Interactive Visual Debugging with UML

Timothy Jacobs and Benjamin Musial
Air Force Institute of Technology
{Timothy.Jacobs, Benjamin.Musial}@afit.edu

ABSTRACT

Software debugging is an extremely difficult cognitive process requiring comprehension of overall application behavior along with detailed understanding of specific application components. Typical debuggers provide inadequate support for this process, focusing primarily on the details accessible through source code. To overcome this deficiency, we link dynamic program execution state to a Unified Modeling Language (UML) object diagram. We enhance the standard UML diagram with focus + context, graph layout, and color encoding techniques that organize and present objects and events in a manner that facilitates analysis of system behavior. We support debugging using high level abstractions commonly used in system design, while maintaining access to low level details with an interactive display.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging – Debugging aids, Distributed debugging; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – Restructuring, reverse engineering, and reengineering; D.1.7 [Programming Techniques]: Visual Programming

General Terms

Design, Performance, Human Factors

Keywords

Software visualization, Unified Modeling Language (UML)

1. INTRODUCTION

Analysis and troubleshooting of dynamically executing software systems is a problem faced by many software developers. Accomplishing this task requires an evaluation of the degree to which software applications, both individually and collectively, satisfy both functional and nonfunctional system requirements. While many tools are available to assist with this evaluation, our goal is provide a visual presentation that facilitates system evaluation using high-level design representations rather than stepping through individual messages or lines of code. As with any tool for monitoring software systems, we also need to leave the system behavior unchanged while minimizing the impact that monitoring has on system performance.

ACM Symposium on Software Visualization, San Diego, CA

As it is difficult to objectively analyze system behavior and requirements satisfaction, we focus instead on system abnormalities and errors that are likely to interfere with requirements satisfaction. These abnormalities might occur as bugs in individual applications, platform failures, invalid or unsynchronized information flows, or network failures. It is our hypothesis that such abnormalities and errors are likely to become more apparent through the visual presentation of dynamic attributes of system behavior, such as information and control flow, in conjunction with well understood structural models of software systems such as UML object diagrams.

Empirical evidence suggests that program debugging is an extremely difficult cognitive task. Even programs of modest size have multiple combinations of inputs and interactions that must be considered. Developers must simultaneously track multiple program details with a high degree of precision while maintaining a mental model of the many relationships between components within the overall program structure. Processes for developing programs require considerable input and collaboration from human developers, thus introducing numerous opportunities for error.

With distributed software systems, the problem is amplified. Interacting software processes can now run asynchronously, in any order, on different processors. Complexity is increased by at least another order of magnitude as reflected in the combinations of processes, interactions, and inputs that are now possible. Additional messages, data structures, and algorithms are required to manage software execution and coordinate information sharing among processes.

Telles and Hsieh [2001] define debugging as "the process of understanding the behavior of a system to facilitate the removal of bugs." Without adequate understanding of the system, it is difficult to identify and correct errors without affecting other parts of the system. One may correct symptoms of the problem without actually resolving the root cause of the problem. Tools that can increase system understanding during execution can facilitate the identification and safe removal of program defects.

This research investigates external visual representations for improving program understanding and execution analysis in a distributed environment. Visual representations capture the relationships and dependencies among the processes and data that are part of the information system. Program understanding is enhanced by linking the execution state of the program to well-understood models of the system such as UML object diagrams. By visualizing the program at multiple levels of abstraction, developers can hide extraneous information until it is needed, at which point they can drill down to the appropriate level of detail. Visual techniques such as animation and color encoding capture the dynamic nature of collaborative information systems by depicting changing information such as method invocations, communications messages, and state changes.

2. BACKGROUND

Visual presentation 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 program visualization techniques, Roman and Cox [1993] propose a taxonomy consisting of five criteria – scope, abstraction, specification method, interface, and presentation. Price et al [1997] identify additional criteria to include fidelity and invasiveness.

More powerful visualization techniques present a broad *scope*, covering many aspects of a program such as code, data state, control state, and behavior. In presenting this scope, these techniques should provide many views with multiple levels of *abstraction* from high-level design to code. The visualization system needs *specification methods* that require minimal effort from the user while providing sufficient flexibility for the viewer to customize the content being explored. The *interface* for interacting with the visual presentation should be intuitive and easy to understand and use. In general, direct manipulation interfaces tend to be more intuitive than interaction through controls. *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. Specification and presentation of program information must minimize *invasiveness* to the program while maintaining *fidelity*. The visualization should not alter the behavior of the program or use misleading semantics that lead to incorrect interpretation.

Debuggers such as those provided in Visual C++ [1998] and ProDev Workshop [2002] illustrate the low end of program visualization techniques. These environments provide little abstraction and minimalist interface capabilities. Typically, users are able to interact with textual representations of code and data state using simple text, menu, or button commands. 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.

At higher levels of abstraction, a number of research systems have been developed for visualization of particular aspects of program scope. State changes, message flow, or object instantiations are frequently depicted with animated representations of these dynamic behaviors during program execution [Baecker et al, 1997] [Jerding and Stasko, 1994, 1996]. Still other systems depict the relationships between objects or methods to facilitate program understanding [Müller et al, 1992]. These systems are primarily concerned with *presentation* of the various program aspects with little emphasis on other program visualization criteria.

Detailed visual diagramming languages (e.g. UML) have been defined for depicting the entities, relationships, state changes, and data flows in object-oriented programs. These "visual languages" provide multiple levels of abstraction and comprehensive presentation semantics for software analysis and design. Computer-aided software engineering (CASE) tools are widely available for developing and editing software designs with these standardized visual languages. Despite such widespread support for object-oriented visual design languages, the resulting diagrams are typically not used during debugging and analysis of executing programs. In fact manual transformation from the visual diagram to executable code is often required, thus severing the link between implementation and design. Only recently have researchers begun exploring languages such as UML for execution analysis. Examples include extensions to UML to incorporate execution semantics of concurrent programs [Mehner and Wagner, 2000].

3. VISUALIZING APPLICATIONS

In this paper, we describe dynamic visual representations for individual, object-oriented applications in a distributed system. We extract execution state and events from object-oriented Java programs using the Java Platform Debug Architecture [2002]. Using a scaled down version of ArgoUML [2002], we display the dynamically changing objects and events of the executing application with UML object diagrams. Using focus + context, graph layout, and color encoding techniques, we organize objects and events from the system to facilitate understanding of system behavior.

Over the past decade the Unified Modeling Language (UML) has become the standard modeling language for software systems. UML supports visual representations for various elements of program scope including static structure (class diagram), state transitions (state diagram), message passing (collaboration diagram), and event sequencing (sequence diagram). The symbols and associated semantics are well known within the software industry. UML is typically used to communicate software analysis and design models, but once these models have been implemented, the UML is seldom referenced. The disconnect between design and implementation is made worse by the lack of automated support for testing and debugging with UML. Since a major part of debugging involves program understanding and impact analysis, we contend that application of UML during debugging and testing activities will improve developer effectiveness. Furthermore, if UML models are referenced and maintained throughout the software life cycle, implementation and design models are more likely to be kept consistent.

The class diagram is the most widely used of all UML diagrams [Cook and Brodsky, 1999]. A class diagram includes rectangular nodes depicting classes with annotated lines between these nodes to indicate the relationships between classes. Closely related to the class diagram is the object diagram that depicts actual instances of classes and relationships in a system. In both diagrams, rectangular nodes depicting classes or objects are annotated with textual labels to identify the object and its associated state variables and behaviors. It is not uncommon for a software system to be composed of hundreds or thousands of unique classes or objects. Analysis and comprehension of such a software system requires both a high level overview of the system structure (consisting of numerous classes or objects and the relationships among them) and a detailed examination of the characteristics of individual classes or small subsets of classes. Software visualization techniques for UML diagrams should provide access to both high level and detailed views.

In our work, we use a UML object diagram to represent the objects and relationships in a Java application as it executes. To build the diagram, we extract Java program information from the Java Virtual Machine using the Java Platform Debugging Architecture (JPDA). Whenever a new object is encountered in the application, we extract information about its attributes, methods, and relationships with other objects. We then modify the object diagram to depict the newly extracted information

about the application. As the application executes, the active object is expanded to its highest level of detail and the active method is highlighted in color. In this manner, one can observe objects and methods as they are invoked.

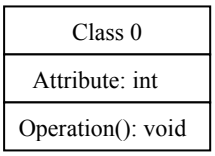
With hundreds or thousands of potential objects active in any application, we anticipate object diagrams that would require multiple pages to present at full detail. Navigating through many pages significantly increases the time to access components of interest. In addition, by spreading the model over multiple pages, it is not possible to simultaneously view high level system structure and individual component details, thus inhibiting understanding of relationships that may be important in understanding the impact of changes necessary to correct an error.

3.1 Degree of Interest Transformations

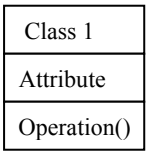
To overcome the problems associated with large diagrams, we have implemented a modified version of UML that maintains the symbology and semantics of UML while improving its space efficiency. To accommodate rapid access to both high level and detailed system information, we apply the concepts of focus + context to an existing, open source, UML editor known as ArgoUML. We develop a fisheye lens that displays less detail for components with a smaller degree of interest and we apply selective aggregation techniques to hide components that are beyond a specified degree of interest. Finally, we apply a graph layout algorithm that arranges components to emphasize hierarchical relationships while improving space efficiency.

We use selective filtering and modified graphical elements to create multiple level of detail (LOD) representations for UML classes. At the highest level of detail, each class consists of a rectangle divided into three segments that include textual labels for the class name, attributes, and methods. At lower levels of detail, the textual labels are selectively filtered according to the perceived relevance of the information. Although other possibilities for aggregation and filtering may exist, our solution supports six levels of detail as summarized here:

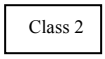
- LOD 0 contains the highest amount of detail. For a UML class representation this includes a full size graphical representation that includes textual labels for all attributes and methods along with type information.



- LOD 1 hides attribute and return types while minimizing margin size.



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



- LOD 3 removes all textual information and only indicates the presence of a component.

- LOD 4 contains no textual information and indicates the presence of a component at a reduced size.

- LOD 5 completely hides the component from the user.

A class is displayed 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 object in focus (Equation 1).

$$(1)\ DOI \approx \log_2(freq) + (max\_visible\_doi - dist)$$

In this equation, *freq* is the number of times the node in question has been accessed during the current debugger session; *dist* is the graphical distance from the node to the focal point in the diagram; and *max_visible_doi* is the maximum DOI at which a node is visible. The focal point is set to the object currently being executed unless the user selects a different focal point by clicking on a node. Similarly, node access is defined as execution flow passing to a particular object or a user selecting a node.

Each time a node is accessed, the degree of interest is recalculated, and the display is updated to reflect the appropriate level of detail for each node. When a class becomes active, it is temporarily displayed at LOD 1 to highlight current program activity. Should a displayed node be further from the focal point than a hidden node, all hidden nodes along the path of associations between the two nodes have their LOD increased to 4 to prevent dissection of the graph. If a cycle exists in the graph, the level of detail is determined by the longest path from the node in focus.

To demonstrate this mapping of LOD representations to the DOI function, consider the sample class diagram in Figure 1. If *Class 0* is accessed most frequently, then it becomes the node of focus and is displayed at LOD 0. *Class 1* is one link away from *Class 0*, so its degree of interest is one lower; consequently, *Class 1* is displayed with less detail at LOD 1. Each subsequent class in this diagram is one additional graphical link away from the node of focus, *Class 0*, so each subsequent class is represented at the next lower level of detail. Beyond LOD 4, additional classes in this path would be hidden using the selective aggregation techniques described below.

Generalization and *aggregation* are key concepts in object-oriented software systems. These relationships are hierarchical, with a generalization relationship linking a more general parent class to one or more specialized child classes. A child class is a special case of the parent class that modifies or adds to attributes and behavior that are inherited from the parent class. Similarly, aggregation links a parent container class to one or more child classes representing parts that are combined within the container class. As hierarchical relationships, both generalization and aggregation are good candidates for selective aggregation techniques that hide lower levels of the hierarchy. For instance, there may be little value in showing that a car consists of wheels, an engine, and chassis, when the user is only interested at the time in seeing that there is a car. UML also includes association

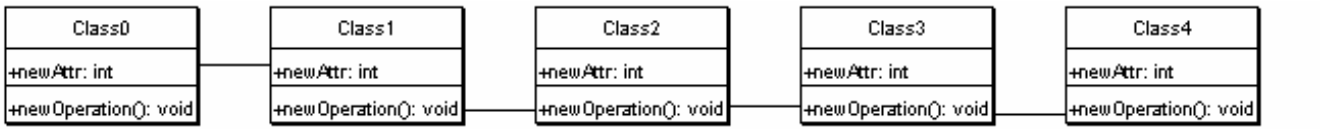


Figure 1a. Original object diagram

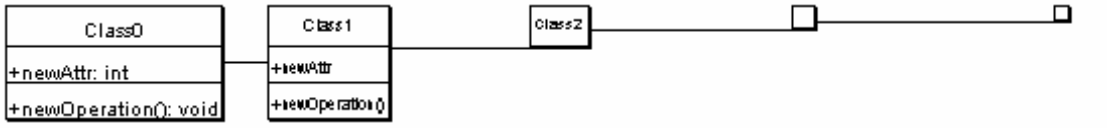


Figure 1b. Object diagram after application of degree of interest function

relationships that indicate a non-hierarchical relationship between two classes.

We apply selective aggregation, based on UML's association, aggregation, and inheritance relationships, to hide classes that are graphically distant from the class in focus. To indicate that classes are hidden, we continue to display the appropriate UML symbology for inheritance or aggregation. For an association relationship we simply display a partial link indicating hidden classes. When the content is brought into focus, the hidden hierarchy expands revealing the aggregate relationships within.

Figures 2a and 2b show the results of our selective aggregation techniques for a simple class diagram. Figure 2a depicts a class diagram before selective aggregation is applied. The link with a triangle at the end depicts an inheritance relationship and the link with a diamond depicts an aggregation relationship. The triangle and the diamond are positioned at the end of the link associated with the parent class. Figure 2b presents the class diagram after applying our selective aggregation techniques. The triangle, diamond, and a small line continue to be displayed to indicate that the hidden classes are connected via some relationship to the class. Any classes of a lower DOI and further from the focal point are also hidden from view. It is expected that if the aggregate class is of little interest and is thereby hidden, then so will those objects that it is associated with.

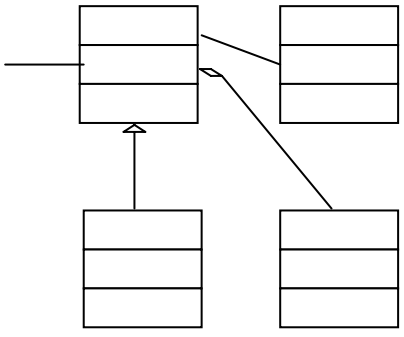


Figure 2a. Partial class diagram

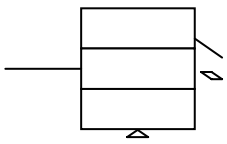


Figure 2b. Class diagram after selective aggregation

3.2 Graph Layout

To improve the space efficiency of our focus + context techniques for UML, the positions of elements in the model must also be modified. We develop a graph layout algorithm that automates this repositioning. In addition to space efficiency, our algorithm emphasizes the hierarchical presentation of generalization relationships and it preserves user context by minimizing the relative reassignment of node positions. Hierarchical presentation and user context are given priority over other factors such as edge crossings and connection points.

Application of our layout algorithm requires the movement of nodes and links in the class diagram. If done instantaneously, this movement may cause the user to lose the original context. To avoid this, we use smooth animation when moving graphical elements between their original and revised positions. We also move the elements in two stages. Using a layered algorithm (see [Battista et al, 1999]), we first position elements in the appropriate layer of the inheritance hierarchy. The second stage then positions leaf nodes to optimize space efficiency. The graph layout algorithm is applied after all degree of interest and selective aggregation functions.

We describe the application of our degree of interest techniques and graph layout algorithm using the initial class diagram shown in Figure 3. In this example, we first set the upper left node as the focal point. After applying our degree of interest and selective aggregation techniques, the diagram is revised as shown in Figure 4. Selective aggregation affects those nodes furthest from the node of focus as observed at the far right of the diagram.

At this point, we are ready to apply our layout algorithm. In the first stage of this algorithm, we assign nodes to a layer based on their inheritance hierarchy. A difference of one layer exists across an inheritance relationship. Each node connected via an association or aggregation relationship will be placed in the same layer. If a cycle exists, a node may be connected to multiple hierarchical layers. In this case, a node that is a direct descendant of an inheritance relationship is placed in a lower layer. All other nodes in the cycle remain in the original layer determined through the association relationships. Figure 4 identifies the layers in our sample diagram.

Once each node is placed in a layer, the algorithm proceeds to position each node within the layer. Nodes descending from a node on another layer are positioned directly below the parent node. Other nodes in a layer are positioned relative to the hierarchically positioned node in the same relative horizontal order that they were positioned in the original diagram. Each

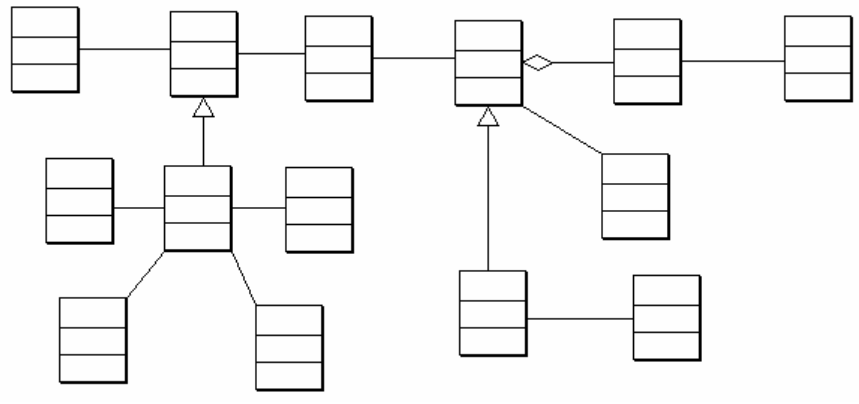


Figure 3. Example class diagram

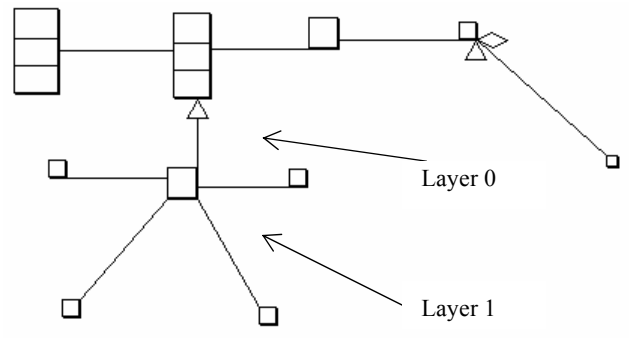


Figure 4. Class diagram after DOI function

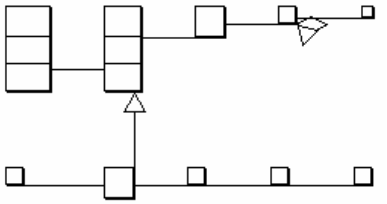


Figure 5. Class diagram after Phase 1 of layout algorithm

node in a row is separated by a horizontal gap of a fixed distance. Similarly the y position for each row is determined by adding a fixed size gap to the lowest point in the above row. This stage of the algorithm has the effect of shrinking the overall display area yet it maintains the general layout of the original diagram. The results of this process for our sample graph are shown in Figure 5.

The second stage of the layout algorithm searches each row for nodes with leaf nodes on the same row. If these leaf nodes have a relatively low LOD (and vertical height), they are shifted to the right side of the associated node. The leaf nodes are arranged radially with the radius proportional to the number of leaf nodes. If possible, the position of other nodes on that row is left unchanged. Should there be insufficient room, then all nodes to the right of the leaf nodes are shifted right to establish a suitable gap.

The bottom left node in our example (Figure 5) has four associated leaf nodes on its row. The second stage of the layout algorithm rearranges these nodes in a radial fashion as shown in Figure 6. Stage 2 repositioning increases the number of nodes that can be displayed in the same amount of space while still

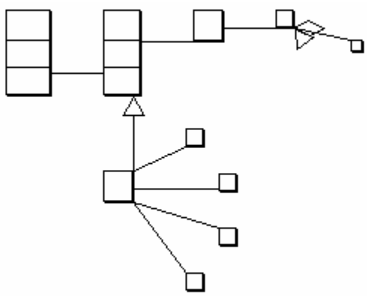


Figure 6. Class diagram after Phase 2 of layout algorithm

preserving user context. This is particularly effective when placing leaf nodes with a low LOD around a node with a high LOD.

4. VISUAL DEBUGGING

By applying our visual techniques, we are able to view a far larger portion of the object diagram for a given application. By doing so, we are better able to grasp the overall structure and relationships between object instances in the application. In displaying a larger portion of the diagram, we are not able to see the detailed information such as attributes and methods for each class; however, by applying focus + context techniques we are able to drill down and display this information for those objects of most interest, while still maintaining our view of the relationships between objects.

For these visual techniques to be effective for debugging, we need to include some indication of what part of the program is currently being executed. We do this by increasing the level of detail displayed for the active object while highlighting the active method in red so that it stands out from the rest of the diagram. By increasing the level of detail, we also increase the relative size of the active object. By changing the size and color of active portions of the application, and displaying a large part of the application model on a single display, it becomes much easier to follow the control flow as the application executes.

Once we identify an area of interest, we can interact with our visual presentation to display more details of class methods or attributes. If necessary, we can also simultaneously display the source code associated with a particular class. Like other debuggers, we can set breakpoints and examine variables and other characteristics of our program.

5. DISCUSSION

To analyze the effectiveness of our focus + context techniques we compare our modified UML displays to the original UML class diagram displays. A major goal of this aspect of our work is to maintain the symbology and semantics of UML while improving its space efficiency to accommodate rapid access to both high level and detailed system information. We analyze the effectiveness of our techniques with respect to this goal.

If we assume fairly simple classes such as those in our earlier examples, then we can fit roughly 24 classes on a single screen using standard UML presentation techniques such as those in ArgoUML. By applying our degree of interest and graph layout algorithms, we can display approximately 75 classes on a single screen. Thus we obtain three times the space efficiency without even resorting to selective aggregation. The degree to which selective aggregation increases our space efficiency is dependent on the level of fan out in the system relationships.

Access to high level and detailed system information is more difficult to evaluate. With original representations, we will need to scroll through multiple pages if we have more than 24 classes. In order to deal with large systems, the user would have to commit much of the overall system information to long term memory. By applying our focus + context techniques, we can readily view 75 classes simultaneously with the structure of hundreds of classes potentially available using selective aggregation. This enables the user to offload more cognitive processing to the external visual presentation.

Using original ArgoUML representations, details are readily available for up to 24 classes at a time; however, the user can still only focus on one area of the screen at a time. With our focus + context view, only a few components are in focus at any given time. The user needs to interactively select another component to bring it into focus. Once the appropriate component is brought into focus, the time to access detailed information is similar to that of the original UML. If all components that the user is likely to access are displayed on the same page, it is likely to be faster to switch focus using the original UML; however, for the common case where components are spread over multiple pages in the original UML, our focus + context techniques are likely to improve access time since more components are accessible from the same screen.

To demonstrate our techniques, we apply them to a research application consisting of 84 classes and roughly 34,000 lines of code. Typical debugging techniques would potentially require browsing all 34,000 lines of code to identify the relationships between the various classes and methods. By debugging with our modified UML, one can readily observe the flow of control between classes along with the static structures indicating how the various classes are related. If we can fit 50 lines of code on a page, 34,000 lines of code would require 680 pages. Based on our theoretical calculations for UML, 84 classes would require between three and four pages to display with typical UML presentation styles. With our modified UML, nearly 90 per cent of the classes should fit on a single page. In actuality, due to the many attributes and methods in the classes in this system, less than two classes fit on a page with standard UML presentation techniques (Figure 7). Our modified UML for this application displays 10 classes on a single page (Figure 8). This number is less than our theoretical calculations due to the flatness of the inheritance hierarchy. With only two layers, all 84 classes are placed in two rows. Thus we have two very long rows that only partially fit on the page, while we have considerable unused space in lower layers of the diagram. To resolve this problem, we must modify our algorithm to create additional layers between the inheritance layers.

These examples clearly demonstrate that our focus + context techniques, while not optimal, provide improved space efficiency over standard UML layouts for a class diagram. Now we analyze our techniques using the program visualization criteria described in Section 2 – scope, abstraction, specification method, interface, presentation, fidelity, and invasiveness.

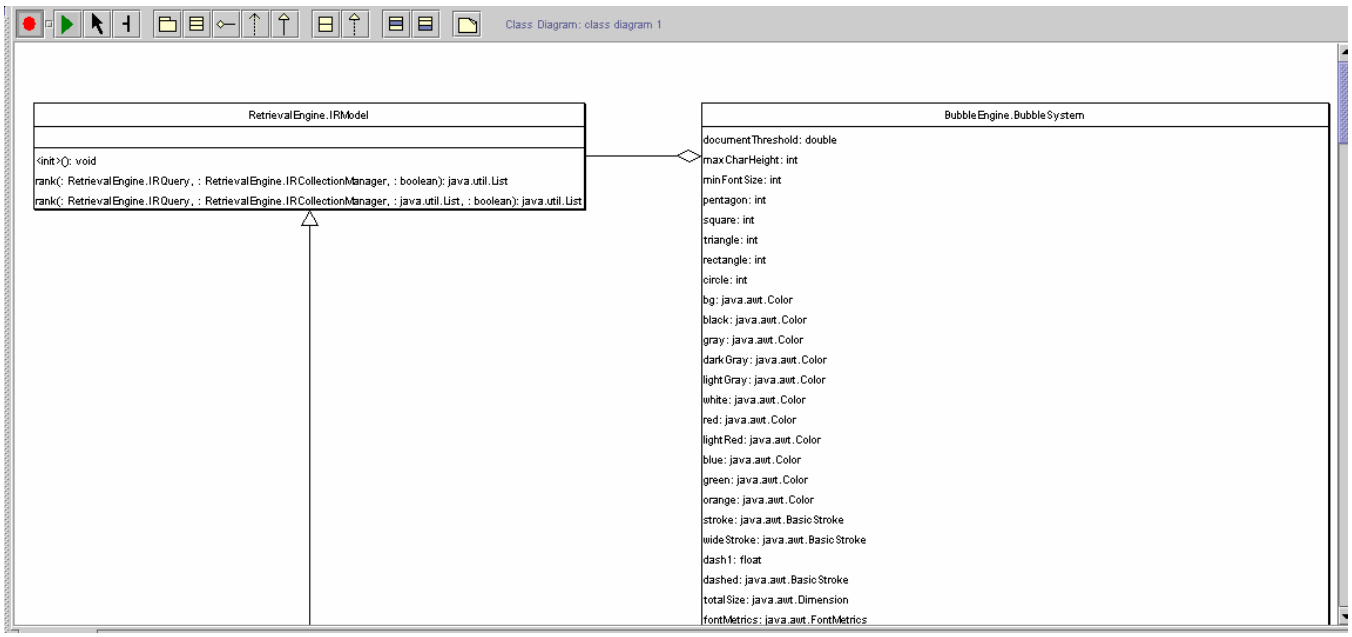


Figure 7. Object Diagram with standard UML techniques

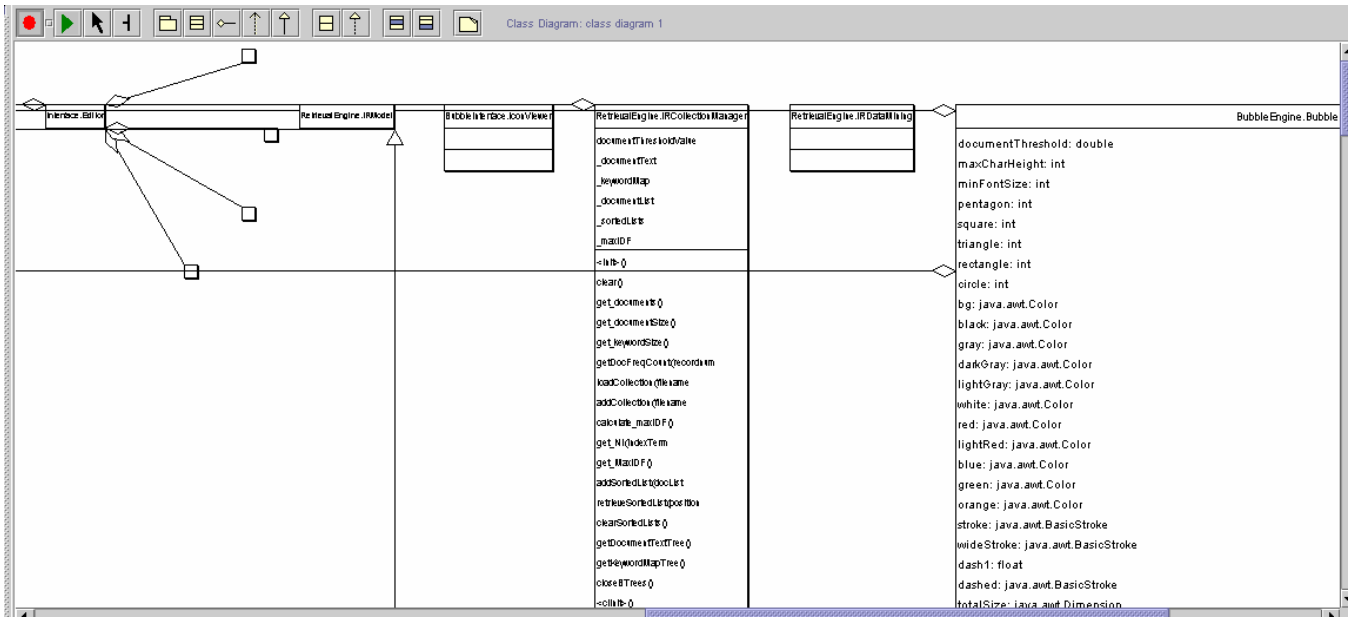


Figure 8. Object Diagram with focus + context applied

Our current implementation limits the scope of our visualization techniques to that provided with the UML class diagram along with source code. We can identify the program structure while monitoring the flow of control as the program executes. Similarly, abstraction is primarily achieved through the visual representations supported by UML. We can view the application at different levels of detail that include overall system structure, classes and relationships, methods and attributes, and source code.

Specification methods are provided through the Java Virtual Machine and require little or no input from the user. In a distributed environment, the user may have to specify the Internet location of the application or the source code. No changes are required to any of the application code, thus the only way in which our techniques might alter the behavior of the application is through the consumption of resources in the Java Virtual Machine that are needed by the application. This problem can be overcome by running the visual debugging system on a separate virtual machine or platform than the application being debugged.

Our user interface and presentation semantics are basically defined by the UML implementation in ArgoUML. We modify the presentation to make it more space efficient using focus + context techniques; however, we do not alter the semantics.

6. DISTRIBUTED SYSTEMS ANALYSIS

In earlier work [Kil, 2002], we emphasized system characteristics unique to distributed systems, such as the information and control flow between applications, rather than trying to capture every aspect of individual application behavior. By exploring information from the messages between collaborating applications, one can learn a considerable amount about distributed system behavior and performance. We facilitate this exploration by presenting multiple visual abstractions of distributed applications and the messages between them. These views provide an overview of system relationships and behavior with which the user can interact to obtain more detailed information about specific messages. Important events such as message transmission and errors are animated or highlighted to immediately attract the user's attention. In this manner, the user can more easily identify the cause and effect of different system behaviors. We intend to integrate this earlier work with the techniques presented here to facilitate analysis and debugging of distributed systems.

7. CONCLUSIONS

Program debugging has proven to be a difficult cognitive task. In this work, we successfully link program execution to a UML class diagram to offload some of the cognitive modeling to an external visual display. To support analysis and modification of large software applications, we modify the standard UML class diagram with focus + context, graph layout, and color encoding techniques. In doing so, we provide access to both high level structural information and low level program details. We also draw attention to the flow of control in the program as it moves from method to method between objects.

While we have not yet demonstrated the effectiveness of these presentations in accomplishing actual debugging tasks, our techniques significantly improve the quantity of information available in a single display while linking program execution to a well-known representation commonly used in application design. By associating debugging with design representations, we take advantage of the higher level abstractions that have proven effective in the engineering of many successful software systems.

8. ACKNOWLEDGMENTS

This work is funded in part by the United States Air Force Office of Scientific Research. The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

9. REFERENCES

ArgoUML, Tigris.org, http://argouml.tigris.org, 2002.

BAECKER, R. ET AL, "Software Visualization for Debugging", *Communications of the ACM*, April 1997, pp 44-54.

BAKER, M.J. AND EICK, S.G., "Space-Filling Software Visualization", *Journal of Visual Languages and Computing, 6*, 1995, pp 119-133.

BATTISTA, G. ET AL, *Graph Drawing Algorithms for the Visualization of Graphs*, Prentice Hall, 1999.

Building the Joint Battlespace Infosphere, Volume 1, U.S. Air Force Scientific Advisory Board, SABTR-99-02, 1999.

CHUAH, M.C. AND EICK, S.G., "Information Rich Glyphs for Software Management Data", *IEEE Computer Graphics and Applications*, July 1998, pp 24-29.

COOK, S. AND S. BRODSKY, *OMG Analysis & Design PTF, UML 2.0, Request for Information, Response from IBM*, IBM Corporation, http://cgi.omg.org/docs/ad/99-12-08.pdf, 1999.

Java Platform Debugging Architecture, Sun Microsystems, http://java.sun.com/products/jpda, 2002.

JERDING, D.F. AND STASKO, J.T., *Using Visualization to Foster Object-Oriented Program Understanding*, Technical Report GIT-GVU-94-33, Georgia Institute of Technology, 1994.

JERDING, D.F. AND STASKO, J.T., *The Information Mural: Increasing Information Bandwidth in Visualizations*, Technical Report GIT-GVU-96-25, Georgia Institute of Technology, 1996.

KIL, C.K., *Visual Execution Analysis for Multiagent Systems*, Master's Thesis, Air Force Institute of Technology, 2002.

MEHNER, K. AND WAGNER, A., "Visualizing the Synchronization of Java-Threads with UML", In *Proceedings of the IEEE International Symposium on Visual Languages*, 2000.

MÜLLER, H.A. ET AL, *A reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models*, ACM, 1992.

PRICE, B., BAECKER, R., AND SMALL, I., "An Introduction to Software Visualization", in *Software Visualization: Programming as a Multimedia Experience*, John Stasko et al, editors, The MIT Press, Cambridge, Massachusetts, 1997.

ProDev Workshop, Silicon Graphics Incorporated, www.sgi.com/developers/devtools/tools/prodev.html, 2002.

ROMAN, G.-C. AND COX, K.C., "A Taxonomy of Program Visualization Systems", *IEEE Computer*, December, 1993.

TELLES, M. AND Y. HSIEH, *The Science of Debugging*, The Coriolis Group, Scottsdale, AZ, 2001.

Visual C++ 6.0 Professional Edition, Microsoft Corporation, 1998.