APPLYING INFORMATION VISUALIZATION TECHNIQUES TO VISUAL DEBUGGING

by

John A. Costigan, III

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master's of Science
Computer Science and Applications

Virginia Polytechnic Institute and State University

June 13, 2003

Approved by ______________________________
Dr. Chris North, Chairperson of Supervisory Committee

____
Dr. Cliff Shaffer, Member of Supervisory Committee

____
Dr. Scott McCrickard, Member of Supervisory Committee

Program Authorized to Offer Degree Department of Computer Science

Date ______________________________

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

ABSTRACT

APPLYING INFORMATION
VISUALIZATION TECHNIQUES TO
VISUAL DEBUGGING

by John A. Costigan

June 13, 2003

Chairperson of the Supervisory Committee: Dr. Chris North
Department of Computer Science

In the arena of software development, implementing a software design (no matter how perfect the design) is rarely done right the first time. Consequently, debugging one's own (or someone else's) software is inevitable, and tools that assist in this often-arduous task become very important with respect to reducing the cost of debugging as well as the cost of the software life cycle as a whole. Many tools exist with this aim, but all are lacking in a key area: information visualization. Applying information visualization techniques such as zooming, focus and context, or graphical representation of numeric data may enhance the visual debugging experience. To this end, drawing data structures as graphs is potentially a step in the right direction, but more must be done to maximize the value of time spent debugging and to minimize the actual amount of time spent debugging. This thesis will address some information visualization techniques that may be helpful in debugging (specifically with respect to visual debugging) and will present the results of a small pilot study intended to illustrate the potential value of such techniques.

TABLE OF CONTENTS

Table of Contents i
List of figures iii
Acknowledgments v
Glossary vi
Chapter 1: Introduction 1
    1.1 Information Visualization 1
    1.2 Data Structure Visualization 1
    1.3 Debugging 2
    1.4 Visual Debugging 3
    1.5 Visual Debugging Prototypes 7
        1.5.1 Visual Debugger (VDB) 7
        1.5.2 Data Structure Visualizer (DSV) 10
        1.5.3 Visual Debugger 2.0 (VDB2) 12
Chapter 2: Review of Related Work 15
    2.1 Instructional Tools 15
        2.1.1 Brown University Algorithm Simulator and Animator 16
        2.1.2 XTango, Polka, and Samba 17
        2.1.3 Language-Independent Visualization Environment 18
    2.2 Visual Debuggers 19
        2.2.1 Dotty 20
        2.2.2 Deet 21
        2.2.3 VDBX 22
        2.2.4 Swan 23
        2.2.5 GNU Visual Debugger 24
        2.2.6 Data Display Debugger 26
        2.2.7 HotWire 27
        2.2.8 Groove and Execution Murals 28
        2.2.9 Lens 29
    2.3 Summary 31
Chapter 3: Information Visualization for Visual Debugging 33
    3.1 Graphical Value Representation 33
    3.2 Zoom and Pan 37
    3.3 Overview+Detail 39
    3.4 Focus+Context 42
    3.5 Direct Manipulation 46
    3.6 Layout 48
        3.6.1 Data Structure-Specific Layout 49
        3.6.2 Universal Layout 50

3.6.3 Stability 52
3.6.4 Issues with Other Information Visualization Techniques 53
3.6.5 Summary 54
3.7 Visual Notification 55
3.8 Summary 58
Chapter 4: Visual Debugger 2.0 60
Chapter 5: Visual Debugging Software Architecture 66
5.1 General Implementation Issues 66
5.2 Integrating with Microsoft Visual C++ 6.0 Debugger 71
5.3 Integrating with Microsoft Visual Studio .Net Debugger 75
5.4 Model-View-Controller (MVC) 78
5.5 Pluggable Graphical Value Representation 79
5.6 Pluggable Layout Algorithms 81
5.7 VDB2 Design Details 82
5.7.1 Backend Database 82
5.7.1 User Interface 85
5.8 Summary 87
Chapter 6: Evaluation 89
6.1 Test Focus 89
6.2 Test Plan 91
6.3 Test Results 94
6.4 Discussion 96
Chapter 7: Conclusions and Future Work 98
7.1 Conclusions 98
7.2 Summary of Contributions 98
7.3 Future Research 99
References 102

LIST OF FIGURES

Number	*Page*
Figure 1: The Visual Debugger	9
Figure 2: The Visual Debugger, in a more compact form.	9
Figure 3: The Data Structure Visualizer	11
Figure 4: The Visual Debugger 2.0	13
Figure 5: The main menu for BALSA	16
Figure 6: A Polka visualization of a sorting algorithm	18
Figure 7: A Polka-3D visualization of the quicksort algorithm	18
Figure 8: LIVE, creating code for a linked list	19
Figure 9: A graph displayed with Dot	20
Figure 10: A graph displayed with Neato	20
Figure 11: Deet	21
Figure 12: VDBX	23
Figure 13: The Swan Data Structure Visualization System	24
Figure 14: The GNU Visual Debugger	25
Figure 15: The Data Display Debugger	27
Figure 16: A tree, using DDD's automatic layout algorithm	27
Figure 17: Hotwire, debugging a word processor.	28
Figure 18: The Groove object-oriented program visualizer	29
Figure 19: The Lens visual debugger, using XTango	30
Figure 20: Table Lens	34
Figure 21: The "bar" visualization mode of VDB2	35
Figure 22: The "color" visualization mode of VDB2	36
Figure 23: The DSV, zoomed out	39
Figure 24: The same graph, zoomed in	39
Figure 25: The DSV, in Overview+Detail mode	40
Figure 26: Defining which data is important to the user	41
Figure 27: VDB, in Detail mode	42
Figure 28: VDB, in Overview mode	42
Figure 29: An interactive fisheye view for browsing a website	43
Figure 30: The DSV, with no focus	45
Figure 31: The DSV, focused on the root node	45
Figure 32: The DSV, after extending the focus	45
Figure 33: Automatic layout of a data structure	48
Figure 34: The same data structure, after the user moves some nodes	48
Figure 35: Result of automatic layout	51
Figure 36: Layout after a few nodes are added	51
Figure 37: VDB2, showing a new node and several modified variables	56

Figure 38: VDB2, showing a deleted node 56
Figure 39: VDB2, showing a memory leak and a dangling pointer. 57
Figure 40: The Visual Debugger 2.0 Main Display 60
Figure 41: VDB2 in text mode 61
Figure 42: VDB2 in bar mode 62
Figure 43: VDB2 in color mode 62
Figure 44: VDB2 zoomed in 62
Figure 45: VDB2 zoomed out 62
Figure 46: VDB2 with Focus+Context 63
Figure 47: VDB2, showing a new node and several modified variables 64
Figure 48: VDB2, showing a deleted node 64
Figure 49: VDB2, showing a memory leak and a dangling pointer. 64
Figure 50: Retrieving information from the runtime 73
Figure 51: Rough pseudocode of the update algorithm 76
Figure 52: Class Hierarchy for Primitive Data Type Visualizers 80
Figure 53: The Layout menu of the DSV 82
Figure 54: Class hierarchy of VDB2's backend database 83
Figure 55: Class hierarchy of VDB2's user interface 85
Figure 56: The user interface classes 87
Figure 57: A typical linked list response test, text mode 92
Figure 58: A typical BST response test, text mode 92
Figure 59: A typical response test, bar mode 93
Figure 60: A typical response test, color mode 93
Figure 61: Summary of pilot test results 94

ACKNOWLEDGMENTS

The author wishes to thank: Dr. Chris North for the guidance that only a competent advisor can provide; Ines Khelifi for her assistance in developing the Visual Debugger 2.0; Dhruv Manek and Thy Nguyen for their assistance in developing the Data Structure Visualizer; Ben Wilhite for his assistance in developing the Visual Debugger; Matthew Sample and Sam Stone for the back end of the Visual Debugger; Dr. Cliff Shaffer and Dr. Scott McCrickard for serving on the thesis defense committee; and the participants in the pilot study.

GLOSSARY

call stack. a block of memory that stores local data used for calls to functions. Also called a "frame stack" or just a "stack."

child. in a rooted tree, the vertex on the sending end of a particular vertex's incoming edge.

data structure. a collection of related data along with the operations that can be performed with the data.

digraph. in graph theory, a graph whose edges have direction. Also called a "directed graph."

directed graph. in graph theory, a graph whose edges have direction. Also called a "digraph."

edge. in graph theory, a relationship (either with or without direction) between two vertices.

frame stack. a block of memory that stores local data used for calls to functions. Also called a "call stack" or just a "stack."

graph. in graph theory, a set of vertices (usually labeled) and a set of corresponding edges (sometimes labeled) between the vertices.

heap. a large block of memory that is dynamically allocated as necessary to store program data. Also called a "memory heap."

layout. the visual organization of the nodes and relationships of a graph.

lay out. to visually organize the nodes and relationships of a graph, usually to make the graph more understandable to humans viewing the graph.

leaf. in a rooted tree, a vertex with no children or whose children are all empty trees.

memory heap. a large block of memory that is dynamically allocated as necessary to store program data. Also called a "heap."

memory leak. memory allocated on the heap without a reference with which the owner program can access it.

node. in graph theory, an element of a graph between which edges are connected. Also called a "vertex."

parent. in a rooted tree, the vertex on the receiving end of a particular vertex's single outgoing edge.

pointer. a piece of data that provides access to another piece of data.

root. in graph theory, a special vertex—with either no outgoing edges or no incoming edges—that is designated to turn a tree into a rooted tree.

rooted. in graph theory, a tree is said to be rooted when there is a single distinguished vertex (often labeled or clearly indicated), called the root, that has no outgoing edges.

stack. in memory allocation, a block of memory that stores local data used for calls to functions. Also called a "frame stack" or a "call stack."

tree. in graph theory, a connected graph with no cycles.

undirected graph. in graph theory, a graph whose edges have no direction.

vertex. in graph theory, an element of a graph between which edges are connected. Also called a "node."

C h a p t e r   1

CHAPTER 1: INTRODUCTION

1.1 Information Visualization

Information visualization is a field of computer science that deals with effective means of displaying or otherwise communicating data to human beings. Commonly, this involves displaying data in a way that is somewhat unrelated to its original form in order to enhance human ability to parse the information. The organization of data in computer memory is often different from the way it is displayed to humans, particularly because computer memory is limited by its linearity. In all situations that involve digital data, the simplest way to display the data would be to output the data as a stream of bits, just as the computer stores it. This approach, however, is not human-friendly, so information visualization techniques are used to present digital data to humans in a more understandable fashion.

When the size of data becomes too large to be easily communicable via standard methods of display, information visualization can be used to summarize or generalize data into a smaller format, allowing more data to be viewed simultaneously at the cost of detail. Analysis of the summarized display may prove more efficient than viewing many pages of detailed display, possibly revealing trends that are not as easily noticed.

1.2 Data Structure Visualization

Data structure visualization deals with the construction of understandable graphical representations of memory-based data structures. Data structures are commonly used when implementing a software solution to a problem, and many computer science students learn about data structures early in the programming curriculum. One problem that hampers students that are new to data structures is their physical invisibility. When coding up the answer to a problem, programmers cannot see the actual data structures—physically, they are only streams of bits in a computer's memory, invisible to the programmer. By creating graphical representation, students—and veteran programmers—can better understand the data structure that it represents.

When representing actual program data, a visualization of a data structure not only enhances comprehension of the data structure—critical for programmers unfamiliar with the data structure—but may also provide important information about specific values in a data structure. In a typical text-based debugger, a programmer might query the debugging tool for each of the values of the data members present in a data structure—a data structure visualization could include this data automatically, concurrently providing detail about several parts of a data structure inside the visualization itself.

1.3 Debugging

Debugging is the process of finding and then fixing bugs in a program. Bugs are generally not known to exist until the impact of the bug is observed, oftentimes during programming or validation testing of the program. The process of debugging can be summarized in five steps:

1. Observe negative impact of bug.
2. Identify state of program that produces negative impact (the error state).
3. Locate source code that produces the identified error state.
4. Modify source code to eliminate possibility of entering error state.
5. Verify that modification fixes bug.

The negative impact in step one could be anything from incorrect output, to a graphical anomaly in the user interface, to a crash that causes the program to end abruptly. The impact may be observed during the programming process (as the programmer verifies the effects of code that he or she writes), during the validation process, or even during regular use by end-users. Many times, the negative impact will be directly related to a data structure. For example, the negative impact may be the direct result of a faulty state in the data structure. In this case, identifying the faulty state in the data structure leads to identifying the faulty state in the program. Data structure visualization, then, can be used to identify these data structure maladies.

1.4 Visual Debugging

Visual debugging is the application of data structure visualization to the debugging experience. Instead of debugging with print statements or with text-based debuggers, the visual debugger offers the programmer an abstract and understandable graphical portrait of his or her program. The ideal visual debugger would replace entirely a text-based debugger and the programmer's need for print statements. Rather than invent the wheel, however, a visual debugger can rely on a text-based debugger as its engine, using the text-based debugger to retrieve certain programming information and providing the same features that an ordinary text-based debugger would provide. The difference is that the visual debugger includes additional, graphical features that better facilitate overview, generalization, and trend identification.

The goal of a visual debugger is to provide visually understandable representations of data structures or other program data during runtime, so that developers can better understand the actual organization of their data structures and respond accordingly to improve the programmatic management of those data structures. Ultimately, these visual representations should increase the productivity of programmers that are developing and debugging a system, especially programmers unfamiliar with the current system.

Visual debuggers generally include all of the features of a standard text-based debugger (sometimes they even provide a text interface with a text-based debugger), plus additional visual features to provide a graphical point of view of the program. For example, a visual debugger may include a graphical flow chart illustrating the basic procedure of the code being debugged.

Visual debuggers can use their graphical nature to enhance understanding of a program. Users new to a system of code, a common situation for code maintainers, can use a visual debugger to get a general graphical overview of a system that they couldn't get without reading the code. In addition to understanding of the system, visual debuggers can help users to observe the effects of certain lines of source code as they are executed. For example, a data structure visualization that lists the values of certain variables would allow the programmer to watch those variables as they change through the execution of the code, much like the watch variable concept in standard text-based debuggers. The difference between text-based watch variables and visual debugger watch variables is context. Text-based debuggers require specification of which variables should be watched, but a visual debugger can automatically provide those variables, plus the context of those variables—information surrounding the variable, such as other variables in that object (if the variable is part of an object).

The main difference between visual debuggers is in their methods of displaying program information. Some debuggers provide general overviews about class types and their structures [14], great for getting to know an unfamiliar system. Other debuggers provide specific visualizations of program data during execution of the program [37] [38] [5] [13]. An important example of this kind of visualization is that of data structures—this is useful not only for getting a feel for the design of data structures, but also for debugging the creation, manipulation, and destruction of these data structures.

A general solution to the problem of visualizing data structures lies in graphs. Most data structures use a method of address referencing to organize pockets of data. For example, in a linked list, each of the "pockets" is an element of the list, and each element contains a reference to the next item in the list. This creates a directional relationship between adjacent elements in a linked list. Each element is related to the next element in the list by the reference that the former holds to the latter. These references can constitute edges of a graph, and the vertices of the graph can represent pockets of data. The problem then reduces to one of properly displaying the graph in a humanly understandable form.

There are several algorithms for **laying out** directed and undirected graphs in general, but they are not necessarily universally appropriate for data structures. In addition, most classes of data structures have a generally accepted way of being drawn (i.e. linked lists as a linear row of blocks connected by arrows); there should be a way to utilize these standards when generating drawings of data structures. This implies advance knowledge of the type of data structures in the program that is being debugged. It may be possible to determine the class of data structure automatically, or at worst it can be queried from the user without excessive trouble.

It is also important that the diagrams display as much pertinent information as possible about the elements in the data structure. When given more information, the developer can make better conclusions regarding the reliability of their program, and the additional information is also useful in determining the sources of bugs and their resulting solutions. Too much information, however, wastes space and can interfere with a programmer's ability to quickly access the information that he or she wants. A healthy balance must be maintained to provide the programmer with as much information as is needed to solve the problem, but no more or less. An information visualization concept known as Focus+Context, which allows a particular detailed focus on one section of information while a context of the rest of the information that consumes less screen space is maintained, might be of use in this department.

In addition to standard graph visualization of data structures, it should be possible to apply other information visualization strategies in the user interface of a visual debugger, particularly with respect to the display of data structure values. Until now, solutions have used only text to represent the values of variables. While text is certainly informative, it is difficult to grasp the "big picture" of a data structure by examining all of the text of the variable values. Representing values, such as integers or strings, as graphical entities is information visualization at its literal.

An overview of the data structure, trading detail for "big picture," might help programmers spot some data structure problems more quickly, or at least point programmers in the right direction. An overview is usually useless without the ability to investigate specific details of parts of the overview, which is why overviews are usually combined with detailed views to form a concept called Overview+Detail.

The important issues at hand are to determine what kind of information visualization ideas can feasibly be applied to the visual debugging domain. How can these information visualization concepts be cultivated and implemented in a visual debugger? What kinds of effects do these features have on the debugging process? To help answer these questions, a series of visual debugger prototypes were designed and implemented incorporating many information visualization ideas, culminating ultimately into the Visual Debugger 2.0 (VDB2), which combines all of the features of its predecessors, as well as some new features of its own.

1.5 Visual Debugging Prototypes

The first prototype, a Visual Basic construction known simply as the Visual Debugger (VDB), was designed to connect to Microsoft's Visual Studio to provide a simple visual representation of data structures. The second, the Data Structure Visualizer (DSV), was built in Java and is a tool designed specifically to target the problem of visualizing data structures (with or without connection to an underlying debugger). The third tool, a second release of the Visual Debugger (VDB2), was redesigned completely in Visual C# and features enhanced connectivity with Visual Studio .Net and additional information visualization capabilities.

Each tool provides an initial example for the implementation of different information visualization features that can be used to enhance the visual debugging experience.

1.5.1 Visual Debugger (VDB)

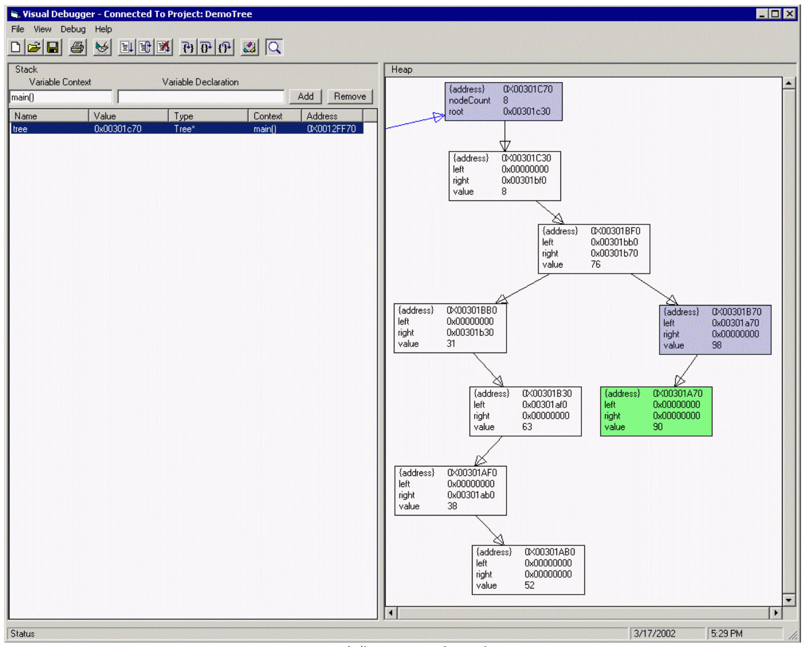
The Visual Debugger project began in 2000 as the joint effort of a pair of undergraduate researchers and a group of graduate students taking a class in information visualization. It had been decided early that the goal of the tool would be to provide visual debugging for the Microsoft Visual Studio framework. This decision proved to be a limiting one, since the API to the Microsoft Visual C++ 6.0 Debugger was lacking in many essential features. Most notably, it was impossible to query the MSVC++ Debugger about the structure of a given class or about the variables on the stack at a given breakpoint.

The workaround was to query not the MSVC++ Debugger but the user about the structure of classes, to query the user about which variables the user was interested in watching. This somewhat cumbersome approach was a necessary tradeoff of maintaining compatibility with Microsoft Visual Studio.

More limitations in the MSVC++ Debugger API required that a small bit of C++ code be included in the target program for the Visual Debugger to track data allocated on the heap. The added code is a simple #include statement, but intrusive to the user nonetheless.

After defining their classes in the Visual Debugger user interface, the user is presented with a real-time, dynamically-updated display of the runtime state of the data structure that the user defined. Figure 1 shows the Visual Debugger being used during the execution of a program that uses a binary search tree. Stack data appears on the left-hand side of the display, and heap data is shown on the right.

By first defining that an object of class "Node" has variables named "left," "right," and "value," the Visual Debugger can keep track of variables of that name and keep them updated as the program progresses. It even uses colors to differentiate between new, modified, and unmodified nodes. Clearly, however, each node uses an enormous amount of screen space, and only a dozen or so nodes can be displayed at once.

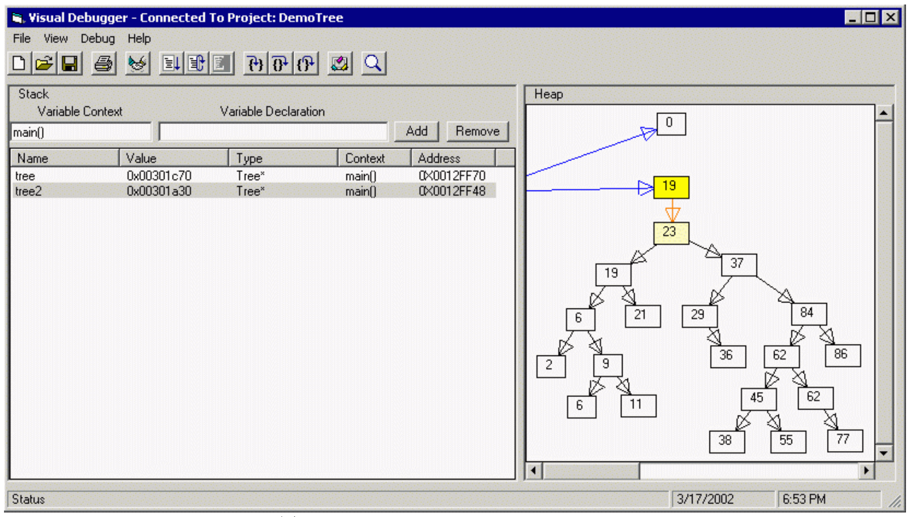


main()

Name	Value	Type	Context	Address
tree	0x00301c70	Tree*	main()	0X0012FF70

```
{address}  0X00301C70
nodeCount  8
root       0x00301c30

{address}  0X00301C30
left       0x00000000
right      0x00301bf0
value      8

{address}  0X00301BF0
left       0x00301bb0
right      0x00301b70
value      76

{address}  0X00301BB0
left       0x00000000
right      0x00301b30
value      31

{address}  0X00301B70
left       0x00301a70
right      0x00000000
value      98

{address}  0X00301B30
left       0x00301af0
right      0x00000000
value      63

{address}  0X00301A70
left       0x00000000
right      0x00000000
value      90

{address}  0X00301AF0
left       0x00000000
right      0x00301ab0
value      38

{address}  0X00301AB0
left       0x00000000
right      0x00000000
value      52
```

Figure 1: The Visual Debugger



main()

Name	Value	Type	Context	Address
tree	0x00301c70	Tree*	main()	0X0012FF70
tree2	0x00301a30	Tree*	main()	0X0012FF48

Figure 2: The Visual Debugger, in a more compact form.

While the Visual Debugger will happily provide every piece of information (including virtual memory address) of every node, VDB offers a second, compact mode that displays only the value of one variable for each node (Figure 2). This allows many more nodes to be viewable at once and still gives important information about the values of nodes.

The "show all information" mode is great when extracting detailed information about every node is important, but there are many cases when only one of the members of a class is of interest to the user. Figure 2 shows a binary search tree, where each node in the tree contains several variables, including pointers to other nodes. In verifying the validity of the binary search tree, the arrows are generally sufficient for verifying appropriate pointer connections, so the only important piece of information in each node is the value of the node. The programmer must make sure that the values are in order, as they should be in a binary search tree. In this case, the programmer can switch to "show one value" mode and does not lose any significant information by hiding the pointer values and can even debug more effectively given the ability to see more nodes at once.

Two major disadvantages of the Visual Debugger are the necessary definition of the user's classes and the speed of execution. Built using Visual Basic with efficiency-killing workarounds due to the suboptimal MSVC++ Debugger, the tool at times takes up to a few seconds to update the display at each breakpoint in the execution, depending on how many nodes are in the data structure. The tool simply cannot scale up.

The last malady of the tool is graph layout. The tool's automatic layout algorithm was primitive and limited to the layout of new nodes. The tool relied on the user's judgment to lay out the graph as he or she desires.

1.5.2 Data Structure Visualizer (DSV)

The Data Structure Visualizer (Figure 3) was conceived on the heels of the release of the Visual Debugger with only one goal in mind: visualization of the data. As such, this tool did not implement connection to any underlying debugger. Its purpose was mainly to experiment with and prototype various methods of visualizing data structures and the interacting with the user. Several improvements were made over the visualization techniques of the original Visual Debugger design.

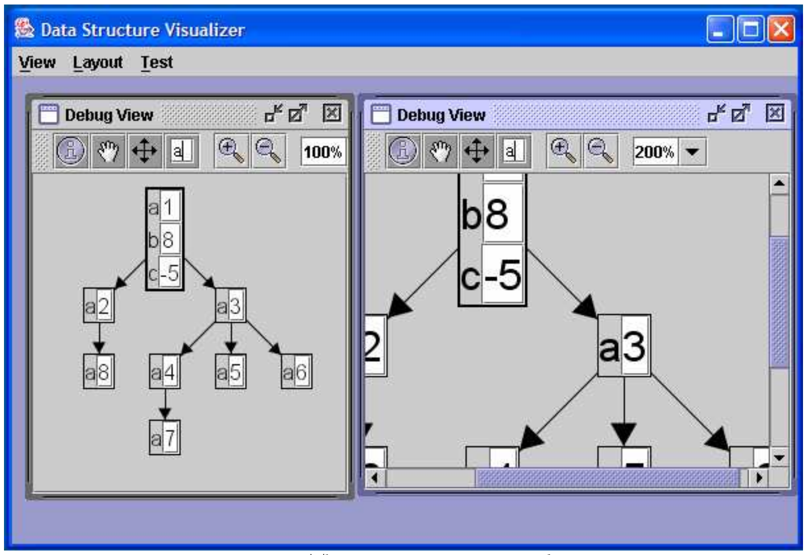


Figure 3: The Data Structure Visualizer

First, the DSV expanded on VDB's hiding of less important class member variables by allowing both modes ("show all information" and "show one value") to coexist simultaneously. Only nodes that the user is particularly interested in need be displayed with all of its member variables. This provides more flexibility to the user about which variables are important during the debugging process.

The second improvement was the added ability to support multiple open windows, each offering a different view of the data structure, so that it could be examined at multiple parts simultaneously. Now, the user can focus in on two particular parts of the data structure, ignoring the rest of the data structure as it becomes less important.

Third, the ability to zoom and pan was added. This is particularly useful for large data structures, as the user can zoom out to get a good general picture of the data structure as a whole and then zoom in on parts that are particularly important to the user.

A framework for more appropriate automatic laying out of the graphs was put in place. Included with the DSV were layout algorithms for linked lists and for general trees, with the added capability of allowing the user to define their own layout techniques via a simple pluggable layout interface.

Finally, the ability to edit values in the data structure was added, potentially allowing users to change parts of their program as it runs to monitor the results of such changes. This is especially useful when a programmer has located a bug in the program and wants to verify, without changing any code or stopping the program, that a certain change in the code will fix the bug.

Of course, the DSV was not designed with an underlying debugger, so these visualizations and interactions cannot immediately be used to debug a program, but the ideas demonstrated in the DSV are ripe for inclusion in a real visual debugger.

1.5.3 Visual Debugger 2.0 (VDB2)

The second release of the Visual Debugger, numbered 2.0, was originally called the "Data Structure Visualizer" (not to be confused with the aforementioned prototype tool), but was renamed as the successor to the original Visual Debugger. It is intended to combine the classic debugging capabilities of the original Visual Debugger with the lessons learned during the development of the Data Structure Visualizer. Powered by a faster programming language, Visual C#, and a more robust debugging platform, the Microsoft Visual Studio .Net Debugger, the major efficiency shortcomings of the old version can be overcome, and the ideas honed in the Data Structure Visualizer can be added.

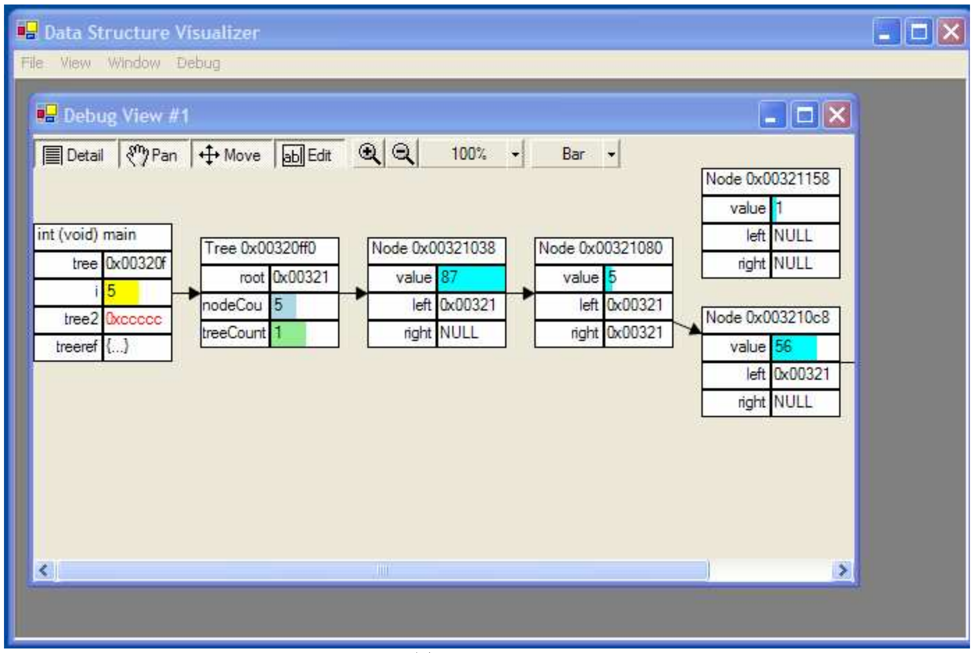


Figure 4: The Visual Debugger 2.0

Aside from the commonalities shared with its predecessors, VDB2 also comes with fresh new ideas about the visualization of data held within the nodes of the data structure. Figure 4 shows VDB2 debugging a binary search tree, where related numerical values are enhanced with bars that vary in size depending on the size of the values. In addition to being able to visualize the values as bars, the user may also choose to visualize them with varying colors.

The Visual Debugger 2 also comes with an improved automatic layout algorithm, a universal layout algorithm intended to appropriately lay out as many data structures as possible. This layout is maintained as the user steps through their code, with emphasis on graph layout stability in an attempt to reduce user confusion between updates in the visualization.

Thanks to the vast improvements in the Visual Studio .Net Debugger over its 6.0 precursor, the execution is orders of magnitude faster, and the user no longer needs to give the tool any information about the structure of their classes. In fact, the tool is now run from *inside* the Visual Studio .Net Integrated Development Environment by way of a Visual Studio Add-In. When the user decides that they want the assistance of the visual debugger, he or she selects a menu item from the Visual Studio menu, and VDB2 starts up immediately with a graphical view of the user's entire program.

C h a p t e r 2

CHAPTER 2: REVIEW OF RELATED WORK

It is widely admitted that one of the most time-consuming parts of programming is finding bugs and fixing them. This realization has led to practically unlimited interest in reducing the possibility of errors and in expediting the process of debugging. Tools such as those included in Microsoft's Visual Studio suite of software development environments are a result of this ongoing effort. One thing that many debuggers lack, however, is visualization.

There are, however, many software projects developed with data structure visualization in mind.

2.1 Instructional Tools

In 1988, Shu [24] described why visual programming should be superior to standard text-based programming, citing in part about ability of pictures to communicate more efficiently than text as well as a picture's ease of understanding and remembrance. This spoke most clearly to the art of computer science instruction, where research turned to using graphical techniques such as "pretty-printing" [2]—a simple reorganization of source code—, automatic program flowchart production [21], and algorithm animation tools [35].

The algorithm animation tools are of particular interest in visual debugging. These tools use animation techniques to demonstrate the execution of an algorithm as it proceeds. While these tools are generally static with respect to underlying program execution and are meant only as instructional tools, they provide a strong basis for appropriate visual techniques for graphical representation of program data structures.

2.1.1 Brown University Algorithm Simulator and Animator

The Brown Univeristy Algorithm Simulator and Animator (BALSA) [4] is a tool with an collection of demonstrations that each attempt to explain the workings of a particular algorithm. Figure 5 shows BALSA's main menu, which is an array of graphical icons, each of which leads the user to a 10-15 minute graphical simulation. Each simulation is built individually to best illustrate and teach the respective algorithm.

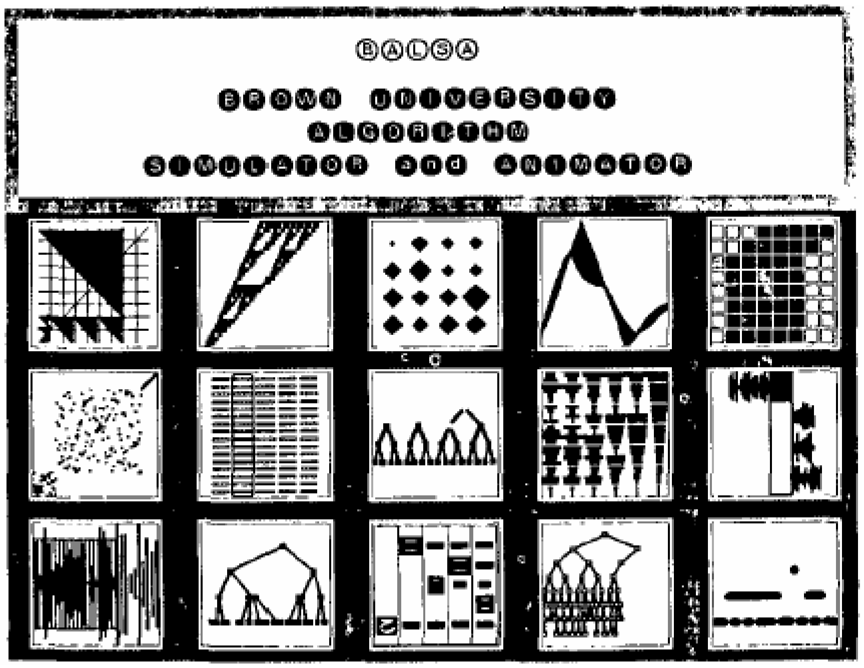


Figure 5: The main menu for BALSA

BALSA performs its purpose well: it shows static simulations for the purpose of teaching. It does this by using domain-specific visualizations to represent the data structures used for each algorithm. This makes each simulation very intuitive and easy to use, but prevents generalization for application to other realms and domains.

BALSA was one of the first algorithm animation tools, and its example has been followed with various spin-offs, including BALSA-II, Tango, and Polka .

2.1.2 XTango, Polka, and Samba

XTango [26], Polka [27], and Samba [32] are all tools developed primarily by Stasko, with the purpose of providing general-purpose algorithm animation. Unlike BALSA, these tools are designed to be used to animate *any* algorithm, not just a predefined set, albeit with some programmer intervention required.

XTango grew out of the Tango project [25]. The programmer uses XTango to visualize his or her algorithm by inserting annotation commands within the code of the algorithm. These commands create and manipulate graphical objects, such as circles, squares, and lines. As the algorithm runs, it changes the properties of the graphical objects, which results in the animation of the algorithm.

Polka is a descendent of XTango, built with more functionality than its predecessor. Its development began targeted at parallel computation, but it has since become useful for visualizing standard serial programs as well. Like XTango, it controls animation through user-inserted annotations. Polka uses a program visualization system called the Parallel Program Animation Development Environment (PARADE). It also uses an object-oriented animation object hierarchy to increase ease of use and flexibility. Programmers can create new animation objects (specific shapes, for example) with particular properties to better craft the animation to his or her application. Figure 6 shows a custom Polka visualization for a sorting algorithm. Each bar represents ten percent of the total number of elements in the list, and each bar is a plot of the minimum, maximum, and average of the elements—the darker a bar is, the more sorted it is.

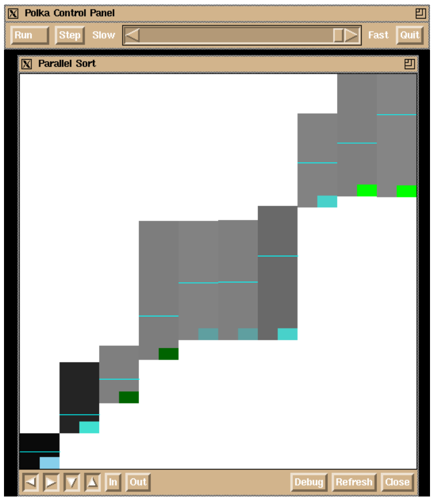


Figure 6: A Polka visualization of a sorting algorithm

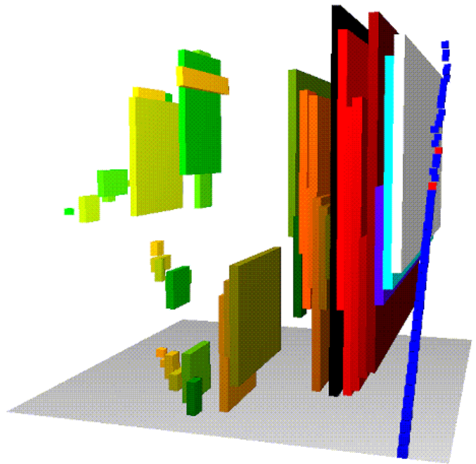


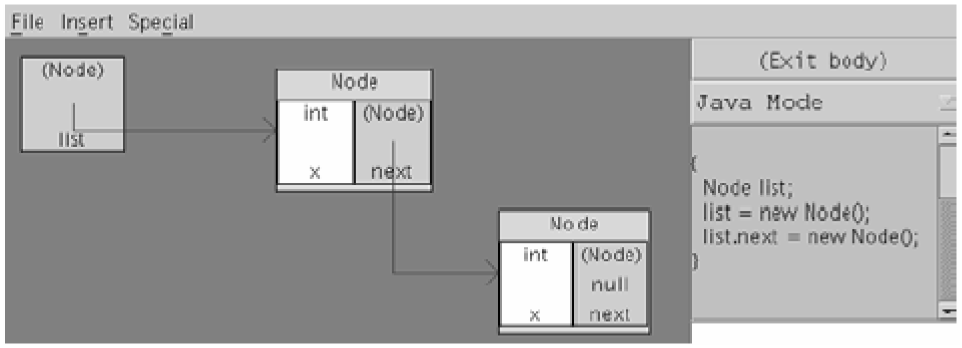
Figure 7: A Polka-3D visualization of the quicksort algorithm

A variation of Polka, Polka-3D [28], uses 3D visualizations instead of 2D visualizations (Figure 7).

Samba is a front-end to Polka that translates input commands (from a file or standard input) into the appropriate Polka commands. This allows Polka to be used by *any* programming language (as opposed to C/C++, the language in which Polka is developed) that can output text to a file.

2.1.3 Language-Independent Visualization Environment

The Language-Independent Visualization Environment (LIVE) [5] uses an entirely different approach to algorithm visualization development. Aiming to provide a tool for professors teaching computer science concepts like demonstrate—structured data, pointers, dynamic memory management, block-structured allocation and deallocation of variables, subprogram flow of control, and recursive subprogram calls—LIVE provides a visual interface for creating relationships between data and automatically generates code to implement those relationships. Implementing a linked list is as easy as a few clicks to create the graphical representations of the nodes, plus a few more clicks to create the relationships. Code to implement the linked list is created automatically.



```
{
  Node list;
  list = new Node();
  list.next = new Node();
}
```

Figure 8: LIVE, creating code for a linked list

LIVE tries to be as language-independent as possible, but it is best-suited for procedural languages like C or Pascal. Also, while some animation is provided by stepping through the code like a debugger, the animation capabilities of the tool are limited.

2.2 Visual Debuggers

In addition to non-debugger tools aimed directly at the instructional use of visualization in teaching data structures, there are many tools designed more broadly as debugging tools. These tools, like the Visual Debugger, intend to support the programmer's debugging process with graphical visualization of certain aspects of the debugged program.

A project called Deet [5] uses a common graph-drawing tool called Dotty [13] to provide visual representations of data structures. VDBX [13] is another Dotty-powered visual debugging project. Swan [20] uses programmer-inserted annotations to build highly flexible data structure visualizations. A larger project called the GNU Visual Debugger has a version that compiles on three different platforms. By far, however, the Data Display Debugger (better known as DDD) [37] is the most well-known and largest of the visual debuggers, and has been under active development since 1993.

2.2.1 Dotty

Dotty [13] is a general-purpose graph utility. It can be used to edit and draw general directed or undirected graphs. Though it can run standalone as a graph editor, it is more important to the art of visual debugging as a front-end tool for visual debugging tools such as Deet and VDBX.

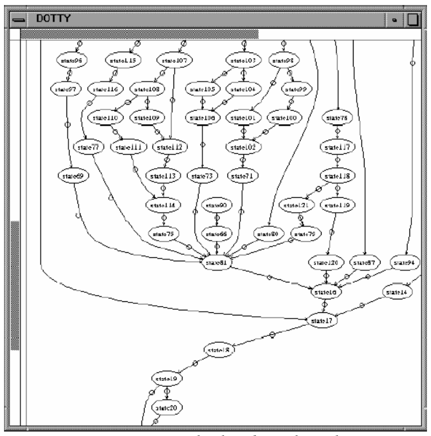


Figure 9: A graph displayed with Dot
(taken, unedited, from [13])

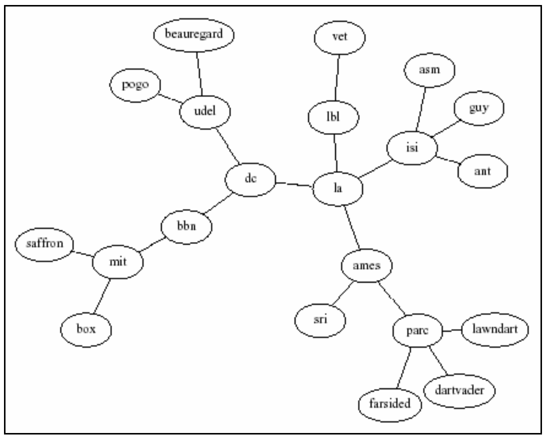


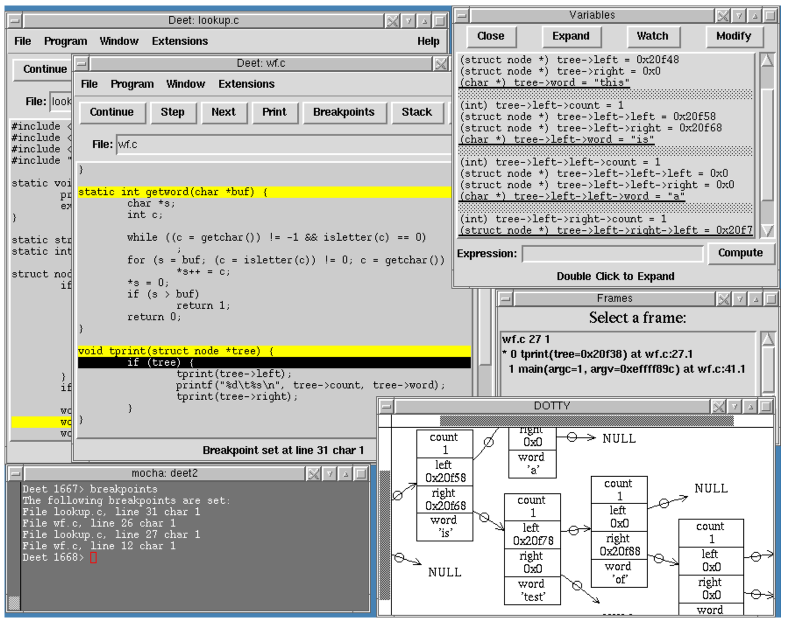
Figure 10: A graph displayed with Neato
(taken, unedited, from [13])

Dotty is partitioned into two sections: a programmable viewer called Lefty; and graph layout generators, implemented with Dot (Figure 9) and Neato (Figure 10). Dot generates layouts for directed graphs and is based on a graph layout technique known as the layering method [20]. Neato generates layouts for undirected graphs and uses a different approach called physical modeling [20].

Together, Dotty provides advanced capabilities to programmers that would otherwise be forced to implement them manually. By using Dotty, visual debugging software need not worry about proper display of the graphs they construct.

2.2.2 Deet

Deet [5] stands for Desktop Error Elimination Tool. It attempts to address the shortcomings of large, complex debuggers, so it is a very small shell program that does a surprisingly large amount of work in only 1500 lines of shell code. Written in tksh, a variant of the Korn shell scripting language, it is machine-independent, but it relies on the concept of a "nub," a small set of machine-dependent functions that must be embedded in the debugged program to allow Deet to collect the necessary data from the program that is being debugged. Thus, Deet requires a slight bit of extra effort on the part of the programmer to provide him or her with debugging capabilities.



```
#include <
#include <
#include <
#include "

static voi
        pr
        ex
}

static str
static int

struct noc
        if



        }
        if

        wc
        wc
        wc
```

```
}

static int getword(char *buf) {
        char *s;
        int c;

        while ((c = getchar()) != -1 && isletter(c) == 0)
                ;
        for (s = buf; (c = isletter(c)) != 0; c = getchar())
                *s++ = c;
        *s = 0;
        if (s > buf)
                return 1;
        return 0;
}

void tprint(struct node *tree) {
        if (tree) {
                tprint(tree->left);
                printf("%d\t%s\n", tree->count, tree->word);
                tprint(tree->right);
        }
}
```

```
(struct node *) tree->left = 0x20f48
(struct node *) tree->right = 0x0
(char *) tree->word = "this"

(int) tree->left->count = 1
(struct node *) tree->left->left = 0x20f58
(struct node *) tree->left->right = 0x20f68
(char *) tree->left->word = "is"

(int) tree->left->left->count = 1
(struct node *) tree->left->left->left = 0x0
(struct node *) tree->left->left->right = 0x0
(char *) tree->left->left->word = "a"

(int) tree->left->right->count = 1
(struct node *) tree->left->right->left = 0x20f7
```

```
wf.c 27 1
* 0 tprint(tree=0x20f38) at wf.c:27.1
  1 main(argc=1, argv=0xeffff89c) at wf.c:41.1
```

```
Deet 1667> breakpoints
The following breakpoints are set:
File lookup.c, line 31 char 1
File wf.c, line 26 char 1
File lookup.c, line 27 char 1
File wf.c, line 12 char 1
Deet 1668>
```

Figure 11: Deet
(taken, unedited, from [5])

Figure 11 shows a sample debugging session that uses Deet. The graphical representation of a data structure appears in the lower right corner, in a window labeled "Dotty."

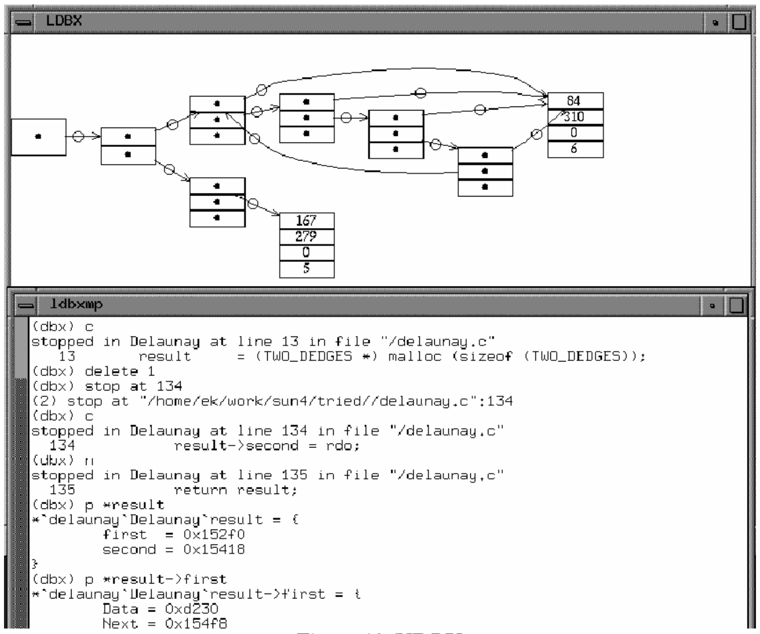
Features of Deet include capability for browsing source files, setting breakpoints and watch variables, and visualizing data structures via Dotty, a separate program that relieves Deet of the complex responsibility of drawing graphs. Deet's creators found problems in larger debuggers such as GDB, which is about 150,000 lines of C code. Large programs are inherently buggy themselves, and Deet's conciseness attempts to avoid that and other problems that go along with size and complexity. Large debuggers are also more difficult to extend, and one of Deet's most attractive features is its extensibility. Because it is so small, it is easier to understand, making it easier to modify and extend.

2.2.3 VDBX

VDBX [13] is another small visual debugging project that delegates to Dotty the complex task of drawing graphs in an aesthetically pleasing organization. Unlike Deet, however, VDBX is not intrusive in that it adds nothing to the program being debugged. Instead, it uses DBX, an existing text-based debugging tool, to gather the required information from the debugged program.

Figure 12 shows VDBX in action. A visual representation of a data structure (generated by Dotty) dominates the upper half of the screen.

Because it uses an external debugger to provide debugging capabilities, it is tied to that debugger in both positive and negative ways. It inherits all the features of DBX, giving it more capabilities than Deet's smaller nub, but with it comes the limitations of DBX. Developers that wish to use VDBX must first be familiar with DBX to use its features. VDBX would be more user-friendly if the developer could use his or her own favorite debugger. Because the developers of VDBX saved themselves the effort of designing their own debugger, however, they were able to focus more on the visualization of data structures, centering more on the use of Dotty to display data structures graphically, contrary to Deet, where Dotty plays a role less significant than Deet's other features.



```
(dbx) c
stopped in Delaunay at line 13 in file "/delaunay.c"
   13       result     = (TWO_DEDGES *) malloc (sizeof (TWO_DEDGES));
(dbx) delete 1
(dbx) stop at 134
(2) stop at "/home/ek/work/sun4/tried//delaunay.c":134
(dbx) c
stopped in Delaunay at line 134 in file "/delaunay.c"
  134           result->second = rdo;
(dbx) n
stopped in Delaunay at line 135 in file "/delaunay.c"
  135           return result;
(dbx) p *result
*`delaunay`Delaunay`result = {
        first  = 0x152f0
        second = 0x15418
}
(dbx) p *result->first
*`delaunay`Delaunay`result->first = {
        Data = 0xd230
        Next = 0x154f8
```

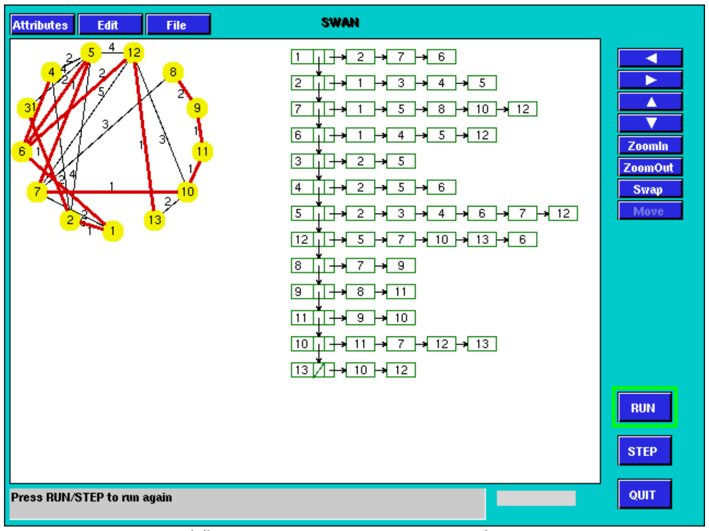
Figure 12: VDBX
(taken, unedited, from [13])

2.2.4 *Swan*

Swan [23] takes a cue from the algorithm animation tools by forming its visualizations through user-inserted annotations to develop an informative picture of the data in a user's program (Figure 13). In much the same way that programmers might use print statements to output the values of certain runtime variables, the Swan user instead uses the Swan Annotation Interface Library to visualize parts or all of a data structure.

While traditional algorithm animation tools generally require a time-consuming process to develop their application-specific visualizations, Swan aims for the debugging realm by attempting to require as little effort as possible. Swan includes various automatic layout techniques, including the general circular method and specific list and tree layouts, but also gives the user the flexibility of defining their own layout for a particular program.

In addition to visualizing data structures, Swan can also step through code to provide a dynamic view of how the program progresses. This is especially useful for debugging, where visualizing the results of particular lines of code can be helpful in locating and fixing bugs.



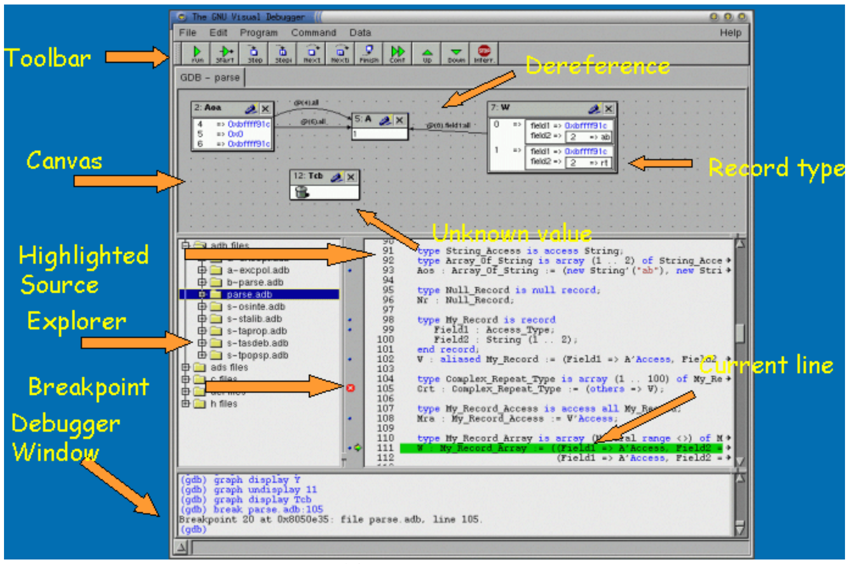
Press RUN/STEP to run again

Figure 13: The Swan Data Structure Visualization System
(taken, unedited, from [20])

2.2.5 GNU Visual Debugger

The GNU Visual Debugger (GVD) [38] is an ongoing visual debugging software project, developed by ACT-Europe. It is robust, implementing many useful debugging features. Like VDBX, it uses a text-based debugger to carry out the actual debugging necessities, and it uses the output to display it on-screen in a user-friendly format. Unlike VDBX, which restricts its users to DBX or debuggers that use the DBX syntax for C structs, it can be configured to use nearly any debugger that the programmer wishes to use. It has native support for GDB, VxWorks, LynxOS, JVM, and others, but if a favorite debugger is not supported natively, GVD can easily be extended to use that particular debugger. GVD even supports using multiple debuggers simultaneously.

Figure 14 shows the main window of the GNU Visual Debugger. The "Canvas" at the top of the screen is of most concern. It displays data nodes as boxes that lists all the fields of the data node, and arrows connect nodes that are related to each other via pointers.



```
type String_Access is access String;
type Array_Of_String is array (1 .. 2) of String_Acce
Aos : Array_Of_String := (new String'("ab"), new Stri

type Null_Record is null record;
Nr : Null_Record;

type My_Record is record
   Field1 : Access_Type;
   Field2 : String (1 .. 2);
end record;
V : aliased My_Record := (Field1 => A'Access, Field2

type Complex_Repeat_Type is array (1 .. 100) of My_Re
Crt : Complex_Repeat_Type := (others => V);

type My_Record_Access is access all My_Record;
Mra : My_Record_Access := V'Access;

type My_Record_Array is array (Natural range <>) of M
W : My_Record_Array := ((Field1 => A'Access, Field2 =
                        (Field1 => A'Access, Field2 =
```

```
(gdb) graph display Y
(gdb) graph undisplay 11
(gdb) graph display Tcb
(gdb) break parse.adb:105
Breakpoint 20 at 0x8050e35: file parse.adb, line 105.
(gdb)
```

Figure 14: The GNU Visual Debugger
(taken, unedited, from [38])

In addition to flexibility with debuggers, it is written in Ada using a highly platform-independent GUI toolkit called GtkAda, so it can be compiled on virtually any modern operating system, including Microsoft Windows, GNU/Linux, and Solaris. As long as the system has a text-based debugger that GVD can use, then the system can use it as a powerful visual debugger.

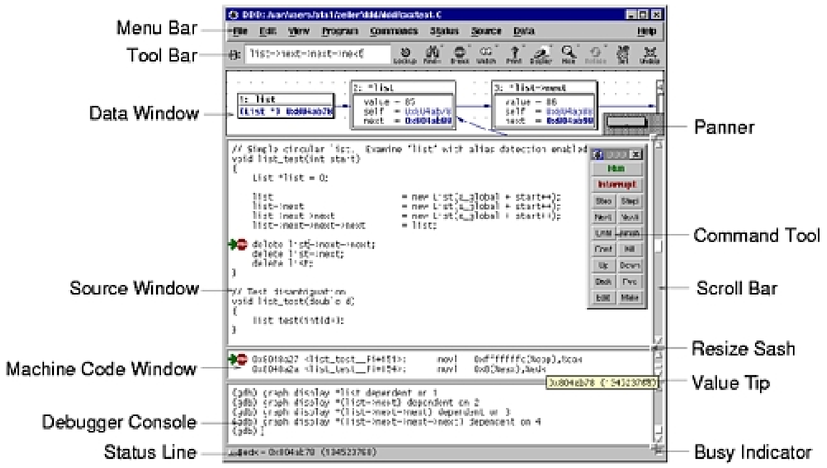
Among its numerous features, it sports the capability to graphically depict data structures, though this feature is downplayed somewhat in comparison to its high flexibility with respect to debuggers and platform-independence. Still, its visualization of data structures is sufficient enough for simple debugging purposes.

2.2.6 Data Display Debugger

The Data Display Debugger (DDD) [37] is a long-established visual debugging program that serves as a graphical front end to the GNU Debugger (GDB). It was created as a free alternative to commercial debuggers that were bundled with proprietary integrated development environments and compilers.

The project started in 1993 as a simple front end to GDB. It inherited GDB's advanced debugging capabilities and added graphical representation of data structures, much in the same way that GVD does, with boxes representing nodes and arrows representing pointers. Since then, it has evolved to include hundreds of new features, including support for many more command-line debuggers, including DBX, WDB, JDB, XDB, the Perl debugger, and the Python debugger. In addition to standard graph representations of data, it also includes support for multidimensional plots of data, a novel approach to visual debugging.

Figure 15 shows a Data Display Debugger session. Like GVD, it's "Data Window" is at the top of the window. DDD takes the graph one step further, however, by providing a method of laying out a Data Window. Figure 16 shows a type of tree data structure after applying one of DDD's automatic layout algorithms.



```
// Simple circular list.  Examine "list" with alias detection enabled
void list_test(int start)
{
    List *list = 0;

    list                        = new List(a_global + start++);
    list->next                  = new List(a_global + start++);
    list->next->next            = new List(a_global + start++);
    list->next->next->next      = list;

    delete list->next->next;
    delete list->next;
    delete list;
}

// Test disambiguation
void list_test(double d)
{
    list_test(int(d));
}
```

```
0x0804a27 <list_test__Fi+51>:      movl    0xfffffffc(%ebp),%eax
0x0804a2a <list_test__Fi+54>:      movl    0x8(%eax),%edx
```

```
(gdb) graph display *list dependent on 1
(gdb) graph display *(list->next) dependent on 2
(gdb) graph display *(list->next->next) dependent on 3
(gdb) graph display *(list->next->next->next) dependent on 4
(gdb) |
```

Figure 15: The Data Display Debugger
(taken, unedited, from [36])

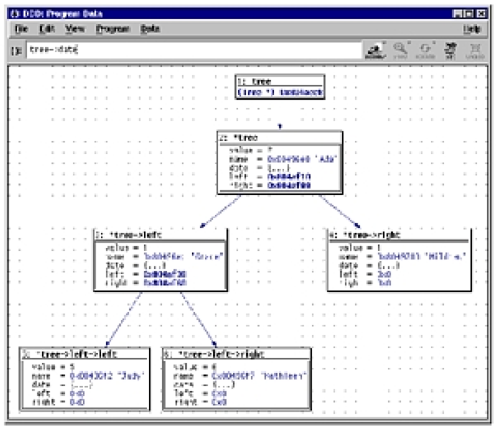
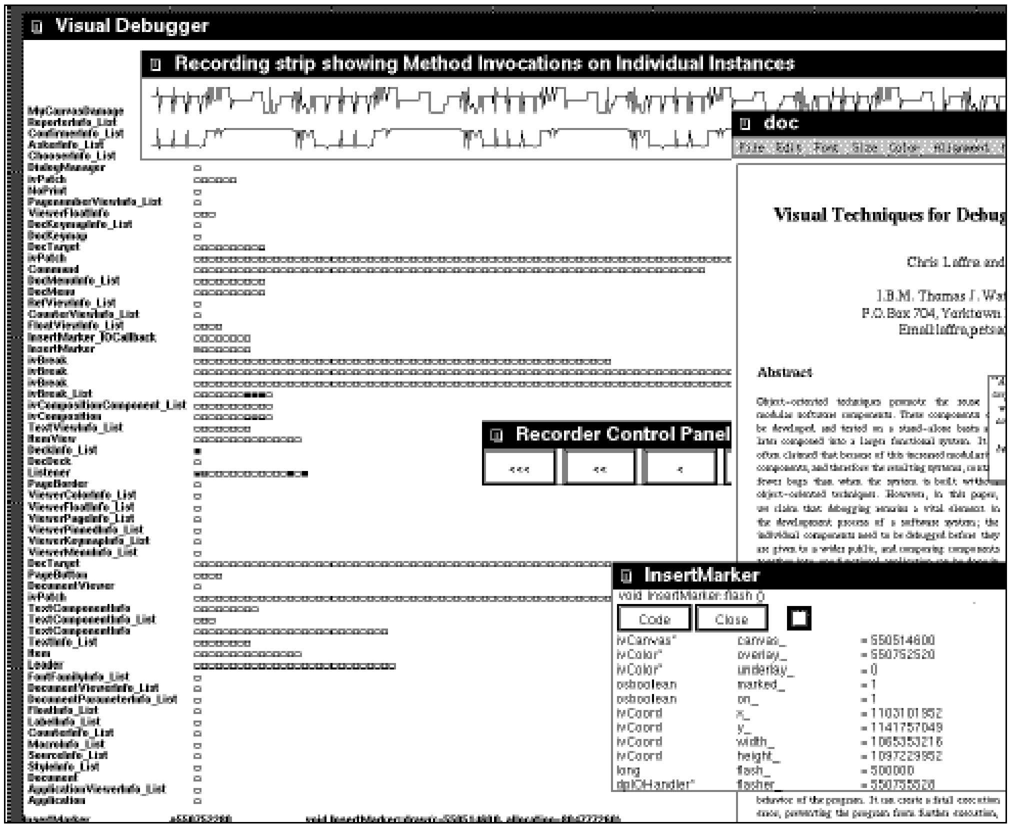


Figure 16: A tree, using DDD's automatic layout algorithm
(taken, unedited, from [36])

2.2.7 *HotWire*

HotWire [14] uses a more information-visualization-oriented approach to visual debugging, designed for debugging object-oriented Smalltalk and C++ code. Instead of creating a graph of the user's data structure, it uses more abstract forms of visualizations to present a unique overview of the system. Figure 2, for example, illustrates two of the programmed visualization methods. The boxes represent instance-tracking, where each box represents one instances of a particular class. A class that has too many boxes (instances) may indicate a memory leak. The other visualization technique is a message invocation graph. This provides the user with a path of execution as methods are called on certain objects from other objects.


Visual Debugger
Recording strip showing Method Invocations on Individual Instances
doc
Visual Techniques for Debug
Abstract
Recorder Control Panel
InsertMarker
Code
Close

```
void InsertMarker::flash ()
ivCanvas*     canvas_     = 550514600
ivColor*      overlay_    = 550752520
ivColor*      underlay_   = 0
osboolean     marked_     = 1
osboolean     on_         = 1
ivCoord       x_          = 1103101952
ivCoord       y_          = 1141757048
ivCoord       width_      = 1065353216
ivCoord       height_     = 1097229952
long          flash_      = 500000
dpIOHandler*  flasher_    = 550755520
```

Figure 17: Hotwire, debugging a word processor.

2.2.8 *Groove and Execution Murals*

Groove [29] uses the properties of the object-orientedness of C++ code to extract a class hierarchy, which it visualizes using various shapes and colors. Its intention is to monitor, trace, debug, and design C++ code. It displays the class hierarchy as a tree structure, with instances listed inside circles. Although its visualization is a bit non-intuitive, cumbersome, and space-consuming, Groove demonstrates that it is possible to use visual debuggers to debug object-oriented code.

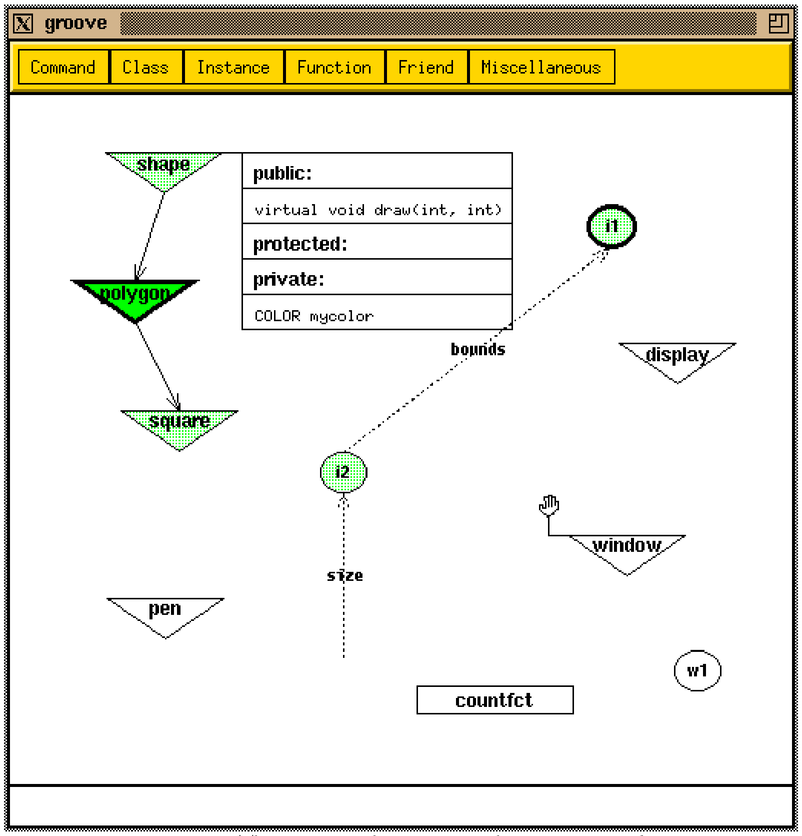


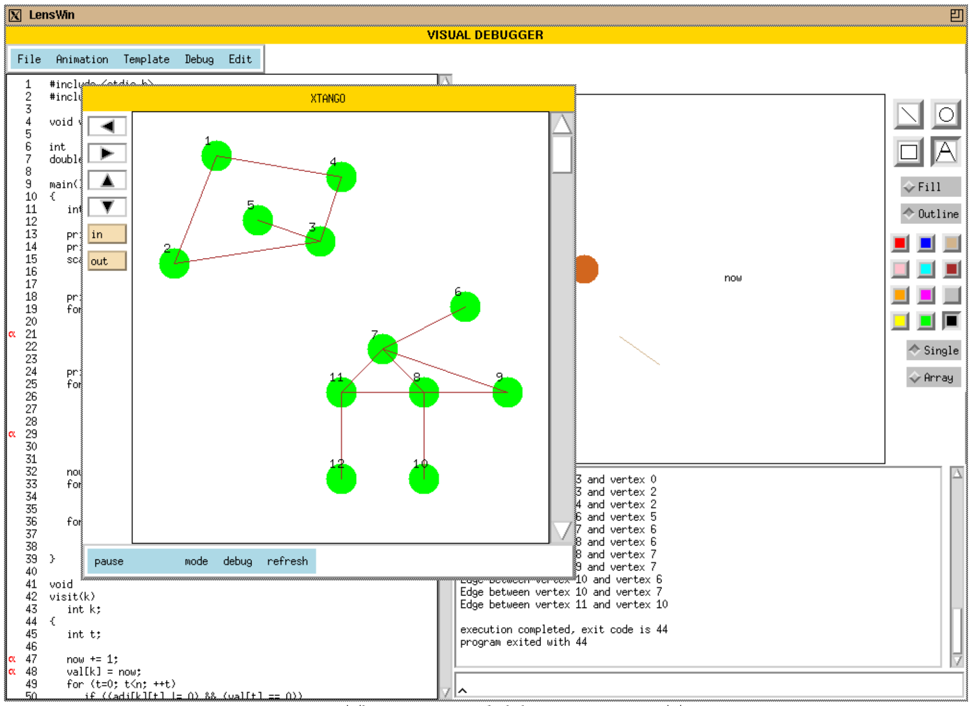
Figure 18: The Groove object-oriented program visualizer

The same research that resulted in Groove led to the application of information murals [31] to the visual debugging of object-oriented programs [33]. This technique, called execution murals, uses anti-aliasing to display a massive number of message-passing events on a screen at a single time, allowing the user to debug the message-passing behavior of an object-oriented program.

2.2.9 Lens

Lens [30] is another annotation-based debugger, like Swan, but Lens approaches the annotations in a completely different way: graphically. Instead of editing their code by inserting annotation commands, the user instead uses a graphical direct manipulation interface to create the annotations. All of the annotations are inserted via point-and-click interaction, requiring keyboard only to specify names of appropriate program variables.

To accomplish such a task, Lens implements only more commonly used animation techniques, such as moving, flashing, and filling. Since these tasks are already well handled by an algorithm animation tool, XTango, Lens uses XTango as to handle the graphical animations. Figure 19 shows a screenshot of Lens in action, using XTango to display a depth-first-search algorithm.



```
#include <stdio.h>
#incl

void

int
double

main(
{
   int

   pr
   pr
   sc


   pr
   for




   pr
   for






   no
   for


   for


}

void
visit(k)
   int k;
{
   int t;

   now += 1;
   val[k] = now;
   for (t=0; t<n; ++t)
      if ((adj[k][t] != 0) && (val[t] == 0))
```

```
3 and vertex 0
3 and vertex 2
4 and vertex 2
6 and vertex 5
7 and vertex 6
8 and vertex 6
8 and vertex 7
9 and vertex 7
Edge between vertex 10 and vertex 6
Edge between vertex 10 and vertex 7
Edge between vertex 11 and vertex 10

execution completed, exit code is 44
program exited with 44
```

Figure 19: The Lens visual debugger, using XTango

In addition to the direct manipulation interface, Lens connects directly to dbx, allowing any program that has been annotated with dbx's debugging symbols to be debugged. This makes it a true visual debugger, instead of just an algorithm animation tool, like XTango.

2.3 Summary

In general, data structure visualization can be divided into two basic groups. The first focuses on instruction, providing very appropriate visualizations of specific types of data structures. The narrow focus of these types of tools allows them to produce highly understandable visual representations of data structures that otherwise might be difficult to decipher, especially to programmers new to those types of data structures. Their fault lies in their lack of generality. They cannot be applied practically to working, real-world problems. Those that try (i.e. Polka, XTango, and Lens) introduce a complex language or process that the programmer must use to define their specific visualization, which has the adverse effect of requiring more time and energy.

The other group of data structure visualization tools attempts to provide one or more general methods of viewing live data structures or the programs that uses it. This allows programmers that are debugging real-world programs to gain visual insight about their code that goes beyond the standard data that a typical text-based debugger provides. The hope is that the programmer can use this additional information to better understand and better debug the system that the programmer is developing.

What all of these visual debuggers is missing, however, is a sound information visualization basis. While most provide some use of color, and HotWire even goes so far as to provide visual overviews of programs, they lack information visualization ideas such as Focus+Context that could be used to better communicate the details of a data structure, and they lack information visualization ideas even so simple as zooming or murals to better communicate the overview of a data structure.

Information visualization can be used to improve the quality and effectiveness of the images produced by a visual debugger by making them more understandable and more revealing. Appropriate application of information visualization can also increase the amount of information displayed at once, allowing the programmer to more easily scan a data structure to identify correctness or anomalies during runtime, ultimately resulting in shorter debugging sessions and less maintenance time overall.

C h a p t e r 3

CHAPTER 3: INFORMATION VISUALIZATION FOR VISUAL DEBUGGING

The most important part of applying information visualization to visual debugging lies in user interface design. There are many information visualization concepts that could easily be applied to the domain of visual debugging, but surprisingly have remained absent. Several of these have been implemented in at least one of the three visual debugging prototypes (VDB, DSV, or VDB2). The ideal visual debugger, hopefully to be materialized in VDB2 as development continues, would include as many useful information visualization concepts as possible.

Described herein are seven different information visualization concepts that can be applied to a visual debugger, such as those illustrated in the previous chapter. None of these information visualization concepts are novel—several have been available for many years—but they are not well applied to existing visual debuggers. This is surprising, considering their potential in the visual debugging domain as well as their relative ease of implementation. Including one or all of these techniques in a visual debugging tool can improve a programmer's ability to use that tool to effectively debug programs that are highly oriented toward data structures.

3.1 Graphical Value Representation

Suppose a user, Gary, is developing code for a binary search tree to store integers. Gary has just put what he thinks are the finishing touches on his binary search tree, and he is beginning rudimentary testing to assure its correctness. He steps through the code using his favorite visual debugger, visually tracking the growth and changes of his data structure as his test program progresses. At each step, he verifies that the orderedness of the binary search tree remains intact. The repetitive nature of this task makes it prone to error, and Gary fails to notice a certain error in the tree that is introduced due to a failure in his code's logic. A node was erroneously swapped with another, and the orderedness of the tree was destroyed. Due to the subtle nature of the change (after all, a 17 can look very similar to a 77), the error remains unnoticed until a more automated verification and validation phase randomly recreates and identifies the faulty state in the binary search tree.

The term "information visualization" begs the possibility of visualizing not just the structure of data structures but the information itself held therein. Tools like Table Lens [20] have tried various ways to try to visualize numbers (Figure 20), including using bars and using colors.

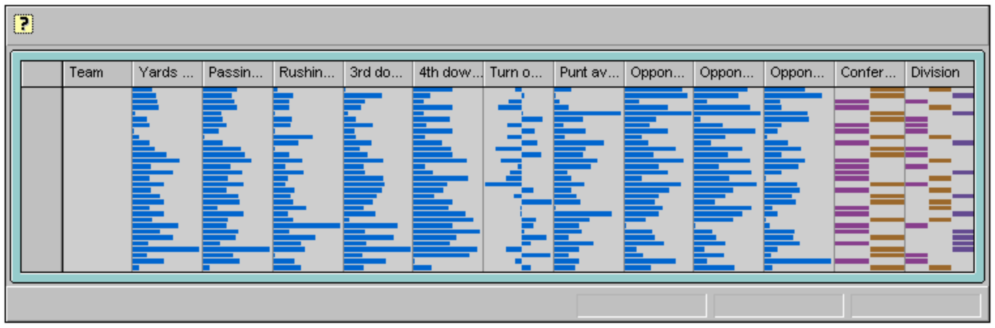
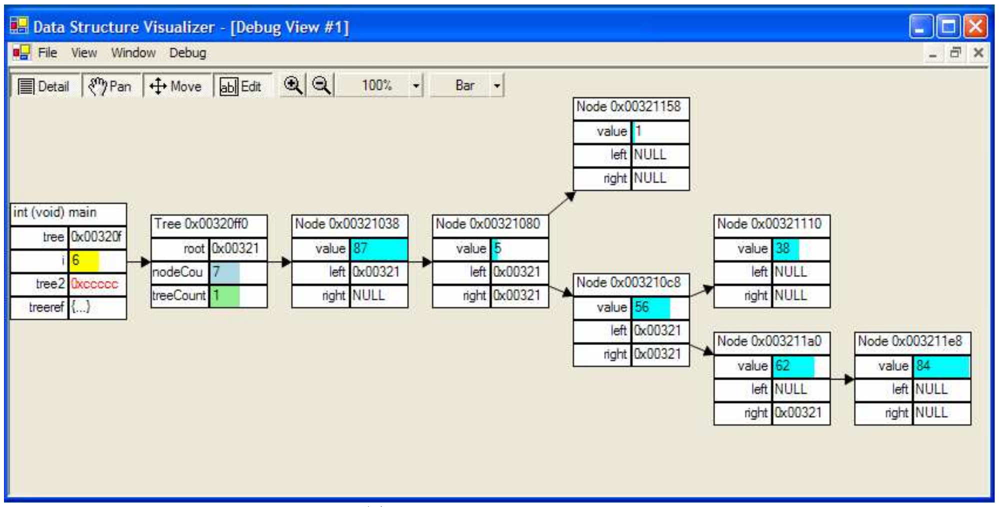


Figure 20: Table Lens

The Visual Debugger 2 employs both bars and colors in an effort to demonstrate the use of such graphical representation of numerical values. Figure 21 shows the Visual Debugger 2.0 in "bar" visualization mode, analogous to the bars in Table Lens, in which each numerical variable is displayed with the textual value framed by a bar of a size that varies with respect to the value of the variable. Larger values are given large bars, and smaller values are given smaller bars. This allows users to more easily point out larger and smaller values, which becomes especially important, for example, in sorted data structures, where a large value would stand out in a crowd of smaller values much more easily with the bars. The large bar would dominate over the smaller bars, allowing the user to more quickly locate the error in the sorted data structure, and faster location of errors means faster resolution of errors.



int (void) main

tree	0x00320f
i	6
tree2	0xcccccc
treeref	{...}

Tree 0x00320ff0

root	0x00321
nodeCou	7
treeCount	1

Node 0x00321038

value	87
left	0x00321
right	NULL

Node 0x00321080

value	5
left	0x00321
right	0x00321

Node 0x00321158

value	1
left	NULL
right	NULL

Node 0x003210c8

value	56
left	0x00321
right	0x00321

Node 0x00321110

value	38
left	NULL
right	NULL

Node 0x003211a0

value	62
left	NULL
right	0x00321

Node 0x003211e8

value	84
left	NULL
right	NULL

Figure 21: The "bar" visualization mode of VDB2

In the example given in Figure 21, unrelated variables are given bars of different colors to help prevent the user from comparing variables that should not be compared. Due to the difference in colors, it is quite apparent that the `nodeCount` variable (shown as `nodeCou` in VDB2, which truncates long variable names to reduce unnecessary use of space) of the `Tree` class is not related to the `value` variable of the `Node` class. As a result, variables of the same type, such as the `value` variable of the `Node` class, are easier to compare with each other.

VDB2 also implements a "color" mode. Figure 22 shows the same graph as in Figure 21, but in "color" mode instead of "bar" mode. Instead of widths of bars, the user now uses the darkness of equally sized bars as the cue for magnitude of the variable's value. Larger values are given darker shades. Again, this helps the user to compare relativity between variables.

Similar to "bar" mode, related variables are given different intensities of the same hue, so that mistakes are not made in comparing unrelated variables.

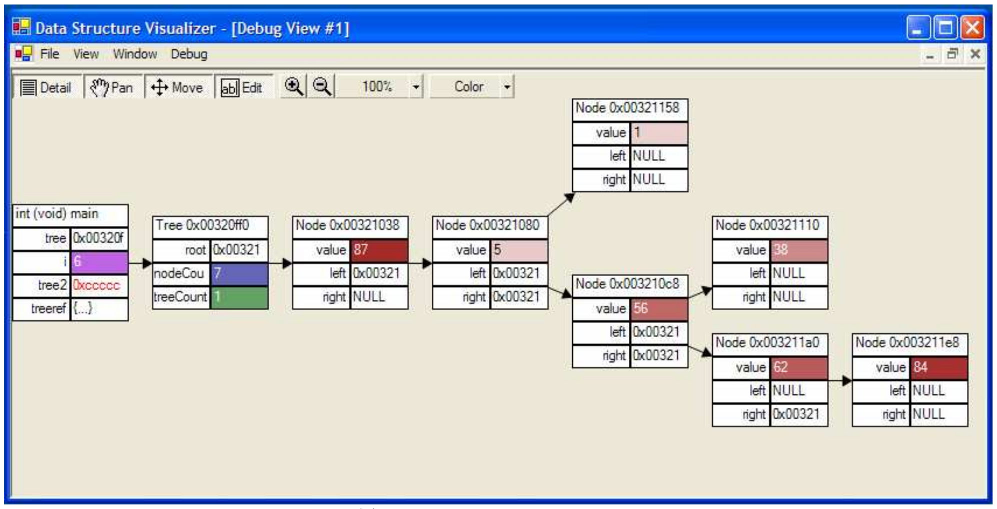


Figure 22: The "color" visualization mode of VDB2

Graphical value representation provides a quick and intuitive way of obtaining relatively accurate values without having to read any text. It makes it easy to identify outliers and can even be used at a resolution much lower than that of text. The obvious downside is the loss in precision. The exact determination of values along a continuous scale is practically impossible, forcing the user to rely on the text values for precision. Variables that are close in value may appear to be the same according to the bars and colors, which is arguably both a good and a bad side effect. While a dark color may stand out easily in a crowd of light colors to signal a significant error in an ordered linked list, it may be less helpful in locating errors with less significant results, i.e. if two close values are flipped in an ordered list.

If Gary had had the Visual Debugger, he would have better been able to notice the subtle error in his binary search tree. A 17 may look similar to a 77 in text, but in magnitude they are very different, and bars or colors add a dimension in addition to the text that Gary could have used to clearly differentiate between a 17 and a 77. By locating the problem earlier, while the design and code of the binary search tree is still fresh in his mind, Gary potentially saves valuable time and money that otherwise would later have been spent finding and fixing the bug.

3.2 Zoom and Pan

Now, suppose Gary is debugging a very large graph structure. As he looks at the seemingly random array of boxes and arrows, he becomes lost about what part of the data structure he is looking at. He navigates around the visualization with the scrollbars, but no matter where he is in the data structure, he cannot figure out which part of the data structure visualization is being operated on by the code he is debugging. Frustrated, he searches for the root of the data structure and retraces his previous steps back to where he thought he should be, losing valuable time in the process.

Zoom and Pan is a way to visualize information that is simply too large to put logically on a single display. Take, for example, a large text document. To read it, it is not necessary to look at the entire document. Instead, only a small part of the document is read at a time. Zoomed in to a particular part of the document, one section can be read, and as the visible extent of your display is reached, panning or scrolling reveals the next part. With most computer-based word processors, it is possible to zoom in even further, in case the user's vision is impaired or the text is small.

Zoom and Pan is also widely used in the graphical editing domain. Especially with images that are larger than the resolution of the display, it is advantageous to be able to zoom in on certain parts of the image and pan around.

Tools like Pad++ [3] have demonstrated that zooming and panning can be applied to the workspace metaphor to increase the potential capacity that the typical monitor has for displaying information. Theoretically, the scale of the workspace can be expanded infinitely, allowing a limitless amount of information to be displayed on the screen at one time, albeit in increasingly less clarity. As more graphical objects are added to the workspace, the user must zoom out to see them all, which reduces the size of each object as it is displayed on the screen, which in turn—due to pixilation or other form of size reduction—may reduce the amount of detail that can be gleaned from the graphical object by the user. The user may choose to zoom on particular objects or groups of objects, which hides other objects but illuminates smaller details of the zoomed objects.

This simple information visualization concept can easily be applied to the visual graphs used to represent data structures. The Data Structure Visualizer uses zooming to allow the user to see as much (Figure 23) or as little (Figure 24) of the graph as they wish. They can view the entire data structure, sacrificing detail if the text is not large enough to be readable, or they can zoom in on particular parts of the data structure in order to take a closer look. Giving the user the freedom of zooming and panning can aide in the debugging process, especially when large data structures are being debugged.

When zooming out to a large factor in an attempt to display as much of the graph as possible, it may be beneficial to apply information mural techniques [31] such as anti-aliasing to increase the amount of information portrayed in the display.

By extending zoom and pan to the visual debugging arena, the programmer can now have access not to just a slice of the program, but the whole picture of all of the variables in a program. The limits of screen space only delegate how much detail the user can get without losing part of the picture.

Gary, our once-frustrated friend, no longer has to search aimlessly for location and direction in his huge data structure. Instead, he can zoom out, get a feel for where he is with respect to the entire data structure, and then zoom back in to where he had been, with a newfound sense of where he had been and what he had been doing.

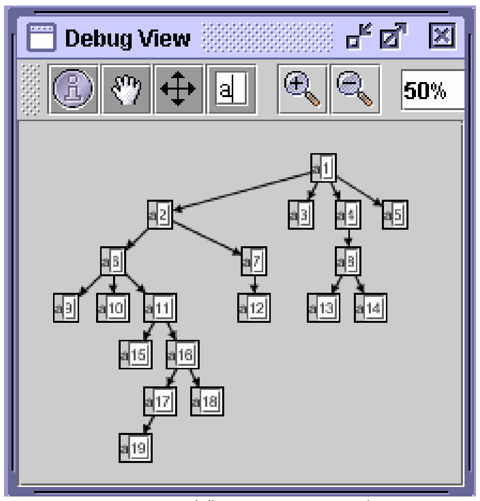


Figure 23: The DSV, zoomed out

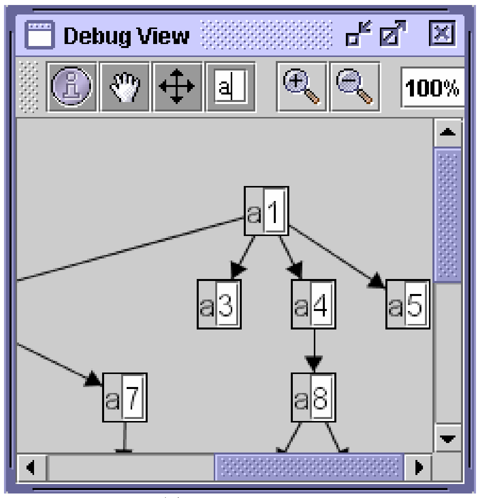


Figure 24: The same graph, zoomed in

3.3 Overview+Detail

What if Gary did not have to leave his place in the large data structure to see where he was? What if he could just glance at a separate display that already contained a zoomed out view of the entire data structure?

Overview+Detail [8] means providing the user with one display that acts as a general overview of the object being viewed, while providing a separate display that acts as a more detailed window into the object. For example, graphic artists may use two windows when editing a particular image: the overview, which is a small window containing a scaled-down version of the entire image, and the detail in a larger window, zoomed in on a particular section of the image. The overview window can even include information about the detail window, such as its location.

Zooming can be helpful in implementing an Overview+Detail display. For example, the Data Structure Visualizer uses zooming and its multiple-display feature to create one zoomed-out display as the overview, and a zoomed-in display as the detail. Figure 25 shows the Data Structure Visualizer in Overview+Detail mode. Two visualization displays are open; the smaller, zoomed-out display on the right provides an overview for the larger, detailed display on the left. The overview is too small to allow examination of particular node values, but instead provides a rough idea of the general structure of the entire data structure.

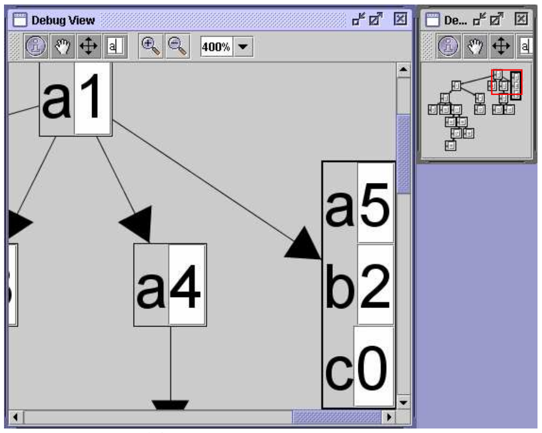
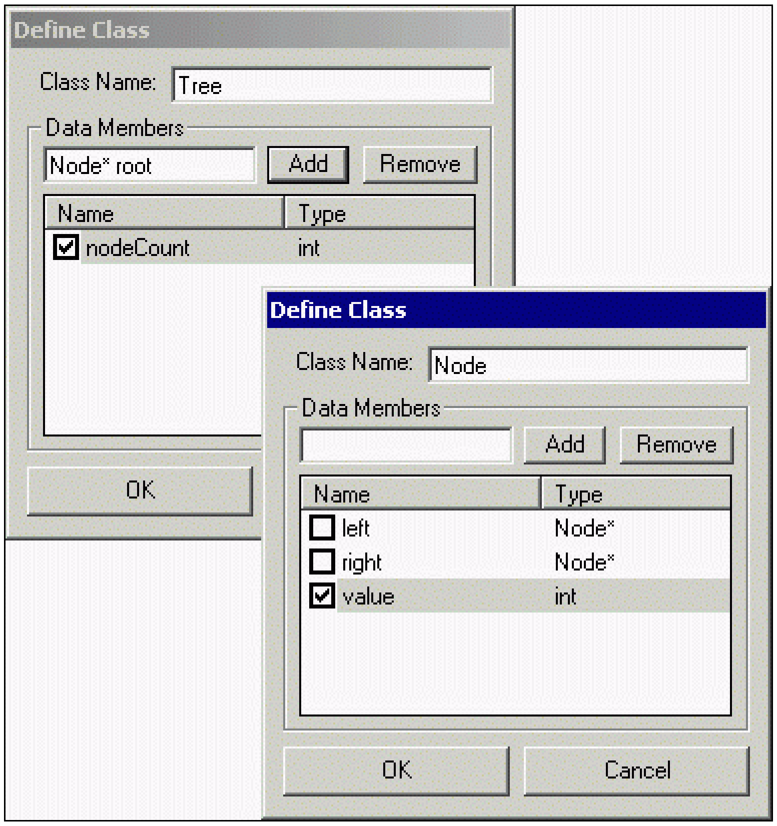


Figure 25: The DSV, in Overview+Detail mode

The original Visual Debugger implements two levels of overview. First, it requires the user to manually define the C++ classes used in the software project (Figure 26). While a bit cumbersome to the user, it also gives the user the flexibility to specify exactly which class members are displayed and which are always hidden. If a particular class member is not important to the user, he or she simply does not define it in VDB interface. A programmer could potentially specify only a single member variable in a certain class, providing an "overview" of instances of that class. This overview cannot be escaped, however, without modifying the user definition of the class.



Class Name: Tree

Node* root

Name	Type
☑ nodeCount	int

Class Name: Node

Name	Type
☐ left	Node*
☐ right	Node*
☑ value	int

Figure 26: Defining which data is important to the user

In addition to defining which fields are important enough to be displayed, the user must also choose one member variable in each class that is the most important (defined with a checkmark in VDB). This special member's value will always be displayed, even when the user puts VDB in Overview mode. Figure 27 shows VDB in its default mode, the detail mode, which gives all of the known information about the classes and variables defined by the user. Figure 28 shows VDB in overview mode, where only the value of the "special" member is displayed for each node. The former allows the user to examine all of the properties of every node, while the latter uses much less screen space and still gives the user the most important information. Used together, they provide Overview+Detail.

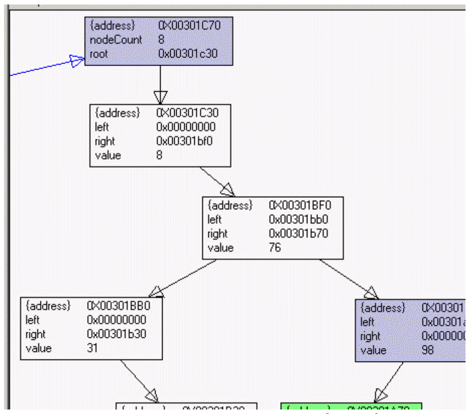


Figure 27: VDB, in Detail mode

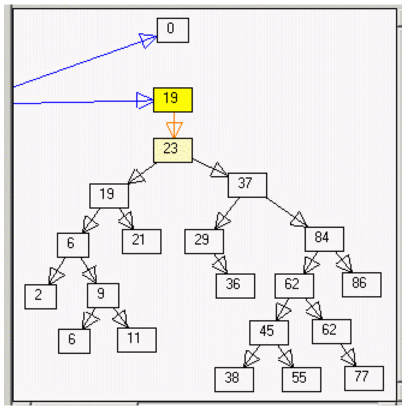


Figure 28: VDB, in Overview mode

Thanks to Overview+Detail, Gary no longer gets lost in large data structures, because he always has a rough overview of the data structure to which to refer when he starts delving into deeper parts of the visualization.

3.4 Focus+Context

But suppose that this Overview+Detail separate display concept still causes a hindrance to Gary during his debugging of the large data structure. The separate display uses valuable screen space that could otherwise be used to display more of the data structure. What if there were a combination of the overview and the detail into a single display?

Focus+Context [8] is a natural progression from the use of Overview+Detail. The former is much like the latter, except that the detail part, now called the focus, is somehow embedded *inside* the overview. The overview that surrounds the focus provides a context to the detailed section. An example of Focus+Context is the fisheye view [5].

The fisheye view acts as a magnifying glass, zooming in on whatever is in focus (usually defined by the location of the mouse) and providing a smooth gradient of zoomed-out graphics around the focus. Objects closer to the focus appear larger, allowing more detailed analysis, while objects further from the focus appear smaller, providing context to the focus.

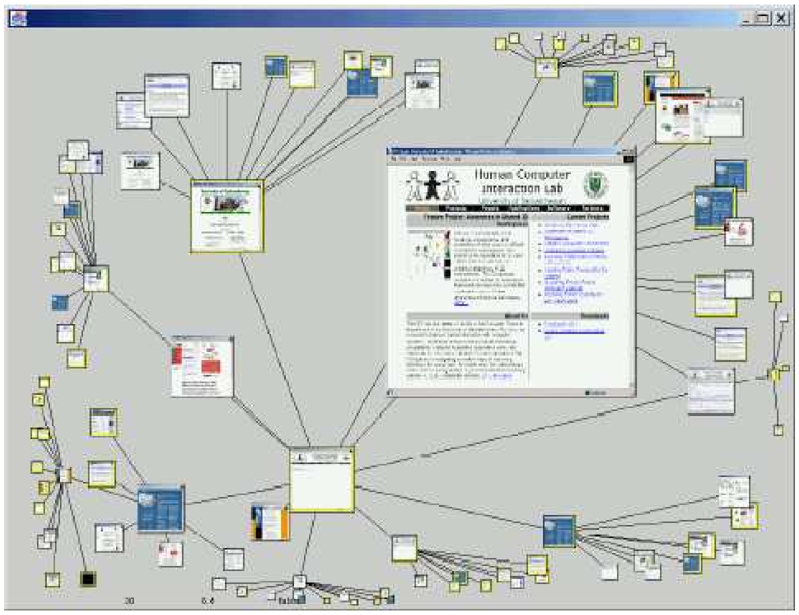


Figure 29: An interactive fisheye view for browsing a website
(taken, unedited, from [5])

When debugging complex data structures visually, screen real estate is consumed quickly. As the size of the data structure in memory increases, its corresponding graphical representation increases in size. A visual debugger can either concede defeat and simply require more screen space, or it can try to cut out some less important information to squeeze as much of the more important information onto the screen as possible.

Applying Focus+Context to visual debugging is not too daunting a task. Like the browser fisheye view in Figure 29, certain nodes or groups of nodes in the data structure can be shown in full detail, allowing the user to observe all elements of the data in those sections, while those parts of the data structure that are not of particular use at the moment can be partially hidden, providing *some* information about the contents of these unimportant nodes, or perhaps providing no information at all save a dot representing its location and relationship in the data structure. Ideally, data should be displayed in only as much detail as its importance to the user prescribes.

With this resolve, the Data Structure Visualizer handles Focus+Context by forming a spectrum of importance with respect to a particular type of information. A type of information can be totally useless to the user, in which case the visual debugger should never waste screen space by displaying it. The most important information should always be displayed on the screen, no matter how crowded the display becomes. Information whose importance lies somewhere in between should occupy an amount of space and time that is somewhere in between – either less space of occupation or less time of visibility than the most important information.

The DSV, like the original Visual Debugger, allows nodes to be in one of two different modes: detail (where all known variables of a node are displayed) and overview (where some variables are hidden). Instead of separating the two in different displays, however, the detail is embedded within the overview to create a Focus+Context effect. Nodes near the focus are displayed in detail mode, while the surrounding nodes are displayed in overview mode. The focus is defined by the location of the mouse. As the mouse passes over a node, it transforms from overview mode (Figure 30) to detail mode (Figure 31).

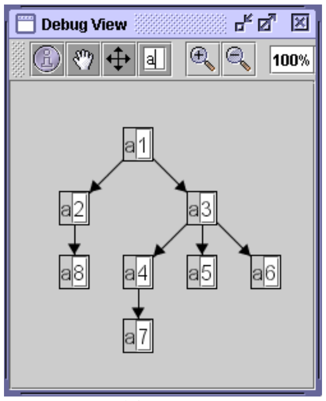


Figure 30: The DSV, with no focus

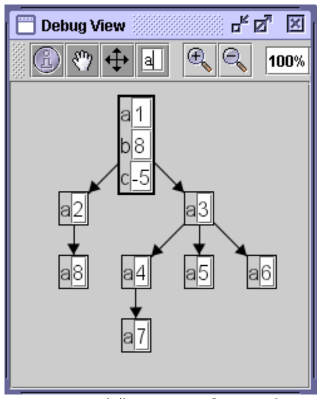


Figure 31: The DSV, focused on the root node

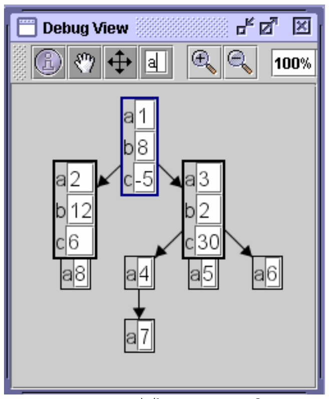


Figure 32: The DSV, after extending the focus

The user can extend the focus beyond a single node, so that the focused node's children and/or parents are displayed in detail as well (Figure 32).

The DSV also allows the user to specify how many variables are displayed in overview mode, so that if two variables are equally important, the user can see them both in overview mode. That is where the spectrum of importance comes to play. Each variable in a class can be assigned an importance level, which is used to determine which variables are displayed in overview mode. The importance level can also be used to define a "threshold of detail," where only variables of at least a particular importance are displayed in overview mode. This threshold of detail could deteriorate further from the mouse, so that more variables are displayed for the nodes that are nearer to the mouse.

Combining zooming with Focus+Context is generally a useful capability. In fact, many Focus+Context implementations use zooming to display the focused area in greater detail. In the case of the DSV, The user may zoom in or out to their desire, and focus detail level is dependent on the zoom level. That is, when the user is zoomed out, information provided visually—such as the number of member variables in a node, which can be inferred from the size of a focused node, or any data displayed using graphical data representation—may be noticeable when zoomed out, but if the text is too small to read anyway, no additional text information is provided.

The use of Focus+Context generally precludes the need for Overview+Detail, but it is not impossible to use both—merely inefficient. The goal of Focus+Context is to eliminate the need for Overview+Detail, so using both defeats the key purpose of Focus+Context. It does not hurt, however, to provide both types of functionality. Particular users may prefer Overview+Detail to Focus+Context, so providing both caters to individual preferences.

With Focus+Context, Gary can see a tremendously large part of his data structure, while being able to look at the details of a particular part of that data structure, depending on which part he is interested in. This allows Gary to focus on one section of the data with enough detail to see exactly what is going on, while still getting the context that helps him identify where he is in the data structure.

3.5 Direct Manipulation

Gary finds the visualizations useful, but he wishes that there was more that he could do with the visualization, other than view it. He would like to be able to drag parts of it around and reorganize the visualization graph as he sees fit. He would like to play with the nodes as if they were direct links to the actual nodes in his data structure, as though changing a value in the visualization could change the corresponding value in his data structure.

Being able to treat a program being debugged in a direct manipulation style provides the user with an easy-to-learn interface with their data structures. Being able to move and arrange nodes by dragging them around in the display, for example, helps the user to organize the data in a way that makes more sense to them. Connecting the modification of the text values in the graphical representation of a node with the modification of the corresponding variables in the user's program is another example of direct manipulation.

Direct manipulation provides users with a metaphor that helps them to better understand their data structures. Objects declared on the heap are much like pockets of memory, which corresponds nicely with the rectangles enclosing the graphical members of an object on the display, and pointers used in C/C++ are almost always visualized as arrows like the ones connecting the rectangles in the graphical representation.

The DSV and VDB2 implement direct manipulation primarily through mouse manipulate of the graphical representations of objects—the nodes. Users can move the nodes in the display, allowing them to be rearranged in a user-specified organization (Figure 33 and Figure 34). Additionally, focus of a node is achieved by pointing the mouse at its graphical representation, and clicking on the node causes the focus to persist even without having the mouse pointed at it. Finally, the DSV allows the user to click on the values of object member variables to edit their values.

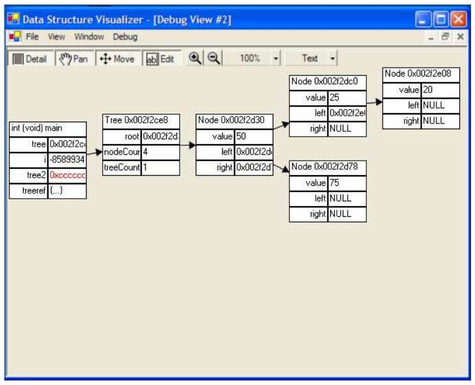


Figure 33: Automatic layout of a data structure

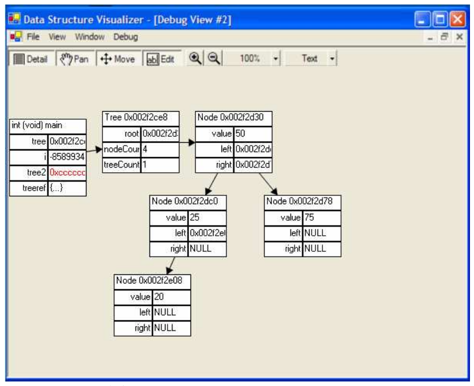


Figure 34: The same data structure, after the user moves some nodes

The DSV, though not connected to a back-end debugger, demonstrates the capability of editing the values in the graphical representation of an object. Ideally, a true visual debugger would associate the editing of such values with the modification of the values with which their graphical representations are associated.

Direct manipulation gives Gary an intuitive and convenient way to do everything he had wanted to do with the visualization. He can move a node around and edit its properties as though it were an actual object.

3.6 Layout

With the newfound flexibility of being able to arrange the nodes in the visualization comes a new trouble for Gary. Now, his data structure is in disarray and he cannot understand what the visualization is supposed to represent. The visualization is a hodgepodge of randomly-placed nodes and confusing arrows.

A key part of any visual debugger is its ability to automatically lay out a graph without user intervention. Being able to build graphs and to display the nodes therein is one thing, but it is paramount that these nodes be presented in an organized fashion that is quickly and easily understood by the user. A graph in disarray is perhaps less useful than no graph at all.

Several existing visual debugging tools have demonstrated the ability to automatically lay out the graphs that they create. Dotty focuses on this very ability, and tools that use Dotty, like Deet and VDBX, are automatically equipped with graph layout capabilities.

Some tools, like Swan and DDD, even go so far as to provide multiple layout algorithms. This provides better support for a wider variety of graphs. But is it enough?

3.6.1 Data Structure-Specific Layout

There are many types of data structures, and many of them have had specific ways of visualizing them since they were first introduced to a classroom. Linked lists, for example, have almost unanimously been visualized with boxes connected by arrows. While many data structures can be represented as graphs in this fashion, the way those boxes are distributed across the visualization canvas can vary greatly from data structure to data structure. Linked lists are usually depicted with a simple linear arrangement, with the head of the list at the left and the rest of the list trailing to the right.

One solution to this variety lies in data structure-specific layout. For every unique way of laying out a graph, corresponding to unique types of data structures with unique layouts, an algorithm is designed to organize the graph according to the most appropriate layout. For example, the linked list algorithm would lay out the linked list data structure as one would expect it to appear, as a linear array of boxes connected by arrows.

Since a visual debugger should be charged with the responsibility of laying out the graph of a data structure in a way that makes it more understandable to its human analyzer, one would expect that domain-specific layouts serve well. Based on the quantity of types of data structures, however, it may not be feasible to devise and implement a layout algorithm for each and every one. A visual debugger can get around this issue by providing a simple way to allow users to expand the visual debugger's library of layout algorithms. If a user cannot find a suitable layout technique already included with the visual debugger, then the user may have the programming skill—depending on the simplicity of the visual debugger's layout algorithm plug-in API—to develop their own specific layout algorithm to complement their data structure. The ability to add user-defined layout algorithms increases flexibility for the user, and it reduces load on the developers of the visual debugger by not requiring them to attempt to design layout algorithms for every possible type of data structure.

In the case that the user does not want to spend the time to develop their own layout algorithm, it is appropriate to at least include a default layout algorithm to handle any data structures that do not fall in any of the categories of data structures for which the visual debugger does not have a specific layout technique—a universal layout technique.

3.6.2 Universal Layout

The silver bullet of data structure graph layout would be to find an algorithm that organizes *any* type of data structure graph in a manner most familiar to and most usable by the largest assortment of users. But different users have different particular tastes, and it may be impossible to find a suitable lowest common denominator.

The Visual Debugger 2.0 uses a simple, universal layout technique derived from the layering method, a graph layout method that is particularly useful in laying out directed graphs like those inherent in reference-based data structures. While still in development, the basic algorithm is a recursive function that, for each node, first places its children to its right, then places itself to the left of its children, centered vertically between its children (Figure 33). Back-edges in cyclic graphs are shown with straight, translucent arrows to increase simplicity without sacrificing readability. The result is a graph whose roots are at the left (where each root is a stack frame, the top-most frame at the top of the display), and all of their children one-level deep are in the next column, and all of *their* children on in the third column, etc.

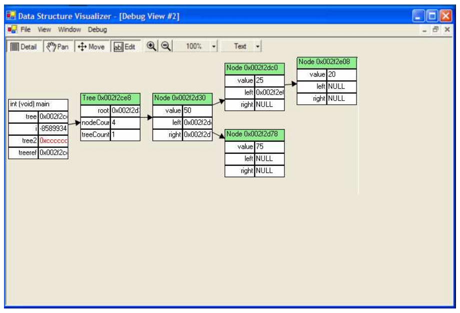

Data Structure Visualizer - [Debug View #2]
File View Window Debug
Detail Pan Move Edit 100% Text

Figure 35: Result of automatic layout

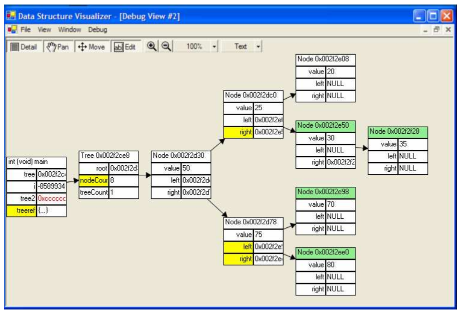

Data Structure Visualizer - [Debug View #2]
File View Window Debug
Detail Pan Move Edit 100% Text

Figure 36: Layout after a few nodes are added

Though initially confusing when applied to trees, one reason children are to the right instead of below their parents (which is normal for some data structures, i.e. trees) is for increased simplicity with respect to the sizes of nodes. All nodes have the same width in VDB2 (values are truncated as necessary, usually for long strings), but their heights vary with respect to the number of data members in the node as well as the focus level of the node. Since only the widths are constant, any alignment can only be made vertically, and it was deemed more important to keep nodes of the same depth aligned vertically than to keep randomly-related nodes aligned vertically.

A second reason for placing children to the right is to increase depth visibility despite the number of data members in a node. For example, if all of the nodes had twenty data members, then perhaps two levels would be visible if children were placed below their parents. With children to the right, you can see more levels, and the number of levels that you can see is constant. The tradeoff is that the number of visible nodes per level varies with the number of data members in the nodes.

3.6.3 Stability

Another attribute of a layout algorithm that is particularly important in the constantly changing state of a visual debugger is that of *stability*. If a layout method is called after each step through a user's code, it is vital that the layout chosen for the latest version of the data structure remain as true to the old version as possible. If a new node is added to the data structure, it would be particularly confusing for the user if that drastically changed the layout of the resulting graph. Small changes to the data structure should result in small changes to the layout.

One way to implement stability would be to develop a dynamic layout method that lays out a graph based on its previous state. If a node is removed from the graph, then the visual layout approach of the dynamic layout algorithm would be to very slightly move the locations of the existing nodes to fill the gap left by the now-missing node. A standard layout method, such as the ones present in Dotty, would simply lay out the entire graph from scratch, which could disorient the user. A further step to reduce disorientation would be to include animation, so that the changes between the states of the data structure between two points in the execution of the program are made even more apparent.

When the state of the data structure changes, VDB2's automatic layout algorithm maintains stability by only moving nodes when necessary (Figure 36). That is, the nodes are not newly placed at each update, unless they are brand new nodes, in which case some existing nodes may need to be moved slightly to provide room for the newly placed nodes. Nodes that are moved by the user remain moved. This contrasts with unstable graph algorithms like the one in Deet, which uses Dotty to lay out the entire graph from scratch at each breakpoint. Node movement is minimized to reduce confusion that the user may perceive when nodes move unexpectedly. Animation may also be used to reduce this confusion.

3.6.4 Issues with Other Information Visualization Techniques

Focus+Context has a profound impact on layout. For instance, how is the layout effected when a certain node or area of nodes is placed in focus. Do the surrounding nodes move away to provide room for the focused nodes, which are probably larger due to their more-detailed nature? If the nodes *don't* move, then the most important context information—the context immediately surrounding the focused information—is lost. One solution is to simply stretch the drawing canvas around the focused nodes after the layout has been performed, using a distortion-oriented technique [15]. If Focus+Context is employed in the visual debugger, the layout must be dynamic enough to support focus-based layout, which may need to be updated in real time.

Direct manipulation also plays a role in layout. If a user moves a node, does the layout allow the anomaly in the layout? Should the layout be strictly enforced? Might the layout adapt to the actions of the user, modifying the layout algorithm to suit the preferences indicated by the user when he or she moves nodes around the canvas? This leads to the idea of using user hints to indirectly and flexibly control graph layout [18]. If direct manipulation is supported in the visual debugger, then these layout issues must be addressed.

3.6.5 Summary

In fact, whatever path is chosen by a visual debugger, it may be more important to give the user as much flexibility as possible with respect to layout. Why not provide *both* kinds of layout algorithms? Allow the user to choose among the various data-structure-targeted layout algorithms, or let the user choose a more universal layout algorithm. What's more, the ideal visual debugger would provide a simple interface for allowing the user to provide their *own* layout algorithm, in much the same way that Swan allows users to manually lay out their graphs via annotations embedded in the code.

If a user's data structure is too unique to warrant a general layout method, and the universal layout algorithm aren't producing the desired look, then the user may with to develop his or her own layout. This may take extra time and effort on the part of the user, but if the user can find profit in it, it certainly cannot hurt to provide the functionality. The Data Structure Visualizer attempts to provide this through a simple java interface.

Gary can now play with the arrangement of his visualization without fear of getting lost in the confusion of randomly placed nodes. When it gets confusing, Gary just asks his visual debugger to lay out the graph, and the visualization makes sense to him again.

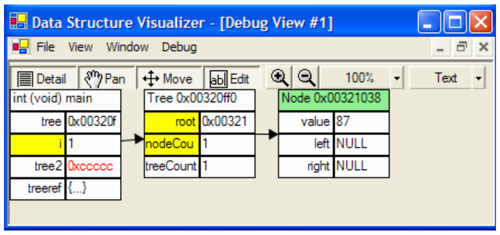
3.7 Visual Notification

Despite all of these information visualization tools, Gary might still accidentally let a small change in his data structure go unnoticed. What if it had been an 80 and 88—instead of the 17 and 77—that had been erroneously swapped? The difference in text is still small, but now, even the difference in magnitude is small. An error, however, is still an error, and it must be detected as soon as possible.

A huge part of debugging a program is identifying errors in a program. Fixing errors is often much easier than finding the errors, but the latter is clearly a prerequisite for the former. It is imperative, then, that a visual debugger provide as much useful information as possible that might help the programmer locate errors. This can be done by keeping the user aware of specific goings on as the program executes with monitoring ideas gleaned from the realm of notification systems [17].

Visual debuggers have a distinct advantage over text-based debuggers at being able to visually assist users in identifying errors in data structures, which leads ultimately to identifying errors in the code. One way that a visual debugger might do this was already discussed in conjunction with graphical representation of numerical data. Being able to spot irregularities in a sorted linked list can lead a programmer to a bug in the code that might otherwise have been more difficult to find.

Another advantage on which visual debuggers should capitalize is that of notification of changes in the state of the underlying data structure. A study shows that programmers can more quickly find and resolve reference-related bugs (like those in data structures) when given immediate feedback [7]. If the value of a certain node changes, the user should be notified about that change so that he or she can verify that the change was consistent with his or her expectancy about what the program should be doing.

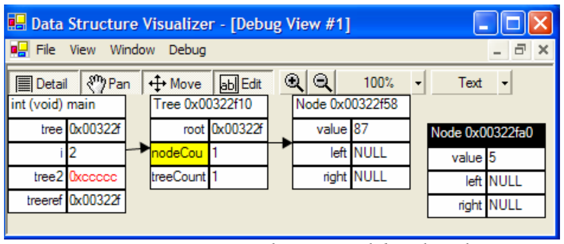


int (void) main
tree 0x00320f
i 1
tree2 0xcccccc
treeref {...}

Tree 0x00320ff0
root 0x00321
nodeCou 1
treeCount 1

Node 0x00321038
value 87
left NULL
right NULL

Figure 37: VDB2, showing a new node and several modified variables



int (void) main
tree 0x00322f
i 2
tree2 0xcccccc
treeref 0x00322f

Tree 0x00322f10
root 0x00322f
nodeCou 1
treeCount 1

Node 0x00322f58
value 87
left NULL
right NULL

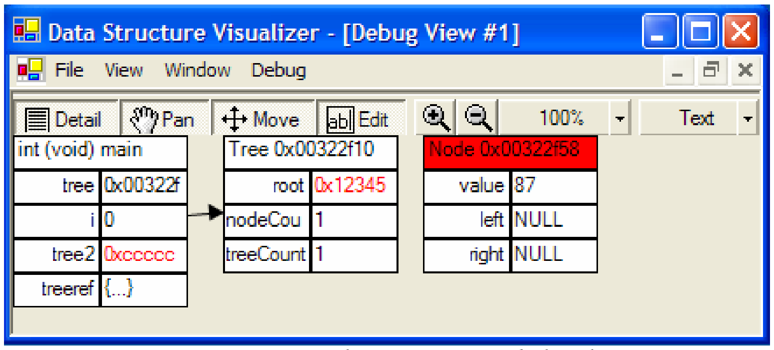
Node 0x00322fa0
value 5
left NULL
right NULL

Figure 38: VDB2, showing a deleted node

VDB2 provides notification of several events. When a new object has been allocated to the heap since the last breakpoint, its node's title bar appears green to inform the user thereof (Figure 37). When a value in an object changes, its label becomes yellow (Figure 37). Deleted nodes appear in black (Figure 38).

In addition to these standard notifications, which may or may not indicate errors, VDB2 also informs the user of confirmed (as opposed to potential) errors such as memory leaks or dangling pointers. Figure 39 shows VDB2 visualizing a program that has just changed the "root" pointer to an erroneous value. The node that the pointer had originally been pointing to is now inaccessible by the program, the definition of a memory leak. VDB2 marks the node in a noticeable red to assure that the user is alerted about the memory leak created by his program.

In addition to the memory leak, the "root" pointer is now a dangling pointer because it does not point to a valid Node object; further dereferences of the "root" pointer will result in either unpredictable results or a segmentation fault. VDB2 alerts the user of this dangerous situation by highlighting the faulty address in red text.



int (void) main	
tree	0x00322f
i	0
tree2	0xcccccc
treeref	{...}

Tree 0x00322f10	
root	0x12345
nodeCou	1
treeCount	1

Node 0x00322f58	
value	87
left	NULL
right	NULL

Figure 39: VDB2, showing a memory leak and a dangling pointer.

Notification of state changes in the data structure is useful in assisting a programmer in finding possible errors, but notification about memory leaks and dangling pointers means that VDB2 has discovered on its own errors in the program. Every programmer has forgotten to free memory after it is no longer being used. VDB2 can help that programmer make sure that this mistake is found as easily and as quickly as possible.

Using Focus+Context may require special cases to handle notification of an object that is in focus and an object that is not in focus. For example, when a node in the DSV is not in focus, it hides less-important member variables to save space, but what happens if one of those less-important member variables changes? Visual notification of that member variable would require either putting the node in focus, at least to the point that the modified member variable is visible, or notifying the user in another way, such as highlighting the node that contains the member variable. Perhaps, however, the "less-important" status of the member variable warrants that the notification not occur at all.

Now, Gary gets instant notification whenever any change occurs in his data structure. Once-unnoticeable changes are now signaled to him on a more noticeable scale. As soon as the 80 and 88 are swapped, Gary notices the change and that it goes against his expectations, which leads to the investigation and eventual solution of the causal bug.

3.8 Summary

The goal of any debugger should be to assist the programmer in locating bugs and fixing them. There are many different ways to do this, and several of them lie in the field of information visualization, specifically when applied to a visual debugger.

Bugs in code often produce bugs in the data structures that they use. If the negative effects on these data structures can be identified, their causes can be traced back to the bugs in the code. Therefore, visual debuggers carry the additional responsibility of helping the programmer locate errors in their data structures. This can be done by notifying the user of important events, such as when a variable changes or when a pointer becomes invalid. It can be done by making it easier for the programmer to identify errors through use of graphical representation of data values.

It can be done by providing the user with a more flexible visualization environment; a visualization environment that allows the programmer to see as much of the data structure as is necessary, via Zoom and Pan, Overview+Detail, or Focus+Context; a visualization environment that allows the programmer to reorganize the display without fear of disorganizing it forever, or to modify their data structure directly simply by making changes in the visualization, via direct manipulation and layouts.

By helping the user find errors in data structures, information visualization techniques allow the visual debugger to better assist the programmer in locating and fixing errors in the underlying code, which saves time, money, and maintenance.

Chapter 4

CHAPTER 4: VISUAL DEBUGGER 2.0

As proof of concept, the Visual Debugger 2.0 represents a complete implementation of all seven of the information visualization ideas presented in the previous chapter. While some of the implementation may be incomplete and relatively basic in some respects, and there are still outstanding issues that need to be resolved, VDB2 at least provides functionality for the ideas, and the issues can be resolved with moderate work.

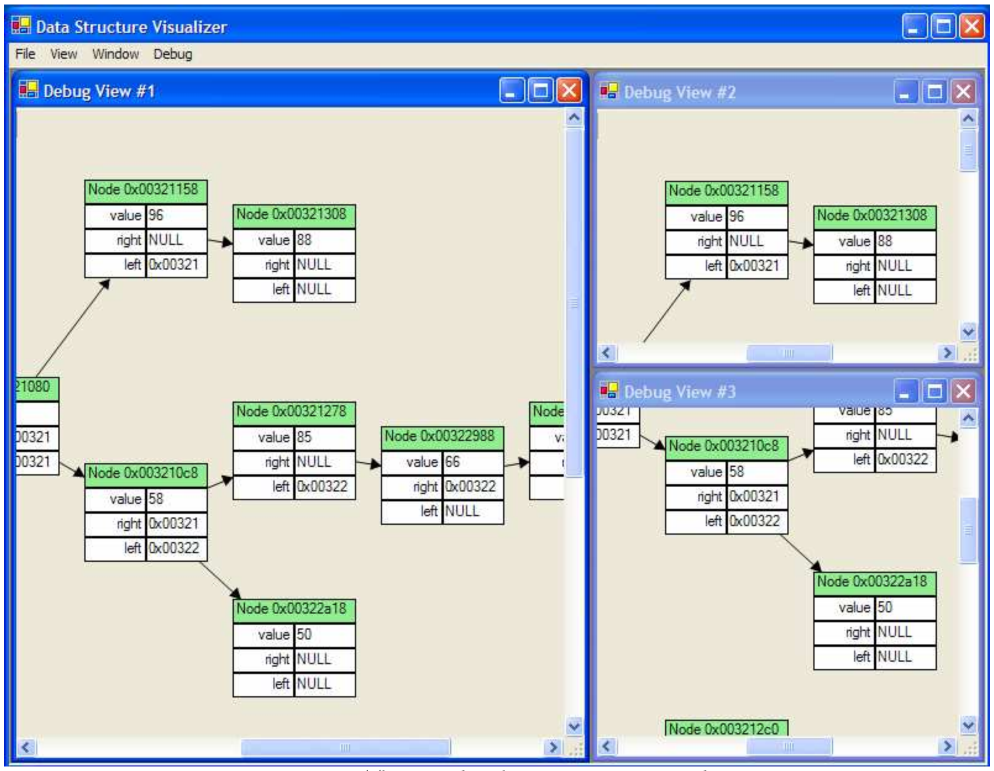


Figure 40: The Visual Debugger 2.0 Main Display

Not to be confused with the java-based Data Structure Visualizer program, the main window of VDB2 is called the Data Structure Visualizer window (Figure

40). Inside this main window, the VDB2 interface is based on the concept of the "debug view," which is a single view of the data structures that the user is debugging. In the main window, the user can create as many debug views as he or she desires, with the capability to move and modify the properties of each one. Each debug view may focus on a different section of the data structures, but all debug views represent windows into the same data structures.

Graphical value representation is implemented in the form of various data display modes. At any one time, a given debug view may be in one of three modes: text, bar, or color. When in text mode (Figure 41), all data is represented by textual representation. That is, strings appear as strings of characters and numerical data appear as strings of numbers. When in bar mode (Figure 42), numerical data is represented by colored bars of varying sizes, where color represents the member variable name and the size of the bar represents the magnitude of the value relative to the minimum and maximum of the other values found in instances of that variable name. Color mode (Figure 43) is like bar mode, except that the bars all are all full-size, instances of the same variable name are colored with the same hue, and intensity of the color indicates relative magnitude of the number. Each debug view may have its own display mode, so all three modes could conceivably be visible at one time.

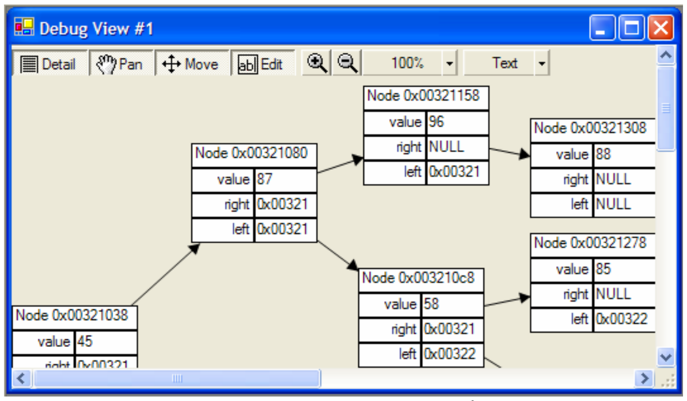
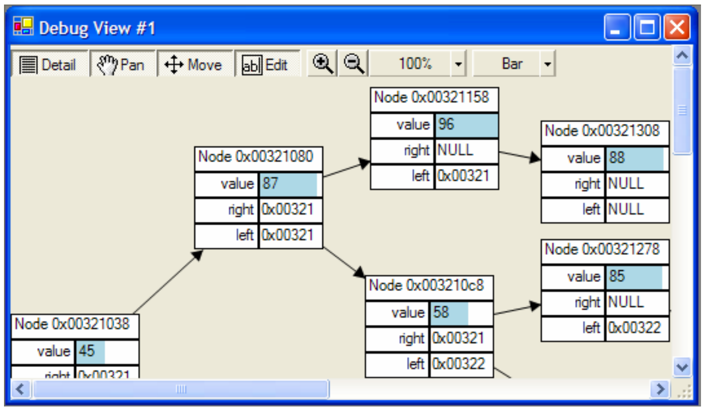


Figure 41: VDB2 in text mode



Node 0x00321158
value 96
right NULL
left 0x00321

Node 0x00321308
value 88
right NULL
left NULL

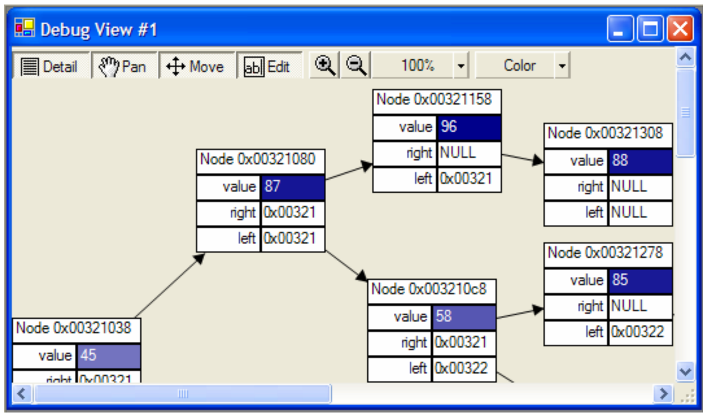
Node 0x00321080
value 87
right 0x00321
left 0x00321

Node 0x00321278
value 85
right NULL
left 0x00322

Node 0x003210c8
value 58
right 0x00321
left 0x00322

Node 0x00321038
value 45
right 0x00321

Figure 42: VDB2 in bar mode



Node 0x00321158
value 96
right NULL
left 0x00321

Node 0x00321308
value 88
right NULL
left NULL

Node 0x00321080
value 87
right 0x00321
left 0x00321

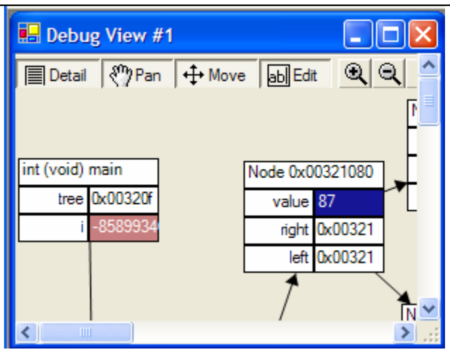
Node 0x00321278
value 85
right NULL
left 0x00322

Node 0x003210c8
value 58
right 0x00321
left 0x00322

Node 0x00321038
value 45
right 0x00321

Figure 43: VDB2 in color mode

Zooming in VDB2 is inherited from the Data Structure Visualizer, allowing the user to zoom each debug view arbitrarily in (Figure 44) and out (Figure 45).



int (void) main
tree 0x00320f
i -8589934

Node 0x00321080
value 87
right 0x00321
left 0x00321

Figure 44: VDB2 zoomed in

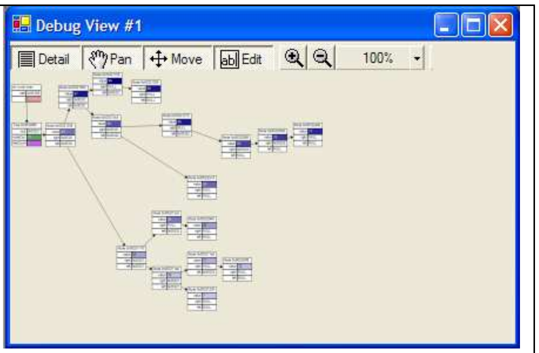


Figure 45: VDB2 zoomed out

VDB2 provides Overview+Context in the same way that the DSV did. The capability to zoom combined with the capacity for multiple views allows the user to create their own custom Overview+Detail displays. For example, the debug views in Figure 45 and Figure 44 could have represented the overview and detail views, respectively.

Focus+Context in VDB2 (Figure 46) is implemented like it is in the Data Structure Visualizer, except that member variable names are only included in detail mode. Use of the bar or color visualization modes increases the ability of the user to distinguish between related member variables, because they are colored with the same hue. As in the DSV, focus for a node is achieved when the mouse is pointing over the node, and the focus can be held by clicking on the node. Further clicking of the node results in focus of that node's reference parents and children, and those nodes' parents and children, etc. Right-clicking on the node retracts this "field of focus" level by level.

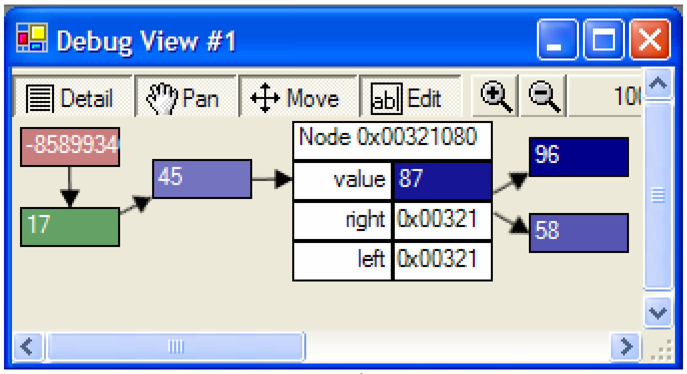
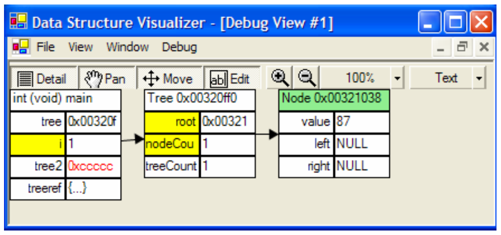


Figure 46: VDB2 with Focus+Context

The default layout algorithm of VDB2 is a variation of the layering method, adapted to go from left-to-right to easily adapt to differences in node sizes (the heights may vary, due to the number of member variables in an object, but the widths of all focused nodes are the same, and the widths of all non-focused nodes are the same). In addition to the default layout algorithm, two application-specific layout algorithms are included (for linked lists and trees), with the capability to add user-defined layout algorithms. The layout is updated dynamically after each break in the execution, though this automatic layout feature can be disabled at the user's leisure. There is currently no specific support for incorporating Focus+Context dynamics in the layout algorithm.

Visual notification for several execution events is included in VDB2. Specifically, the following events are signaled to the user:

1. New object instantiated: green node title bar (Figure 47)
2. Variable modified: yellow variable name label (Figure 47)
3. Object deleted: black node title bar (Figure 48)
4. Memory leak: red node title bar (Figure 49)
5. Dangling pointer: red pointer value (Figure 49)

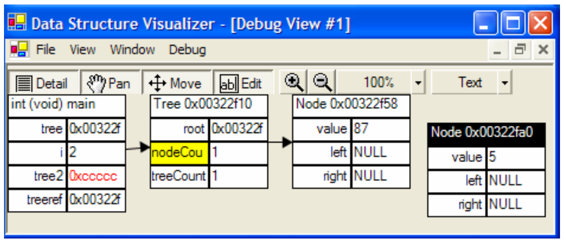


int (void) main	
tree	0x00320f
i	1
tree2	0xcccccc
treeref	{...}

Tree 0x00320ff0	
root	0x00321
nodeCou	1
treeCount	1

Node 0x00321038	
value	87
left	NULL
right	NULL

Figure 47: VDB2, showing a new node and several modified variables



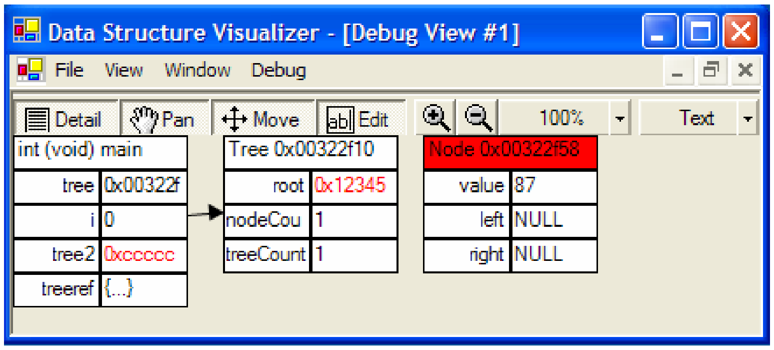
int (void) main	
tree	0x00322f
i	2
tree2	0xcccccc
treeref	0x00322f

Tree 0x00322f10	
root	0x00322f
nodeCou	1
treeCount	1

Node 0x00322f58	
value	87
left	NULL
right	NULL

Node 0x00322fa0	
value	5
left	NULL
right	NULL

Figure 48: VDB2, showing a deleted node



int (void) main	
tree	0x00322f
i	0
tree2	0xcccccc
treeref	{...}

Tree 0x00322f10	
root	0x12345
nodeCou	1
treeCount	1

Node 0x00322f58	
value	87
left	NULL
right	NULL

Figure 49: VDB2, showing a memory leak and a dangling pointer.

As is evident by these features, VDB2 hopes to be a better visual debugger than its competitors by including many more information visualization concepts than its competitors. More importantly, the development of VDB2 has proven that such concepts can be included in new visual debuggers or even added to existing visual debuggers without reinventing the wheel. In the case of VDB2, using the Microsoft Visual Studio .Net Debugger as the underlying text-based debugger—instead of writing a brand new debugger or re-implementing the features of one—allowed a single programmer to build a highly functional information visualization-based visual debugger in less than a year.

C h a p t e r   5

CHAPTER 5: VISUAL DEBUGGING SOFTWARE ARCHITECTURE

Building a visual debugger is inherently more difficult than building a standard text-based debugger. With a text-based debugger, he user generally provides all of the necessary information about what exactly is needed from the debugger. For example, if a user wants to know the value of variable `myInteger`, the user tells the debugger what the variable is and how to find it. The debugger retrieves the value from the computer's memory and returns it to the user.

With visual debugging, a major goal is to display the data structure in its entirety, and it is far too tedious a task to ask the user to provide the names and locations of all of the variables in his or her data structure, especially considering that the number of variables increases as the data structure grows. It is up to the visual debugger to explore and discover as much of the data structure as is possible, with as little user intervention as possible. How this is done varies from debugger to debugger.

Some visual debuggers rely on underlying debuggers text-based debuggers, such as gdb, which are already great at retrieving values from memory. Both versions of the Visual Debugger are built around the Microsoft Visual Studio suite of development environments.

5.1 General Implementation Issues

There are various pieces of information to which a visual debugger should have access in order to build a useful visual representation of a data structure.

- Program stack data: variables on the debugged program's stack and their values.
- Heap data: variables allocated on the heap and their values.
- Class structure: the pieces of user-defined classes that make up the data members of an object.

In general, a visual debugger must have access to all of this program data in order to display useful visual information about a running program. Typically, the easiest way to do this is to use an underlying text-based debugger that is already set up to debug programs. In most cases, the debugged program must be compiled with debugging information that allows the debugger tool to query certain information such as the names of variables, which are not stored in standard binary executables.

Using an underlying text-based debugger allows the visual debugger to maintain all of the features of a text-based debugger. All that remains, then, is determining how to extract—through the text-based debugger—all of the information that is needed to construct a useful visual representation of the data structure. If the necessary information cannot be extracted through the text-based debugger, then an alternate means of extracting the information must be devised, which may involve direct manipulation of the binary image and will generally require much greater involvement and effort. Fortunately, a text-based debugger will generally be sufficient.

At some point during the execution of the program being debugged, information will need to be queried from the text-based debugger in order to build the visual representation. This may happen, for example, when something changes in the program image, requiring an update in the visual debugger. This update can occur on a continual recurring basis, assuming data can be extracted from the program image during execution; this method allows real-time visualizations to be provided to the programmer, but will severely slow the execution of the program since the visual debugger is using as much CPU cycles as it can get. It may therefore be more appropriate to update only when the user wants an update, via push-button, intermittent, or breakpoint-triggered update. VDB, DSV, and VDB2 all update when a breakpoint is reached. A breakpoint in visual debugging is analogous to that in text-based debugging—when the given line of code is reached during execution, the program is stopped temporarily until the programmer resumes execution. When a breakpoint is reached, an update occurs.

Depending on the nature of the visual debugger, an update retrieves only as much information as it needs. In the case of the original VDB, the user provided VDB with all of the data that was to be collected, specifically with respect to information on the stack. The VDB2, on the other hand, uses its underlying debugger to query all of the data in the program stack, providing instant access to every variable in the program stack. Given access to a text-based debugger, the values of the stack variables can easily be queried. Pointers to primitive data types—such as integers or floating point numbers—can easily be resolved by following the pointer, asking the text-based debugger for the value of the data at the given address. Difficulties arise when non-primitive objects are involved.

When an object or a pointer to an object is encountered, then if the visual debugger is interested in providing information about the contents of the object, then the visual debugger must determine the memory structure of the object in order to query the text-based debugger about the values of the member variables of the object. In some cases, this may require querying the user about the structure of their data structures. Using a more sophisticated underlying text-based debugger, however, the visual debugger may be able to query the text-based debugger about the structure of the object.

As the structure of the object is explored, more objects or pointers to objects may be discovered embedded within the object. Depending on how deep the visual debugger chooses to go, recursion can be used to further explore the depths of the data structure. Care must be taken that infinite loops do not occur in the survey of data structures that include cycles, such as circular linked lists. This can easily be handled by marking nodes as visited. Ultimately, the process amounts to either a breadth-first or depth-first search, the choice of which depends on the preference of the developer of the visual debugger.

How deep the data structure is traversed is dependent on the intention of the visual debugger. For example, a shallow scan of a data structure might provide a useful initial overview of the shallow parts of a data structure, where certain branches could be explored in detail, in further depth.

As each object is encountered, the visual debugger must store information about it as it is collected from the underlying text-based debugger. This information can be stored in a graph structure that very nearly mirrors the structure of the user's data structure. The result is two data structures: the *user's* data structure and the *visual debugger's* data structure. The objects in either data structure are also referred to as **nodes**. The visual debugger's memory requirements expand with the size of the user's data structures, and the update process can generally be completed in $O(n)$ for n primitives and objects in the user's data structures.

An update amounts to collecting information about the current state of the user's program and comparing it to the state recorded at the time of the last update. When a node in the visual debugger's data structure corresponds to a node in the user's data structure that is no longer reached during an update (easily found by collecting the nodes that have not been marked as visited), then one of two cases occurs.

If the visual debugger is collecting only partial data (i.e. by querying the user as VDB does), then there are three sub-cases. The node may have been deleted by the user's code, which is a perfectly valid cause. The node may now be accessible by a different pointer, one not specified by the user—another valid cause. Finally, the node may not be accessible by any part of the user's code—a memory leak. While the first two cases are acceptable for working code, it is impossible for the visual debugger to distinguish between the second and third case without having knowledge about all of the pointers in the debugged program. If the first case can be ruled out through dialogue with the text-based debugger, then it is safest to flag the node as a potential anomaly, informing the user.

If the visual debugger is collecting all of the program data information at once, as VDB2 does, then if no references are found pointing to the node, then the node was either properly deleted or is a memory leak. In some cases, this can be discovered by querying the underlying text-based debugger. For example, the Microsoft Visual Studio .Net Debugger used by VDB2 includes a method, `IsValidValue()`, that can be used to determine whether a variable was properly deleted.

When the update is completed, all that remains is updating the display to reflect the new data. When a user requests that a variable be modified, then functionality generally exists in text-based debuggers to handle this task, but this is usually only possible for variables that can be referenced by name (i.e. via a stack frame) or by address. For this reason, it is helpful to include the address of each node in the visual debugger's data structure. Addresses are usually easy to query from a text-based debugger, except in programming languages that do not acknowledge the existence of concrete, fixed addresses (i.e. Java).

Any other static user manipulation that may occur during a breakpoint should be independent of the underlying program data, assuming the visual debugger has all of the data that it needs. For example, a visual debugger can query initially only parts of a data structure, to save loading time, but if more detail is requested from the user, then the visual debugger must further query the text-based debugger in order to ascertain the data that the user requests. These details are, of course, dependent on the implementation of the visual debugger.

5.2 Integrating with Microsoft Visual C++ 6.0 Debugger

Integrating the original Visual Debugger with Microsoft Visual C++ 6.0 Debugger (MSVC++ Debugger) proved to be a tremendous challenge. The API for the MSVC++ Debugger is far too simple to do any serious debugging without manipulating a few factors.

Connection to the MSVC++ Debugger is relatively simple through the API, assuming the connecting party is using a Microsoft programming language, such as Visual Basic, the language in which the original Visual Debugger is designed. A simple method connects the Visual Debugger to the currently open project in the MSVC++ Debugger, a method that is called when the Visual Debugger is run by the user. In case a project is not open at the time VDB is initialized, a button is provided that allows the user to manually connect VDB to the MSVC++ Debugger. Connection to a project allows the connecting party to use the functions in the MSVC++ Debugger API to access information about the project.

The main limitation of the MSVC++ Debugger is that it provides only one function that is really of any use. The method, `GetExpression()`, allows the Visual Debugger to query the value of any variable in any current stack frame, assuming the name of the variable and in which stack frame its scope lies is known (see Figure 50). The argument to `GetExpression()`, a string, is evaluated in much the same way that strings are evaluated using the MSVC++ Debugger watch window.

Most primitive values are easy enough to retrieve, but any objects of any class would return only "{...}" as the value (see Figure 50). This makes it impossible for the Visual Debugger to perform any queries about the specifics of a particular class or object. The only solution available is to ask the user to provide the specifics that the Visual Debugger needs about class structure. The values of member variables of objects could be obtained, but the names of the variables are essential for this task.

The first piece of information needed is the name and type of all variables on the stack that are to be watched. For example, if the user wants to visualize a tree data structure, they might have a `Tree` class and create a new `Tree` object on the heap, using a pointer (called `tree`) on the stack to refer to it. The user first gives the name and type of `tree`.

If the Visual Debugger is not already familiar with objects of class `Tree`, then it must ask the user to define the specifics of the `Tree` class. The user need only define members of the class that he or she is interested in watching in the Visual Debugger display.

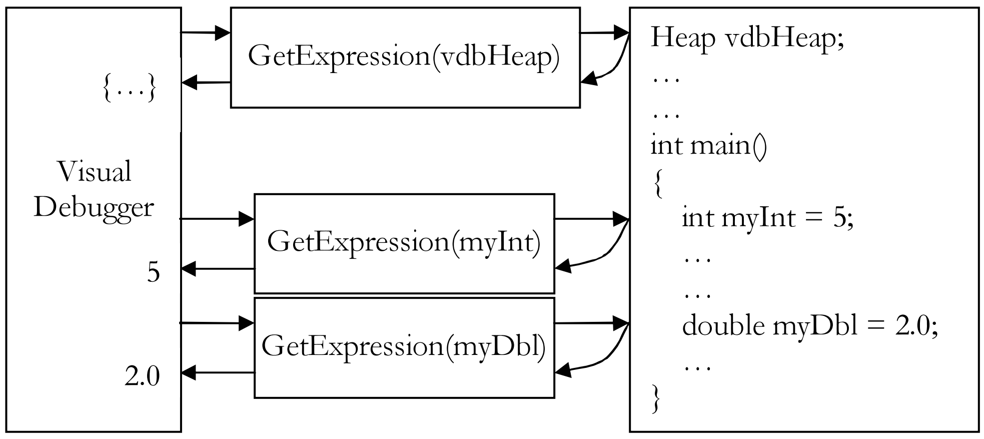


Figure 50: Retrieving information from the runtime

Given the names of the member variables of a particular class and the name of a pointer (located on the stack) to such an object, that object's member values can then be retrieved by passing to `GetExpression()` a string like this: `pointer->member1`

This value can then be displayed to the user as the value of one of the members of a class that is referenced by a pointer on the stack. If one of the members of that class is another pointer to another object, then the members of that object can be dereferenced in the same way, and so on recursively until NULL pointers are reached. Each object has a special textual address that has its roots in a stack frame. For example, the following might be the textual address of a leaf node in a binary tree: `myTree->root->left->left->right->left`

As an object is encountered, the values and states of its member variables are stored in a VDB-side object in a database. If the object has been encountered before, then the values are updated. To facilitate lookup of these objects, a hash table is maintained mapping addresses to VDB-side objects. Addresses can be retrieved simply by inserting an ampersand around the textual address of the object. This hash table is especially important in preventing VDB from going into an infinite loop caused by cycles in users' data structures.

To keep track of allocated and deallocated objects, the user must insert a line of code that adds `#include` statements for our special overloaded versions of the `new` and `delete` operators. Also included are a few Visual Debugger data structures to keep a debugged-program-side database of all heap objects. This program-side database stores the name, type, and address of heap objects and can be retrieved by the Visual Basic side of the Visual Debugger because the name and structure of the program-side heap is fixed and known by the Visual Debugger (see Figure 50). The contents of the database are retrieved via `GetExpression()`.

The database allows the Visual Debugger to keep track of all objects created on the heap using the `new` operator, as well as any objects deleted using the `delete` operator. This is especially useful in identifying memory leaks.

This process provides the Visual Debugger with all of the essential data to build a reasonably informative picture of a data structure, assuming the user is willing to put in the time and effort to define the specifics of their classes. A modification to VDB could be made to allow it to parse C++ header files and automatically harvest the necessary information, but this undertaking requires building a relatively robust C++ code parser.

The main drawback to this approach is the cost in time. The `GetExpression()` is an expensive function, taking up to several milliseconds to complete depending on the size of the string response, and the quantity of `GetExpression()` calls put a tremendous strain on the Visual Debugger, causing it to take up to a few seconds to update the display, depending on how many nodes are present in the system. At best, VDB can reasonably handle up to 50 nodes before the pause becomes unbearable. The lack of efficiency and the cumbersomeness of use were the two largest reasons to move away from the MSVC++ 6.0 Debugger and toward its successor, the Microsoft Visual Studio .Net Debugger.

5.3 Integrating with Microsoft Visual Studio .Net Debugger

The Microsoft Visual Studio .Net Debugger provided a much more robust and efficient API, allowing the Visual Debugger 2.0 to provide fast and accurate visual debugging without requiring the user to enter any information about his or her data structures.

The most significant changes in the .Net Debugger include methods for retrieving all of the stack frames in the current stack, as well as all of the variables in each of those stack frames, and the new and improved `GetExpression()` function. In addition to values, the `GetExpression()` function now returns the type and all of the data members of the queried variable, wrapped up neatly in an `Expression` object. All of these features create a graph made up of the `Expression` objects on the stack frames, which can contain references—thanks to the data members that the `Expression` object supplies—to other `Expression` objects.

The new API makes it easier to retrieve all of the information about a program without awkward workarounds. The biggest issue then becomes keeping updated.

Again, a hash table is used to keep track of objects and to prevent infinite loops caused by data structures with cycles. The objects in the hash table also make up the nodes of a graph that represents VDB2's local copy of the graph induced by the nodes of the debugged data structure. At each breakpoint, the .Net Debugger's graph of `Expression` objects is traversed, continuously comparing the .Net Debugger's graph with the local graph maintained by VDB2. New nodes in the .Net Debugger's graph are added to VDB2's graph, and changes in any of the nodes in the .Net Debugger's graph are reflected in VDB2's version. The entire update algorithm (Figure 51) is completed in linear time with respect to the number of Expression objects that are being monitored by VDB2, which is defined in part by the sum of the number of variables on the stack and the number of variables allocated on the heap.

```
Procedure Update
   For each node n in list of nodes
      Mark n as unvisited
   End For
   For each Stack Frame f in the runtime
      For each Stack Variable v in f
         Run UpdateRecursive on v
      End For
   End For
   For each node n in list of nodes
      If n is marked as unvisited
         Assure that n was properly deleted
            (otherwise, inform user of memory leak)
      End If
   End For
End Procedure

Procedure UpdateRecursive on a Variable v:
   Check if value of v is new or changed
   If v is not in our list of nodes
   Add v to our list of nodes
   End If
   If v is a class
   Then
      Mark v as visited
      For each Data Member m in v
         Run UpdateRecursive on m
      End For
   End If
End Procedure
```
Figure 51: Rough pseudocode of the update algorithm

An update is performed every time the .Net Debugger enters Break mode, which includes breaks caused either by breakpoints in the code or by the user stepping through the code line by line. It may be possible to keep the graph updated even when the .Net Debugger is not in Break mode, but it was decided early that such a strategy would consume too many CPU cycles and could not be practically used by the programmer.

If, during the update process, any of the nodes in VDB2's graph was not visited, then that node is no longer accessible by the program. Before labeling the node a memory leak, however, VDB2 must determine whether or not the node was properly deleted. A boolean function in the `Expression` object called `IsValidValue()` helps in deciding that. This same function can be used to identify whether pointers point to valid data, which can aide in the discovery of dangling pointers.

Already, the .Net Debugger provides VDB2 with more information than was available to the original Visual Debugger through the MSVC++ Debugger, at a fraction of the effort.

New to the .Net series of Microsoft integrated development environments is the concept of the .Net Debugger Add-In. Any software tool can be developed as a .Net Debugger Add-In, allowing it to be accessed directly through the .Net Debugger. Instead of opening the .Net Debugger and the Visual Debugger 2.0 separately, there is a menu item in the .Net Debugger menu that allows direct access to VDB2. This is much more convenient for the user and allows VDB2 to register itself to receive events from the .Net Debugger, such as those that occur when the user's program arrives at a breakpoint or as the user steps through the program code.

The .Net Debugger still lacks a method of distinguishing between whether a pointer points to a single object or an array of objects. Overridden `new` and `delete` operators like those used in the original Visual Debugger may be used to monitor this type of information, but it seems perfectly reasonable that a debugger like the .Net Debugger keep track of that information.

The direct access and the use of Visual C# instead of Visual Basic have also greatly increased the efficiency of the tool. Update time is now negligible even as the number of nodes in the system exceeds he upper limit of what the original Visual Debugger could reasonably handle.

5.4 Model-View-Controller (MVC)

The Model-View-Controller software architecture focuses on separating the data model, the view or views of that data model, and the control of that data and the views.

To provide the user with multiple, coordinated views such as those in the Data Structure Visualizer and the Visual Debugger 2.0, the Model-View-Controller architecture is the most appropriate architecture to use. It prevents unnecessary data duplication while still providing the user the flexibility of multiple, independent displays.

Specifically, a single data structure is maintained that reflects the data in the user's data structure, retrieved through an underlying debugger (in VDB2's case, the Microsoft Visual Studio .Net Debugger). The objects containing this data structure information includes hooks for notification about changes in the corresponding data. Depending on the programming language, the format of the hooks may be different (Java uses "listeners," C# uses "events" and "delegates"), but the basic event-based concept is the same. For example, when an update occurs that causes a change in the value of a variable, an event is fired that notifies all interested parties of the change.

The interested parties are the multiple views that each connect to the same data. Each view registers itself to receive events about changes in the underlying data that it is assigned to visualize, and when these changes occur, corresponding changes are made in the display.

The controller part of the MVC is manipulated by the update method and user interaction. When an update is performed, a chain reaction occurs that first causes the data to change, which in turn causes the visual aspects of the program to change. Similarly, when the user changes something in the view, such as the value of a variable, the controller makes sure that the change propagates to the data model as needed.

This general software architecture allows the Visual Debugger 2.0 to easily maintain multiple views while worrying about the underlying data totally separately. This makes it especially easy to improve the visualization of the tool as new ideas are developed to improve the tool. It is also much more efficient than copying the underlying data for each view.

5.5 Pluggable Graphical Value Representation

The Visual Debugger 2.0 includes support for graphical value representations, specifically for using bars and color shades to help compare relative values of related variables. The object-oriented nature of VDB2 allows these types of value visualizations to be easily created and added to the tool.

First, it is clear that the easiest visualizations are going to involve the primitives contained by the nodes and not the nodes themselves. It is certainly possible to create visualizations that involve an entire node or a group of nodes, but the Visual Debugger focused only on visualization of primitive data values such as integers, floating point numbers, and strings.

To better facilitate this kind of primitive data type visualization, a base class was developed that acts as the super-class for any classes that wish to act as visualization widgets for primitive values. Specifically, there is an abstract `PrimitiveViewBase` class and a simple `TextPrimitiveView` class that provides the default, text-only representation of a primitive data type. There is also a `PointerPrimitiveView` class that extends from `TextPrimitiveView` (since it also needs to display the textual address of the object that it points to) and adds an arrow to the node to which the pointer points.

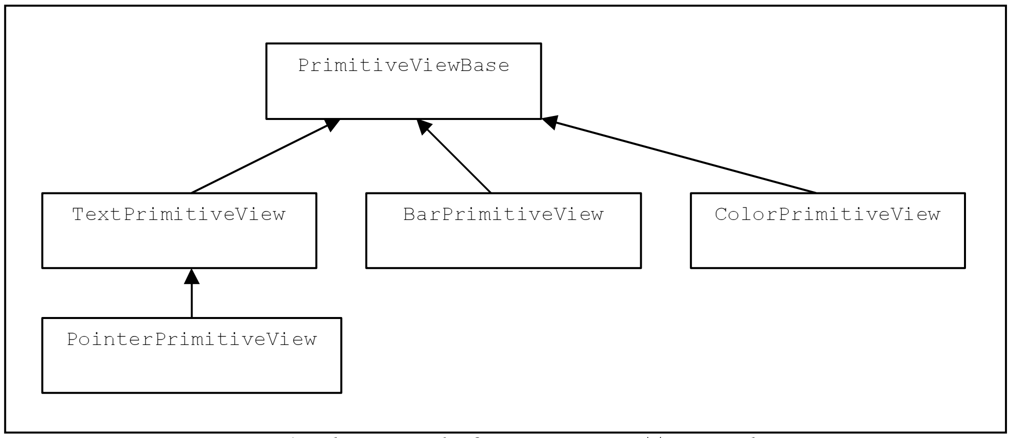


Figure 52: Class Hierarchy for Primitive Data Type Visualizers

Figure 52 shows the class hierarchy used to allow the user to select the graphical value representation type. As long as a class is a subclass of `PrimitiveViewBase`, it can be used to visualize primitives. Notice that the Bar and Color classes do not extend `TextPrimitiveView`, even though they both also use text. This stems from the inability in C# to draw on top of Textbox widgets, which are used in `TextPrimitiveView`.

The class hierarchy makes it easy to implement pluggable visualizers. The user can even switch visualizers for individual node variables, if they so desire. Whenever the user selects a new visualization type, the old visualizer object is destroyed and replaced with an instance of the new type.

An interesting side effect of this object-oriented architecture and open source is that users can implement their own visualizers and plug them into the existing tool extremely easily. A little knowledge of C# allows the user freedom in visualizing their data that is bounded only by their imagination and their skill with C#.

5.6 Pluggable Layout Algorithms

In much the same way that the architecture of the Visual Debugger 2.0 allows pluggable visualizations of primitive data types, the Data Structure Visualizer provides an interface for pluggable graph layout algorithms.

The key lies in the Layouter interface, which defines a single key method. The method, called `layout()`, takes an array of nodes as the only argument, and the method should modify the locations of each of the nodes based on the algorithm defined in the method. There are also methods for defining the name and description of the layout algorithm.

Two built-in implementations of `Layouter`, `ListLayouter` and `TreeLayouter`, are included in the Data Structure Visualizer, and lay out graphs as their names suggest. Their existence does not force the user to use either. If the user chooses, he or she may write their own Java file to implement the `Layouter` interface. In fact, a base class is provided, `LayouterBase`, that includes useful helper functions that may assist the layout developer.

When the Java file is written, the user need only compile the source file into a Java Class file and load the .class file in the Layout menu (Figure 53). The List and Tree layout techniques are already included in the Layout menu, and the user's custom layout algorithm will join them.

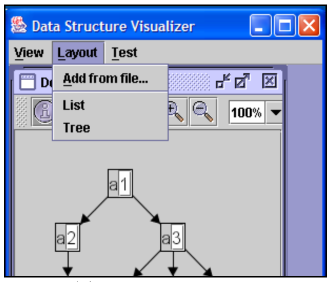


Figure 53: The Layout menu of the DSV

5.7 VDB2 Design Details

5.7.1 Backend Database

During a debugging session, VDB2 maintains a database of all of the variables in the debugged program, as well as their values at the last update. The visual debugger's database consists of a single object of type DebugTree, which contains all of the information needed to draw VDB2's views (Figure 54). Its most important part is the directed graph that contains information about all of the variables accessible by the debugged program.

Every node in the general graph is of type DebugNode and represents a variable or class in the user's data structure. The DebugNode class is an abstract class that includes information about the address of the variable, the type of the variable as a string (i.e. integer, string, or some user-defined type), the name of the variable, and its state (i.e. new, unchanged, modified, etc.), all current as of the last update. The subclasses of DebugNode are ContainerDebugNode and DisplayableDebugNode. A ContainerDebugNode is a DebugNode that contains (through aggregation) one or more DisplayableDebugNode objects. The

DisplayableDebugNode class gets its name from the fact that it is a node in the general graph that is the DebugTree data structure, and it is a relatively primitive data type that can be directly displayed. For example, integers, floating point values, and strings can be directly displayed by using their text values. Only objects of type DisplayableDebugNode have a value, which is stored as a string.

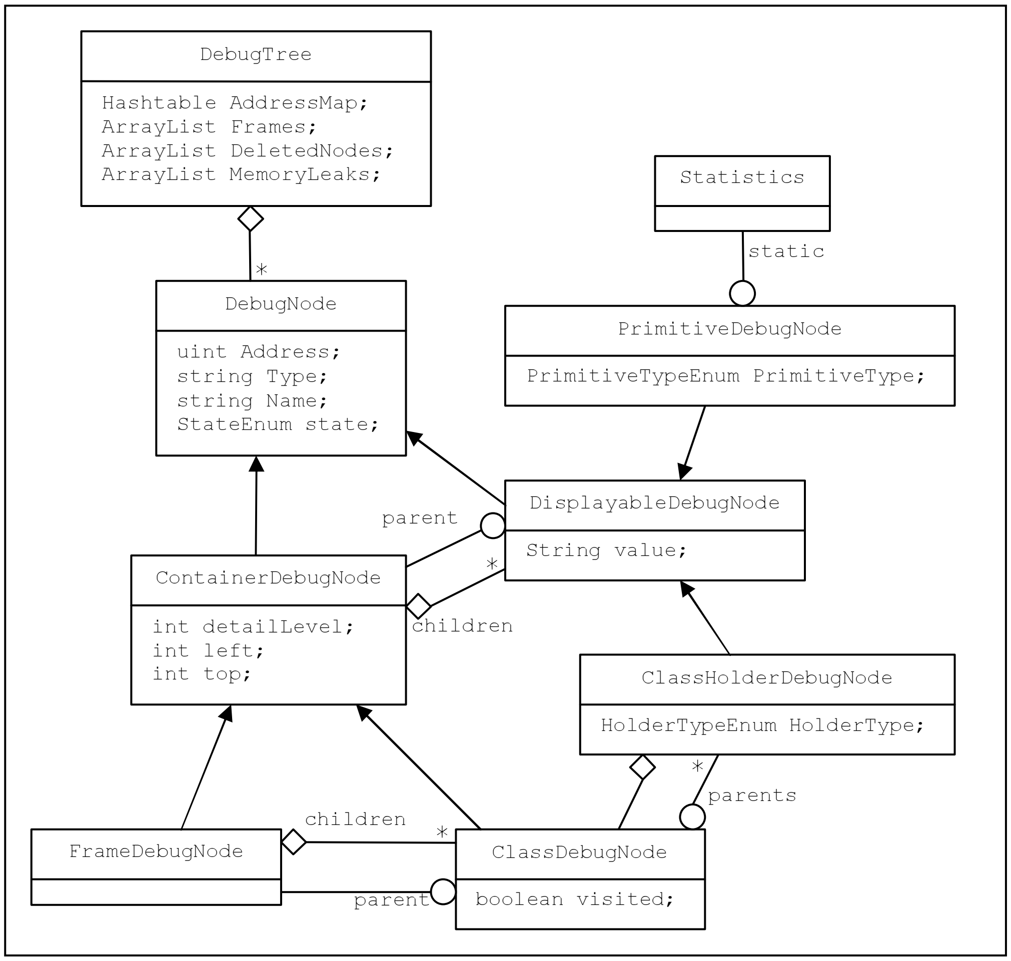


Figure 54: Class hierarchy of VDB2's backend database

Classes in the debugged program are represented as a subclass of ContainerDebugNode called ClassDebugNode, since classes contain member variables (which may themselves be primitives or other classes). Just as any other ContainerDebugNode, each ClassDebugNode contains one or more

DisplayableDebugNode objects. ContainerDebugNode objects also store coordinates, which correspond to the location on the drawing canvas to which the node belongs.

DisplayableDebugNode is an abstract class with two subclasses: PrimitiveDebugNode and ClassHolderDebugNode. The former represents a primitive data type such as an integer or string. The latter represents any variable that leads to a class. This variable can be either a pointer to a class, a reference to a class, or a class itself. In any case, the ClassHolderDebugNode itself contains a single ClassDebugNode object representing the object to which it refers.

DebugTree includes several lists, each of which consist of objects of type DebugNode. The most important of the lists is called Frames, which is an array of all of the frames on the program frame stack, each frame of which is represented by a subclass of DebugNode called FrameDebugNode. FrameDebugNode is a sister class of ClassDebugNode, almost identical except that a FrameDebugNode cannot have a parent; they are the only roots of the directed graph. Each FrameDebugNode contains stack variables, including primitives, classes, references to classes, and pointers to classes.

In addition to the Frames list, there are several other lists of DebugNode objects. The AddressMap contains references to all DebugNode objects, mapping them to the address at which they are located in memory. This allows the DebugTree to quickly access a DebugNode given a particular address, which is especially important in identifying whether or not a given address has been visited and recorded by the update method.

The DeletedNodes and MemoryLeaks lists store all DebugNodes which represent variables that are no longer accessible by the debugged program. Since these variables are not accessible, they cannot be stored in the DebugTree graph itself, since there are no references to these variables. This apples only to variables that are allocated on the heap, since all stack variables are already included in FrameDebugNode objects. Once a variable is determined to be inaccessible, it is removed from the DebugTree data structure and placed in the DeletedNodes or MemoryLeaks list, depending on whether the variable was properly deleted.

5.7.1 User Interface

The user interface of VDB2 is based around the MainForm class (Figure 55), which is the class with direct communication with the Microsoft Visual Studio .Net Debugger (through the Connect class).

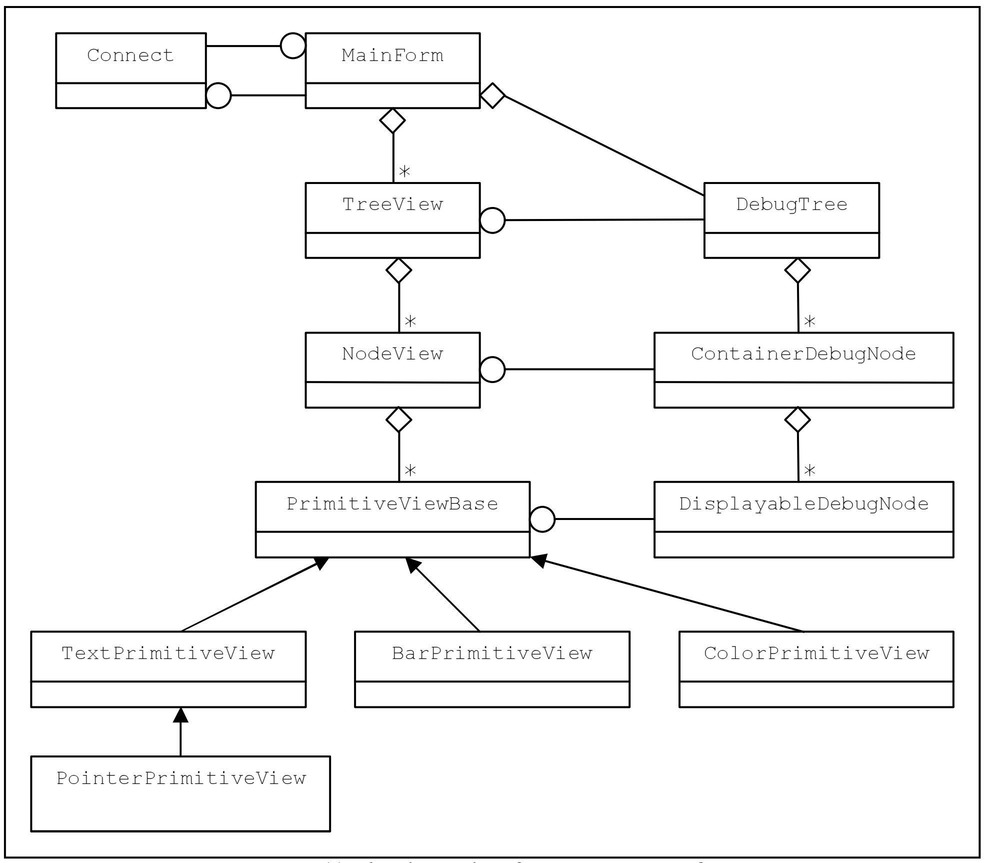


Figure 55: Class hierarchy of VDB2's user interface

The MainForm class is the main display window of VDB2, called the Data Structure Visualizer window. The Connect object signals the MainForm object whenever a breakpoint occurs, so that the MainForm object can update its DebugTree object.

The MainForm class contains any number of sub-windows, each of which is an instance of class TreeView. Each TreeView object connects directly to its MainForm parent's DebugTree object, so that all TreeView objects are graphical representations of the same data.

A TreeView object contains one or more NodeView objects, which correspond to the ContainerDebugNode objects in the DebugTree. Each ContainerDebugNode is represented by a single NodeView object in a particular TreeView, and it appears as a box containing information about the data members in the ContainerDebugNode (Figure 56). The data members are represented by PrimitiveViewBase objects, which are contained by the NodeView object.

Since all of the user interface objects (TreeView, NodeView, and PrimitiveViewBase) correspond to the same objects in the DebugTree data structure, all updates to the DebugTree graph are automatically reflected in the user interface objects. Once a change is detected, the user interface objects are refreshed to reflect the change. Since coordinate information is also stored in the DebugTree, each NodeView object in a given TreeView object remains in the same coordinate with respect to their corresponding NodeView objects in other TreeView objects. In this way, each of the TreeView objects represents the same image of the data structure, but each TreeView object can be scrolled or panned to focus on a different part of the data structure.

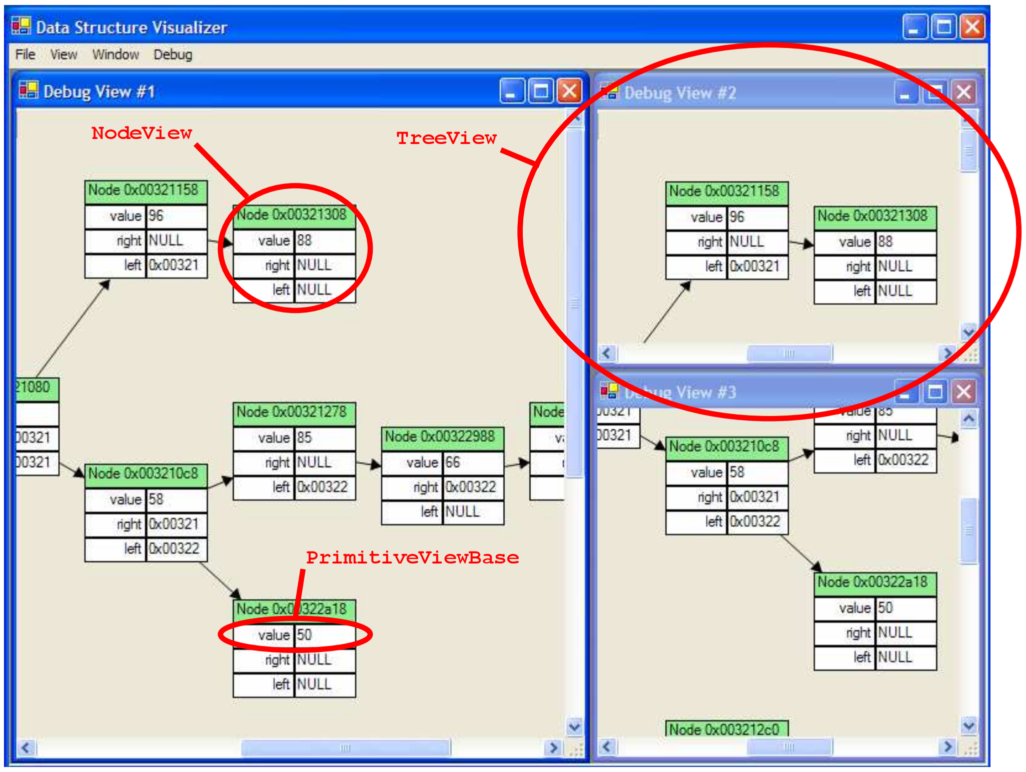


Figure 56: The user interface classes

5.8 Summary

Clearly, visual debugging is more involved that standard text-based debugging, as it requires the functionality of the latter (usually provided through an underlying debugger) plus additional functionality to support the graphical nature of the visual debugger. Particularly, the visual debugger should not simply display only the variables that the user specifies, as this requires far too much of the user. A data structure is made up of many variables, and if the data structure is of any non-trivial size, it is likely that the programmer will not easily be convinced to provide the necessary information for all of those variables. Yet, it is important to represent truly the data structure in its entirety, so a visual debugger must be able to collect the information.

Other than connection with an underlying debugger, typical programming concepts like Model-View-Controller and object-oriented programming allow for the relative ease of creation, flexibility, and extension of a usable visual debugger.

While the Microsoft Visual Studio .Net debugger is clearly a significant step in the right direction from its Microsoft-bred predecessors, the API can still be improved to better facilitate the addition of a visual debugger such as VDB2. What would be most welcome is a more comprehensive API for querying the structure of an object; current support is rudimentary and lacks sufficient and appropriate differentiation between objects, references to objects, and pointers to objects. Also, functionality for recognizing state of an object (new, modified, deleted, etc.) is absent, requiring the visual debugger to determine and store this information on its own.

Despite these technical issues, none are show-stoppers, and VDB and VDB2 have demonstrated that use of an underlying text-based debugger such as those in the Microsoft Visual Studio suite of integrated programming environments can be used to produce visual representations of a user's data structures with minimal effort on the part of the user.

Chapter 6

CHAPTER 6: EVALUATION

As a simple demonstration of the positive effects of information visualization on visual debugging, a small pilot study was conducted. The evaluation aims to be focused, and thus its purpose is to provide a brief glimpse at the potential benefits of a single aspect of using information visualization techniques on a single aspect of visual debugging. The size of the study certainly does not warrant a hard conclusion, but should pave the way for future studies that can test more formally the exact effects of information visualization on visual debugging.

The evaluation is related to studies performed by Cleveland [6] in which he was tasked with studying the effectiveness of various visual parameters when used to indicate numerical data. His results ranked several visual parameters (such as length, position, rotation, and color) in how well they could be mapped by humans to their numerical equivalents. For example, he found that position was a more effective visual parameter than color. This pilot study will focus similarly on how well users can identify errors based on the use of text (which uses no visual parameters), bars (which use position/length as a visual parameter), or colors.

6.1 Test Focus

One key element of debugging of any sort, visual or otherwise, is the ability to find errors in a program. Bugs in a program cannot be fixed until they are first found. This doesn't necessarily mean locating the offending line of code, but instead discovering that there is a problem in the first place, which is the first step in fixing a bug.

Errors apparent in data structures are sure signs of errors in the underlying code that created the data structures. It follows then that bugs of a certain nature can be found by first looking at defects in the resulting data structures. The specific defining feature of these types of bugs is that they result in a clear, negative effect on parts of a data structure. Not all errors fall into this category—an infinite loop need not involve a data structure at all and is clearly a defect in code. Many errors, however, will have an effect on at least one data structure.

It would seem relatively important, therefore, to focus a visual debugger's attention on being able to assist a programmer in discovering errors in data structures. Increasing efficiency in finding errors in data structures increases the efficiency of finding errors in the underlying code. One way that debuggers can increase this efficiency is by providing feedback that is more informative (or at least more quickly informative) than the classic textual information that typical debuggers give to programmers.

The focus of this evaluation was to try to determine a rough estimate of the effects, negative or positive, that information visualization—specifically that of graphically representing primitive data types—has on a programmer's capacity for and efficiency in finding errors in data structures.

A simple, empirical, lab-based study using the Visual Debugger 2.0 was used to obtain this information.

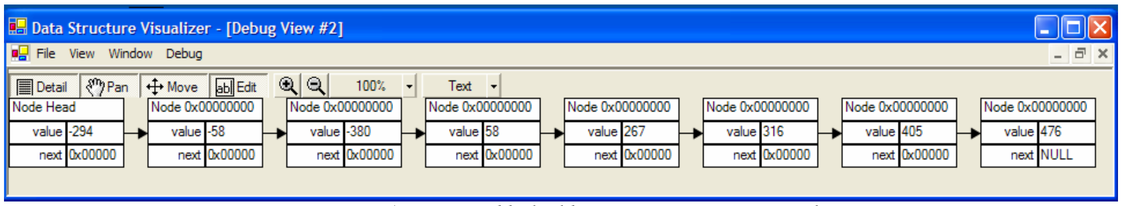
6.2 Test Plan

The critical parameter involved in measuring the user's efficiency in identifying errors in a data structure is the time required for the user to notice and locate the error. By controlling the visualization type and measuring response time (in locating the data structure anomaly) and accuracy (in identifying the exact change) as the dependent variables, the visualization methods can be compared to discover whether information visualization features have an effect on data structure error location.

First, each subject was given a pre-survey to determine how skilled the subject was with related topics such as computers, programming, debugging, graph theory, linked lists, and binary search trees. The target user base is made up of computer programmers, so the subjects were all relatively experienced with all but perhaps graph theory.

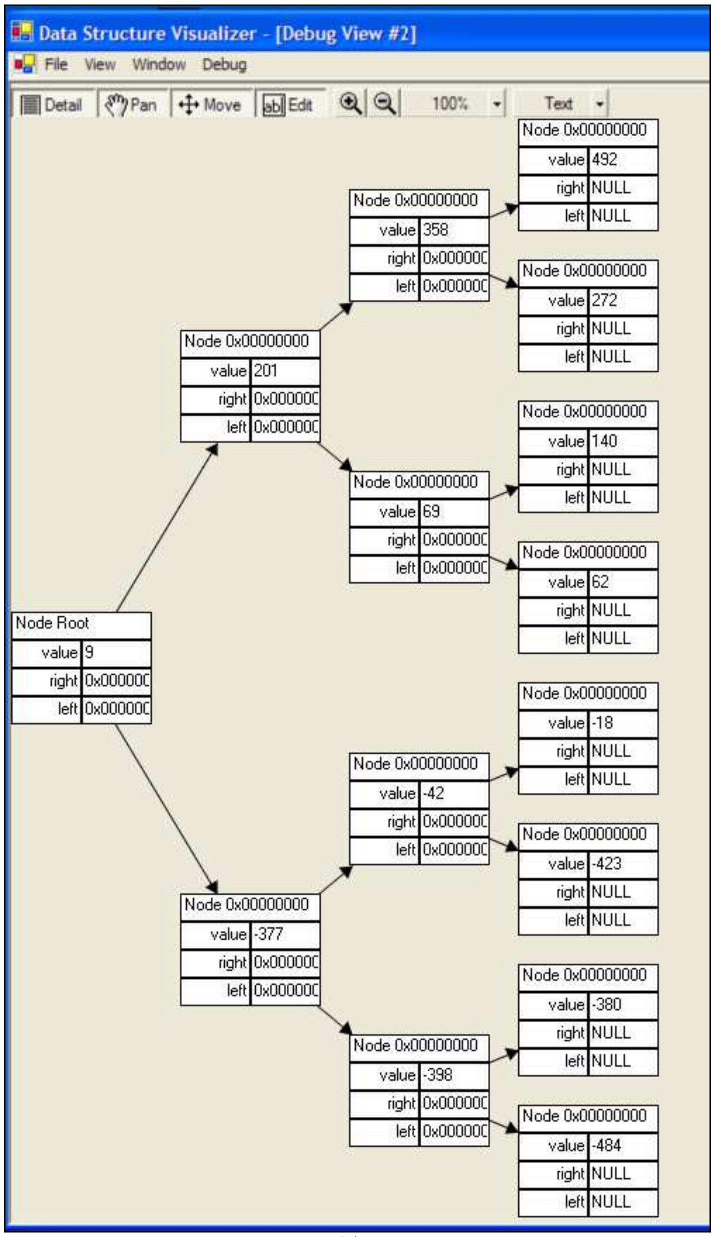
Each subject was placed in front of a computer running the Visual Debugger 2.0. A special version was developed for the pilot test. The visualization techniques remained unchanged, but code was added to monitor the users as they proceeded through the examination.

VDB2 presented the subjects with a series of screens. Initially, each screen consisted of VDB2 visualization of a linked list of eight nodes (Figure 57). The labels at the top of each node and the "next" variable are unimportant to the subject's task and serve only as a distraction that is likely to be typical in many visual debuggers. The subjects were told to focus on the "value" variable, the only variable whose value differed between nodes.



```
Node Head        Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000
value -294       value -58        value -380       value 58         value 267        value 316        value 405        value 476
next 0x00000     next 0x00000     next 0x00000     next 0x00000     next 0x00000     next 0x00000     next 0x00000     next NULL
```

Figure 57: A typical linked list response test, text mode



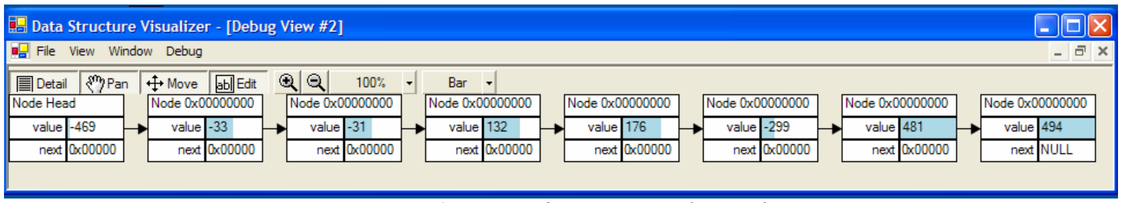
```
Node Root
value 9
right 0x00000C
left 0x00000C

Node 0x00000000       Node 0x00000000       Node 0x00000000
value 201             value 358             value 492
right 0x00000C        right 0x00000C        right NULL
left 0x00000C         left 0x00000C         left NULL

                                            Node 0x00000000
                                            value 272
                                            right NULL
                                            left NULL

                      Node 0x00000000       Node 0x00000000
                      value 69              value 140
                      right 0x00000C        right NULL
                      left 0x00000C         left NULL

                                            Node 0x00000000
                                            value 62
                                            right NULL
                                            left NULL

Node 0x00000000       Node 0x00000000       Node 0x00000000
value -377            value -42             value -18
right 0x00000C        right 0x00000C        right NULL
left 0x00000C         left 0x00000C         left NULL

                                            Node 0x00000000
                                            value -423
                                            right NULL
                                            left NULL

                      Node 0x00000000       Node 0x00000000
                      value -398            value -380
                      right 0x00000C        right NULL
                      left 0x00000C         left NULL

                                            Node 0x00000000
                                            value -484
                                            right NULL
                                            left NULL
```

Figure 58: A typical BST response test, text mode

The list was intended to be an *ordered* linked list of numbers ranging from -500 to 500, except that there was exactly one node in the list that destroyed the sorted property of the list. There was exactly one node such that, if that node had been removed from the list, the remainder of the list would have been in sorted order. The goal of the user was to find the anomaly node and click on it with the mouse. As soon as the anomaly node was found and clicked, a confirmatory beep sounded and the numbers were replaced by new values. This process was repeated a set number of times. Incorrect clicks were recorded to monitor error rate. An "incorrect click" is defined as a click on a wrong node—clicks on the background were not recorded as errors.

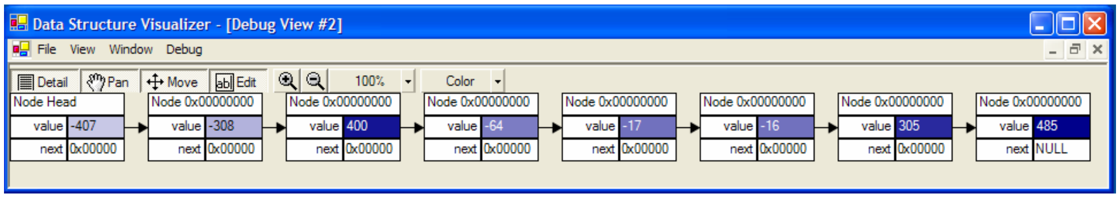
Following the series of linked list screens was a more difficult variation of the same test that used a complete binary search tree of fifteen nodes instead of a linked list (Figure 58). Again, each tree was supposed to be in sorted order, but a single node ruined the order. The user's goal was to locate and click the faulty node. The same rules applied as did during the linked list part of the test.

The pair of tests was repeated for both the bar (Figure 59) and color (Figure 60) visualization modes. The rules again remained the same.



```
Data Structure Visualizer - [Debug View #2]
100%   Bar

Node Head      Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000
value -469     value -33        value -31        value 132        value 176        value -299       value 481        value 494
next 0x00000   next 0x00000     next 0x00000     next 0x00000     next 0x00000     next 0x00000     next 0x00000     next NULL
```

Figure 59: A typical response test, bar mode



```
Data Structure Visualizer - [Debug View #2]
100%   Color

Node Head      Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000  Node 0x00000000
value -407     value -308       value 400        value -64        value -17        value -16        value 305        value 485
next 0x00000   next 0x00000     next 0x00000     next 0x00000     next 0x00000     next 0x00000     next 0x00000     next NULL
```

Figure 60: A typical response test, color mode

Consequently, there were three sections (corresponding to the three visualization types), each of which had two sub-sections (the linked list and the binary search tree). In a more formal examination, each subject would execute the six total sub-sections in random order, to balance the learning effect. Due to the small size of the pilot test, however, the order was assigned to assure fairness to each visualization mode.

Each sub-section consisted of ten practice tests, followed by twenty recorded tests.

The modified version of the Visual Debugger 2.0 recorded the time for each click in each sub-category (including practice tests), the number of errors received during the sub-category, and the average click time for the twenty non-practice tests.

The pilot test was performed on four subjects, each of which account for twenty data points, resulting in eighty data points for each test category. The subjects were recruited locally from the Computer Science Department at Virginia Tech.

6.3 Test Results

Figure 61 shows a summary of the results of the pilot test.

Test	Avg. Click Time	Error Rate*
Text – List	6.0442	0.075
Bar – List	5.9505	0.3
Color – List	5.6433	0.2
Text – Tree	11.1361	0.3125
Bar – Tree	10.5108	0.25
Color – Tree	8.2218	0.2375

* Error Rate = Ratio of error clicks to correct clicks

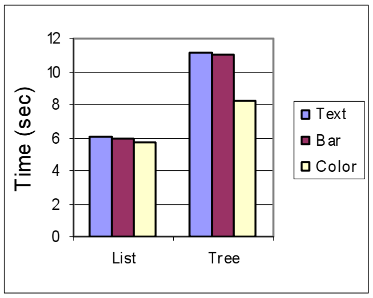


Figure 61: Summary of pilot test results

Although the results favor the information visualization methods (bar and color) with respect to response time in all categories, the results for linked lists are too close to make any definitive judgment, especially given the size of the test. The results for the binary search trees were more revealing.

For all users, the binary search tree task was much more difficult than the linked list task. This is to be expected, as the tree included nearly twice as many nodes in a more scattered arrangement than the simple linear layout of the linked list.

According to the average click times for trees, the bar visualization mode performed only slightly better then the plain text mode, but the color visualization mode performed more than 20% better than both of them.

A 2-way ANOVA analysis reveals that both test type (list or tree) and visualization type (text, bar, or color) have a significant effect (test type, $p<0.0001$; visualization type, $p<0.0301$) on response time. The significant effect of test type makes sense—the tree was invariably more difficult that the list.

For lists, visualization type had a significant effect ($p=0.0441$) on response time. The response times for the bar visualization mode were not significantly less than those for text ($p=0.4543$). The response times for color were not significantly less than those for text ($p=0.2235$). The response times for color were not significantly less than those for bar ($p=0.3536$).

For trees, visualization type had a significant effect ($p=0.0172$) on response time. The response times for the bar visualization mode were not significantly less than those for text ($p=0.2475$). The response times for color were significantly less than those for text ($p=0.0064$). The response times for color were significantly less than those for bar ($p=0.0011$).

The statistical results reveal that, while the color visualization mode resulted in significantly faster user response for more complex data (trees, as opposed to lists), there were no other significant differences.

Error rates favored text mode by far in the linked list task, but both information visualization modes beat out text mode in error rates for the tree task. This may imply that the information visualization techniques are useful in reducing errors in analysis of more complex data structures.

A 2-way ANOVA analysis reveals that only test type (list or tree) had a significant effect ($p<0.0486$) on error rate. The significant effect of test type makes sense—the tree was invariably more difficult that the list.

6.4 Discussion

The subjects expressed interest in the information visualization aspect of the tool, but voiced complaints about some minor bugs experienced during the bar and color visualization mode tests. These bugs may have affected the results, reducing the response times and increasing the error rates for the bar and color modes.

During and after testing, users expressed their opinions of the tool, with and without the bar/color information visualization modes. They mentioned that the tool was useful for identifying that there is a bug in a program and for locating the bug in the data structure, but was less useful during the process of actually fixing the bug. This may be due to the external nature of the tool itself—it gives a good picture of the data structure as a whole, but does little to illuminate the code behind the data structure, save to notify the user of the effects of a line of code.

Overall, the results are promising for this specific information visualization concept, graphical data representation, concluding that it can to increase the programmer's ability to more quickly locate errors in data structures, at least for complex data structures. The more quickly an error can be located, the more quickly it can be solved. The results of this experiment imply on a broader scale that information visualization concepts can assist the programmer in validation testing and in debugging. Whether or not other information visualization concepts fare as well as graphical data representation will require testing focused on those other information visualization concepts.

With respect to the difference between information visualization modes and plain text mode, users mentioned that the bars and colors were more helpful in locating major anomalies in data structures than the plain text mode. Of the four users, three preferred using the bar or color modes to using the plain text mode.

Chapter 7

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

1. The following information visualization techniques can be adapted to the domain of visual debugging to improve the visual debugging experience: graphical value representation, zoom and pan, Overview+Detail, Focus+Context, direct manipulation, layout, and visual notification.

2. Visual Debugging requires more in-depth implementation than that of a text-based debugger, but using the latter as an underlying debugger can make implementation of a visual debugger easier.

3. Information visualization features such as those described can be implemented without requiring additional input from the user. Specifically, these features can be added with requiring the user to modify their code or to describe to the visual debugger the structure of their user-defined classes.

4. Based on a small pilot study, information visualization techniques have been shown to increase efficiency in detecting errors, a key step in fixing bugs.

7.2 Summary of Contributions

1. **Design.** A list was enumerated of seven information visualization techniques that can be added to and should be able to improve today's visual debuggers, and how these techniques can be used in visual debuggers. Benefits and consequences of individual techniques as well as combinations of techniques

were provided. In addition, a design was developed to combine all of these techniques into a single user interface.

2. **Architecture.** General visual debugging issues were described, a list was enumerated of several software design techniques that are helpful in implementation of a visual debugger, and issues specific to incorporation of a visual debugger into real commercial debugging software (e.g. Microsoft Visual Studio) were discussed.

3. **Evaluation.** A small pilot test was conducted to demonstrate the potential of information visualization in visual debugging.

4. **Implementation.** Three tools were developed that demonstrate the ability to impart information visualization concepts onto the visual debugging domain. VDB is available for download at http://infovis.cs.vt.edu/datastruct/, and VDB2 will soon be available for download at the same location.

7.3 Future Research

1. The enumerated list of information visualization techniques is certainly not exhaustive. As new ideas are developed in the relatively new field of information visualization, many of these may play well into the hands of visual debugger developers. As these concepts arise, they too should be studied to determine their potential in the visual debugging domain.

2. The Microsoft Visual Studio suite of debuggers is not the only debugger with which a visual debugger may be integrated. In fact, other visual debuggers have been developed on top of several other debuggers. These visual debuggers may be enhanced, or new visual debuggers created, to expand information visualization to debuggers other than Visual Studio's.

3. There is much formal evaluation work that should be done to better understand the full effects of information visualization concepts on visual debugging. The pilot test was a simple demonstration that should be expanded on through larger and more formal testing. For instance, VDB2 can now be used in longitudinal studies of data structures students to discover the long-term effects of using such a visual debugger.

4. While the Visual Debugger 2.0 is intended to implement them all, it is still only a debugger, ideally to be included with an Integrated Development Environment such as Microsoft Visual Studio. Though VDB2 is tightly integrated with Visual Studio as an AddIn, a visual debugger built directly into Visual Studio is certainly an area of research that could be developed.

5. User response during the empirical evaluation revealed that the tool was much less useful in fixing bugs than it was for finding bugs. Something should be done to increase the user's ability to use the Visual Debugger 2.0 to fix bugs after the user has noted their existence. This may be done by increasing the association between a data structure and parts of the code that manipulate it. For example, clicking on a member variable of an object in the visual representation of the data structure could highlight the specific lines of code that refer to that member variable.

6. Variations of the graphical data representation technique can easily be extended. The visual debugger currently employs only bar and color modes, but a visual debugger might use a combination of the bar and color techniques, using colored bars of varying shade *and* width. A particular type of variable could use *two* colors instead of one—one for positive values and one for negative values. Circular, pie-chart-like visualizations can be concocted, or even histograms can be used for more complex data.

7. The Visual Debugger 2.0 currently focuses only on visualizations that represent primitive data values such as integers and floating point values, but with enough imagination, one can come up with visualization techniques that can be applied to entire nodes or groups of nodes. For example, if a graphic could be conjured as a suitable representation for a node of a particular state, where the state is derived from a sequence of multiple member variables, then that graphic could substitute or add to the node and its data in much the same way that the bars and colors supplement the textual values of the integer values.

REFERENCES

[1] S. Baase and A. V. Gelder. *Computer Algorithms: Introduction to Design and Analysis*, Third Edition. Addison Wesley Longman, Reading, MA. Apr. 2000.

[2] R. M. Baecker and A. Marcus. "Design Principles for the Enhanced Presentation of Computer Program Source Text". In *Proceedings of CHI '86, Human Factors in Computing Systems*. ACM Press. 1986.

[3] B. B. Bederson, et. al. "Pad++: A zoomable graphical sketchpad for exploring alternate interface physics." J. Vis. Lang. Comput. 1996.

[4] M. H. Brown and R. Sedgewick. "A system for algorithm animation." In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 177-186. ACM Press. 1984.

[5] A. E. R. Campbell, G. L. Catto, and E. E. Hansen. "Language-independent interactive data visualization." In *Proceedings of the 34th technical symposium on Computer science education*, pages 215-219. ACM Press. 2003.

[6] W. S. Cleveland and R. McGill. "Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods." In *Journal of the American Statistical Association*, pages 531—554. Sept. 1984.

[7] C. Cook, M. Burnett, and D. Boom. "A bug's eye view of immediate visual feedback in direct-manipulation programming systems." In *Papers presented at the seventh workshop on Empirical studies of programmers*, pages 20-41, Alexandria, VA. ACM Press. 1997.

[8] D. Dee-Lucas and J. H. Larkin. "Learning From Electronic Texts: Effects of Interactive Overview for Information Access." In *Cognition and Instruction*, pages 431-468. 1995.

[9] G. W. Furnas. "Generalized fisheye views." In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16-23, Boston, MA. ACM Press. April 1986.

[10] C. Gutwin. "Improving focus targeting in interactive fisheye views." In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 267-274. ACM Press. 2002.

[11] D. R. Hanson and J. L. Korn. A Simple and Extensible Graphical Debugger. In *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 174-183, Anaheim, CA. Jan. 1997.

[12] M. C. Kasten. Data Structures in COBOL. Mar. 18, 2002. <http://home.swbell.net/mck9/cobol/tech/struct.html>

[13] E. Koutsofios and S. C. North. Applications of Graph Visualization. In *Proceedings of Graphics Interface*, pages 235-245, Banff, Alberta, Canada. ACM Press. 1994.

[14] C. Laffra and A. Malhotra. "HotWire -- A Visual Debugger for C++." In *USENIX Sixth C++ Technical Conference*, Cambridge, MA. 1994.

[15] Y. K. Leung and M. D. Apperley. "A review and taxonomy of distortion-oriented presentation techniques." In *ACM Transactions on Computer-Human Interaction*, pages 126-160. June 1994.

[16] S. C. Locke. Graph Theory. Sept. 1, 2001. Mar. 18, 2002. <http://www.math.fau.edu/locke/graphthe.htm>

[17] S. McCrickard. "Attuning notification design to user goals and attention costs." In *Communications of the ACM*, Volume 46, Issue 3, pages 67-72. ACM Press. 2003.

[18] H. Nascimento. "A framework for human-computer interaction in directed graph drawing." In *Australian symposium on Information visualization*, pages 63-69, Sydney, Australia. 2001.

[19] W. Pierson and S. H. Rodger. Web-based Animations of Data Structures Using JAWAA. In *Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, pages 267-271, Reno, Nevada. ACM Press. 1998.

[20] R. Rao and S. K. Card. "The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information." In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*. ACM Press. April 1994.

[21] P. Roy and R. St. Denis. "Linear Flowchart Generator for a Structured Language". ACM *SIGPLAN Notices*, pages 58-64. ACM Press. 1976.

[22] P. Saraiya. *Effective Features of Algorithm Visualizations*. Thesis. August, 2002.

[23] C. Shafer, L. S. Heath, and J. Yang. "Using the Swan Data Structure Visualization System for Computer Science Education". In *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium of Computer Science Education*, pages 140-144, Philadelphia, PA. ACM Press. 1996.

[24] N. C. Shu. *Visual Programming*. New York: Van Nostrand Reinhold Co, 1988.

[25] J. T. Stasko. "TANGO: A Framework and System for Algorithm Animation." *IEEE Computer*, 23(9), pages 27-39. 1990.

[26] J. T. Stasko. "Animating Algorithms with XTANGO." *SIGACT News*, 23(2), pages 67-7 1. 1992.

[27] J. T. Stasko and Eileen Kraemer. "A Methodology for Building Application-Specific Visualizations of Parallel Programs." In *Journal of Parallel and Distributed Computing,* 18(2):258-264. June 1993.

[28] J. T. Stasko and J. F. Wehrli. "Three-Dimensional Computation Visualization," In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 100-107, Bergen, Norway. August 1993.

[29] J. T. Stasko and D. F. Jerding. "Using Visualization to Foster Object-Oriented Program Understanding." Technical Report GIT-GVU-94/33, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA. July 1994.

[30] J. T. Stasko and S. Mukherjea. "Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger." In *ACM Transactions on Computer-Human Interaction (TOCHI)*, Volume 1, Issue 3, pages 215-244. September 1994.

[31] J. T. Stasko and D. F. Jerding. "The Information Mural: A technique for displaying and navigating large information spaces." In *Proceedings of the IEEE Visualization '95 Symposium on Information Visualization*, pages 43-50, Atlanta, GA. October 1995.

[32] J. T. Stasko. "Using Student-Built Algorithm Animations as Learning Aids." Technical Report GIT-GVU-96/19, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA. August 1996.

[33] J. T. Stasko and D. F. Jerding. "Visualizing interactions in program executions." In Proceedings of the 19th international conference on Software engineering, pages 360-370, Boston, MA. ACM Press. 1997.

[34] D. Tunkelang. A Practical Approach to Drawing Undirected Graphs. *Technical Report CMU-CS-94-161*, Carnegie Mellon University, School of Computer Science. 1994.

[35] M. Wiggins. "An overview of program visualization tools and systems". In Proceedings of the 36th annual Southeast regional conference, pages 194-200. 1998.

[36] A. Zeller. *Debugging with DDD.* Jan. 3, 2000. Free Software Foundation, Inc. Mar. 18, 2002. <http://www.gnu.org/manual/ddd/html_mono/ddd.html>

[37] A. Zeller and D. Lütkehaus. DDD — A Free Graphical Frontend for UNIX Debuggers. *ACM SIGPLAN Notices*, 31(1):22–27. ACM Press. Jan. 1996.

[38] Using the GNU Visual Debugger. Jan. 24, 2002. ACT-Europe. Mar. 18, 2002. <http://libre.act-europe.fr/gvd/gvd.html>