mainly event driven. The requirements of this component are that it communicates with the GUI to request or display information provided by the debuggee JVM. This usage scenario suggests that an event driven interface might also be effective between the GUI and the debuggee process.

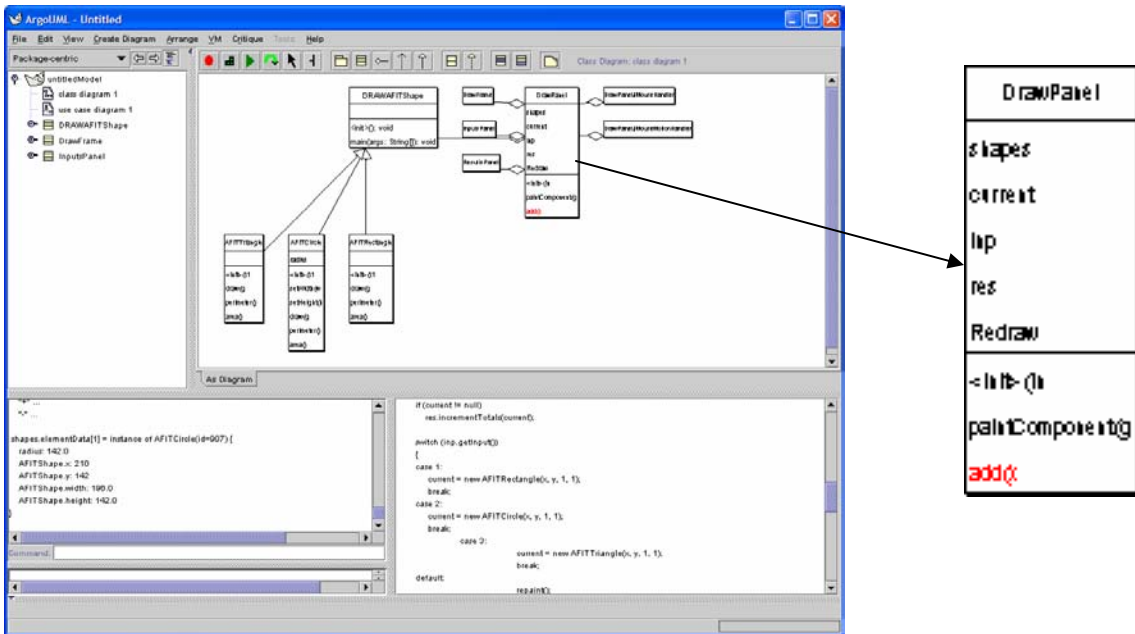


Figure 20 – Debugging screen layout

```

shapes.elementData[1] = instance of AFITCircle(id=907) {
    radius: 142.0
    AFITShape.x: 210
    AFITShape.y: 142
    AFITShape.width: 196.0
    AFITShape.height: 142.0
}
  
```

Figure 21 – Accessing object information in ArgoUML

The observer thread handles communication between the debuggee process and the GUI. This thread periodically accesses the JVM to determine if there are any changes to the observed system. If a change is detected, an event triggers the reverse engineering process and updates the model on the screen.

A variety of commands may be issued via a command prompt facility. On another window pane, results from these commands are displayed. It is via this command prompt that the user can access debugging information that may not be accessible from the

ArgoUML GUI. For example, this research adds a ‘dump’ command that dumps object information, not available on the class diagram displayed. Here the user can access attribute values of objects of interest. An example of this is shown in Figure 20 with the enlarged command panel shown in Figure 21. Here the user can access required values for variables or instance counts for classes. A great amount of consideration was given to the possibility of representing this object information graphically rather than in a pretty printed format. Another Java debugger, JSwat by Blue Marsh Softworks, uses jtrees to represent this information [JSW04]. Figure 22 shows such a representation from JSwat. It was eventually decided that a pretty printed format, although not ideal, would be preferable to the possibility of having to slow down the reverse engineering process further with the addition of another data structure.

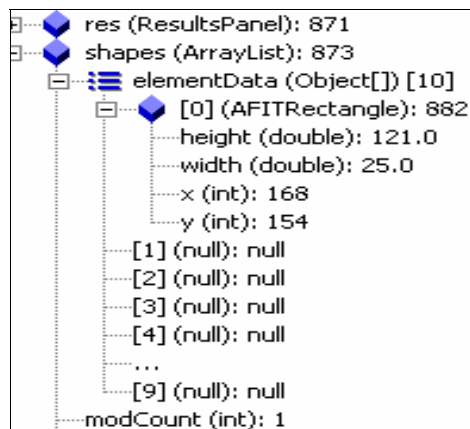


Figure 22 – Jtree representation of object information in JSwat

```
public void add(int x, int y)
{
  if (current != null)
    res.incrementTotals(current);

  switch (inp.getInput())
  {
  case 1:
    current = new AFITRectangle(x, y, 1, 1);
    break;
  case 2:
    current = new AFITCircle(x, y, 1, 1);
    break;
  case 3:
    current = new AFITTriangle(x, y, 1, 1);
    break;
  default:
    repaint();
    res.totalArea = res.totalArea - (int)current.area();
    res.totalPerimeter = res.totalPerimeter - (int)current.perimeter();
    break;
  }
}
```

Figure 23 – Breakpoint setting in ArgoUML

To be effective as a visual debugging system, access to the source code should be provided. The ability to enable and disable breakpoints via direct manipulation of the source code is also highly desirable. The system allows breakpoints to be set and removed by double-clicking the mouse on the appropriate line. The observer queries the execution manager to determine if the breakpoint is valid. This research highlights the chosen breakpoint line in red as shown by the example in Figure 23. The source code for a class with a “main” method is automatically requested from the user when the process begins. The user can then navigate to the appropriate file before debugging commences. In addition, the current execution point is also highlighted red on the displayed class diagram. As shown by the magnified node on the side of the diagram in Figure 20, the user is presented with a visualization of the control flow by the highlighting of the current method in execution. In this particular instance, we can see that the `add()` method in the `DrawPanel` class is highlighted red. This capability is highly desirable in debuggers as it provides for easier cognition than manual tracking of the program’s execution. It

allows the user to follow execution at a higher level of abstraction rather than follow the execution through each line of code. The ability to follow program execution provides the user with an understanding of the order of executed components and some insight into system behavior.

This visual debugging system allows the user to control the execution of the debuggee process as do most other debuggers. That is, controls exist to provide for unrestricted execution or resumption from a suspended state (the green play button), stepping through the code line-by-line (step), a step up command (the green next arrow), suspension of execution (a red circle). The step up command is incorporated by this research to allow for faster stepping through code. It essentially completes execution of the current method and resumes debugging one process higher in the stack. If connections to multiple JVMs exist, the user must select a component from the segment of interest prior to selecting the control. The listener then determines which component to select and carries out the appropriate action in the selected JVM. The control buttons are shown in Figure 24.

To allow a flexible and easily maintainable interface between the debugger component and ArgoUML, patterns are used. An adapter pattern in the `CommandInterpreter` class allows the observer thread to issue simple commands without an underlying knowledge of the operation of the system. The `ExecutionManager` class maintains knowledge of the JVM parameters. This relationship is depicted in Figure 25.

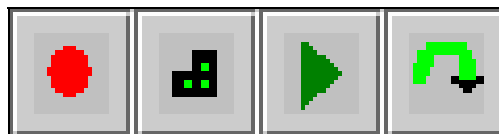


Figure 24 – The ArgoUML Debug Control buttons

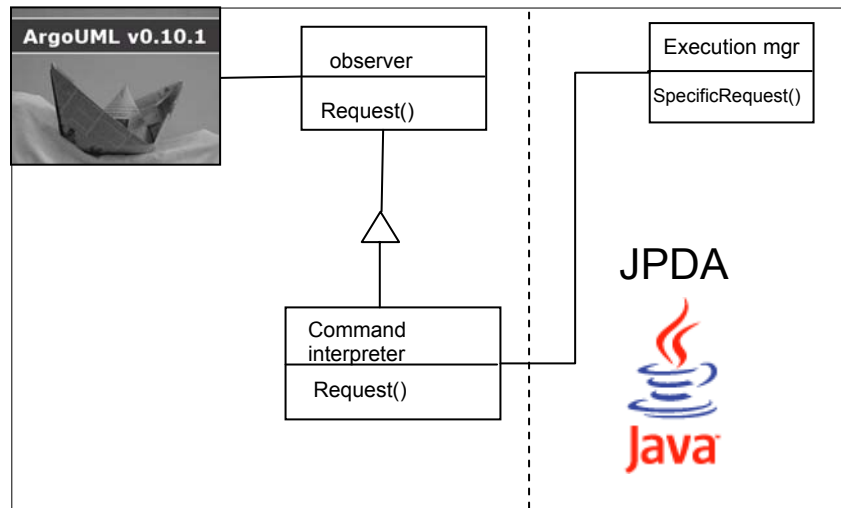


Figure 25 – Adapter Pattern used in the visual debugger system

4.4 Experimental Design

This section outlines the resources and methods to be used for the experiment to determine the effectiveness of the developed visual debugger prototype.

4.4.1 Experimental Objectives

In this research, the debugger prototype is compared against debuggers currently available in IDEs such as Borland’s JBuilder or TogetherControlCenter. The objective of the experiment is to examine the effectiveness of the prototype application in terms of the following:

1. Software Visualization criteria
2. Debugging criteria
3. Usability criteria

Each of the above criteria is measured via a questionnaire and are described in the following sections.

4.4.2 Criteria

The following briefly describes the criteria used for the experiment. Each section also includes the questions included in the corresponding portion of the questionnaire.

4.4.2.1 Software Visualization

Several SV techniques are used in the prototype application. Table 1 shows questions designed to determine the level that the application satisfied some generic SV criteria.

SOFTWARE VISUALIZATION CRITERIA		1	2	3	4	5	6	7	8	9	10	
1. SPACE ECONOMY (focus+context, layout algorithm)	Inefficient											Very efficient
2. METAPHORS (graphical symbology used)	Difficult to understand											Easy to understand
3. INTERCONNECTION (relationships between components)	Confusing											Very clear
4. INTERFACE	Hard to manipulate data											Easy to manipulate data
5. SCOPE (what can you see? code? diagram?)	narrow											wide
6. LEVELS OF ABSTRACTION	Very limited											Multiple levels
7. PRESENTATION	Difficult to interpret											Easy to interpret

Table 1 – Survey: Software Visualization Criteria

The above questions do not directly address the level at which UML semantics were adhered to. UML is a standard and is either met or breached. Since the application directly makes use of the Novosoft UML library, it is accepted that the semantics of UML are met. However, the questions do address the ease of understanding the graphical metaphors and the interaction between components within the visualization presented. The criteria selected are based on the Roman and Cox Program Visualization taxonomy discussed in Chapter 2.

Space Economy refers to the application's ability to efficiently display Class information. Participants will respond in terms of its perceived efficiency. This criterion addresses the effectiveness of the layout algorithm and focus+context features. *Metaphors* refer to the graphical symbology used in the application. This addresses the suitability of using UML semantics in a debugging application. *Interconnection* refers to the relationship between graphical components used to display Class information. Since standard UML is used in ArgoUML, this question addresses whether or not the interconnection between nodes are intuitive and clear to the user. The *interface* consists of graphical objects presented to the viewer and interaction with the presentation using buttons, menus, and other controls or through direct manipulation of the graphical objects. This addresses whether or not the interface controls used in ArgoUML allow for easy manipulation of data. The *level of abstraction* criterion refers to the application's ability to display multiple levels of information, from source code right up to high-level structural representations. Related to this is the *scope*, which refers to the amount of different information made visually available by the application to aid the user in debugging. *Presentation* refers to the semantics of the graphical objects that are presented to the viewer. The presentation is that aspect of the visualization that facilitates interpretation and understanding of the graphics. This covers issues concerning human cognition and effective visual communication such as the use of color, size, spatial relationships and other visual concepts to depict additional meanings.

4.4.2.2 Debugging

A debugger is a programming tool used to execute a program, test (and view) its states, update its environments, and set breakpoints. The questions in Table 2 do not

address the ease of finding a particular bug with using the prototype application. Instead, they present general desired functionality for debuggers and the level at which the prototype application satisfied those. Section 4.5.3 further discusses other activities that determine the debugging effectiveness of the tool.

DEBUGGING CRITERIA		1	2	3	4	5	6	7	8	9	10	
1. Suspend, Step, Resume program execution	Difficult to perform/not satisfied											Easy to perform/adequately satisfied
2. View threads, method calls	Difficult to perform/ not satisfied											Easy to perform/adequately satisfied
3. View object information	Difficult to perform/ not satisfied											Easy to perform/adequately satisfied
4. View variable information	Difficult to perform/ not satisfied											Easy to perform/adequately satisfied
5. Breakpoint specification	Difficult to perform/ not satisfied											Easy to perform/adequately satisfied

Table 2 – Survey: Debugging Criteria

4.4.2.3 Usability

Usability considerations are vital in determining the ‘effectiveness’ of a debugger, or any other software application for that matter. Since one of the focuses of this research is to determine the effectiveness of providing a dynamic UML class diagram visualization of an executing object-oriented program for the purposes of debugging, it is important to consider the major usability factors that can greatly influence this activity. Even though the application may be capable of certain functionality, it will not be ‘effective’ if sound usability concepts are not incorporated into the design of the program. The following sets of questions (Tables 3, 4 and 5) are separated into three categories – screen, learning, and system capabilities. This questionnaire is a slightly edited form of a *Questionnaire for User Interface Satisfaction* designed by Chin, Diehl and Norman [CDN88].

SCREEN		1	2	3	4	5	6	7	8	9	10	
1. Reading characters on diagram and debugger panel	hard											easy
2. Organization of information	confusing											very clear
3. Sequence of screens	confusing											very clear

Table 3 – Survey: Usability Criteria-Screen

LEARNING		1	2	3	4	5	6	7	8	9	10	
1. Learning to operate the system	difficult											easy
2. Exploring features by trial and error	difficult											easy
3. Remembering names and use of commands	difficult											easy
4. Performing tasks is straight forward	never											always
5. Help messages on the screen	unhelpful											helpful
6. Supplemental reference material	confusing											clear

Table 4 – Survey: Usability Criteria-Learning

SYSTEM CAPABILITIES		1	2	3	4	5	6	7	8	9	10	
1. System speed	too slow											fast enough
2. System tends to be	noisy/ unstable											quiet/stable
3. Correcting your mistakes	difficult											easy
4. Designed for all levels of users	never											always

Table 5 – Survey: Usability Criteria-System Capabilities

4.4.3 Approach

Apart from using the results of the questionnaire, this research gathers quantitative results based on the number of classes displayable in a set display area. Empirical discussion is also generated to support the hypothesis that these visualization techniques more effectively display object-oriented systems than standard debuggers. In this case, evaluation of the effectiveness of the overall debugging system is empirically based. A java profiler will also be used to present performance bottlenecks within the debugger application.

A test program with a simple bug is used in the experiment. Participants are coached in using ArgoUML prior to the experiment and are informed of the correct behavior of the test program. During the experiment, participants were asked to perform a number of simple tasks with the test program as input. For example, listing down certain variable values, setting breakpoints, etc. Success or failure to detect the bug within the allocated time (25 min) are recorded on the bottom of the questionnaire. Five participants were chosen to take part in the survey in accordance with the heuristic for usability testing of computer systems [BAR03]. Selection of participants were based on individual experience with standard debuggers as well as general software engineering knowledge. Since only a small sample size was used, this research does not apply any predictive or descriptive statistical analysis to the results. The average scores for each criteria were evaluated and a deduction was made as to what those scores meant, based on user comments and general responses.

As well as marking responses to the questions presented in Section 3, participants were asked to comment on and list down the three most positive and negative aspects of the application. These will be used to incorporate enhancements, or remove certain functionality in the prototype application. A general overall comment on how the prototype application compares with standard IDE debuggers will also be requested.

4.5 Summary

The hypothesis for this research states that visualizations can be of assistance to users in distributed and large system debugging. Based on this hypothesis, the visualizations are of great importance. This chapter defined the algorithms and processes developed to produce these visualizations. It describes the modifications to ArgoUML to

include these visual transformations and additional debugging capabilities. The research interfaces ArgoUML to the debuggee JVM via JPDA and other debug code. The class diagram of the model for the display is reverse-engineered from data accessed in the debuggee JVM. It then presents a guideline for the experiment to test the research hypothesis. The experiment relies on three sets of criteria – Software Visualization, Debugging and Usability.

5 Results and Analysis

5.1 Introduction

This chapter presents the results of the conducted experiments discussed in Chapters 3 and 4 and provides an analysis of the effectiveness of the modified ArgoUML visual debugger based on these results.

5.1.1 Presentation of Results

Testing debuggers relies heavily on the analysis of other programs and verifying that the debugger returns expected values. This however only forms part of the testing portion of this research since much of it is focused on the visualization aspects of the application. The approach taken by this research in meeting the requirements stated in Chapter 3 is divided into 4 phases. Firstly, it verifies that previous distributed system capability is met at an equal or higher standard. In essence, this phase is a regression test phase – an activity that is typically required in any software development process that involves updating an existing platform. The second phase of the test analyzes the graphical display capability of the system. This primarily focuses on the analysis of the layout algorithm discussed in Chapter 4. The third phase of the test is debugging applications and analyzing the visual display capability and how it aids in the debugging process. This includes debugging both small and large object-oriented systems and provides quantitative results in terms of its space efficiency and other visualization and debugging criteria as stated in Chapter 3. The fourth portion of the test is the Usability survey outlined in Chapter 4. This involves subjects coached at using the prototype application and providing responses to a Usability survey. Since Visualization can be a subjective technique, and may work differently for different people, the survey is also an

important part of the research as it establishes not only how Visualization criteria or debugging criteria are met, but also its perceived effectiveness.

5.2 Testing Preparation and Procedure

All four phases of the experiment use the same hardware and operating system configuration. The PC in use for testing contains a Pentium 4, 1.7GHz processor and 512 MB RAM running under the Windows 2000 environment. The tests use Java Runtime Environment v1.3.7 on each PC. The initial part of the test (phase 1) uses additional PCs with the same configuration as it tests the distributed system capability of the application. The following sub sections highlight details of each test.

5.3 Phase I – Regression Test

The only test performed in this phase was that of the distributed system connectivity capability of ArgoUML. This was the only function that was not changed as part of this research. However from a debugging standpoint, it remains a critical portion of the application as it allows multiple programs to be debugged. Table 6 summarizes the procedure of this regression test.

Figure 26 shows a screen capture of ArgoUML with two simple OOP agents ported and ready for debugging activity.

ArgoUML was able to test each agent using the debugging buttons or through the command line facility. Each of the agents was able to be manipulated separately, and was distinguished from each other with the use of a segment number. Figure 27 shows the same two agents as in Figure 26 but with a node in the second agent being set as the focal point of the display.

Regression Test Objective	Ensure that a distributed system can be connected to ArgoUML.
Technique:	<ol style="list-style-type: none"> 1. Run ArgoUML 2. Run batch files created for two agents*. 3. For each agent, select 'Generate from JVM', then set port and machine name information. 4. Ensure each system is displayed on the screen. 5. Ensure display layout is correct (e.g. check horizontal separation of agents) 6. Test the following debug controls with a node from each agent selected: <ol style="list-style-type: none"> a) Step up b) Step c) Pause d) Resume
Special Considerations:	<p>Evaluation of the effectiveness of displays will be empirically based.</p> <p>*selected agent is a simple 200 line Object-oriented Program called AFITShape.</p>

Table 6 – Regression Test Procedure

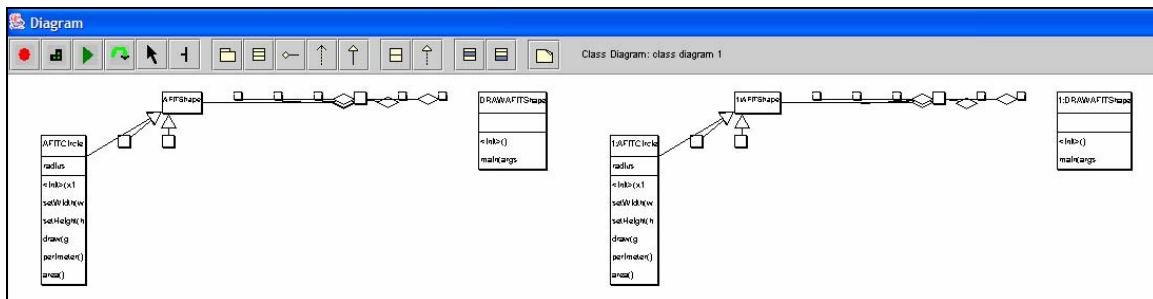


Figure 26 – Diagram Panel after two OO agents are connected to ArgoUML

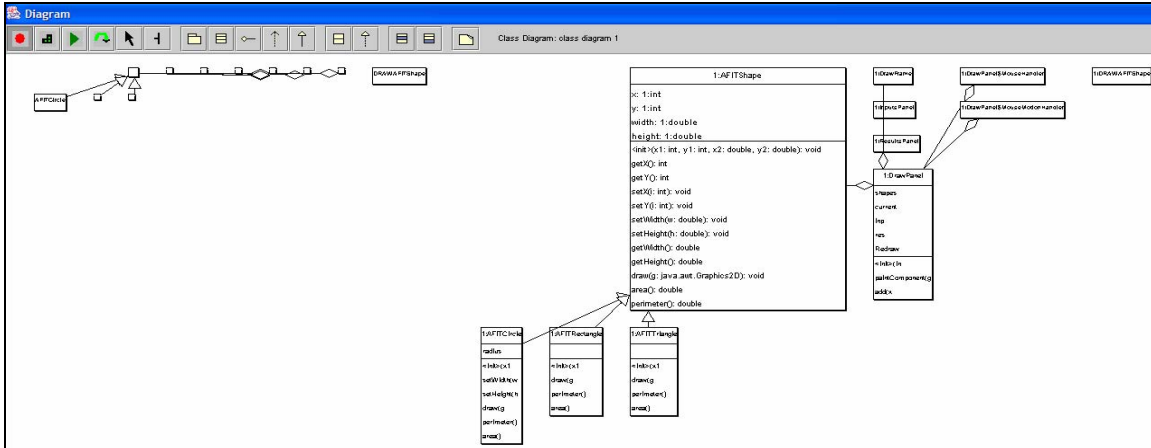


Figure 27 – ArgoUML with two agents loaded and focus point on the second agent

5.4 Phase II – General System Display Analysis

Chapter 4 explained how the graph layout algorithm in ArgoUML operates. This section now analyzes the graph layout algorithm. It presents a simple example that illustrates the effects of the algorithm and primarily focuses on the space efficiency aspects that the algorithm provides.

With the screen resolution set at 1400×1050 pixels, a display area is created and a set of nodes similar to those in Figure 12 are connected in a 3 × 3 configuration as shown in Figure 28. After applying the graph layout algorithm coupled with the degree of interest display capability (with the focus point set at the top left node), the number of nodes displayable in the same set area is 32 as shown in Figure 29. This is an improvement of 355%. It is obvious from Figure 28 that this improvement would be even greater if more nodes were initially considered (e.g. if a 5x5 configuration was used rather than a 3x3).

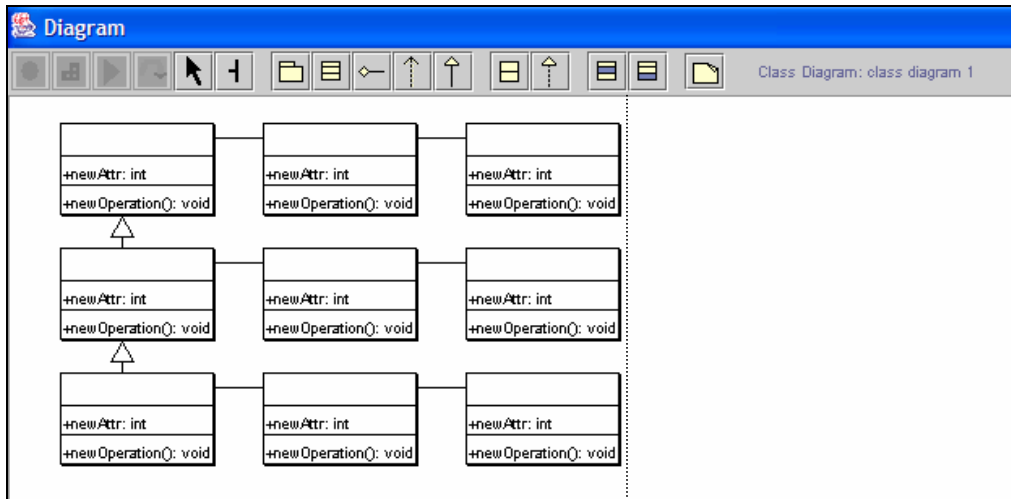


Figure 28 – 9 nodes linked up with no visualization techniques applied

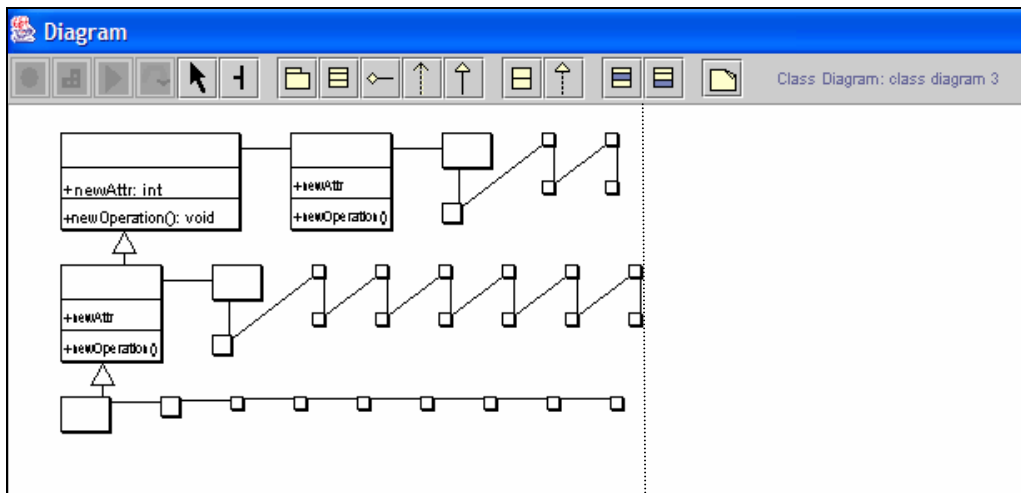


Figure 29 – Nodes with degree of interest display and hierarchical layout applied

The above example illustrates the main advantage of the focus+context technique. With the use of this technique, only one component can be in focus at any given time. If the original UML representation of the system is to be used, then details are available for as many full classes that can fit in a display, however the user can still only focus on one area of the screen at a time. With focus+context enabled, the user needs to select focal points of interest to access detailed information on that node. If the system being analyzed is small, then it is likely that switching focus would be faster with the original

UML representation. However for systems where the UML diagram would span several pages, the focus+context features are likely to improve access time since a much greater amount of information is accessible in the same set display area.

5.5 Phase III – Visual Debugging Analysis

This phase involves general display testing with small, medium and large systems. It covers the visualization capabilities of ArgoUML and how its interface to the JPDA effectively aids in the debugging process. The following sub-sections covers each of the three system tests.

5.5.1 Small System Testing

The program analyzed in this phase of the experiment is AFITShape, a simple graphical program that allows users to draw three different shapes on a display panel and outputs the area and perimeter of each along with the total area and perimeter of all shapes drawn. AFITShape comprises 9 main classes and approximately 400 lines of code. When being debugged, this program would be represented in a typical IDE debugger such as TogetherControlCenter as shown in Figure 30.

A significant amount of direct manipulation is required in IDE's to view different levels of abstraction, even for a small system like AFITShape. In Figure 30, the top pane shows a small part of the class diagram. The user would need to rearrange windows as well as directly manipulate the class diagram in order to display all required information. On the other hand, loading AFITShape into ArgoUML yields the display shown in Figure 31.

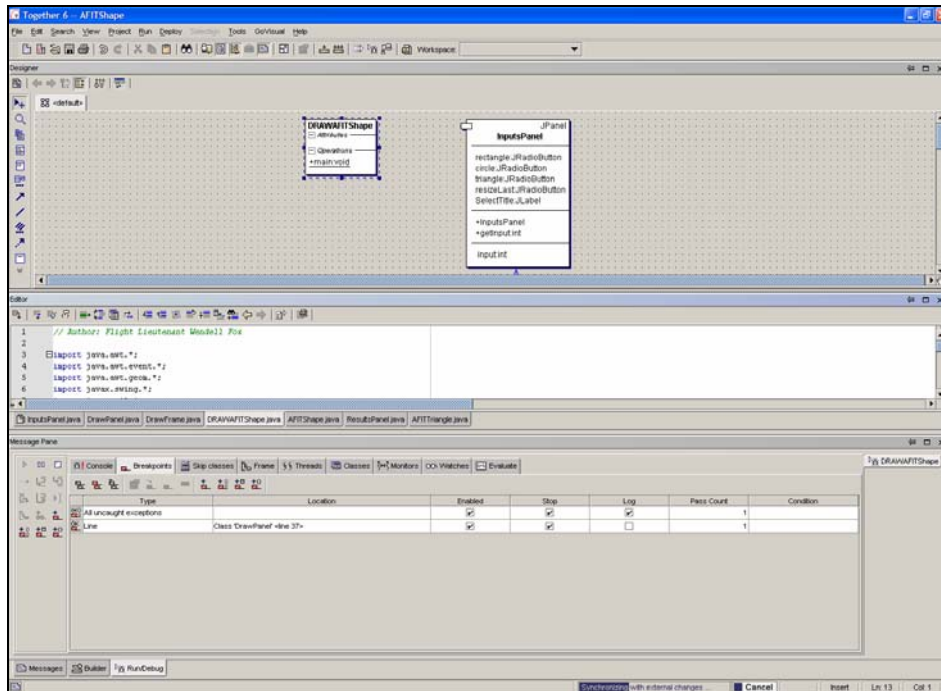


Figure 30 – Typical display of a small system in an IDE

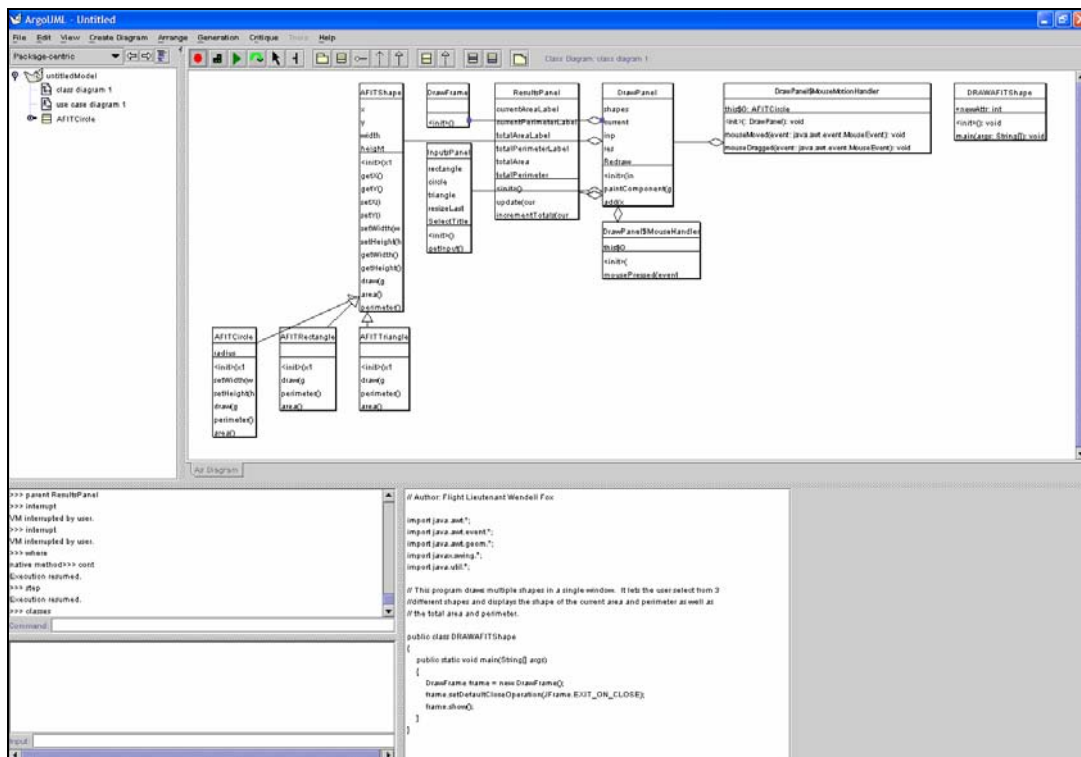


Figure 31 – Initial display on ArgoUML after loading AFITShape

Here, several items stand out. Firstly, the class diagram shown on the display panel is automatically laid out hierarchically as a result of the graph layout algorithm that is applied. In this instance, no filtering or focus+context techniques are applied to the display. However the application still manages to fit all classes into the editor pane. Notice in this case that there are 11 classes displayed. This is due to two internal classes within the `DrawPanel` class of `AFITShape`. Figure 31 also shows the general display configuration of ArgoUML. In the top left hand pane, is the navigation tree that allows the user to navigate to a selected class. Clicking onto a node in the navigation tree enables that node to be the node of focus in the display panel. The top right hand pane is the display pane that contains the class diagram. The user is also given the option to open a diagram window that is dedicated only to this pane. The two panels on the lower left hand side are the JPDA input and output panels. These allow the user to enter in specific debugging commands and inspect different text outputs as a result of those commands. The panel on the lower right is the source code panel. As discussed in Chapter 3, the visualization objectives of this research includes that of presenting the information through multiple levels of abstraction. This is clearly achieved by the modified ArgoUML as discussed and illustrated in Figure 31. In this example, no focus+context was necessary as the system analyzed was small enough to be represented fully without any filtering techniques.

5.5.2 Medium System Testing

The system chosen to be analyzed for this phase of the testing was JUnit. JUnit is an open source unit testing suite for Java programs [JUN02]. It is ideal for this phase of the analysis as it comprises approximately 3000 lines of code and 52 classes. The diagram

generated by TogetherControlCenter for this system is shown in a package diagram view in Figure 32.

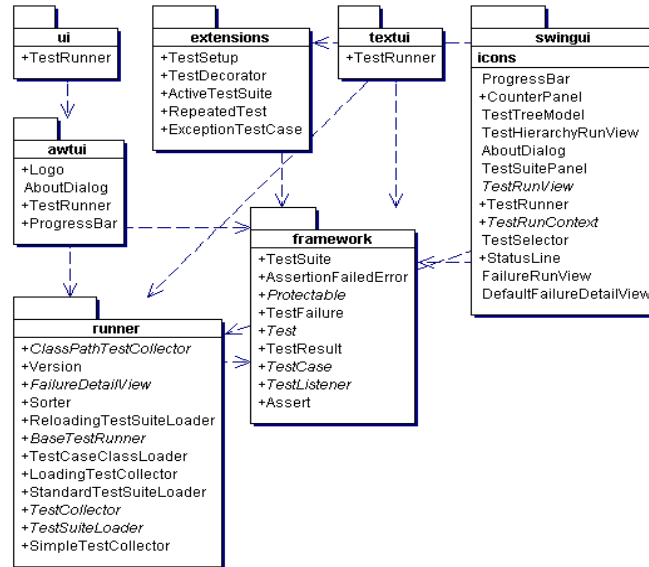


Figure 32 – Package Diagram representation of JUnit in TogetherControlCenter

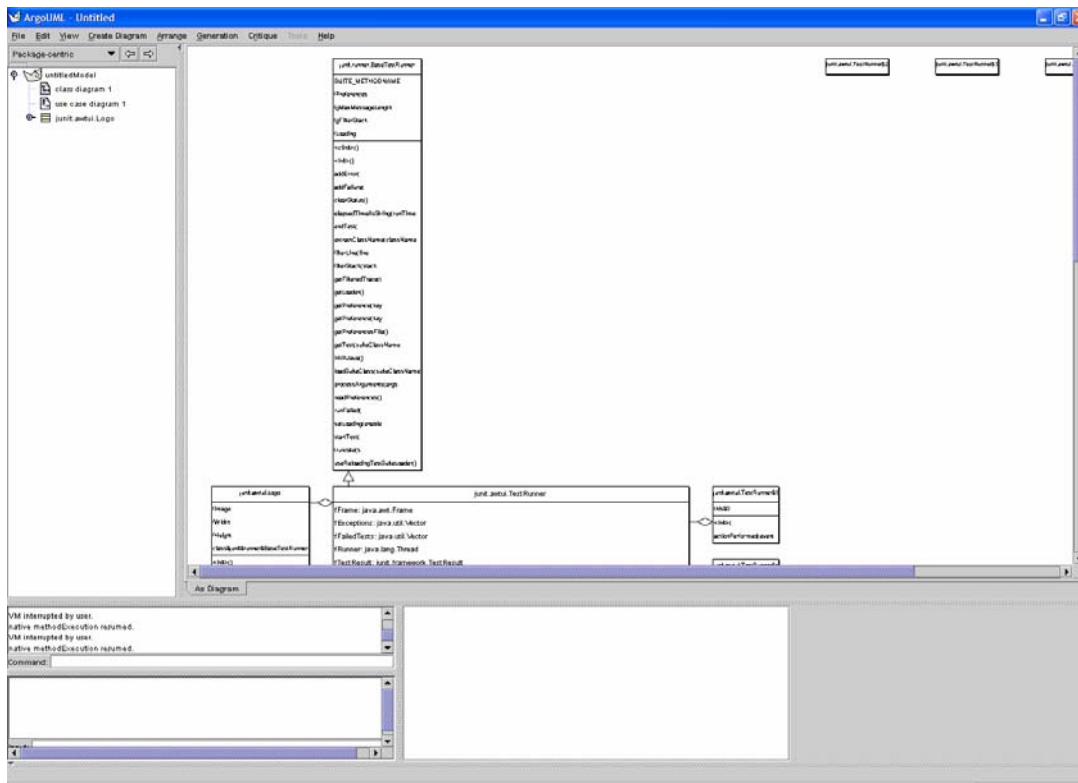


Figure 33 – Initial view of JUnit

Here, TogetherControlCenter uses a drill down capability technique rather than showing the whole class diagram. The user can click on a package of interest to see the classes within that package, and each package may then be viewed through multiple windows. As mentioned in Chapter 2, although this method is space efficient, it still suffers from the possibility that the ‘big picture’ may be lost.

Loading JUnit into ArgoUML yields the display shown in Figure 33. This particular screen capture only shows 5 classes (where the name is visible) and part of one more. This is because there are several Java classes within JUnit that has been filtered out by ArgoUML since they only contain native methods. Also this figure only shows JUnit at its initial status. Here we see that only part of the class diagram is displayed. Figure 34 shows the diagram after additional filtering techniques are applied. This is a combination of both focus+context and information hiding. Another feature added to ArgoUML is the ability to hide node compartments. Some classes may end up being too large to be accommodated into one page. Therefore the user is given the option to close either or both of the attribute and operation compartments to result in a smaller graph that still preserves context and does not diminish program understanding. In Figure 34, the operation compartment of the largest node is hidden so that the user can see the full high-level structural view of JUnit without having to span multiple pages. We now see that 16 classes are displayed, an improvement of 320%.

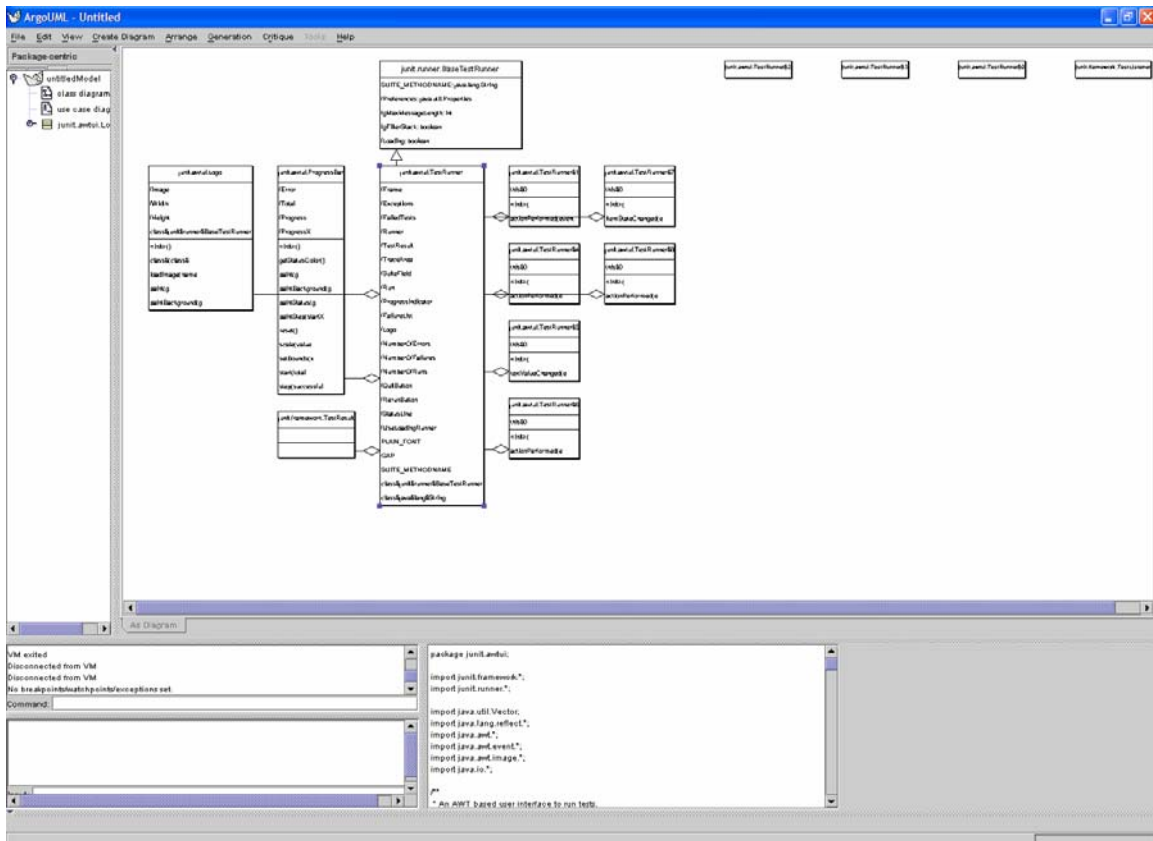


Figure 34 – The JUnit display after application of filtering techniques

5.5.3 Large System Testing

One of the major aims of this research is to be able to utilize the developed application in an environment where it is analyzing either a singular large object-oriented systems or multiple systems connected through a distributed network. Prior large system testing of ArgoUML was limited to analyzing BubbleWorld – an AFIT developed information retrieval system that comprised approximately 30,000 lines of code and 84 classes. With the original layout algorithm, the system became too recursive to handle anything larger than BubbleWorld. Since much larger applications are likely to be used, this phase of the experiment loads ArgoUML for debugging into ArgoUML. The modified version of ArgoUML comprises more than 1,000 classes and a little over 200,000 lines of code.

At the commencement of this testing activity, similar to both the medium and small system testing, reverse engineering is applied to the model to immediately provide the user with a class diagram representation to eliminate the need to form a mental model by either reading through documentation or browsing through the source code. Assuming that a monitor set at a resolution of 1400×1050 would fit approximately 50 lines of code in one page, then browsing through all of ArgoUML's source code would require more than 4,000 pages.

As was the case for JUnit, ArgoUML contains many classes with a large number of attributes and methods. This results in very large nodes as shown in Figure 35, where only 3 classes in the diagram panel are displayed. Further on in the execution cycle however, when the graph layout has been applied and different nodes are in focus, we are able to fit 17 classes as shown in Figure 36, an improvement of 566%. Further filtering techniques, by closing compartments yields the graph in Figure 37. Although only the same 17 classes are visible in this filtered display, only approximately half the display area is used compared to the display in Figure 37 so the effective improvement actually totals to approximately 1100%. Notice that in Figure 37, the process was already at a stage where the class diagram would span many pages as indicated by the scroll bars below the diagram panel.

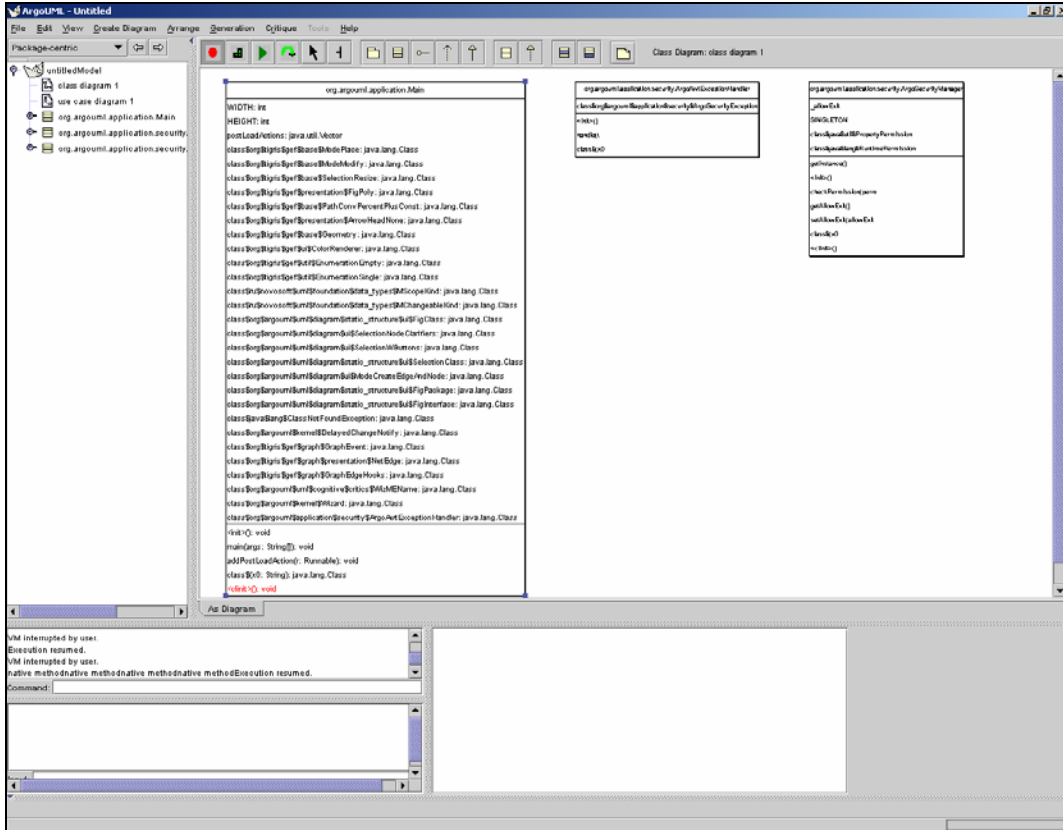


Figure 35 – Initial graph display for ArgoUML

The debugger connects to ArgoUML remotely via socket. The processing cycle for the display activity consists firstly of extracting data out of the debuggee process, then reverse engineering the data to form a model, then finally application of the graph layout algorithm prior to the display of that graph on the diagram panel. The initial processing cycle for debugging ArgoUML is slow and consumes a considerable amount of resources due to the enormous amount of data being extracted from the JPDA. This initial cycle takes approximately 120 seconds with the CPU utilization remaining at 100% for the entire duration. Furthermore the graph layout process always consumes more resources at load up time since objects are not displayed immediately near their final positions, requiring extensive animation. For this stage of the experiment, the debuggee ArgoUML consumes 32,000KB of memory while the debugger ArgoUML consumes 96,000KB of

memory. Once this initial processing is complete the requirements reduce significantly allowing for consecutive debugging of large programs or multiple programs in a networked system.

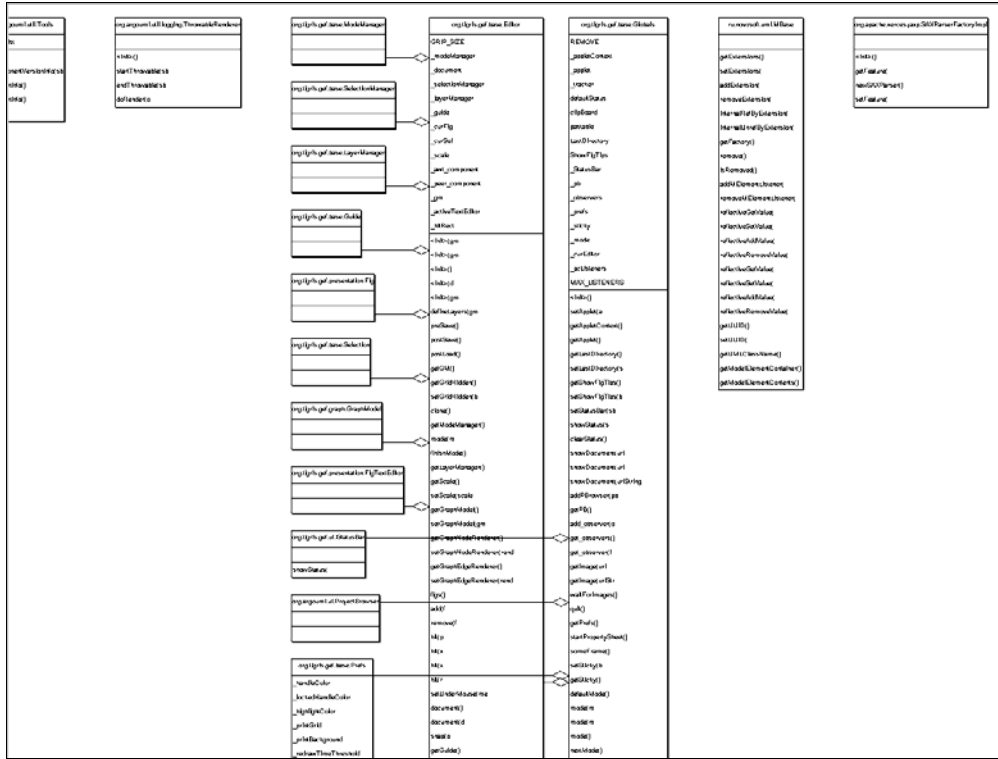


Figure 36 – ArgoUML later in its execution cycle

The debug controls incorporated into ArgoUML allow the user to step through the program execution and follow the control flow on the displayed UML class diagram. The node containing the method currently executing becomes the node in focus. Due to the filtered native classes, the method that is currently executing may not belong to any of the classes contained in the display. This change in level of detail as each method executes is valuable as it reduces cognitive search when viewing a large graph.

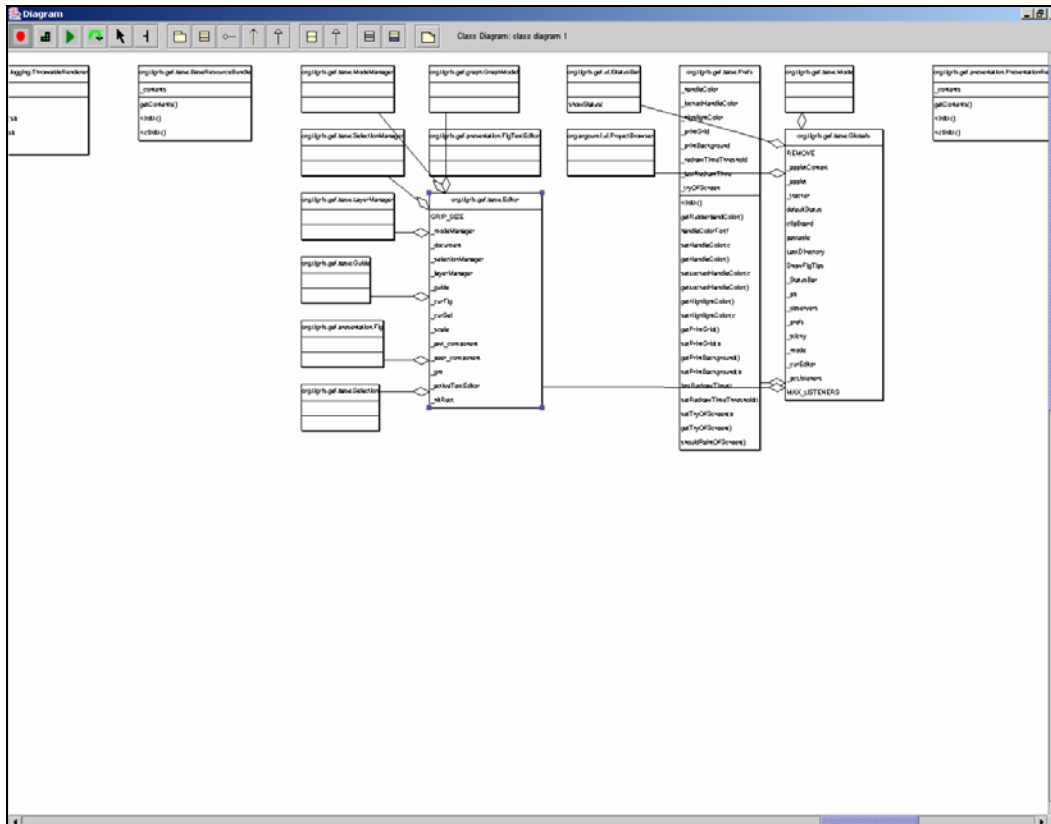


Figure 37 – ArgoUML layout after filtering is applied

5.6 Phase IV - Usability Tests

This section discusses the results obtained from the usability survey conducted at AFIT to establish the effectiveness of the modified ArgoUML debugger. Chapter 4 discussed the experimental setup of this usability test. Table 7 lists the average score of each criteria addressed by the survey. The following sections analyze the results in terms of each of the measured criteria.

5.6.1 Software Visualization

This portion of the survey, taken from the Roman and Cox PV taxonomy discussed in Chapter 2, received the highest scores from all five participants.

	Criteria	Score (/10)
Software Visualization Criteria	Space Economy	9.4
	Metaphors	9.2
	Interconnection	9.4
	Interface	8.6
	Scope	9.2
	Levels of Abstraction	9.6
	Presentation	8.8
Debugging Criteria	Suspend, Step, Resume program execution	7.4
	View threads, method calls	7.4
	View object information	7.0
	View variable information	7.2
	Breakpoint specification	8.2
Usability Criteria	Reading characters on diagram panel	8.8
	Organization of information	9.2
	Sequence of Screens	8.6
	Learning to operate the system	7.2
	Exploring features by trial and error	8.6
	Remembering names and use of commands	7.6
	Straightforwardness of performing tasks	7.4
	Help messages on the screen	6.2
	Reference Material	6.6
	System Speed	6.6
	System Stability	7
	Ease of correcting mistakes	7
	Design for all levels of users	5.8

Table 7 – Survey Results showing average scores for each criteria

Figure 38 is a chart graphing the results from the software visualization criteria. This group of criteria scored the highest among the three tested with an average of 9.17.

Space economy earned an average score of 9.4. It is obvious that the incorporation of the focus+context, graph layout algorithm and filtering techniques greatly satisfied this criteria. All users responded positively to the space economy that these techniques provided.

Comprehension of the metaphors used in ArgoUML scored 9.2. Metaphors refer to the graphical symbols used in the application. Since ArgoUML uses the standard UML class diagram semantics, and all participants were familiar with UML, they found the

display very easy to understand. Similarly with the Interconnection criteria scoring 9.4, users responded positively since the interconnections between nodes (i.e. inheritance links, aggregation links) follow the UML class diagram semantics.

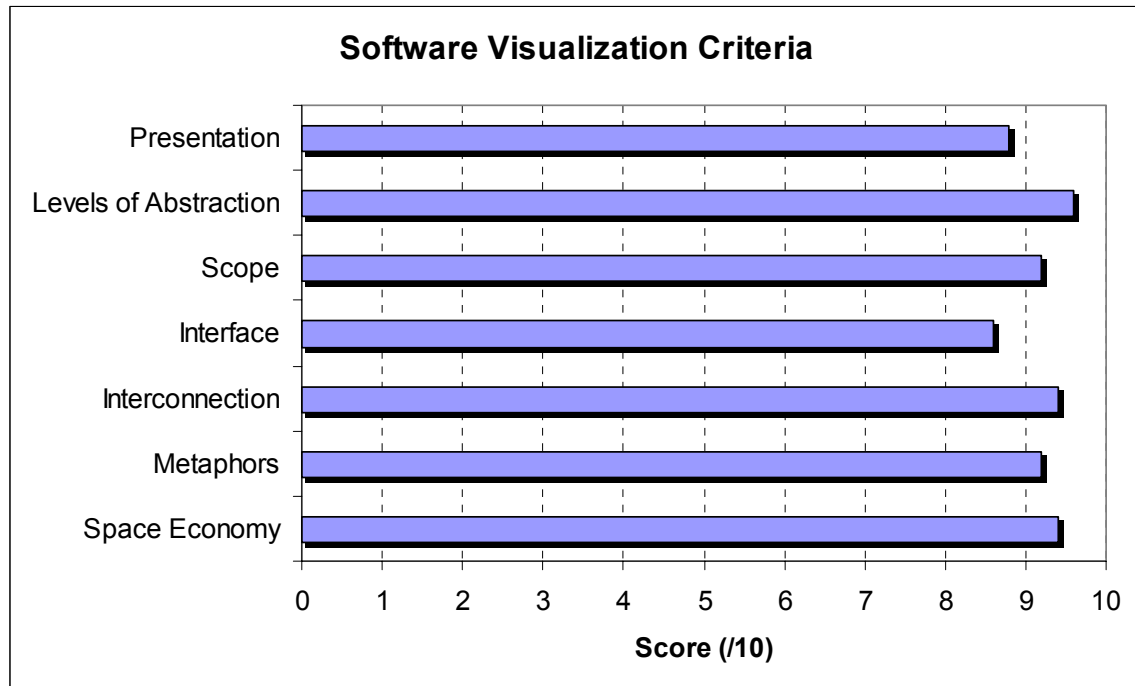


Figure 38 – Software Visualization Criteria Survey Results

Although the interface criteria still scored highly, receiving 8.6, this was the lowest rated criteria within the SV section. ArgoUML’s interface was designed to be user friendly and all modifications to the interface as a result of this research used the existing GUI framework within ArgoUML and GEF. However the interface scored as low as 7/10 from one of the participants. This is expected due to the behavior of nodes in ArgoUML. Sometimes it is difficult to select a node on the display panel. It may require the user to click around the node before it is selected. This behavior is not due to any modifications introduced by this research. Even the latest release of ArgoUML from its originators behave similarly [ARG02]. Since this criteria measures the ease of manipulating data, it is expected that the deficiency above would have a negative impact.

The scope and level of abstraction criteria scored 9.2 and 9.6 respectively. Participants responded well to the general layout of ArgoUML, showing a number of different types and levels of information.

Presentation earned a score of 8.8. Participants were able to easily interpret, not only the UML representation but all the other cognitive visual aspects such as use of color to highlight breakpoints or to indicate the current method in execution.

5.6.2 Debugging

This portion of the survey earned an average score of 7.44 across its five sub criteria. Figure 39 shows the average scores of each criterion. The first criteria – Suspend/Step/Resume capability scored an average of 7.4 from the five participants. The general response was although it was simple to perform these steps with the use of the ArgoUML debug buttons incorporated by this research, it was still not complete. As discussed in Chapter 4, this research added a step up function to the step capabilities of ArgoUML. This allows for rapid stepping through code. However a more precise control of program execution would still be preferable, such as the inclusion of step filters that allows the user to specify classes, packages, or class patterns that they do not want to step into. This is a very useful suggestion as it will allow for code tracing while avoiding external libraries that are not of interest.

View threads/method calls criteria also scored 7.4. The general response was that although current methods in execution are highlighted, no visualization was provided for viewing threads. Only a command line output is provided by ArgoUML to display threads and threadgroups.

Viewing object information and variable information scored 7 and 7.2 respectively. Once again, only a command line output is provided for these types of information rather than a displayed graphical visualization.

Breakpoint specification earned a fair score of 8.2 from the participants. Users are able to set and clear breakpoints either through a command line input or mouse selection over a chosen line on the source code display.

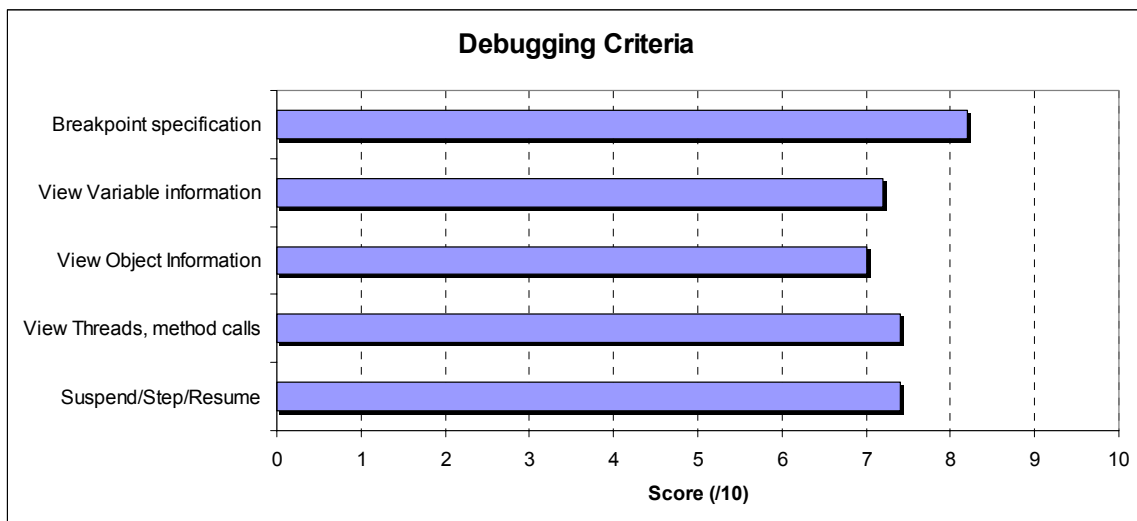


Figure 39 – Debugging Criteria Survey Results

5.6.3 Usability

The usability criteria were divided into three sections – Screen, Learning, and System Capabilities. The first criterion – reading characters on the display – scored an average of 8.8. ArgoUML provides a clear display of text on all display panels. However during animation, text can sometimes be distorted due to the smaller size of nodes.

All participants responded well to the organization of information, with the criterion earning an average score of 9.2. Similarly, the screen sequence scored in the top end among the usability criteria, earning an average of 8.6. The general response was that no

major aspect of the ArgoUML screen presentation was confusing. The graph comparing the average scores of these criteria is shown in Figure 40.

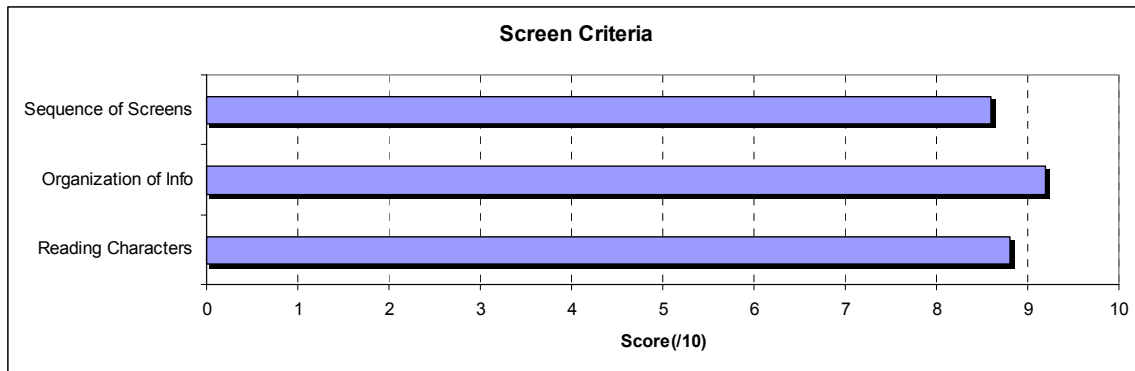


Figure 40 – Screen Criteria Survey Results

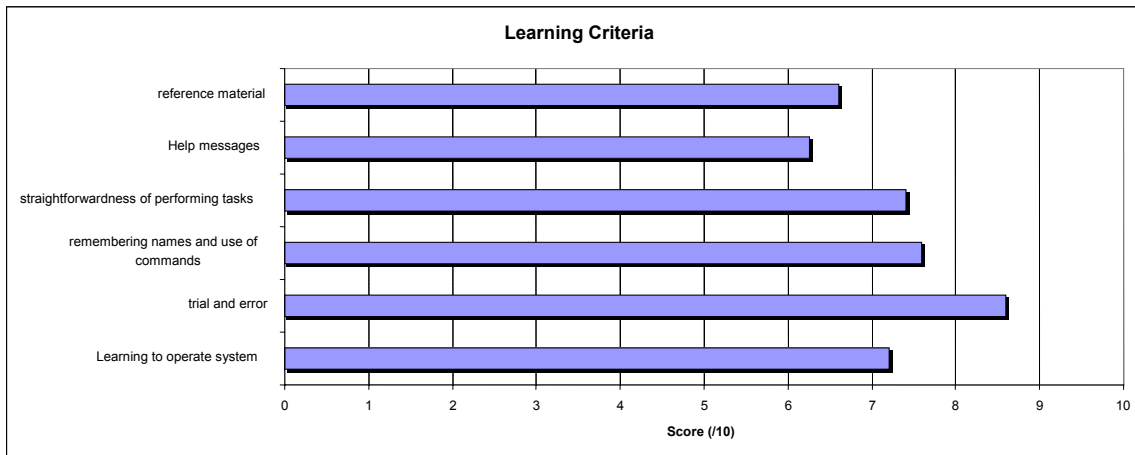


Figure 41 – Learning Criteria Survey Results

The learning criteria were fairly difficult to establish since participants are only briefly coached on the operation of ArgoUML and only basic operation was covered. Average scores are graphed in Figure 41. The score of the first learning criterion was 7.2. Although it is fairly easy to perform simple tasks, some of the more complex tasks like viewing object information requires remembering a sequence of commands. This is related to both the ‘straight forward task execution’ criterion and the ‘remembering of names and commands’ criterion which both earned similar scores at 7.6 and 7.4

respectively. The trial and error criterion scored an average of 8.5. Users were in general able to figure out how to perform debugging tasks or display modification tasks from trial and error. The help messages and reference material criteria scored a low 6.2 and 6.6 respectively. Only minimal help screens were included by this research and the current documentation on ArgoUML is still lacking. This criteria was included as part of the research to flag this documentation deficiency with ArgoUML.

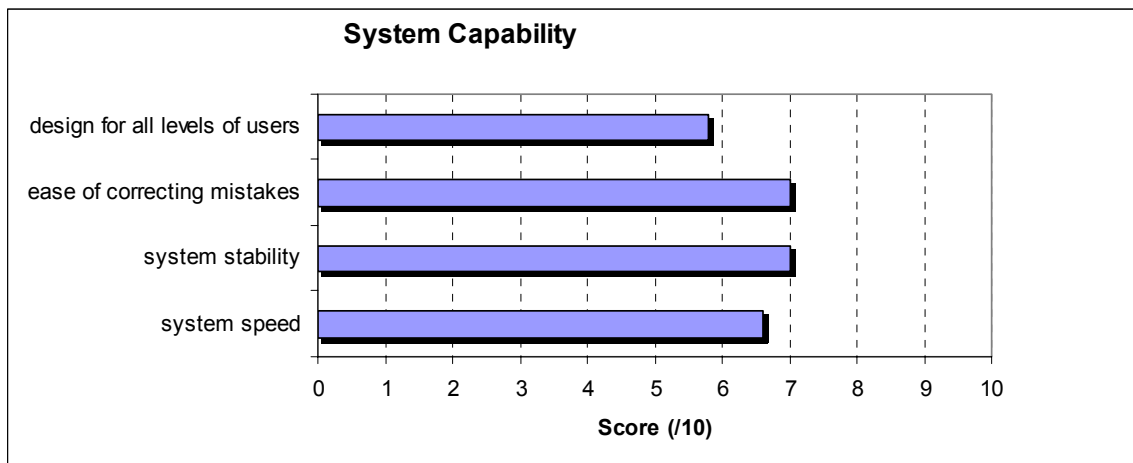


Figure 42 – System Capability Survey Results

The system capability criteria scored relatively low in comparison with the other criteria in the survey. The average scores are shown in Figure 42. Participants accepted that when using ArgoUML to debug systems, consideration must be given to the platform being used due to the CPU intensive nature of the system during the reverse engineering process. The speed criteria earned an average score of 6.6 while system stability scored 7. This is again related to the slower nature of the system during the reverse engineering process and animated display. ‘Correcting mistakes’ earned a score of 7. The user is not allowed to correct a command entered into the command line. The system will flag if the command is unknown, however if the command was accidentally entered, then there is no undo step to return to the original state. In terms of the diagram panel, users are

allowed to add and remove graphical components as they please. The lowest score of the survey was received by the last criterion – designing for all levels of users, which scored 5.8. Although all the participants had considerable experience in software engineering and using IDEs and standard debuggers, the general response was that ArgoUML was not at a stage where it can easily be used by novice users. An experience with simpler debuggers is required and an appreciation of UML semantics is preferred prior to using the modified introduced by this research ArgoUML.

5.7 Summary

This chapter presents the results obtained from the testing and experiments conducted to test the research hypothesis. It discusses the initial experimental set up and procedure for the different phases of testing. This is followed by an analysis of the results obtained from each phase. The phases comprise a Regression Test phase, General Display testing, Visual Debugging for different sized systems and the Usability test phase.

In the first three phases, the results are analyzed using resource requirements and visual effectiveness. The system showed an improvement in space efficiency over the standard ArgoUML layout, of at least 200% and as much as 1110% in the tests conducted. The fourth phase supports the initial investigation through a usability test that validates ArgoUML in terms of its usability aspects. The evaluation that resulted suggests that the visualization techniques used are promising, with their perceived visual effectiveness gaining relatively high scores. On the other hand, the experiment also flagged deficiencies in the debugging functionality and system capability aspects. These primarily relate to the slow reverse engineering process to display the class diagram on

the panel and lack of visual support in accessing some debugging information. Overall, feedback from participants suggests that the inclusion of a visual representation in the form of a dynamic UML class diagram aids in the debugging process as it negates the need to form a mental model of the program structure, while dealing with the lower level details.

A combination of the results from all phases leads to the conclusion that the developed system aids the user in the debugging process by increasing space efficiency and providing a wide scope of information as well as multiple levels of abstraction. These advantages provide a much greater understanding of the relationship between classes of interest and the remainder of the system.

6 Conclusion and Future Work

6.1 Introduction

This research designed, implemented and validated a visual debugger that effectively makes use of the standard UML class diagram to aid in the debugging of object-oriented systems. This chapter summarizes the research in terms of the goals stated in Chapter 1. First, it reviews the problem definition for the research. It then revisits the goals and the process used to demonstrate how these goals were met, as well as providing a brief discussion of results obtained from the tests conducted. Finally, it presents possible avenues for future work.

6.2 Motivation and Objectives

This section reviews the need for motivation behind this research, the objectives set out at the beginning of the research and its success at meeting those objectives.

6.2.1 Background

The JBI is a large distributed system that links many types of applications and databases to multiple users over multiple protocols. Debugging and analysis tools are required support this structure. Debugging of large object-oriented systems is a difficult cognitive process that requires understanding of both the overall and detailed behavior of the application. In addition, many such applications linked through a distributed system such as the JBI adds to this complexity.

There are many debugging tools currently available in the software development industry that deal with object-oriented systems. However few are capable of effectively promoting program understanding by presenting large amounts of data efficiently. Traditional debuggers limits the user to examine the source code line by line.

The motivation behind this research is to alleviate the problem discussed above. The presented hypothesis is that the incorporation of visualization techniques will reduce the complexity involved in the debugging process as it reduces cognitive load involved in forming a mental mapping of the program structure. More specifically, this research ports a UML class diagram display capability to an existing debug architecture. It considers UML class diagrams to be appropriate for this purpose as it is a familiar standard commonly used in the software engineering and design industry. The class diagram is obtained by reverse engineering the data extracted from the debuggee process by a debug architecture – the JPDA. This method ensures that the debuggee process is not affected at compile time and that there is only minimal effect at run-time.

Even with moderately sized systems, the UML class diagram may take up many pages. This research introduces a focus+context system that aims to provide detail for those objects which are considered interesting to the user while maintaining access to the overall context in which the detail exists. This allows the overall system structure to be displayed in a much smaller area while still preserving UML notation. The system's space efficiency is further improved with the application of an automatic hierarchical graph layout algorithm.

6.2.2 Research Impact

The effectiveness of the methodology introduced by this research is tested by implementing the proposed features in a modified version of ArgoUML. The presentation of UML class diagrams with a focus+context system provides a considerable improvement (up to 1110%) in the number of classes that can be displayed in a set display area while still preserving the semantics of UML. This great improvement in

space efficiency will allow users to debug large systems more easily based on the amount of data that is made manageable in a set information space.

The research applies the visual debugger to a number of different systems of varying sizes, to determine its effectiveness in different situations. Standard debug control features such as breakpoint setting and stepping functions are included. The developed visual debugger also allows the user to follow the control flow. This is achieved by highlighting the current method in execution. Various levels of abstraction are provided to the user, ranging from object and variable information (direct), to source code (direct), right up to the high-level class diagram representation (structural).

The research further supports the tests by conducting a usability type survey that involves participants testing the functionality of the application and responding to a developed questionnaire that involves 3 criteria – Software Visualization, Debugging, and Usability. The result obtained from these tests further supported the hypothesis of this research. The survey flagged a number of deficiencies but overall gave very positive feedback to the methodology introduced by this research.

The visual and debugger objectives systems are considered throughout the design and implementation processes. Objectives were initially defined in Chapter 1, then further refined in Chapter 3. All these objectives were satisfied and are summarized as follows:

- *Present information through multiple levels of abstraction, ranging from source code to high-level architectural views:-* this objective was clearly met by the modified ArgoUML. The system tests and the usability evaluation concluded that one of the main advantages of the system is its ability to present information through multiple levels of abstraction. Access to low-level details such as object information, variable information, and source code is combined

with a visualization of the high-level architectural view in the form of a UML Class Diagram.

- *Present the visualization model in a familiar manner:-* This objective was met with the use of UML. UML is a familiar modeling language for software and was deemed appropriate as the visualization model of this research. The effectiveness of using UML was evident from the results obtained in the system tests and usability survey.
- *Preserve context to aid in better program understanding:-* This objective was achieved with the incorporation of smooth animation when repositioning nodes. The overall effect of the animation enabled the user to maintain context, thus aiding programming understanding and the debugging process.
- *Present a dynamic visualization display such that it is suitable for real time debugging and match the changing runtime nature of programs:-* This objective was achieved by using the JPDA to extract data from the debuggee process and by creating an observer thread that reengineers this extracted data for the purpose of displaying its UML class diagram representation. The system tests verified that ArgoUML was able to map systems of various magnitude, ranging from the small simple systems to systems that comprise hundreds of classes.
- *Improve space efficiency to reduce search while presenting the user with detail for a region of interest:-* This objective was greatly satisfied by the incorporation of focus+context capability into the display. Space efficiency was further improved by coupling this focus+context feature with a hierarchical graph layout algorithm and filtering techniques.
- *Provide the ability to monitor processes for debugging:-* This was also achieved by porting an existing debug architecture (JPDA) into ArgoUML.

- *Have the ability to automatically reverse engineer object-oriented systems:-* Once again, the development of an observer thread that automatically reverse engineers data taken from the JPDA satisfies this requirement.
- *Have the ability to display multiple systems over socket connections for debugging:-* this requirement was satisfied during the Regression Test phase of the System Test.
- *Have no effect on the system behavior of monitored program:-* minimal system degradation was experienced, mainly occurring at load up time for large systems. This was caused by the large amount of data being reverse engineered.
- *Have minimal effect on system performance to enable real time debugging:-* Only minimal degradation in system performance was experienced for same reason as the previous requirement.
- *Provide standard debugging controls:-* Although other debugging controls were identified to be beneficial to the application, ArgoUML incorporates basic controls to enable the user to perform standard debugging procedures.
- *Provide a well designed GUI with consideration of Shneiderman's principles as stated in Chapter 2 [SHN98]. –* the usability survey verified that these principles were satisfactorily met by ArgoUML
- *Have a low CPU load and memory usage for the monitoring system:-* This requirement was generally met except at load up time of large debuggee programs.
- *Have a design adhering to software engineering principles wherever possible to ensure it adapts to future requirements:-* Use of software engineering patterns throughout the design and implementation of the system satisfactorily met this requirement.

6.3 Future Developments

In the course of this research, a number of improvements have been identified to further aid the debugging of object-oriented systems. The primary future addition identified by this research effort would be the visual representation of object information. Currently this is achieved only in ‘pretty-printed’ format by entering a command on the ArgoUML command line. The work presented here will require restructuring the data structures taken in by the observer thread in a form similar to Jtrees. This will allow easy navigation of object information in a given stack frame. The idea behind this is to be able to access local variables and objects in a presented scrollable view to allow easy navigation to values of interest.

Another avenue for a future addition is to vary the LOD algorithm to include user input. This will allow the user the option of setting the relative importance of each type of information displayed in the UML class diagram. This addition will further improve program understanding and will decrease cognitive search as it will allow important nodes to be displayed at a level of detail specified by the user.

A user controlled JPDA data filtering system would also be very beneficial to this application. Currently, the developer is required to edit the source code from the JPDA interface to filter certain packages that may not be of interest. A possible option within the ArgoUML menu is a data filtering option that allows users to enter packages that are not of interest, thereby negating the need to display those irrelevant classes in the diagram panel.

An intelligent stepping system would be advantageous, as flagged during the usability test. The current system although able to step up the execution tree, does not have any intelligent features that will allow users to specify classes that they do not wish

to step into. This would be very useful for users working in developing an integrated system like the JBI and would only be interested in tracing their own code. Moreover an intelligent stepping system may also allow users to control the speed of execution/stepping. This is particularly useful when the user has inadequate system knowledge or for examining new systems for idiosyncrasies.

Finally, a visual watch functionality is a desirable feature as part of the debugging features of ArgoUML. As stated previously, although object information can be accessed via command line in the current system, the addition of a ‘view watch’ capability whereby users are allowed to add local variables, fields, or expressions to a list that will be evaluated whenever the debugger is paused will significantly improve the debugging capability. In effect, addition of this functionality automatically informs users of information regarding objects or variables of interest.

6.4 Summary

Large software systems are difficult to debug. This complexity is further amplified by the introduction of object-oriented systems and systems connected in a distributed network such as those involved with the JBI. This research introduces a methodology that uses visualization techniques in order to aid in this debugging process. The research effort also includes the validation of the methodology in terms of its effectiveness as a visual debugger, though usability tests.

The research implements the methodology in ArgoUML – a graphical CASE tool, and interfaces it with the JPDA. The resulting ‘visual’ debugger monitors debuggee processes through the attached JPDA framework and reverse engineers the extracted data to form a UML class diagram representation. The user is able to use standard debugging

controls with the debuggee program. The visual debugger highlights the current method of execution on the class diagram to allow the user to track control flow within the program.

The main visualization technique incorporated is the focus+context feature which enables the user to view large quantities of information by focusing on an area of interest and applying a fisheye lens effect across the remainder of the displayed information. Despite the application of this technique, the visual debugger maintains the semantics of UML. These visualization techniques improve access to the information and allows the user to take in more information in the same amount of time. Moreover, a hierarchical graph layout algorithm is applied to the display to improve space efficiency even further. To enhance user understanding of the underlying software system, access to multiple levels of abstraction and a wide scope of information is provided. These include detailed information such as variable and object data and source code, to higher level representations displayed in the form of a UML class diagram.

These visualization techniques provide a more effective way of visually debugging object-oriented systems, both small and large. The effectiveness of the resulting system is supported by positive results from system testing and usability test. The system developed as part of this research was successful at meeting all objectives.

Appendix: Usability Survey Responses

RESEARCH SURVEY

for Master Thesis – Visual Debugging of Object-Oriented Systems with the Unified Modeling Language

by Flight Lieutenant Wendell Fox, RAAF, AFIT GCS-04M

Participant's No: 1

Date: 22 Jan 04

Time Commenced: 0835h

Time Finished: 0855h

1.0 Software Visualization Effectiveness

Please rate each criterion from the scale of 1-10 as indicated in the table below. A definition of each of these criteria is included below.

SOFTWARE VISUALIZATION CRITERIA		1	2	3	4	5	6	7	8	9	10	
1. SPACE ECONOMY	Inefficient									✓		Very efficient
2. METAPHORS	Difficult to understand								✓			Easy to understand
3. INTERCONNECTION	Confusing										✓	Very clear
4. INTERFACE	Hard to manipulate data								✓			Easy to manipulate data
5. SCOPE	narrow									✓		wide
6. LEVELS OF ABSTRACTION	Very limited										✓	Multiple levels
7. PRESENTATION	Difficult to interpret							✓				Easy to interpret

General Comments:

- *Use of visualization techniques aided in program understanding*
- *The levels of abstraction provided in one screen was effective*

Space Economy refers to the Application's ability to efficiently display Class information. Were visualization techniques used effectively? (e.g. Focus+Context, effective layout algorithm).

Metaphors refer to the graphical symbology used in the application.

Interconnection refers to the relationship between graphical components used to display Class information.

The *interface* consists of graphical objects presented to the viewer and interaction with the presentation using buttons, menus, and other controls or through direct manipulation of the graphical objects. The interface should be intuitive and easy to understand and use. In general, direct manipulation interfaces tend to be more intuitive than interaction through controls.

Scope refers to the amount of different information made available by the application to aid the user in debugging object oriented programs.

Presentation refers to the semantics of the graphical objects that are presented to the viewer. The presentation is that aspect of the visualization that facilitates interpretation and understanding of the graphics. This will cover issues concerning human cognition and effective visual communication such as the use of color, size, spatial relationships and other visual concepts to depict additional meanings.

2.0 Debugging Functionality

Please rate the level at which each of the listed debugging functionality was satisfied by ArgoUML from a scale of 1-10 as indicated in the table below.

DEBUGGING CRITERIA		1	2	3	4	5	6	7	8	9	10	
1. Suspend, Step, Resume program execution	Difficult to perform/not satisfied							✓				Easy to perform/adequately satisfied
2. View threads, method calls	Difficult to perform/ not satisfied							✓				Easy to perform/adequately satisfied
3. View object information	Difficult to perform/ not satisfied						✓					Easy to perform/adequately satisfied
4. View variable information	Difficult to perform/ not satisfied					✓						Easy to perform/adequately satisfied
5. Breakpoint specification	Difficult to perform/ not satisfied									✓		Easy to perform/adequately satisfied

General Comments:

- *Prefer if viewing of object info is incorporated into GUI*

3.0 Software Usability

The following survey items covers usability characteristics associated with the prototype debugger. Each criteria is self explanatory and is separated into three categories – screen, learning, and system capabilities. Please rate each item on a scale of 1-10 as indicated in the table below.

SCREEN		1	2	3	4	5	6	7	8	9	10	
1. Reading characters on diagram and debugger panel	hard									✓		easy
2. Organization of information	confusing										✓	very clear
3. Sequence of screens	confusing									✓		very clear

LEARNING		1	2	3	4	5	6	7	8	9	10	
1. Learning to operate the system	difficult						✓					easy
2. Exploring features by trial and error	difficult								✓			easy
3. Remembering names and use of commands	difficult										✓	easy
4. Performing tasks is straight forward	never							✓				always
5. Help messages on the screen	unhelpful						✓					helpful
6. Supplemental reference material	confusing							✓				clear

SYSTEM CAPABILITIES		1	2	3	4	5	6	7	8	9	10	
1. System speed	too slow							✓				fast enough
2. System tends to be	noisy/ unstable								✓			quiet/stable
3. Correcting your mistakes	difficult							✓				easy
4. Designed for all levels of users	never					✓						always

General Comments:

- Help function at command line is helpful

4.0 Comments

List 3 most positive aspects of the ArgoUML Visual Debugger:

1. *Focus+Context display looks good*
2. *Layout algorithm very space efficient*
3. *wide scope of info presented*

List 3 most negative aspects of the ArgoUML Visual Debugger:

1. *No way to visually (point and click) at object data (e.g. array)*
2. *slow at start up*
3. *java debugger not very intuitive*

RESEARCH SURVEY

for Master Thesis – Visual Debugging of Object-Oriented Systems with the Unified Modeling Language

by Flight Lieutenant Wendell Fox, RAAF, AFIT GCS-04M

Participant's No: 2

Date: 22 Jan 04

Time Commenced: 0900h

Time Finished: 0930h

1.0 Software Visualization Effectiveness

Please rate each criterion from the scale of 1-10 as indicated in the table below. A definition of each of these criteria is included below.

SOFTWARE VISUALIZATION CRITERIA		1	2	3	4	5	6	7	8	9	10		
1. SPACE ECONOMY	Inefficient											✓	Very efficient
2. METAPHORS	Difficult to understand											✓	Easy to understand
3. INTERCONNECTION	Confusing											✓	Very clear
4. INTERFACE	Hard to manipulate data											✓	Easy to manipulate data
5. SCOPE	narrow											✓	wide
6. LEVELS OF ABSTRACTION	Very limited											✓	Multiple levels
7. PRESENTATION	Difficult to interpret											✓	Easy to interpret

General Comments:

- *Very well organized. I like the way the boxes auto arrange themselves*

Space Economy refers to the Application's ability to efficiently display Class information. Were visualization techniques used effectively? (e.g. Focus+Context, effective layout algorithm).

Metaphors refer to the graphical symbology used in the application.

Interconnection refers to the relationship between graphical components used to display Class information.

The *interface* consists of graphical objects presented to the viewer and interaction with the presentation using buttons, menus, and other controls or through direct manipulation of the graphical objects. The

interface should be intuitive and easy to understand and use. In general, direct manipulation interfaces tend to be more intuitive than interaction through controls.

Scope refers to the amount of different information made available by the application to aid the user in debugging object oriented programs.

Presentation refers to the semantics of the graphical objects that are presented to the viewer. The presentation is that aspect of the visualization that facilitates interpretation and understanding of the graphics. This will cover issues concerning human cognition and effective visual communication such as the use of color, size, spatial relationships and other visual concepts to depict additional meanings.

2.0 Debugging Functionality

Please rate the level at which each of the listed debugging functionality was satisfied by ArgoUML from a scale of 1-10 as indicated in the table below.

DEBUGGING CRITERIA		1	2	3	4	5	6	7	8	9	10	
1. Suspend, Step, Resume program execution	Difficult to perform/not satisfied						✓					Easy to perform/adequately satisfied
2. View threads, method calls	Difficult to perform/ not satisfied						✓					Easy to perform/adequately satisfied
3. View object information	Difficult to perform/ not satisfied				✓							Easy to perform/adequately satisfied
4. View variable information	Difficult to perform/ not satisfied					✓						Easy to perform/adequately satisfied
5. Breakpoint specification	Difficult to perform/ not satisfied					✓						Easy to perform/adequately satisfied

General Comments:

- *Limited by the debugger*
- *Software still allows the setting of breakpoints and view object information – however this requires significant user interaction.*

3.0 Software Usability

The following survey items covers usability characteristics associated with the prototype debugger. Each criteria is self explanatory and is separated into three categories – screen, learning, and system capabilities. Please rate each item on a scale of 1-10 as indicated in the table below.

SCREEN		1	2	3	4	5	6	7	8	9	10	
1. Reading characters on diagram and debugger panel	hard										✓	easy
2. Organization of information	confusing										✓	very clear
3. Sequence of screens	confusing										✓	very clear

LEARNING		1	2	3	4	5	6	7	8	9	10	
1. Learning to operate the system	difficult									✓		easy
2. Exploring features by trial and error	difficult									✓		easy
3. Remembering names and use of commands	difficult									✓		easy
4. Performing tasks is straight forward	never									✓		always
5. Help messages on the screen	unhelpful									✓		helpful
6. Supplemental reference material	confusing									✓		clear
SYSTEM CAPABILITIES		1	2	3	4	5	6	7	8	9	10	
1. System speed	too slow							✓				fast enough
2. System tends to be	noisy/ unstable										✓	quiet/stable
3. Correcting your mistakes	difficult									✓		easy
4. Designed for all levels of users	never								✓			always

General Comments:

- *Knowledge of Java is required*
- *Especially knowledge of objects and classes*
- *Speed is limited by the debugger*

4.0 Comments

List 3 most positive aspects of the ArgoUML Visual Debugger:

1. *UML representation of java program (reverse engineered!).*
2. *Very clear layout and great self-organizing display.*
3. *easy and intuitive to use.*

List 3 most negative aspects of the ArgoUML Visual Debugger:

1. *Difficult to get object information*
2. *speed can be an issue for slower computers*
3. *need documentation*

RESEARCH SURVEY

for Master Thesis – Visual Debugging of Object-Oriented Systems with the Unified Modeling Language

by Flight Lieutenant Wendell Fox, RAAF, AFIT GCS-04M

Participant's No: 3

Date: 22 Jan 04

Time Commenced: 0945h

Time Finished: 1010h

1.0 Software Visualization Effectiveness

Please rate each criterion from the scale of 1-10 as indicated in the table below. A definition of each of these criteria is included below.

SOFTWARE VISUALIZATION CRITERIA		1	2	3	4	5	6	7	8	9	10	
1. SPACE ECONOMY	Inefficient								✓			Very efficient
2. METAPHORS	Difficult to understand									✓		Easy to understand
3. INTERCONNECTION	Confusing								✓			Very clear
4. INTERFACE	Hard to manipulate data									✓		Easy to manipulate data
5. SCOPE	narrow									✓		wide
6. LEVELS OF ABSTRACTION	Very limited									✓		Multiple levels
7. PRESENTATION	Difficult to interpret								✓			Easy to interpret

General Comments:

- *Appropriate levels of detail used for class nodes*

Space Economy refers to the Application's ability to efficiently display Class information. Were visualization techniques used effectively? (e.g. Focus+Context, effective layout algorithm).

Metaphors refer to the graphical symbology used in the application.

Interconnection refers to the relationship between graphical components used to display Class information.

The *interface* consists of graphical objects presented to the viewer and interaction with the presentation using buttons, menus, and other controls or through direct manipulation of the graphical objects. The interface should be intuitive and easy to understand and use. In general, direct manipulation interfaces tend to be more intuitive than interaction through controls.

Scope refers to the amount of different information made available by the application to aid the user in debugging object oriented programs.

Presentation refers to the semantics of the graphical objects that are presented to the viewer. The presentation is that aspect of the visualization that facilitates interpretation and understanding of the graphics. This will cover issues concerning human cognition and effective visual communication such as the use of color, size, spatial relationships and other visual concepts to depict additional meanings.

2.0 Debugging Functionality

Please rate the level at which each of the listed debugging functionality was satisfied by ArgoUML from a scale of 1-10 as indicated in the table below.

DEBUGGING CRITERIA		1	2	3	4	5	6	7	8	9	10	
1. Suspend, Step, Resume program execution	Difficult to perform/not satisfied								✓			Easy to perform/adequately satisfied
2. View threads, method calls	Difficult to perform/ not satisfied								✓			Easy to perform/adequately satisfied
3. View object information	Difficult to perform/ not satisfied									✓		Easy to perform/adequately satisfied
4. View variable information	Difficult to perform/ not satisfied								✓			Easy to perform/adequately satisfied
5. Breakpoint specification	Difficult to perform/ not satisfied								✓			Easy to perform/adequately satisfied

General Comments:

- *Command line arguments not intuitively obvious*

3.0 Software Usability

The following survey items covers usability characteristics associated with the prototype debugger. Each criteria is self explanatory and is separated into three categories – screen, learning, and system capabilities. Please rate each item on a scale of 1-10 as indicated in the table below.

SCREEN		1	2	3	4	5	6	7	8	9	10	
1. Reading characters on diagram and debugger panel	hard								✓			easy
2. Organization of information	confusing								✓			very clear
3. Sequence of screens	confusing								✓			very clear

LEARNING		1	2	3	4	5	6	7	8	9	10	
1. Learning to operate the system	difficult								✓			easy
2. Exploring features by trial and error	difficult								✓			easy
3. Remembering names and use of commands	difficult								✓			easy
4. Performing tasks is straight forward	never								✓			always
5. Help messages on the screen	unhelpful							✓				helpful
6. Supplemental reference material	confusing						✓					clear

SYSTEM CAPABILITIES		1	2	3	4	5	6	7	8	9	10	
1. System speed	too slow							✓				fast enough
2. System tends to be	noisy/ unstable				✓							quiet/stable
3. Correcting your mistakes	difficult							✓				easy
4. Designed for all levels of users	never						✓					always

General Comments:

- *Need faster pc to be more effective*

4.0 Comments

List 3 most positive aspects of the ArgoUML Visual Debugger:

1. *Class Diagram from reverse engineered data*
2. *Good info on values of variables*
3. *Not too difficult to use*

List 3 most negative aspects of the ArgoUML Visual Debugger:

1. *CPU-intensive at load up*
2. *too much command line arguments required for access to object information*
3. *need more help messages on visualization stuff*

RESEARCH SURVEY

for Master Thesis – Visual Debugging of Object-Oriented Systems with the Unified Modeling Language

by Flight Lieutenant Wendell Fox, RAAF, AFIT GCS-04M

Participant's No: 4

Date: 22 Jan 04

Time Commenced: 1450h

Time Finished: 1510h

1.0 Software Visualization Effectiveness

Please rate each criterion from the scale of 1-10 as indicated in the table below. A definition of each of these criteria is included below.

SOFTWARE VISUALIZATION CRITERIA		1	2	3	4	5	6	7	8	9	10		
1. SPACE ECONOMY	Inefficient											✓	Very efficient
2. METAPHORS	Difficult to understand											✓	Easy to understand
3. INTERCONNECTION	Confusing											✓	Very clear
4. INTERFACE	Hard to manipulate data											✓	Easy to manipulate data
5. SCOPE	narrow											✓	wide
6. LEVELS OF ABSTRACTION	Very limited											✓	Multiple levels
7. PRESENTATION	Difficult to interpret											✓	Easy to interpret

General Comments:

- *Use of standard UML notation useful for higher level understanding of system.*

Space Economy refers to the Application's ability to efficiently display Class information. Were visualization techniques used effectively? (e.g. Focus+Context, effective layout algorithm).

Metaphors refer to the graphical symbology used in the application.

Interconnection refers to the relationship between graphical components used to display Class information.

The *interface* consists of graphical objects presented to the viewer and interaction with the presentation using buttons, menus, and other controls or through direct manipulation of the graphical objects. The

interface should be intuitive and easy to understand and use. In general, direct manipulation interfaces tend to be more intuitive than interaction through controls.

Scope refers to the amount of different information made available by the application to aid the user in debugging object oriented programs.

Presentation refers to the semantics of the graphical objects that are presented to the viewer. The presentation is that aspect of the visualization that facilitates interpretation and understanding of the graphics. This will cover issues concerning human cognition and effective visual communication such as the use of color, size, spatial relationships and other visual concepts to depict additional meanings.

2.0 Debugging Functionality

Please rate the level at which each of the listed debugging functionality was satisfied by ArgoUML from a scale of 1-10 as indicated in the table below.

DEBUGGING CRITERIA		1	2	3	4	5	6	7	8	9	10	
1. Suspend, Step, Resume program execution	Difficult to perform/not satisfied										✓	Easy to perform/adequately satisfied
2. View threads, method calls	Difficult to perform/ not satisfied								✓			Easy to perform/adequately satisfied
3. View object information	Difficult to perform/ not satisfied									✓		Easy to perform/adequately satisfied
4. View variable information	Difficult to perform/ not satisfied									✓		Easy to perform/adequately satisfied
5. Breakpoint specification	Difficult to perform/ not satisfied									✓		Easy to perform/adequately satisfied

General Comments:

- *Most of the stuff only available through use of command line.*

3.0 Software Usability

The following survey items covers usability characteristics associated with the prototype debugger. Each criteria is self explanatory and is separated into three categories – screen, learning, and system capabilities. Please rate each item on a scale of 1-10 as indicated in the table below.

SCREEN		1	2	3	4	5	6	7	8	9	10	
1. Reading characters on diagram and debugger panel	hard									✓		easy
2. Organization of information	confusing										✓	very clear
3. Sequence of screens	confusing									✓		very clear

LEARNING		1	2	3	4	5	6	7	8	9	10	
1. Learning to operate the system	difficult							✓				easy
2. Exploring features by trial and error	difficult									✓		easy
3. Remembering names and use of commands	difficult							✓				easy
4. Performing tasks is straight forward	never								✓			always
5. Help messages on the screen	unhelpful							✓				helpful
6. Supplemental reference material	confusing								✓			clear

SYSTEM CAPABILITIES		1	2	3	4	5	6	7	8	9	10	
1. System speed	too slow							✓				fast enough
2. System tends to be	noisy/ unstable								✓			quiet/stable
3. Correcting your mistakes	difficult								✓			easy
4. Designed for all levels of users	never						✓					always

General Comments:

- *User must have some computer/software background to understand the UML visualization*
- *Animations could be faster*

4.0 Comments

List 3 most positive aspects of the ArgoUML Visual Debugger:

1. *use of standard UML semantics.*
2. *great application of focus+context.*
3. *allows view of code, UML and object/debugger data all in the same screen.*

List 3 most negative aspects of the ArgoUML Visual Debugger:

1. *A little on the slow side at load up time*
2. *some stuff only available by command line*
3. *difficult to click on nodes at times*

RESEARCH SURVEY

for Master Thesis – Visual Debugging of Object-Oriented Systems with the Unified Modeling Language

by Flight Lieutenant Wendell Fox, RAAF, AFIT GCS-04M

Participant's No: 5

Date: 22 Jan 04

Time Commenced: 1530h

Time Finished: 1555h

1.0 Software Visualization Effectiveness

Please rate each criterion from the scale of 1-10 as indicated in the table below. A definition of each of these criteria is included below.

SOFTWARE VISUALIZATION CRITERIA		1	2	3	4	5	6	7	8	9	10		
1. SPACE ECONOMY	Inefficient											✓	Very efficient
2. METAPHORS	Difficult to understand											✓	Easy to understand
3. INTERCONNECTION	Confusing											✓	Very clear
4. INTERFACE	Hard to manipulate data							✓					Easy to manipulate data
5. SCOPE	narrow										✓		wide
6. LEVELS OF ABSTRACTION	Very limited											✓	Multiple levels
7. PRESENTATION	Difficult to interpret											✓	Easy to interpret

General Comments:

- *Presentation of visuals is well organized and easy to understand.*
- *Animation useful in maintaining context of diagram*

Space Economy refers to the Application's ability to efficiently display class information. Were visualization techniques used effectively? (e.g. Focus+Context, effective layout algorithm).

Metaphors refer to the graphical symbology used in the application.

Interconnection refers to the relationship between graphical components used to display Class information.

The *interface* consists of graphical objects presented to the viewer and interaction with the presentation using buttons, menus, and other controls or through direct manipulation of the graphical objects. The

interface should be intuitive and easy to understand and use. In general, direct manipulation interfaces tend to be more intuitive than interaction through controls.

Scope refers to the amount of different information made available by the application to aid the user in debugging object oriented programs.

Presentation refers to the semantics of the graphical objects that are presented to the viewer. The presentation is that aspect of the visualization that facilitates interpretation and understanding of the graphics. This will cover issues concerning human cognition and effective visual communication such as the use of color, size, spatial relationships and other visual concepts to depict additional meanings.

2.0 Debugging Functionality

Please rate the level at which each of the listed debugging functionality was satisfied by ArgoUML from a scale of 1-10 as indicated in the table below.

DEBUGGING CRITERIA		1	2	3	4	5	6	7	8	9	10	
1. Suspend, Step, Resume program execution	Difficult to perform/not satisfied						✓					Easy to perform/adequately satisfied
2. View threads, method calls	Difficult to perform/ not satisfied								✓			Easy to perform/adequately satisfied
3. View object information	Difficult to perform/ not satisfied					✓						Easy to perform/adequately satisfied
4. View variable information	Difficult to perform/ not satisfied									✓		Easy to perform/adequately satisfied
5. Breakpoint specification	Difficult to perform/ not satisfied										✓	Easy to perform/adequately satisfied

General Comments:

- *Not all debugging function are incorporated into the GUI.*

3.0 Software Usability

The following survey items covers usability characteristics associated with the prototype debugger. Each criteria is self explanatory and is separated into three categories – screen, learning, and system capabilities. Please rate each item on a scale of 1-10 as indicated in the table below.

SCREEN		1	2	3	4	5	6	7	8	9	10	
1. Reading characters on diagram and debugger panel	hard								✓			easy
2. Organization of information	confusing								✓			very clear
3. Sequence of screens	confusing							✓				very clear

LEARNING		1	2	3	4	5	6	7	8	9	10	
1. Learning to operate the system	difficult						✓					easy
2. Exploring features by trial and error	difficult									✓		easy
3. Remembering names and use of commands	difficult				✓							easy
4. Performing tasks is straight forward	never					✓						always
5. Help messages on the screen	unhelpful		✓									helpful
6. Supplemental reference material	confusing			✓								clear

SYSTEM CAPABILITIES		1	2	3	4	5	6	7	8	9	10	
1. System speed	too slow					✓						fast enough
2. System tends to be	noisy/ unstable					✓						quiet/stable
3. Correcting your mistakes	difficult				✓							easy
4. Designed for all levels of users	never					✓						always

General Comments:

- *More help messages needed, but trial and error is usually successful*

4.0 Comments

List 3 most positive aspects of the ArgoUML Visual Debugger:

1. *excellent use of visualization techniques.*
2. *good use of animation.*
3. *screen presentation is well organized.*

List 3 most negative aspects of the ArgoUML Visual Debugger:

1. *need more help messages*
2. *need more advanced debugger*
3. *need more GUI debug interfaces*

Bibliography

- [ABS] Absolute360.com - Usability Testing
http://www.absolute360.com/NewletterInfo/useability_testing.htm
- [ARG02] ArgoUML User Manual,
<http://argouml.tigris.org/documentation/defaulthtml/manual/>, 20 June 2002.
- [AUB02] Auburn University, Graphical Representations of Algorithms, Structures, and Processes (JGRASP), <http://www.eng.auburn.edu/grasp/> 18 September 2002.
- [BAR03] Barnum, C. "What's in a Number?" *The Usability SIG Newsletter Vol 9 Issue 3*, January 2003.
- [BDM97] Baecker, R., C. DiGiano, and A. Marcus "Software Visualization for Debugging", *Communications of the ACM, Vol 4 Issue 4*, pp.44-54, Apr 1997.
- [BFN85] Batini, C., L. Furlani and E. Nardelli, "What is a Good Diagram? A Pragmatic Approach" in *Proc. 4th Internat. Conf. on the Entity Relationship Approach*, 1985.
- [BOR02] Borland TogetherControlCenter, <http://www.borland.com/together/>, version 6.0, 2002.
- [CB99] Cook, S., and S. Brodsky, *OMG Analysis & Design PTF UML 2.0 REQUEST FOR INFORMATION Response from IBM*, Online Document, <http://cgi.omg.org/docs/ad/99-12-08.pdf> 1999.
- [CDN88] Chin, J.P, V.A. Diehl, and K.L. Norman "Development of an Instrument Measuring User Satisfaction of the Human Computer Interface", *Proceeding of ACM CHI'88 Conference on Human Factors in Computing Systems*, 1988, pp.213-218.
- [CPHEB97] Chang-Hyun, J., S.K. Phil, S.I. Hyeung, H.P. Eui, and S.L. Byung, "A Design and Prototyping of an Object-Oriented Program Debugger", *Proceedings of the 1997 ACM symposium on Applied computing*, pp45-51, 1997.
- [CS99] Card, S., J. Mackinlay, and B. Schneiderman, *Readings in Information Visualization – Using Vision to Think*, San Francisco: Morgan Kauffman Publishers, 1999.

- [DETT99] Di Battista, G., P. Eades, R. Tamassia, and I. Tollis, *Graph Drawing- Algorithms for the Visualization of Graphs* Upper Saddle River: Prentice Hall, 1999.
- [DHKV93] De Pauw, W., R. Helm, D. Kimelman, J. Vlissides, “Visualizing the Behavior of Object-Oriented Systems” *Proceedings of the OOPSLA '93 Conference on Object- Oriented Programming Systems* , pp326-337, 1993.
- [EKS03] Eiglsperger, M., M. Kaufmann, and M. Siebenhaller “A Topology-Shape-Metrics Approach for the Automatic Layout of UML Class Diagrams”, *Proceedings of ACM 2003 Symposium on Software Visualization*, pp189-197, June 2003.
- [FUR81] Furnas, G.W. “The Fisheye View: A New Look at Structured Files”, in *Readings in Information Visualization – Using Vision to Think*, S. Card *et al.*, editors, San Francisco, Morgan Kauffman Publishers, 1981.
- [GJKKLM03] Gutwenger, C., M. Junger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel “A New Approach for Visualizing UML Class Diagrams”, *Proceedings of ACM 2003 Symposium on Software Visualization*, pp179-188, June 2003.
- [HOL02] Holzmann, G. “The Logic of Bugs” *Proceedings of the tenth ACM SIGSOFT symposium on Foundation of software engineering*, pp81-87, November 2002.
- [JM03] Jacobs, T. and B. Musial, “Interactive Visual Debugging with UML”, *Proceedings of ACM 2003 Symposium on Software Visualization*, pp115-122, June 2003.
- [JPD02] Java Platform Debugger Architecture Overview, Sun <http://java.sun.com/products/jpda/doc>, 28 April 2002.
- [JSW04] Blue Marsh Softworks - Jswat Online Documentation <http://www.bluemarsh.com/java/jswat/> version 2.2.1, 16 February 2004
- [JUN02] JUnit Online Documentation: <http://junit.sourceforge.net>, version 3.8 August 2002.
- [JVM03] Java Virtual Machine Profiler Interface, Sun <http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html>, November 2003.
- [KAN95] Kan, S., *Metrics and Models in Software Quality Engineering*, Reading Ma: Addison-Wesley Publishing, 1995.

- [KIL02] Kil, C.K., *Visual Execution Analysis for Multiagent Systems*. MS Thesis, AFIT/GCS/ENG/02-12. Department of Electrical and Computer Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2002.
- [KM01] Köth, O and M. Minas, “Abstraction in graph-transformation based diagram editors” *Second International Workshop on Graph Transformation and Visual Modeling Techniques*. Vol 50, Elsevier Science Publishers, 2001.
- [KOL96] Kolawa, A., “The Evolution of Software Debugging” *Parasoft IT Paper* <http://www.parasoft.com/jsp/products/article.jsp?articleId=490>, Parasoft 1996-2004.
- [MIL56] Miller, G., “The Magical Number Seven, plus or minus two: Some limits on our capability for processing information” *Psychological Science*, 63, 1956.
- [MUS03] Musial, B.R., *UML Assisted Visual Debugging for Distributed Systems*. MS Thesis, AFIT/GCS/ENG/03-12. Department of Electrical and Computer Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2003.
- [MW00] Mehner, K. and A. Wagner, “Visualizing the Synchronization of Java-Threads with UML”, In *Proceedings of the IEEE International Symposium on Visual Languages*, 2000.
- [OMG00] Object Management Group (OMG), Inc, Unified Modeling Language (UML) Specification Version 1.4, <http://www.omg.org/technology/documents/formal/uml.htm>
- [PBS93] Price B.A, Baekar R.M., Small I.S., “A Principled Taxonomy of Software Visualization”, *Journal of Visual Languages and Computing*, No. 4., pp211-266, 1993.
- [POB00] Paige, R., J. Ostroff, and P. Brooke, “Principles for Modeling Language Design”, *Information and Software Technology*, 42: 665-675, July 2000, UK.
- [RAT02] RATIONAL ROSE, Rational Software Corporation, <http://www.rational.com/products/rose/index.jsp> 18 September 2002.
- [RAU96] Rauterberg, M. “An Empirical Validation of Four Different Measures to Quantify User Interface Characteristics Based on a General Descriptive Concept for Interaction Points”, *Proceedings IEEE Symposium and Workshop on Engineering of Compute-Based Systems*, March 1996

- [RC93] Roman, G.C., and K.C. Cox, “A Taxonomy of Program Visualization Systems”, *IEEE Computer*, December, 1993.
- [REI03] Reiss, S. P., “Visualizing Java in Action”, *Proceedings of ACM 2003 Symposium on Software Visualization*, pp57-65, pp435-441, June 2003.
- [SBG99] Schmidt, A., M. Beigl, and H. Gellersen “There is more to context than location”, *Computers and Graphics volume 23 no 6*, pp893-901, 1999.
- [SHN98] Shneiderman, B. *Designing the User Interface. Strategies for Effective Human-Computer Interaction*. Third Edition, Reading MA: Addison-Wesley, 1998.
- [STA98] Stasko, J. *et al.*, *Software Visualization-Programming as a Multimedia Experience*, Cambridge MA: The MIT Press, 1998.
- [TH01] Telles, M. and Y. Hsieh, *The Science of Debugging*, Scottsdale AZ: The Coriolis Group, 2001.
- [TUF01] Tufte, E. *The Visual Display of Quantitative Information 2nd Ed*, Cheshire CT: Graphics Press, 2001.
- [ULF97] Ungar, D., H. Liebermann, and C. Fry. “Debugging and the Experience of Immediacy”, *Communications of the ACM Vol 40 Issue 4*, pp38-43, 1997.
- [USA00] United States Air Force Scientific Advisory Board. *Report on Building the Joint Battlespace Infosphere*, Volume 1, December 17 2000.
- [USA02] “Usability Evaluation”, www.pages.drexel.edu/~zwz22/UsabilityHome.html , 04 December 2002
- [WAR00] Ware, C. *Information Visualization – Perception for Design*, San Diego CA: Morgan Kaufmann Publishers, 2000.
- [WTMS95] Wong, K., S.R. Tilley, H.A. Müller, and M.A. Storey, “Structural redocumentation: A case Study”, *IEEE Software Vol 12, Issue 1*, pp46-54, January 1995.

Vita

Flight Lieutenant Wendell E. Fox joined the Royal Australian Air Force (RAAF) in July 1996 through the RAAF Undergraduate Sponsorship Scheme. He attended the Queensland University of Technology (QUT) in Brisbane and graduated with Honors in a Bachelor of Engineering degree majoring in Aerospace Avionics in 1999. After completing studies with QUT, he was posted to RAAF Amberley, near Brisbane. In January 1999, he received his commission and attended the RAAF Officer Training School (OTS) at Point Cook near Melbourne, Victoria.

Following graduation from OTS, Flight Lieutenant Fox attended several basic Avionics Engineering courses and following these courses, was employed as a design engineer at the Strike Reconnaissance Systems Program Office (SRSPPO) at RAAF Base Amberley, working for one year in the Avionics Support Section, and 2 years in the F111 Weapon System Support Facility.

In August 2002, Flight Lieutenant Fox entered the Air Force Institute of Technology as a graduate student in the Computer Science and Engineering Department. He graduated the institute with a Master of Science degree in Computer Systems, and went on to work at the Australian Defence Force's Directorate General of Technical Airworthiness as a Software Systems Certification Engineer.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 074-0188		
<p>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p> <p>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</p>					
1. REPORT DATE (DD-MM-YYYY) 23-03-2004		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From - To) Mar 2003 - Mar 2004	
4. TITLE AND SUBTITLE VISUAL DEBUGGING OF OBJECT-ORIENTED SYSTEMS WITH THE UNIFIED MODELING LANGUAGE			5a. CONTRACT NUMBER		
			5b. GRANT NUMBER		
			5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Fox, Wendell E., Flight Lieutenant, RAAF			5d. PROJECT NUMBER If funded, enter ENR #2001001		
			5e. TASK NUMBER		
			5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way, Building 640 Wright Patterson AFB OH 45433-7765			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/04-07		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR / Software and Systems Attn: Robert L. Herklotz, Ph.D. 801 N. Randolph St., Rm 732 Arlington VA 22203-1977 e-mail: Robert.herklotz@afosr.af.mil Comm: (703) 696-6565			10. SPONSOR/MONITOR'S ACRONYM(S) AFOSR/Software and Systems		
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT The Department of Defense (DoD) is developing a Joint Battlespace Infosphere, linking a large number of data sources and user applications. Debugging and analysis tools are required to aid in this process. Debugging of large object-oriented systems is a difficult cognitive process that requires understanding of both the overall and detailed behavior of the application. In addition, many such applications linked through a distributed system add to this complexity. Standard debuggers do not utilize visualization techniques, focusing mainly on information extracted directly from the source code. To overcome this deficiency, this research designs and implements a methodology that enables developers to analyze, troubleshoot and evaluate object-oriented systems using visualization techniques. It uses the standard UML class diagram coupled with visualization features such as focus+context, animation, graph layout, color encoding and filtering techniques to organize and present information in a manner that facilitates greater program and system comprehension. Multiple levels of abstraction, from low-level details such as source code and variable information to high-level structural detail in the form of a UML class diagram are accessible along with views of the program's control flow. The methods applied provide a considerable improvement (up to 1110%) in the number of classes that can be displayed in a set display area while still preserving user context and the semantics of UML, thus maintaining system understanding. Usability tests validated the application in terms of three criteria - software visualization, debugging, and general system usability.					
15. SUBJECT TERMS Software Engineering, Debugging (Computers), Visual Aids,					
16. SECURITY CLASSIFICATION OF:		17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT	b. ABSTRAC T			c. THIS PAGE	Maj Robert P. Graham
U	U	U	142	19b. TELEPHONE NUMBER (Include area code) (937) 255-6565 x4279; e-mail: Robert.Graham@afit.edu	