SOFTVIZ: A RUNTIME SOFTWARE VISUALIZATION ENVIRONMENT

by

Benjamin Kurtz

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

APPROVED:

_____	_____
Professor George T. Heineman, Advisor	Professor Matthew Ward, Advisor
_____	_____
Professor David Brown, Department Reader	Professor Micha Hofri, Head of Department

Computer Science Department
Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609
{bk2@alum.wpi.edu}

Abstract

As software systems become more complex, so does the task of understanding them. To modify even a simple component of a complex system, at least a rudimentary understanding of the structure and behavior of the whole system is necessary. Although currently available development tools can provide a static representation of a complex system, these utilities are limited and prohibitively expensive. As a result, most programmers working on large software systems today resort to classic debuggers and time-consuming plain-text searches through hundreds or thousands of source files. This thesis describes a software development environment that uses static representations of hierarchically structured source code side by side with dynamic visualizations of software systems as they run. This environment provides an intuitive, visual means of easily comprehending complex systems, and has been provided as an open-source development tool for both professionals and students of software engineering.

Table of Contents

List of Figures

Acknowledgments

I would like to thank Dr. George Heineman, whose guidance and contagious enthusiasm made this thesis possible.

I would also like to thank Dr. Matthew Ward and Dr. David Brown for their vital input and feedback.

Special thanks to Jing Yang for showing me her implementation of the Sunburst visualization.

I would be remiss if I did not acknowledge the support of my friends Paul Calnan, Charles McAuley, and Leonard Frank, who have given me so much and asked for so little in return.

1 Introduction

1.1 Motivation

Modern software engineers working on large software systems are faced with the daunting task of understanding enough about the structure and behavior of the system to correctly modify the system. Software systems involving thousands of source files and millions of lines of code are becoming commonplace, and system documentation is almost always a poor reflection of the actual system implementation, since it is not always properly updated when changes occur.

Armed only with debugger stack traces and a plain-text searching tool such as `grep`, a newly hired developer may need several frustrating and unproductive months to begin to comprehend how the system functions as a whole. Developers experienced with the structure of the system may find it difficult and time-consuming to trace its behavior adequately for debugging and optimization purposes. By minimizing the man-hours required for a developer to understand the structure and behavior of a system, the development cost may be substantially reduced.

Although many tools are available for the static analysis of object-oriented hierarchies, modern component-based software systems have additional properties that make useful static analysis difficult or impossible. First, complex component based systems may be dynamically reconfigured using components unknown to the designers of the system. This dynamic assembly and reassembly does not lend itself easily to static analysis. Additionally, the actual runtime behavior of a system may differ considerably from what might be reasonably deduced from a static hierarchical representation.

In addition to the challenges faced in debugging and optimizing, systems which can be dynamically reassembled present additional difficulties in maintenance. In these systems, certain runtime events generated by the system trigger architectural changes, such as the replacement of a particular component. In some situations, where there is a clear causal relationship between certain system events and particular architectural changes, these changes may be automated. In situations where there may be mitigating

factors or relevant external conditions unknown to the system, decisions regarding architectural changes must be made by a human being.

To provide them with the necessary information, these decision-makers require a tool that shows a user-configurable map of the system that tells the user "You are here."

1.2 What is Visualization?

WordNet [40] defines visualization as "a mental image that is similar to a visual perception". Visualization, perhaps counter-intuitively, is not the creation of visual images, but of mental images in the mind of the viewer. The goal is the creation of visual representations of large amounts of data that can be easily understood, even by viewers with limited technical knowledge. In visualizations, various attributes of the data set are mapped to visual attributes such as size, color, texture, or shape. Common examples of visualizations are bar charts, line graphs, maps, and organizational charts. The idea of visualization of numerical data is not new, as the following figure illustrates.

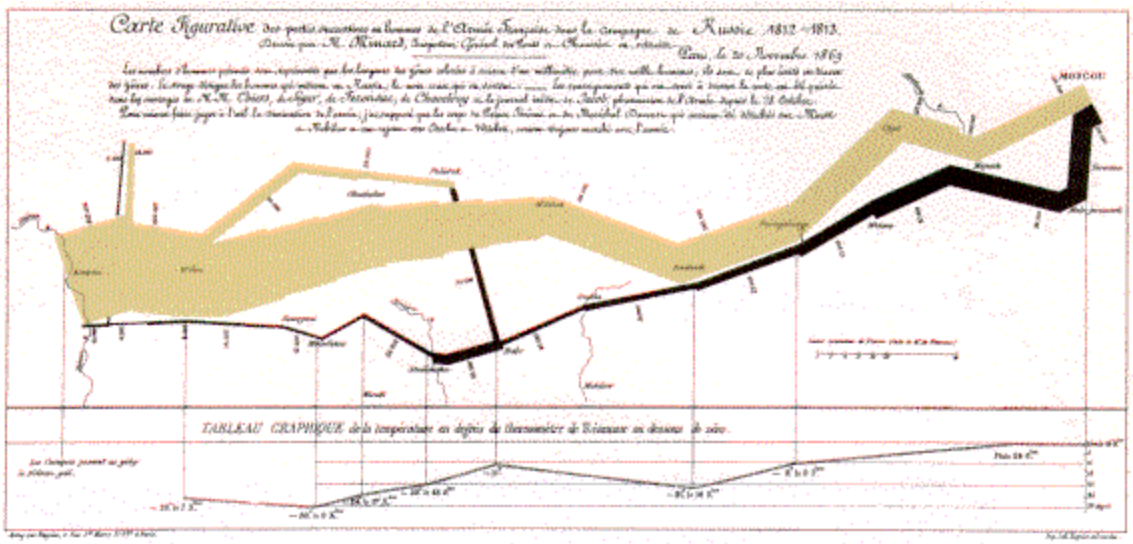


Figure 1.1 – Minard's Visualization of Napoleon's 1812 Russian Campaign [50]

This graph done by a French engineer in 1861 represents Napoleon's attempt to invade Russia in 1812 [50]. The width of the band measures the size of the French army, with beige representing attacking forces and black representing retreating forces. The line graph on the bottom records the temperature at each location the army passed through

during the retreat. With no other knowledge of the underlying data, the viewer can almost feel the horror of the French as their massive advancing beige band is reduced to a thin black line.

The example in Figure 1.1 demonstrates two desirable properties of visualizations: incorporate a large amount of information into a relatively small space, and use color, size, and spatial relationships effectively to convey information.

We can envision using the same type of visualization for any military campaign, or for any data set with similar properties. The same type of visualization may be appropriate for a stock portfolio's catastrophic performance in 1929, mapping the width of the band to monetary value, and the color beige to the time period preceding Black Tuesday.

We can envision an entire class of data sets for which Minard's graph would be appropriate. Each of these data sets has a corresponding Minard visualization, creating a class of visualizations defined by the mapping of data attributes to visual attributes. These mappings describe a *visualization template*, which can be instantiated with a suitable data set to create new instances of Minard's visualization. The concept of general-purpose visualization templates is central to our discussion of software visualization environments.

1.3 What is a Probe?

Probes are information gathering code segments inserted into strategic locations within the source code of the software system targeted for visualization. The target source code can be instrumented with probes manually, or the task can be automated. For our purposes, we are relying on probes that were developed for the Defense Advanced Research Projects Agency's (DARPA) Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA) program [14]. Although these probes can be customized to extract meaningful information from the target system, we used the standard probes supported by the Active Interface Development Environment (AIDE).

When a probe is activated, it generates an event that will be used by our software visualization system to create and animate run-time visualizations. In Chapter 2 we will

discuss probes in more depth, including an evaluation of alternative methods of extracting program data.

1.4 What is AIDE?

AIDE [24] is a modified pre-processor for Java source code developed by George Heineman at the Worcester Polytechnic Institute. The AIDE pre-processor can automatically instrument Java source code with probes that determine when a method is invoked and when it is completed, greatly reducing the time required for source code instrumentation. One can envision numerous other types of probes, for example, a probe that determines when an exception is generated or when an assertion is checked. We have used the available probes while remaining open to including new types of probes into the visualization environment.

1.5 What is SIENA?

SIENA (Scalable Internet Event Notification Architectures) is a middleware event notification package developed at the Software Engineering Laboratory at the University of Colorado [43]. Using SIENA allows our visualization environment to be decoupled from the underlying probe infrastructure; in addition, the visualization can execute on a separate machine from the software systems being visualized. SIENA, combined with the AIDE compiler, and the probe monitoring infrastructure encapsulated by the APIs developed for this project form the core of the environment's infrastructure.

1.6 Goals

By combining run-time monitoring with software visualization, we hope to create a development environment capable of delivering meaningful visual representations of the structure and behavior of multiple software systems to the user.

We have adhered to several design principles in developing our system, including:

- Scalability – the target system may range from simple programs to complex systems with millions of lines of code.

- Portability – the environment must run on many platforms and allow for multi-language software systems.
- Flexibility – users should be able to perform a wide variety of tasks, including debugging and optimization.
- Extensibility – the environment should allow for integration with existing software tools and for the addition of new user-defined visualizations.
- Openness – the environment should work alongside source control systems and integrated development environments.
- Adaptability – the environment must adapt to changing architectures and behaviors and provide for task-specific configuration.

We discuss these design principles in greater detail in Section 3.1.

1.7 What is SoftViz?

The *SoftViz* environment, designed and implemented for this thesis, provides a framework for the static visualization of the abstract structures of object-oriented software systems, as well as the dynamic visualization of the system's behavior. *SoftViz* uses a probe-based approach for the extraction of interesting runtime events, and then publishes these events using the SIENA wide-area event distribution bus. This configuration allows for the visualization system to be run remotely from the systems it is observing.

SoftViz can aide several target audiences, including the new developer trying to become familiar with a complex system, the experienced developer working on debugging and optimization, and the maintainer of a complex component-based system allowing dynamic assembly. These users all need a tool allowing them to easily analyze the runtime behavior as well as the static structure of their software systems. *SoftViz* meets these needs through the use of both static and dynamic visualizations.

1.8 Outline

This thesis is divided into six chapters. In Chapter 2 we examine other software visualization packages and the existing DASADA technologies that comprise the

infrastructure of *SoftViz*, as well as their underlying principles. In Chapter 3, we incorporate the principles of software engineering and visualization with DASADA technology into a high level design for a software visualization environment. In Chapter 4, we discuss the design and implementation of the *SoftViz* environment. In Chapter 5, we apply this environment to several case studies, and in Chapter 6 we discuss our final conclusions and recommendations.

2 Background

This chapter describes the history of software visualization, and presents a discussion of previous software visualization systems and general design considerations for a software visualization system.

Software Visualization refers to the formulation of a mental image of the structure or behavior of software. Visualizations of software can convey information about structure, memory allocation, processor utilization, algorithm efficiency, or many other aspects of software systems.

2.1 Existing Software Visualization Systems

The idea of achieving greater understanding of software through the use of visualization began more than 50 years ago, with the use of flowcharts to represent the behavior of programs [22]. In the 1950's and early 1960's, Knuth and others made advances in the automatic generation of flowcharts from program source code [23][33].

Interest in the static visualization of source code continued in the 1970's with the development of *pretty printing* [34] of source code, and Nassi-Shneiderman charts [38]. Pretty printing makes source code easier to read with the standardizing of spacing and indentation, and sometimes the use of colors or different fonts for keywords. Nassi-Shneiderman charts are more space-efficient than traditional flow-charts and can be easily understood. An example of a simple Nassi-Shneiderman diagram appears in Figure 2.1.

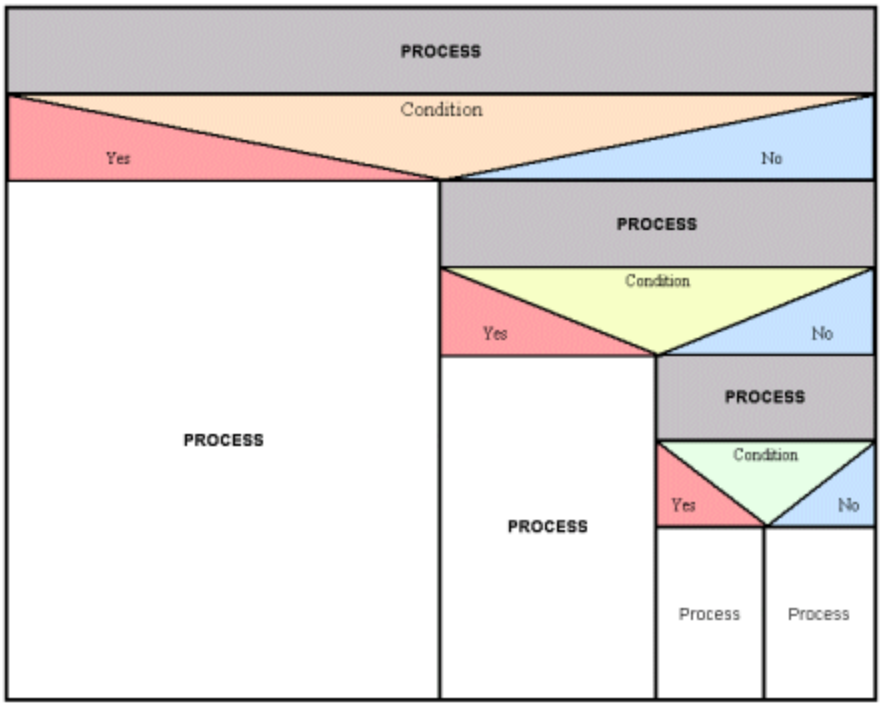


Figure 2.1 – A Nassi-Shneiderman Chart [42]

The first example of dynamic visualizations of a program running appeared in a series of films by Knowlton in 1966 [30][31][32], showing operations on lists in Bell Laboratories' Low-Level Linked-List Language (a.k.a. L[6]). These films were also the first to attempt the visualization of data structures, rather than just charting the control flow of the program. A 1981 film by Baecker, *Sorting Out Sorting* [1], showed dynamic visual representations of 9 different sorting algorithms individually, and then racing with each other to sort a large data set. All of these films visualize the behavior of algorithms, and not that of any particular implementation.

The traditional uses of software visualization are teaching, debugging, and presentation. The early algorithm animation tools, such as BALSA from Brown University [11][12], were used to demonstrate the relative efficiency of algorithms to classes of 50 students. Modern software visualization tools, such as Jinsight from IBM [26], allow software developers to locate memory leaks and perform other debugging and optimization tasks. Both the teaching tools and the debugging tools can be used to create easy-to-understand visualizations for presentations to conferences or managers. The

ANIM tool, developed at Bell Laboratories [2][3], provides the option of generating a still image from an animated visualization for use in presentations.

Since the beginning of the 1980's, many software visualization systems have been developed. To motivate our discussion of the *SoftViz* visualization system, we will now discuss several of the most historical or relevant software visualization systems.

2.1.1 BALSA and Balsa-II

Brown and Sedgewick developed the first fully interactive software visualization system, the Brown University Algorithm Simulator and Animator (BALSA), in 1983 [11][12]. BALSA could display multiple algorithms executing at the same time, and also supported multiple simultaneous viewing of their data structures. The system primarily visualized programs written in Pascal, and provided a pretty-printed display of the source code, with an indication of the currently running line of code.

BALSA accepted regular Pascal source code that was instrumented to notify the BALSA event manager of interesting events, like the entry into a function or the access of a particular data structure. The system was successfully used as a teaching aid in Computer Science classes at Brown for a number of years, until it was replaced by later versions.

In 1988, Brown released Balsa-II [4][5], which provided additional scripting facilities. In addition, it was in color and allowed the use of sound in addition to visual representations.

2.1.2 Zeus

Zeus, also developed by Brown, is another descendant of BALSA [6][7]. Released as a prototype in 1991, Zeus provides the user of the system the ability to alter the visual representation of data at runtime. Additional support for sound was added using MIDI, allowing sounds to be incorporated into the visualizations. Zeus was implemented with multi-processor, multi-threaded platforms in mind, and offers greater capabilities for visualizing parallel programs.

2.1.3 TANGO and POLKA

Stasko produced another software visualization system at Brown University in the late 1980's called TANGO [44]. TANGO was built on a new architecture for algorithm animation he called a *path transition paradigm*. This architecture allowed for smoother animations and less overhead for the visualization designer.

The path transition paradigm's supporting architecture consists of three parts: defining the interesting events in a program that drive the animation, the design of the animations, and the mapping of the interesting events to their corresponding animations. To define the interesting events, special calls are inserted into the target C source code manually. Animations are created by defining transitions, rather than every step in the animation. This is the central advantage of the path transition paradigm. These transitions are defined in terms of trajectory, size, visibility, and color. The mappings between events and animation components in TANGO are also performed manually.

Unlike BALSA, an animation could receive events from several different program sources, and a program source could supply events to several different animations. When the visualization is running, the user is given control over the view, such as pan and zoom, as well as temporal controls such as pause.

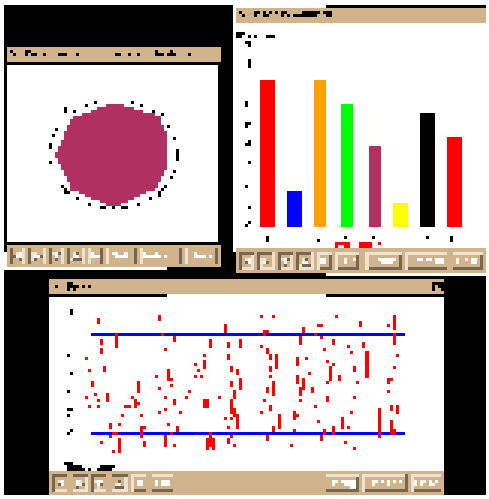


Figure 2.2 – A POLKA Screenshot ft. a Kiviat diagram and a Feynman diagram [20]

TANGO and its many descendants are widely used and freely available, along with a wide variety of pre-made visualizations. Among TANGO's descendants are

XTANGO, the X windows version, and POLKA with its graphical front-end Samba [20]. A sample screenshot of Polka featuring a Feynman diagram, a Kiviat diagram and a bar chart is shown in Figure 2.2.

2.1.4 ANIM

Developed at Bell Laboratories in the early 90's, ANIM takes an interesting "post mortem" approach to software visualization [2][3]. ANIM involves a scripting language of just eight commands (*line, text*, *box*, *circle, view*, *click*, *erase,* and *clear*). To instrument a program for visualization, a user must simply add file output commands, using this scripting language, to interesting parts of the program source. When the program runs, it generates a script file, which can later be compiled into an animation or a static representation.

This approach provides a couple of unique advantages. First, the user can have arbitrary control over the speed and direction of animation playback, as no new information comes into the animation once it is compiled. Second, programs in any language can be instrumented by simply using that language's file output procedures.

2.1.5 PV

PV (Program Visualization) [27] is an extensible visualization tool that uses trace capabilities in the AIX kernel to provide visual displays of information ranging from hardware latencies and operating system overhead to higher-level application-level issues. Although PV can be attached to software systems running several languages, including C, FORTRAN, Ada, and High-Performance Fortran (HPF), it is limited to the AIX operating system. PV is capable of handling large, distributed applications and provides support for plug-ins.

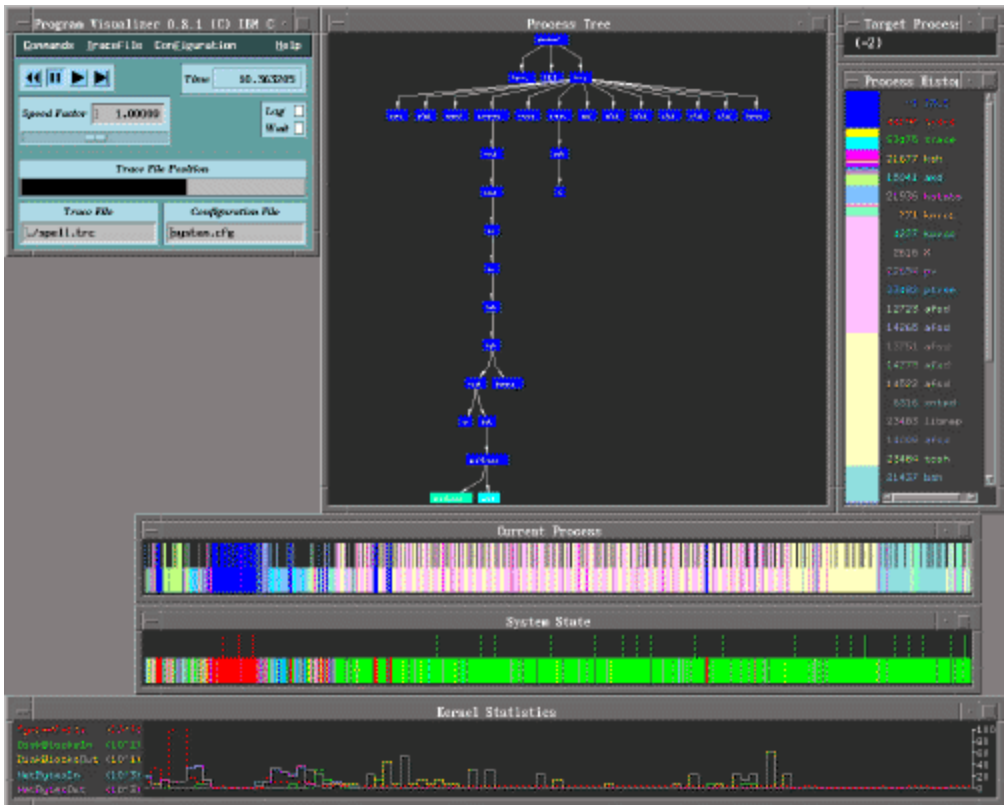


Figure 2.3 – Screenshot of PV [27]

The above figure shows several visualizations in PV, including a representation of the control flow of the program, a list of processes, and a display of system statistics.

2.1.6 Jinsight

Like PV, Jinsight [26] also makes use of trace data for visualizations. Jinsight can run on AIX or Windows platforms, but can only be attached to Java software systems. Jinsight provides capabilities for the filtering of unimportant events and special tools for common debugging issues, such as memory leaks.

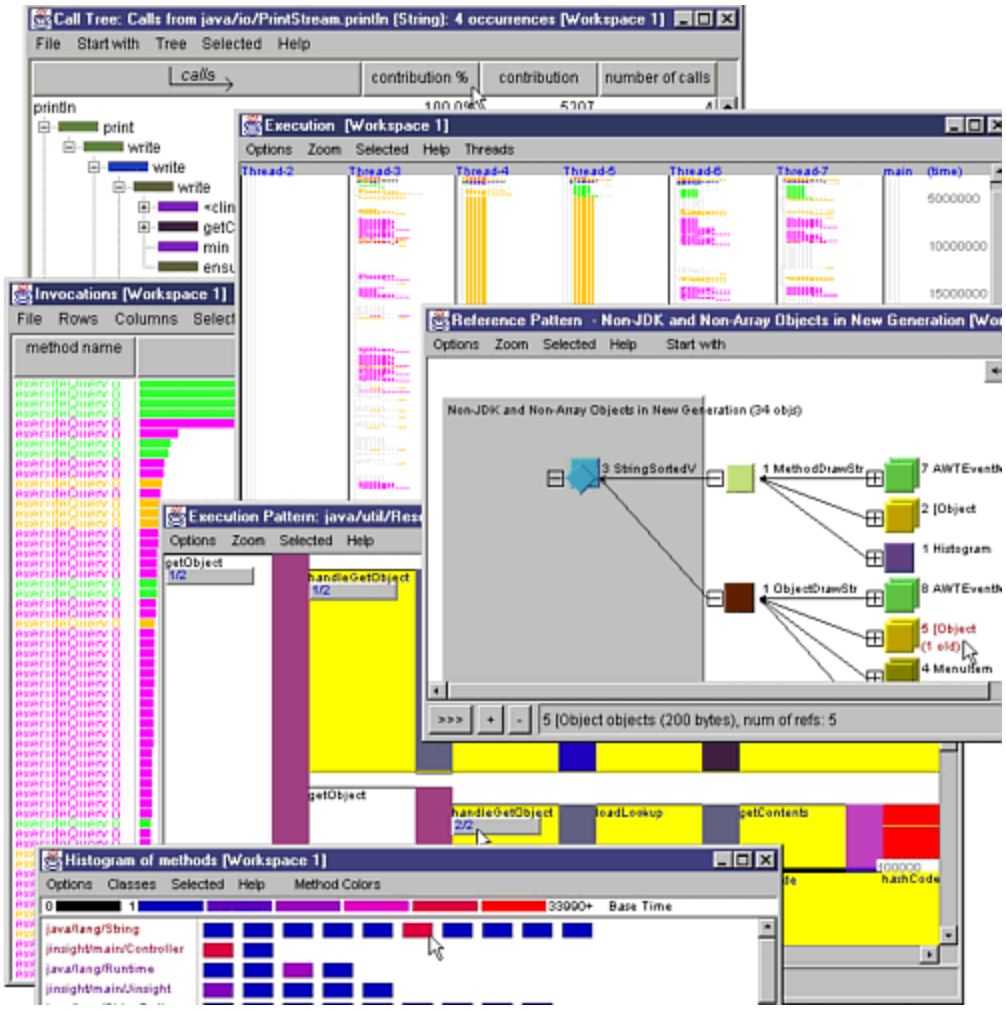


Figure 2.4 – Screenshot of Jinsight [26]

Additionally, Jinsight can create visualizations for common patterns of method calls. PV and Jinsight both allow for the display of both static structural visualizations and dynamic event-driven visualizations, and each makes use of color to show correlations between dynamic visualizations and hierarchal source code representations. Figure 2.4 demonstrates Jinsight's pattern-visualization capabilities and its use of color.

2.2 A Taxonomy of Software Visualization

Several taxonomies of software visualization systems have been derived in the past two decades, beginning with that of Myers [35]. In this taxonomy and its subsequent updates [36][37], Myers categorized software visualization tools along two main axes. The first

axis is the level of abstraction of what is being visualized, ranging from an abstract look at the behavior of an algorithm to a concrete in-depth look at a particular implementation. The second axis measures the amount of animation in the visualization, or whether it is static or dynamic.

As new software visualization systems emerge with a wider variety of capabilities and features, taxonomies of the field of software visualization have become more complex and detailed. The taxonomy of Price, Baecker, and Small (PBS) [39] again divides the broad category of Software Visualization into *algorithm visualization*, which concentrates on the abstract logic of a program, and *program visualization*, which concentrates on a particular implementation. BALSA is an example of an algorithm visualization system, while PV and Jinsight are program visualization tools. By examining this detailed taxonomy in the context of the previously described software visualization systems, we will now reveal common design patterns of modern SV systems.

In the PBS taxonomy, visualizations of software are classified by six major traits:

A. Scope – the range of programs that can be visualized and the generality of the visualizations.
B. Content – the aspects of the software that are visualized.
C. Form – the style and granularity of the visualizations.
D. Method – how visualizations are specified in the visualization system.
E. Interaction – how the user interacts with the visualizations.
F. Effectiveness – how well the visualizations communicate information.

Each of these categories is broken down into sub-categories, shown below:

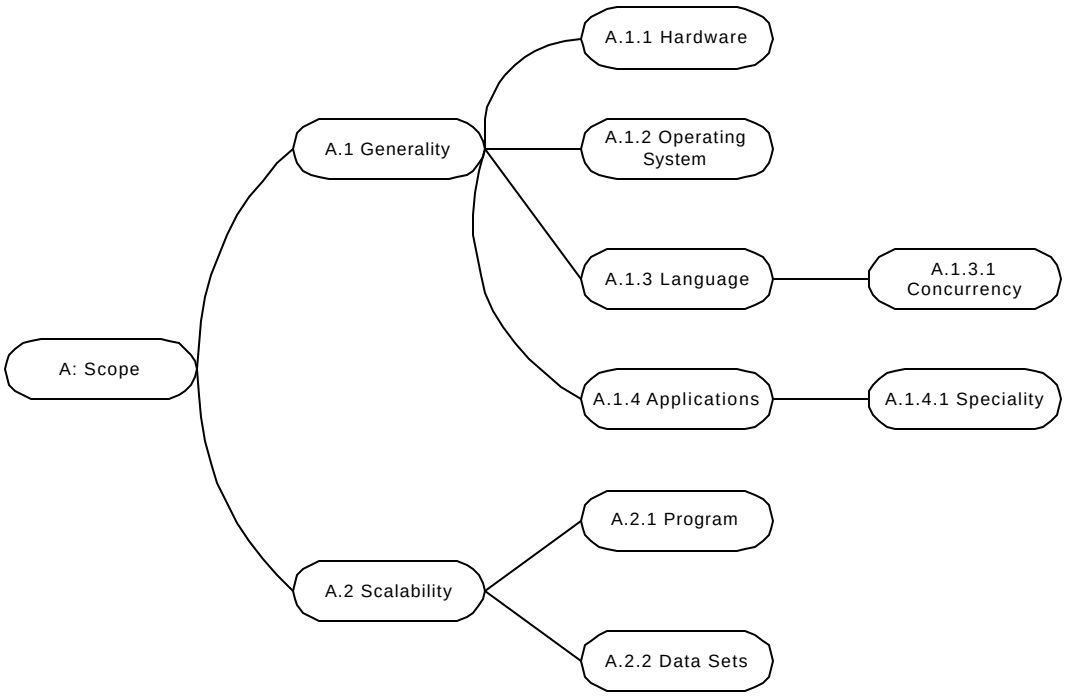


Figure 2.5 – The Scope of a Software Visualization System

A: Scope

This category describes the requirements of a visualization system and the restrictions on the programs it can visualize. This branch of the taxonomy is shown in Figure 2.5.

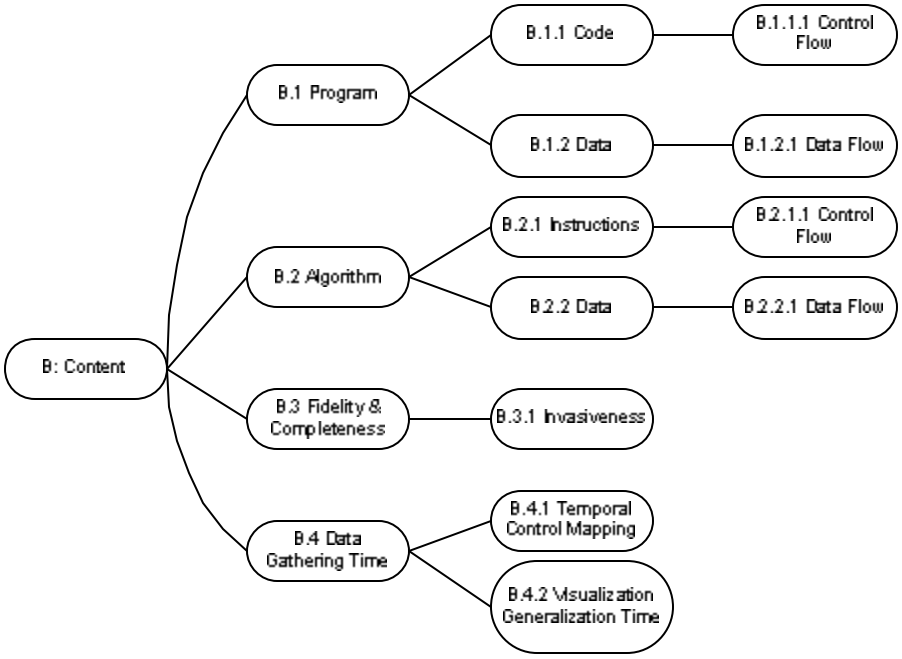
A.1.	*Generality* -	Will the system work on any program, or only one?
A.1.1.	*Hardware* -	What hardware will it run on?
A.1.2.	*Operating System* -	What Operating System is required?
A.1.3.	*Language* -	The languages that visualized programs must be written in.
A.1.3.1.	*Concurrency* -	Does the system allow the use of concurrency? Can it visualize concurrent behavior?
A.1.4.	*Applications* -	Are there any restrictions imposed on programs to be visualized?
A.1.4.1.	*Specialty* -	What types of programs are visualized particularly well?

A generalized system can run on any platform or operating system, and can accept programs in any language and of any type as input. Most of the programs we have examined above are restricted to a particular operating system. ANIM is restricted to UNIX environments, while PV can only run on AIX. Additionally, all of these systems are restricted by what types of programs they can visualize. ANIM is unique in that it can visualize programs in any language, where BALSA can only accept programs written in Pascal.

Support for concurrency, or multi-threaded target programs, is another important design consideration for software visualization systems. Although a visualization system may accept programs written in languages that support concurrency, the system itself may have no capacity for visualizing concurrent behavior. Of the systems we have surveyed, only PV and Jinsight offer sophisticated visualizations of concurrency.

A.2.	*Scalability*	What is the largest example the system can visualize?
A.2.1.	*Program*	What is the largest program that the system can handle?
A.2.2.	*Data Sets*	What is the largest data set it can handle?

Scalability is a measurement of the size of the largest program that a visualization system can handle. Many of the tools reviewed above, including BALSA and TANGO, have not been formally tested or used in a production environment. ANIM is run post-mortem on a generated script file, so it may not be as limited by system memory and processor speed in the same way as a runtime visualization system. PV and Jinsight, both used in production environments, have been tested on reasonably large programs and data sets.



B: Content

This category describes the important differences between *algorithm visualization* systems and *program visualization* systems, and is shown above in Figure 2.6.

B.1.	*Program*	How does the system visualize the implementation of a program?
B.1.1.	*Code*	How does the system visualize the source code instructions of a program?
B.1.1.1.	*Control Flow*	How does the system visualize the flow of control in a program?
B.1.2.	*Data*	Can the system visualize the data structures in a program?
B.1.2.1.	*Data Flow*	How does the system visualize the data flow in a program?

The ability to visualize the implementation-specific details of a program is the defining characteristic of a *program visualization* system. This includes the ability to visualize the control flow of the program through the source code as well as the concrete data structures used in the implementation. PV is an example of a program visualization system that features all of these capabilities.

B.2.	*Algorithm*	How is the abstract logic behind a program visualized?
B.2.1.	*Instructions*	Are the instructions in an algorithm visualized?
B.2.1.1.	*Control Flow*	Is the control flow of an algorithm visualized?
B.2.2.	*Data*	Are the algorithm's high-level data structures visualized?
B.2.2.1.	*Data Flow*	Can the data flow of the algorithm be visualized?

Algorithm visualization systems provide the user with a more abstract view of the behavior of an algorithm. The instructions of an algorithm are an abstraction of the source code of a program. Similarly, abstractions can be made on concrete data structures. BALSA and the video *Sorting Out Sorting* are examples of abstract algorithm visualization systems.

B.3.	*Fidelity and Completeness*	The degree to which the visualizations represent the accurate and complete behavior of the program.
B.3.1.	*Invasiveness*	Does the visualization of a program disrupt its behavior?

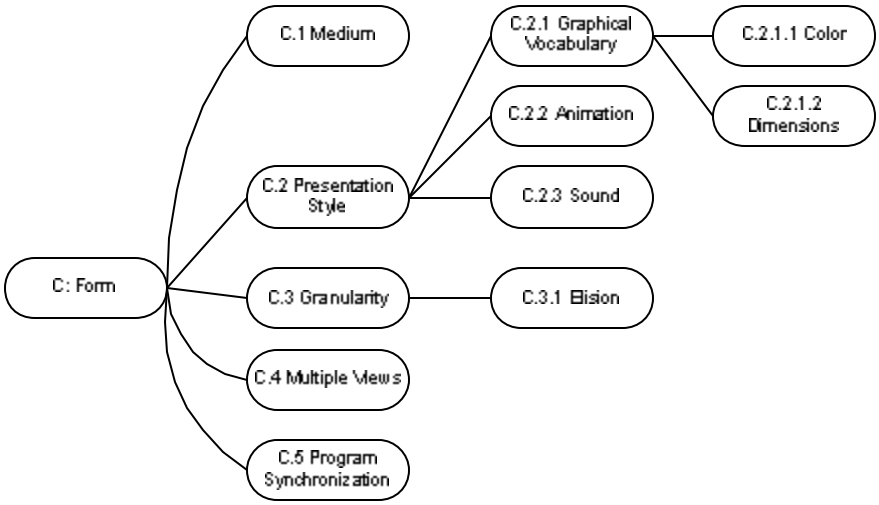
With any abstraction a certain amount of detail is lost. The more abstract algorithm visualization systems may not provide as accurate an image of the targeted program as a system more closely tied to the actual implementation's execution.

It is in the area of invasiveness that some of the most important design distinctions in our surveyed visualization systems are revealed. Many of the systems require the instrumentation of the target program's source code with additional code that provides information to power the visualizations. While this method is effective, the act of altering a program's source code may itself alter the program's behavior. Obviously, the amount of code added to a program and its runtime cost should be minimized when this method is employed.

B.4.	*Data Gathering Time*	When is the data for the visualization gathered?
B.4.1.	*Temporal Control Mapping*	What is the relationship between program time and the time of the visualization system?
B.4.2.	*Visualization Generation Time*	Is the visualization performed before, during, or after program execution?

The time the program data is gathered is tied closely to the invasiveness of the software visualization system. Program information is gathered either at compile-time, at run-time, or both. While compile-time methods are less invasive, they cannot gather information about the dynamic behavior of a program. Therefore, systems which use only compile-time data extraction are limited to static visualizations.

In the run-time methods, we see an additional distinction in the generation time of the visualization. Post-mortem systems such as ANIM collect information at run-time, but do not generate the visualization until execution has halted. Run-time visualization systems such as Jinsight, generate visualizations while the program runs.


C: Form
C.1 Medium
C.2 Presentation Style
C.3 Granularity
C.4 Multiple Views
C.5 Program Synchronization
C.2.1 Graphical Vocabulary
C.2.2 Animation
C.2.3 Sound
C.3.1 Elision
C.2.1.1 Color
C.2.1.2 Dimensions

system reviewed that supports sound is Zeus, although POLKA and Zeus both have three dimensional successors [10][48].

In all the run-time visualization systems surveyed, animation is used to represent the dynamic behavior of the program as time passes. Of the systems that use color, PV and Jinsight show the connection between different visualization windows by assigning the same color to the common elements. Other systems use color for emphasis or simply to make it easier to distinguish visualization elements.

C.3.	*Granularity*	Does the system allow for coarse and fine granularities?
C.3.1.	*Elision*	Does the system allow for the hiding of irrelevant data?

Granularity refers to a system's ability to "zoom out" and filter out unnecessary details. Elision refers to the ability to hide information that has been deemed as irrelevant. While these features are recognized as valuable in the field of visualization, the only surveyed software visualization system that allows for any degree of granularity or elision is Jinsight.

C.4.	*Multiple Views*	Can the system provide multiple synchronized views of the program?
C.5.	*Program Synchronization*	Can the system visualize multiple programs simultaneously?

Multiple views refers to the capability of showing different views of the same program simultaneously. Zeus, Jinsight, PV, and ANIM to a lesser extent, allow the user multiple views of the same program as it runs. This feature can provide the user with more information about the inner workings of the target program.

Program synchronization is the ability to show views of multiple programs simultaneously. This allows the user to race two different programs, or two show client and server programs interact.

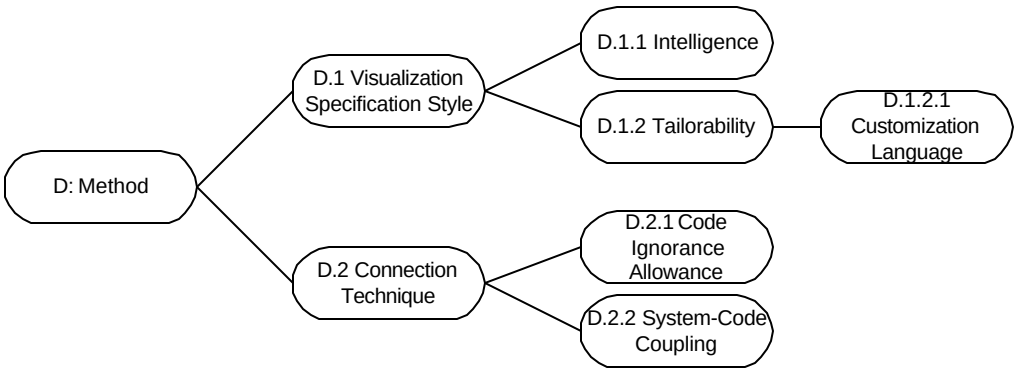


Figure 2.8 – The Method of a Software Visualization System

D: Method

This category describes the way in which new visualizations are specified, and how the visualizations get information from the target program, and is shown above in Figure 2.8.

D.1.	*Visualization Specification Style*	How are visualizations defined? (library, hand-coded)
D.1.1.	*Intelligence*	Is the visualization automatic? How advanced is the AI?
D.1.2.	*Tailorability*	How customizable are the visualizations?
D.1.2.1.	*Customization Language*	How are visualizations customized?

Software visualization systems differ only slightly in the method in which visualizations are defined. Some systems, like BALSA and TANGO, provide the visualization programmer with a library of pre-built visualizations from which to construct a new visualization. Other systems, such as ANIM, require hand-coded visualizations. Another option, which is not widely used in software visualization, is the automatic generation of visualizations.

Another aspect of SV systems is tailorability, or the degree to which the user can alter the appearance of a visualization. Zeus, for example, gives the user the ability to alter the view, and this alteration propagates through the other visualization windows.

D.2.	*Connection Technique*	How does the visualization system get information from the program?
D.2.1.	*Code Ignorance Allowance*	How much knowledge of the program is required to make a visualization?
D.2.2.	*System-Code Coupling*	How closely tied are the program and visualization?

The evaluation of a visualization system's connection method again brings into focus the idea of invasiveness. The most common method of extracting information from a running program is the *instrumentation* of the program's source code with additional code that exports program data. This instrumentation can be performed automatically, which allows the visualization designer an additional degree of code ignorance. Both run-time and post-mortem systems use this method to extract program data.

Any instrumentation of source code is considered an invasive technique, and may result in undesirable consequences to the performance or behavior of the target program at runtime. Some systems use a non-invasive technique, in that it does not modify the source code, called *probing* to get information about the execution of the program from the operating system or from the program interpreter, if one is being used. Although this method does not require alterations to the executable, it cannot produce the same detailed information as invasive techniques.

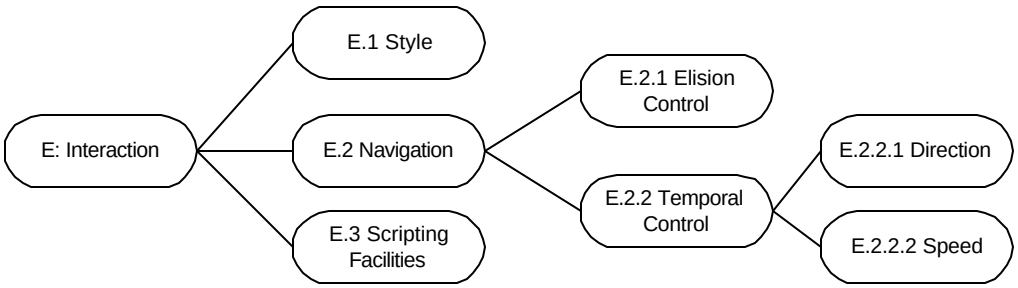


Figure 2.9 – The Interaction of a Software Visualization System

E: Interaction

This category describes how the user can interact with a software visualization system. This branch of the taxonomy is shown in Figure 2.9.

E.1.	*Style*	How does the user give commands to the system?
E.2.	*Navigation*	Does the system support navigation through a visualization?
E.2.1.	*Elision Control*	How can the user hide irrelevant portions of the visualization?
E.2.2.	*Temporal Control*	How does the user control the temporal elements of the system?
E.2.2.1.	*Direction*	Can the user reverse the temporal direction of the visualization?
E.2.2.2.	*Speed*	Can the user control the speed of the visualization?

E.3. *Scripting Facilities* Does the system allow for the recording of interactions with visualizations?

The visualization systems surveyed made only a limited usage of the interaction methods described above. While most of the systems allowed interaction with a mouse, navigation and elision controls were non-existent or severely limited. TANGO, for example, allows navigation through a visualization, but only through resizing or zooming a window. All systems allow some degree of temporal control, the most basic being the ability to start and stop the system. Many systems, including BALSA and Zeus, allow the user total control over the speed of the visualization playback, as well as the ability to record the user's interaction with the system.

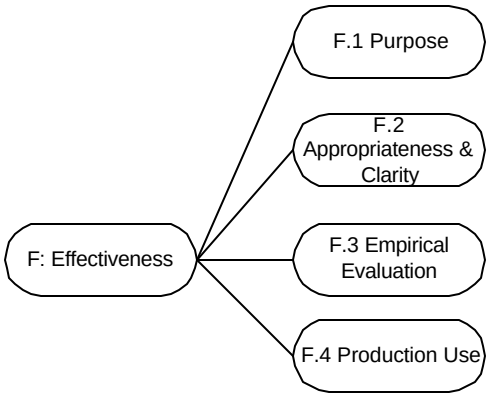


Figure 2.10 – The Effectiveness of a Software Visualization System

F: Effectiveness

This category describes how the system has been tested and used, as well as a general measure of its effectiveness, and is shown in Figure 2.10.

F.1.	*Purpose*	What purpose does the system serve?
F.2.	*Appropriateness and Clarity*	How effective and accurate are the visualizations?
F.3.	*Empirical Evaluation*	Has the system been subjected to a serious evaluation?
F.4.	*Production Use*	Has the system been used in a production setting?

Of the systems surveyed, only PV and Jinsight were subjected to a true empirical evaluation and were used in production [26][27]. The other tools, like BALSA, were never formally evaluated, but were successfully used as teaching aides [11][12].

2.3 Taxonomical Analysis of the SoftViz Environment

In this section, we employ the PBS taxonomy of software visualization to better describe and categorize the *SoftViz* environment.

2.3.1 Scope

The *SoftViz* environment involves two main elements: a "back-end" which is responsible for data gathering and a “front-end which uses this data to generate and animate visualizations. These two components are decoupled through a common interface, so the implementation of each side is not dependant on the implementation of the other. The environment also allows the front-end and the back-end to be run on separate computers connected across a network. In addition, several different back-ends, on different networked computers, can provide data to a single front-end. These properties of the environment are important when considering the scope of the visualization system. The hardware and operating system requirements, for example, must be considered for each component, rather than for the system as a whole. To simplify this situation somewhat, the current implementation of the *SoftViz* environment uses the Java programming language, which is somewhat platform-independent. Both the front-end and back-end will operate on any platform which supports an implementation of the Java Virtual Machine.

The decoupling of the visualization component from the data gathering component also allows the environment a degree of flexibility with regard to what programming language the visualized programs must be written in. The design of the environment would support any number of back-end implementations capable of gathering data from programs written in any Object-Oriented language. Current implementations of the data-gathering components support only the instrumentation of programs written in Java. However, nothing in the current implementation of the front-end requires the use of instrumented Java code.

SoftViz does not place many restrictions on the programs that are visualized. The environment allows the use of concurrency and the visualization of concurrent behavior, but the specifics of how concurrent behavior is visualized are left to the visualization

module developer. The restrictions on programs that can be visualized come from the availability, or lack of availability, of back-end components that provide support for a particular programming language.

While the scalability of the *SoftViz* environment was a primary design goal, the current implementation has not yet been empirically evaluated with regard to scalability. As this is a continuing project, it is hoped that future work with this environment will determine the maximum number of probes and maximum rate of events that the system can handle.

2.3.2 Content

The *SoftViz* environment relies on data extracted by source code intrusive methods. The source code of the target program is instrumented with probes, which produce the data used in visualization. This method produces a stream of probe data which is intrinsically tied to both the source code and the control flow of the instrumented program. In the language of the taxonomy, this environment is a *program visualization* system, in that it creates visualizations based on the source code instructions of a program rather than on the abstract algorithms the program is based on.

The environment currently instruments only method calls, which allows it to follow only the control flow of the instrumented program. The control flow of a program is the primary focus of the *SoftViz* environment. Data flow of an instrumented program can be approximated by examining the arguments passed to calls to the accessors of abstract data structures. This ability can be extended to primitives through the use of instrumented wrappers.

While the *SoftViz* environment relies on source code invasive methods of extracting program data, the extent of the effect of this invasiveness on the performance of the program being visualized has not yet been analyzed. It is expected that the performance of the instrumented program will be inversely related to the number of active probes in the program. In programs which are time-sensitive, the act of instrumenting and visualizing the program may actually alter its behavior.

Structural data about the target program is gathered at the time the program is instrumented. This structural data is combined with dynamic data gathered from the instrumented probes at execution time to generate and animate visualizations. The environment marks each probe-generated event with a timestamp. This gives the front-end the ability to determine the correct order and relative temporal positioning of events, as they occurred on their system of origin. These timestamps allow the front-end to compensate for any variations in network latency that may otherwise result in inaccurate or incomplete visualizations.

The accuracy and completeness of the visualizations themselves is dependant on the developers of the visualization modules themselves.

2.3.3 Form

The visualizations generated by *SoftViz* are intended solely for output to a computer monitor. The actual visual style of a visualization, including the use of animation, 3D graphics, and sound, is left to the discretion of the visualization module developer. Support for elision and the allowance of coarser and finer granularities is also provided at the visualization module level. The current Sunburst visualization module implementation demonstrates the use of animation, elision, and various levels of granularity.

The environment does control the use of color by a visualization, ensuring that all visualizations which target the same program use the same associations between color and structural elements. This provides a degree of visual consistency between visualizations of the same program.

In addition to the capacity of the *SoftViz* environment to provide multiple views of the same program, it also allows the visualization of multiple different programs simultaneously.

2.3.4 Method

The visualization system receives events from instrumented programs across a network connection. Both the instrumentation of the source code and the extraction of the structure of the program are performed by automated tools. For basic operation of these

tools, no changes are required to the default configuration. Visualization modules themselves are written to an interface which generalizes the nature of events and program structure. As a result, absolutely no specific knowledge of the instrumented program is required to write a visualization for it. In fact, the design of the environment allows the creation of a generalized set of visualization modules which may be usefully applied to a wide variety of programs. These features guarantee a large degree of *code ignorance allowance* and a low degree of *system-code coupling*.

2.3.5 Interaction

The specifics of how a user interacts with a visualization, including navigation and elision control, are left to the discretion of the visualization developer. It is expected that the primary method of interaction will be the mouse, but nothing in the design of the environment precludes the use of another input device.

The environment does handle the temporal control of a visualization. A tool resembling the buttons of a VCR allows the user to control the speed and direction of playback of previously recorded events. When events are being received from a program during program execution, this tool only allows the user to stop and resume the reception of events.

Although no scripting facilities are provided, the environment does allow for saving the result of previous user interactions with a visualization by allowing the user to save and restore the state of the GUI workbench.

2.3.6 Effectiveness

The *SoftViz* environment has not been evaluated for use as a development tool. Future work on the environment will incorporate qualitative analysis of the effectiveness of various visualization modules as well as a quantitative analysis of the performance of the system as a whole.

2.4 Design Considerations for Software Visualization

In this section, we will discuss the design concepts that have been revealed in our examination of previous software visualization systems and their features.

2.4.1 Extraction

Software visualization systems vary greatly in their methods of extracting information from target systems. Some visualizations require runtime information, and some are run after the target program's execution has ended. The visualization systems that require runtime information can gather it by altering the source code, by over-riding data structures, by using non-invasive techniques, or a combination of these methods. In this section, we discuss these methods and their impact on the design of a software visualization system.

2.4.2 Post-Mortem Extraction vs. Runtime Extraction

As we have seen in the review of software visualization systems, there are two main approaches to the extraction of dynamic program information: Post-Mortem Extraction and Runtime Extraction.

The "post-mortem" approach collects information about a program while it runs and deposits it in a script file. Later, this script file is compiled into a visualization. This approach has the advantage of knowing all events that will occur during program execution in advance, so it allows the user the ability to move backwards and forwards in the event stream arbitrarily. This method also allows for analysis of the event data and the calculation of an optimal layout for the visualizations. Post-mortem extraction can also make it easier to generate and view visualizations using a single screen.

The "runtime" approach attempts to create and animate a visualization while the target program is running. This approach allows the user the ability to interact with the program while the visualization is running, a feature which could be invaluable to the user in performing any debugging or optimization task.

2.4.3 Code Intrusive

Both the post-mortem method and the run-time method of extraction are what Henry referred to as "code-intrusive" [25]. A code-intrusive extraction method requires that the source code or byte code of the targeted program is *instrumented* with special code that provides information to the visualization system. In the case of post-mortem systems like ANIM, this special code dumps script instructions to a file on disk. For runtime systems, the inserted code must somehow deliver the program information to the visualization system. The obvious solution to this is to use sockets, as this approach accommodates both local and remote visualization systems.

There is some inherent danger in using code-intrusive extraction methods. The inserted code may cause the instrumented program to alter its behavior or performance in unexpected ways. It is expected that an instrumented program will run slower than its non-instrumented counterpart. If the instrumented program is time-sensitive, this may cause aberrant behavior during execution. In an extreme case, automatically inserted code could loop infinitely, causing the instrumented program to freeze. The inserted code will be compiled and run as if it were part of the program, so the inserted code segments must be carefully tested. If remote source code instrumentation is allowed, security precautions must be taken to ensure that no malicious code is added by compromising the instrumentation mechanism.

2.4.4 Data Intrusive

Data-intrusive extraction methods [25] are similar to code-intrusive methods in that they require the insertion of special instructions into the source code of the targeted program. This approach, however, focuses on instrumenting the entry points of the abstract data types characteristic of Object-Oriented programming languages, such as the "get" and "set" methods of an Object-Oriented data type. This instrumentation is accomplished by creating sub-classes of data types and over-riding class methods to serve the needs of the visualization system. Data-intrusive extraction provides another method of providing program information for data-driven visualizations.

2.4.5 Non-Invasive Methods

Non-invasive extraction methods do not affect the source code or byte code of a program, instead relying on querying the operating system or interpreter running the program. On operating systems like AIX, which provide additional kernel-level debugging facilities, this approach can yield large amounts of information. A software visualization system can also extract useful information about programs written in interpreted languages, like Java or BASIC, by querying the interpreter or virtual machine while the program is running.

The obvious advantage to this approach is the ability to visualize programs without having access to the source code. Another advantage to non-invasive methods is the fact that the source code remains unchanged from its normal state. The GNU debugger *gdb* is an example of a system that uses a non-invasive extraction method.

There are several disadvantages to this approach, however. First, non-invasive methods are usually reliant on a particular implementation of a virtual machine or an operating system as debugging methods are rarely standardized. Second, non-invasive techniques usually yield less information than their invasive counterparts. Although *gdb* provides stack traces and can be a helpful debugging tool, sometimes you still have to litter your code with print statements. Finally, the additional overhead for the extraction of program data from the operating system, interpreter, or hardware platform may mean that the program has fewer resources available to it, and may execute more slowly. Although this method is non-invasive in that it does not change the source code, it may still affect program performance.

2.4.6 Instrumentation Methods

Visualization systems that use intrusive methods can either require the programmer to instrument the program by hand, or provide a tool for the automatic instrumentation of source code. Algorithm visualization systems, which concentrate on the more abstract elements of structure and behavior, tend to rely on the programmer to instrument the source code, as automatic systems cannot yet perform conceptual abstraction.

2.5 Summary

In this chapter, we discussed the history of software visualization, including a survey of previous software visualization systems and a taxonomy of these systems. We discovered that at each branch of the taxonomy there are trade-offs to be made in the system design, and discussed the most relevant of these design considerations. In the next chapter, we apply this taxonomy and the revealed design considerations to our desired system, and devise a design accordingly.

3 Design of the *SoftViz* Environment

This chapter describes the design of the *SoftViz* environment, including a summary of the goals and design considerations of the project and brief descriptions of relevant design patterns.

3.1 Global Design Considerations

The general design goals outlined in the introduction provided a basis for subsequent application-specific design decisions. Each of these goals contributed to many aspects of the final system design, and are discussed in detail below.

Our general goals were:

- **Scalability**
- **Portability**
- **Flexibility**
- **Extensibility**
- **Openness**
- **Adaptability**

3.1.1 Scalability

Scalability in a Software Visualization system refers to the maximum size and scope of target systems. The code-intrusive method of data acquisition raises a variety of concerns with regard to scalability. First, the size and complexity of the probe itself should be limited to minimize overhead. If each probe executes slowly, even a small number of inserted probes could significantly degrade performance. In addition to the execution time of a single probe, the number of inserted probes is another factor influencing scalability.

Consider a visualization system that uses an automatic tool to instrument the source of target programs. The user of the tool can choose to instrument many or few aspects of the source code as desired, including methods and data accesses. For a complex program of several hundred thousand lines, the user could choose to instrument only the methods and data elements that are interesting for their purposes. These

instrumented elements generate events which are sent to the visualization system. Even though the target program may be huge, not all of its methods have been instrumented, so the number of events may be relatively small.

This approach of *selective instrumentation* can be used to allow a visualization system to target systems that would be too large for it to handle otherwise. Ideally, we would like our system to be able to handle many thousands, or hundreds of thousands, of events per second. The total number of events that can be handled every second is limited by bottlenecks in the event transport software, in the probe monitoring infrastructure, and in the visualization system itself. Additional limitations on the events that can be realistically handled in a second are imposed by the network infrastructure and the hardware that the target and visualization systems run on.

To maximize the scalability of our environment, we concentrated both on minimizing the bottlenecks imposed by our software and on the use of selective instrumentation.

3.1.2 Portability

The *Portability* of a system is a measure of its ability to run in many different hardware and software environments. Portability is especially important to a software visualization system, as it is likely to be used in many diverse development environments. Additionally, the visualization system and the targeted system should be able to run on separate network-connected computers. Considering this ability, extra measures should be taken for portability, especially for the visualization system code that goes on the target machine. The portability of the visualization system limits the scope of the programs that it can visualize, and thus its own usefulness.

An additional design concern for the portability of a software visualization system is the kinds of languages that target systems can be written in. Although the system may be intended for use with only one programming language initially, the system design should be sufficiently general to allow the addition of support for more programming languages. Limiting the number of supported programming languages again limits the usefulness of a software visualization system to the general public.

3.1.3 Flexibility

The third general design consideration is *Flexibility,* a trait which accommodates the users' need to perform a variety of different tasks and operations. Not all of these tasks may be known at the time of the design of the system, so it is important to consider the future addition of new features to allow the user to perform new tasks.

For a software visualization system, users will have certain expectations of tasks they should be able to perform, like debugging and optimization. The system should also allow for other common software visualization tasks related to teaching and presentation. These tasks have radically different goals, and the environment must be flexible enough to accommodate them.

3.1.4 Extensibility

Extensibility is a measure of the system's ability to integrate with existing software development tools and to expand to meet user needs. The first aspect of extensibility describes the use of a software visualization tool as an extension to a previous development environment. The actual usefulness of a development tool such as a software visualization system is greatly impacted by its ability to work alongside integrated development environments and other common tools. The design of a software visualization system, especially a code-intrusive system which modifies source code, must take into account its effect on other development tools being used on the same program as well as differences in presentation style.

The second aspect of extensibility describes the ease of extending the visualization system itself to meet new needs and allow new tasks. The design of the system must take the addition of new features into account. In a software visualization system, extensibility is most often concerned with the ability to add new types of visualizations, but it could also involve adding support for new programming languages or integration with other tools.

3.1.5 Openness

Openness is another attribute that allows integration with integrated development environments and source control systems. The system design should allow for the import of information generated by other software visualization systems, including maps of the target system's structure or making use of another visualization system's probes. Additionally, our desired system design should allow for the export of similar information for the use of other development tools.

3.1.6 Adaptability

Adaptability is a measure of how well the system can be applied to a new task through user-configuration. An adaptable software visualization system must manage changing architecture and program behavior, as well as provide pathways for the user to further configure the system for specific tasks. This differs from the aforementioned goal of Extensibility in that it is the user, and not the implementer of the visualization system, who must be able to adapt the visualization system for new tasks. For a visualization system to be adaptable, it should provide a set of basic but powerful tools which can be recombined and adjusted to suit a wide variety of tasks.

3.2 Software Visualization Design Considerations

Our examinations of previous software visualization systems and a detailed taxonomy of software visualization in Chapter 2 have revealed many design considerations specific to software visualization. We will now revisit that taxonomy briefly to better illuminate the design decisions of the *SoftViz* environment.

A: Scope

The scope of a software visualization system refers to the restrictions imposed on the target software system by the visualization system. One of our primary design goals is portability, so our desired system should run on many different platforms and accept a

wide variety of different types of programs. We should avoid requiring a particular platform, operating system, or programming language in the design of the system. As shown in Chapter 2, all currently existing software visualization systems are limited to a particular platform, operating system, or programming language, which limits their usefulness to the development community.

Additional considerations should be made for multi-threaded systems, and the visualization system should accommodate and be able to visualize multi-threaded programs. Although there should be no specific restrictions on the types of programs to be visualized, the visualization system will concentrate on the visualization of object-oriented programs and provide special facilities to take advantage of the varied granularity allowed by the different levels of abstraction of object-oriented software.

Another primary design goal of the *SoftViz* environment is scalability. In addition to handling the visualization of systems on various platforms and operating systems, *SoftViz* should be able to visualize programs ranging from only a few lines of code to thousands or millions of lines of code, as it is the largest and most complex programs that developers have the most trouble understanding unaided.

B: Content

The content of a visualization system refers to what aspects of a system are being visualized, and how this information is gathered. Many of the systems described in Chapter 2 are algorithm visualization systems, which concentrate on the abstract behavior of algorithms rather than the details of a specific implementation. Although algorithm visualization is useful for teaching and demonstration, it is of limited use for problems of debugging or optimization, which are largely implementation-specific problems. *SoftViz* will concentrate on the visualization of a specific implementation, and can be categorized as a program visualization tool.

However, the *SoftViz* environment will not necessarily visualize the line-by-line behavior of a program. Prudent use of the AIDE compiler to selectively instrument the method calls of classes may allow the user to visualize the behavior of the target system at a higher level of abstraction than provided by source-code level visualization systems. For example, a user of *SoftViz* interested in algorithm visualization could write and

instrument a program demonstrating the abstract logic of an algorithm and *SoftViz* could function equally well as an algorithm visualization tool.

The *SoftViz* environment must be able to accommodate the visualization of the control flow of a program. Visualization of object-oriented data structures and data flow should also be possible, although *SoftViz* will not provide for the visualizations of low-level data structures, as these vary from system to system.

SoftViz will employ an invasive data-gathering technique, which may affect the behavior of the target system. First, the target system is instrumented with event-generating segments of source code, called probes, by the AIDE compiler. Using AIDE, probes can be selectively inserted. While it would be possible to insert probes at every line of code, this would result in a serious negative impact to performance when running the instrumented program. This effect can be minimized by inserting probes in only the interesting parts of the source, such as the beginning and end of method calls.

Even with selective probing, the impact on the performance of the target system may be quite severe. The *SoftViz* environment incorporates another technology which may help reduce the impact of probes on system performance. The *KX* probe monitoring infrastructure, developed by Peter Gill at WPI [21], allows for probes to be dynamically activated and deactivated remotely at runtime, allowing the user to deactivate all inserted probes except the probes of immediate interest. Although *KX* must be running as a background process, and therefore using system resources, it is expected that the performance gained through selective probing will outweigh the costs in most cases.

Visualizations in *SoftViz* will be created by combining visualization templates, deployed as dynamically loaded modules, with a map of the static structure of the target system, which will be gathered at the time the system is instrumented. These visualizations will then be animated at runtime with data generated by probes in the target system. This probe data can also be saved for playback and analysis at a later date.

C: Form

Form describes the presentation of the visualization to the user. Although the *SoftViz* environment will focus on the monitor as a medium, printer support is not ruled out by the design. The use of most elements of the visual language, such as size, shape,

and animation are left to the discretion of the visualization designer, as are the use of 3D graphics or sound. Color, however, is reserved by *SoftViz* to show the relationship between identical structural elements across all visualizations of the same system. Each discrete structural element is assigned a unique hue by the visualization system, although the visualization modules may alter the saturation or brightness for purposes of animation or user selection.

Developers of visualization modules for *SoftViz* are encouraged to support methods of hiding irrelevant data. This includes support for elision, or data hiding, as well as coarser and finer granularities. Support for these methods is left entirely up to the visualization designer.

The *SoftViz* environment requires the simultaneous viewing of multiple visualizations by using a multi-document interface design. Additionally, these visualizations can all be viewing the same system or they can be viewing different systems. All visualizations targeting the same system share a common color scheme, and share user-generated events like selection operations.

D: Method

The method of a software visualization system describes how visualizations are specified. The *SoftViz* environment relies on a modular system for the specification of visualization templates, and visualizations must be written and compiled in a programming language that the implementation of the environment can understand. The visualization module API provides for the interaction between a *SoftViz* module and the environment. All that is necessary for a visualization developer to complete a module is to write the specific graphics and initialization code.

Visualization modules will be dynamically loaded by the environment, and instantiated with a map of the structure of the target system and each element's associated color. Once instantiated, the visualization will begin to receive events from the target system as well as user-generated events, such as selection operations, from the other visualizations in its user-defined visualization group.

Events generated by the instrumented system are published to the SIENA wide-area event notification bus, where they are received by the visualization environment, which in turn routes them to the visualizations targeting that system.

This decoupled design allows the visualization designer an extremely high level of ignorance of the target system's source code and an extremely low level of coupling between the target system and the visualization. In fact, visualization modules are actually generalized visualization templates which deal with only static structure maps and dynamic event data. Once written, a visualization module can be re-used on any instrumented system, although the use of certain visualizations for a problem may make more sense than others. The versatility of this design ensures that the primary goals of adaptability and flexibility are met by the *SoftViz* environment.

E: Interaction

Interaction describes how the user interacts with the visualization system. In *SoftViz*, the specifics of how a user interacts with and navigates through a visualization are left up to the visualization designer. If the visualization module provides elision or control over granularity, such as zooming, these interactions must be handled by the visualization itself.

Each visualization is responsible for user-generated events that happen while the focus is on that visualization's window. The environment does provide a temporal control similar to the buttons on a VCR, allowing the user to control the speed and direction of playback of a stored stream, and to play and pause live streams.

The visualization module API provides for the saving of state for a visualization, but does not provide any sort of scripting language for interacting with visualizations.

F: Effectiveness

Effectiveness is a description of the purpose of a visualization system and a measurement of its success at achieving that purpose. *SoftViz* is intended as a development tool to aid in the understanding and debugging of complex software systems, which will reduce the amount of time required by these tasks. At this time, no

empirical evaluation of *SoftViz* has been performed, and its use in a production environment has not been assessed.

3.3 Architecture Overview

The *SoftViz* software visualization environment has three layers, with a clear separation of responsibilities. This model allows for extensibility and compatibility with other visualization systems. The bottom layer, called the *Event Substrate Layer*, extracts events from the software systems being analyzed. The *Laboratory Engine Layer* provides for filtering and aggregation of event streams as well as the storage of events from live streams for later playback. The *Visualization Layer* combines live and virtual event streams with visualization plug-ins and structural information about the instrumented program to produce the final visualizations. The relationship between the three layers, the visualization modules, and persistent storage is shown in Figure 3.1.

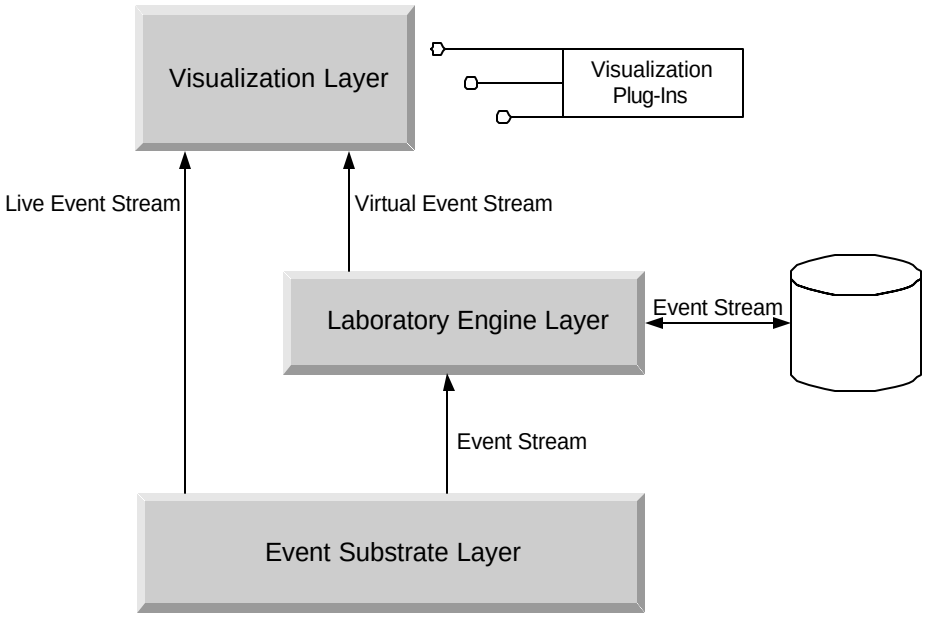


Figure 3.1 – The Three Layer Model

3.3.1 Event Substrate Layer

The Event Substrate Layer provides the laboratory engine layer with the stream of events generated by each target software system as it runs. First, the source files are instrumented with *probes*. These probes generate *events*, containing information about what time the method was called, when it returned, how deep the call was in the stack, which thread called it, and the method's parameters and return values. These generated events are then collected by the probe-monitoring infrastructure [21] and transmitted across the event transport bus, creating an *event stream*.

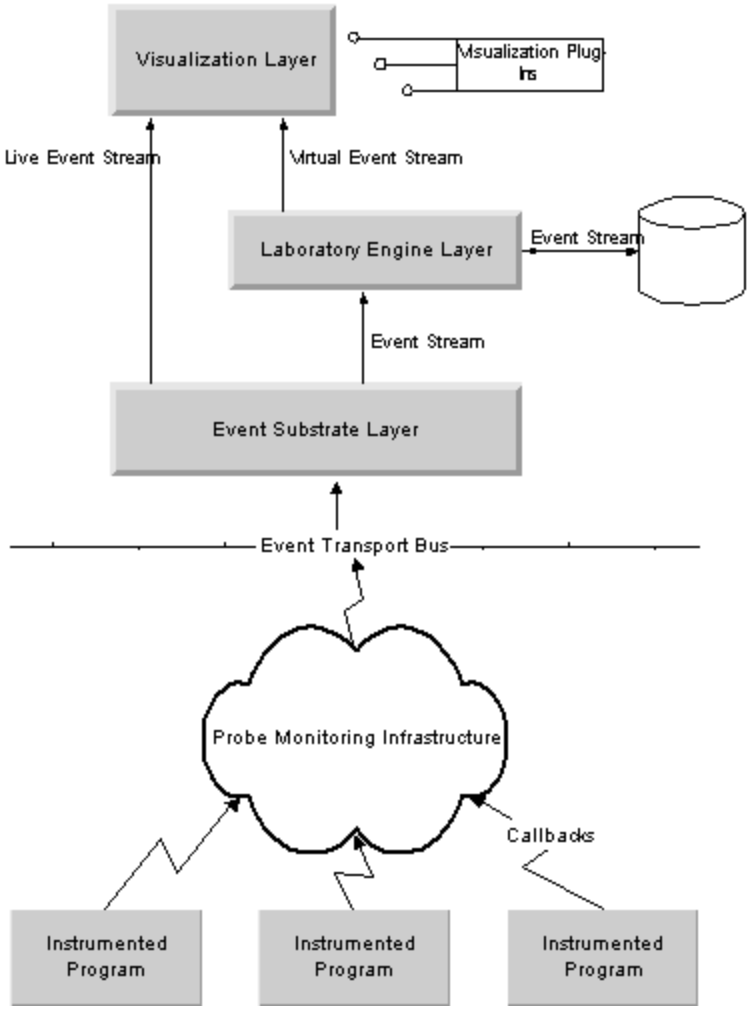


Figure 3.2 – The Three Layer Model with Instrumented Programs

The Event Substrate Layer connects the environment to the instrumented systems, as shown in Figure 3.2. This diagram traces the path of events generated by the probes in the instrumented software. These probes inserted into the source code of the target system are actually callbacks to the probe monitoring infrastructure, which is capable of dynamically changing the way these callbacks are handled. Since the inserted code is nothing but a callback, the actual behavior of a probe can be changed during program execution by interacting with the probe monitoring infrastructure.

The event transport bus is responsible for the transport of probe-generated events as well as for the communication between the probe monitoring infrastructure and the user interface. As described in Chapter 1, the *SoftViz* environment incorporates the *KX* probe-monitoring infrastructure with the *SIENA* event-transport bus. The *SIENA* bus provides a unique identifier for each client and server on the bus, and servers only propagate events to those clients who are subscribed to them. Clients can choose to subscribe to events from certain sources or of certain types. It is this capacity of the *SIENA* bus that allows the *SoftViz* environment to separate incoming events by source.

3.3.2 Laboratory Engine Layer

The Laboratory Engine Layer manages the multiple event streams provided by the event substrate layer, and has the capability for filtering and aggregation of these events. This layer can pass the *virtual event streams* created by the filtration and aggregation of event streams as well as unmodified live event streams to the visualization layer. The Laboratory Engine Layer also has the capability to store event streams to disk for later playback.

Although the events contain information about where in the source code they occurred, the actual structure of the source code must be extracted by the Laboratory Engine Layer at the time the source code is instrumented and saved as a *structure map*. The structure map of a software system contains hierarchical information regarding packages, files, interfaces, and class inheritance. This layer also provides for the extraction and persistent storage of these structure maps from the instrumented software systems.

When a new event stream or virtual event stream is created, the user can associate the new stream with the structure map of the system that is generating that stream's events. In order to ensure consistent coloring of corresponding elements across multiple visualizations, the Laboratory Engine Layer also generates a *color map* of each element in the structure map to a unique hue. The structure map and the color map are both provided to the Visualization Layer for use in instantiating visualization modules, and to provide context for the dynamic event data.

3.3.3 Visualization Layer

The Visualization Layer unifies the event streams and structure maps from the lower layers with visualization templates, or plug-ins, to create the final visualizations. Visualizations in this environment can make use of a structure map, an event stream, or both. This allows for static representations of structure, dynamic representations of stream events over the structure or dynamic interpretations of events without using the associated structure map. The Laboratory Engine Layer can provide event streams and structure maps for several different instrumented systems simultaneously, and the Visualization Layer can show visualizations for separate instrumented systems side-by-side, in a Multi-Document Interface (MDI). This feature would allow for the simultaneous and dynamic visualization of multiple clients and a server, with all of the instrumented systems running on separate machines. The MDI interface allows the user to move the visualizations around on the screen and to hide visualizations that are not of immediate interest.

This layer is comprised primarily of a GUI which allows the user to initiate new event streams, to load and instantiate visualization modules to receive events from a particular stream, and to allow for the grouping of visualizations by the user. Visualizations which are grouped together can communicate with each other and share user-generated events such as selection or elision of a particular structural element. When a structural element in a visualization is selected by the user, for instance, that selection event is propagated to all of the other visualizations that the user has grouped together.

3.4 Visualization Modules

Although different visualization modules, or combinations of visualization modules, will certainly be better suited for different problems or tasks, the visualization modules used by *SoftViz* must be general-purpose. Although a developer may write a visualization module for a specific targeted system, the module should be able to be re-used for all future instrumented programs, at least as far as the environment itself is concerned. For this reason, the specification of a visualization module enumerates all data that can be provided to a module by the environment but allows for any type of output.

When a visualization module is instantiated by the Visualization Layer, the module is provided with the structure map of the target system and the color map, containing the hues for each structure element. The module is also subscribed to the appropriate event stream. Additionally, the grouping mechanism of the Visualization Layer allows modules that are grouped together to communicate with each other.

Each individual module has access to a colored structure map of a software system, dynamically generated events from that system, and the ability to receive and transmit events to other visualizations. What the module does with these capabilities is left entirely to the discretion of the visualization designer.

3.5 Summary

In this chapter we discussed the major design goals of the *SoftViz* environment and applied these goals to the taxonomy of software visualization introduced in Chapter 2 to produce a set of design decisions. Next, we described the overall architecture of the resulting design, including the three-layer model and the idea of a generalized visualization module representing a class of visualizations.

4 Implementation of the *SoftViz* Environment

This chapter describes an implementation of the *SoftViz* environment.

4.1 Programming Language Selection

This implementation was written in pure Java using SDK 1.3.1. The primary design goals of the environment included portability and extensibility. Writing source code that can be easily maintained and extended is made easier by using any object-oriented language, but very few languages provide out-of-the-box portability as well as Java. A Java program compiled on any machine can run on any other machine that supports a Java Virtual Machine, from Alphas to Palms.

The design of the environment also called for a GUI and for the dynamic loading of packaged visualization modules. Java provides extensive API's for GUI development as well as capabilities for the packaging, deployment, and dynamic loading of compiled code. Additionally, Java provides API's for sound and 3D graphics programming, which may be of interest to visualization module developers.

4.2 Protocol and Format Selection

The design of the environment calls for an event notification protocol, a standardized event format, a file format for storing event data, and a means of saving state for visualizations.

4.2.1 Protocol

As the SIENA event notification bus is another technology under the DASADA program, we chose to incorporate it in this implementation. Several implementations of SIENA exist, and we chose the Java version for ease of integration. Future implementers should be aware that the Java version of SIENA exhibits very poor performance compared to more recent versions in C. This may not be a difference in the relative speed of the

languages so much as an inefficient Java implementation. To maximize the scalability of the environment, a more efficient version of SIENA should be used or a more efficient event notification bus should be adopted.

4.2.2 Event Format

The standard event format developed for this implementation contains five fields: timestamp, classname, methodsignature, interfacename, and threadid. The timestamp field is a long integer containing the number of milliseconds since January 1, 1970, 00:00:00 GMT. The classname field is a String of the fully qualified Java class name of the class which produced the event. The methodsignature field contains a String representation of the current method's name and arguments at the time the event was generated, formatted as the access modifier followed by the return type followed by a fully qualified Java class name followed by a period, the method name, and a comma-seperated list of arguments enclosed in parentheses.

The interfacename field contains a String, a comma-separated list of the interfaces that the class which produced the event implements, i.e. "java.io.Serializable, java.lang.Comparable, java.lang.Clonable". The threadid field contains a unique String identifier of the originating thread. This allows the visualization system to handle targeting a multi-threaded software system.

4.2.3 File Formats

The file format used for storing event data is a simple indexing scheme, which uses two files. The first file has an .idx extension and contains a list of colon separated pairs, the left side of each pair is a timestamp for an event and the right side an offset. The second file has a .dat extension and contains the list of all stored events. The offset in the first file is used to quickly find the location of the full event data in the data file. Although this works extremely well for this implementation, future implementers may consider using a full-fledged database, such as MySQL, which may improve speed and offer increased capabilities.

Saving the state of the environment is achieved by having all key classes used by the GUI, including the visualization modules themselves, implement Java's Serializable

interface. This allows the use of ObjectOutputStreams and ObjectInputStreams for the saving and recovery of the environment's state. Although this greatly simplifies some of the internal workings of the implementation, developers of visualization modules should be aware that this has two major ramifications for them. First, all classes used by a visualization module must extend the provided tvs.api.VizElement class, which ensures that they will implement the Serializable interface. Second, if a visualization module makes use of any class that does not implement the Serializable interface, such as java.util.Hashtable, they must declare the field volatile and override the default constructor, which is the constructor with no arguments, of the module's main class. When the visualization is deserialized from disk, the default constructor will be called and must re-instantiate all volatile fields.

4.2.4 Structure and Color Maps

The structure map used in this implementation of the *SoftViz* environment is defined by the provided classes tvs.jah.PackageTree and tvs.jah.PackageTreeNode. A PackageTree is comprised of a linked tree of PackageTreeNodes. A PackageTreeNode represents an object in the Java package hierarchy. In Java, fully qualified names of classes and interfaces include the name of the package, i.e. "java.lang.String" is in the java.lang package. Each PackageTreeNode contains a Vector of its child nodes and a link to its parent node in the package tree.

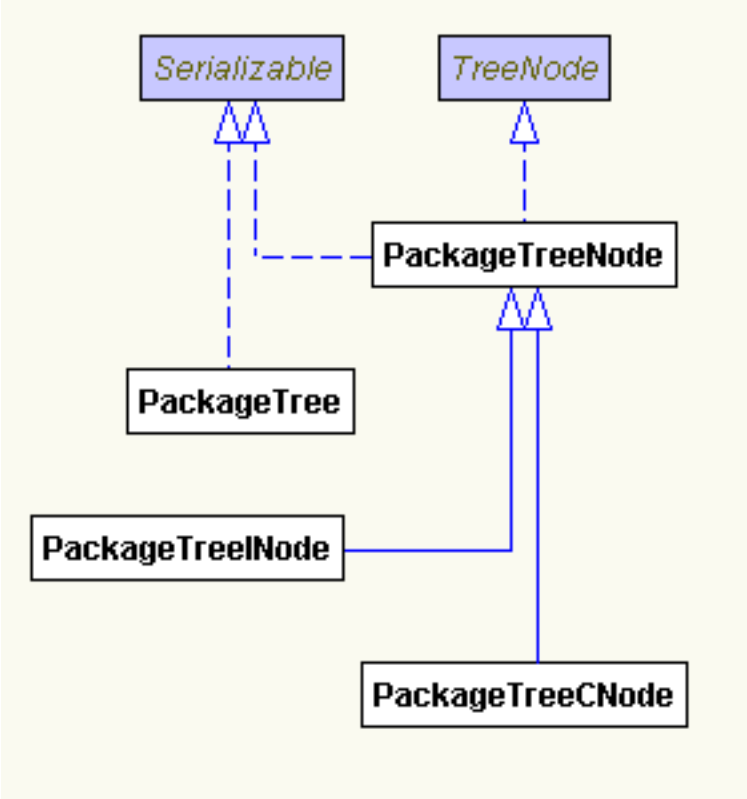


Figure 4.1 – UML Diagram of the Structure Map Data Structure

As shown above, PackageTreeNode has two subclasses: PackageTreeCNode and PackageTreeINode. PackageTreeINodes represent interfaces in the structure map. In addition to the parent and children provided by its superclass, PackageTreeINode contains a Vector of PackageTreeINodes which represent the interfaces that it extends.

PackageTreeCNodes represent classes in the structure map. In addition to their parent and children in the package tree, these nodes contain a link to the PackageTreeCNode representing their superclass and a Vector of the PackageTreeCNodes representing their subclasses. Additionally, PackageTreeCNodes contain a Vector of PackageTreeINodes representing all of the interfaces that the class implements.

This data structure comprises three trees: the package hierarchy, the class hierarchy, and the interface hierarchy, merged into one. The structure map is created in three steps by the Java Hierarchy Extractor (JAH), a modified Java parser created for this project. In the first step, all of the Java source files are parsed and the package hierarchy is created, stored as a tree containing packages, classes, and interfaces. Then a second

pass is made over the package tree to link the interface nodes together by inheritance. The third pass links the class nodes together by inheritance, and also links the class nodes to the interface nodes. The result of this is a web of linked nodes that contains all information about the relationships of packages, classes, and interfaces.

The color map used in this implementation of *SoftViz* is a simple hash table associating each element of a structure map to a particular color. The color map is created from the package hierarchy using the following algorithm. First, the number of leaf nodes in the package hierarchy is counted, and the color space is partitioned. Next, each leaf node in the package hierarchy is assigned a unique color based on the total number of leaf nodes. Finally, each parent node in the package tree is assigned the average color of its children. The implementation of this color map is provided in the class `tvs.api.TVSColor`.

4.3 The Three-Layer Model Implemented

This section describes the key interfaces involved in this implementation and the communication between the three layers of the model and the visualization modules.

4.3.1 Event-Substrate Layer

The most important interfaces for the Event-Substrate Layer in this implementation are `EventStream`, `StreamSubscriber`, and `ITVSEvent`.

The `EventStream` interface defines the basic operations allowed on all event streams, such as `play()` and `stop()`, and is a contract for communication between all three layers. This interface provides methods which publish information to the Visualization Layer about what operations are allowed on this stream, which the Visualization Layer uses to allow or disallow stream controls to the user.

The `EventStream` interface is implemented by the class `SienaEventStream` in the Event-Substrate Layer. This class provides all of the specific functionality involved in connecting to a *SIENA* server and receiving events.

The EventStream interface also provides the subscribe(StreamSubscriber) and unsubscribe(StreamSubscriber) methods, which is how visualization modules connect to a particular EventStream to receive events.

Visualization modules are required by the API to implement the StreamSubscriber interface, which handles the other side of these transactions by providing the handleEvent(ITVSEvent) method, which is how EventStreams notify Visualization modules of a new event.

The ITVSEvent interface describes the methods that every layer will be expecting an event to implement. This interface exists to allow for the later addition of different implementations. In this implementation, ITVSEvent has only one implementing class, TVSEvent, which is the common event representation used throughout the entire environment. Only TVSEvent is responsible for the translation of *SIENA* events or stored events into TVSEvents.

4.3.2 Laboratory Engine

The Laboratory Engine Layer consists of a sub-interface of the EventStream interface called AugmentedEventStream, which describes the additional controls available to the Visualization Layer when dealing with stored event data instead of live data. This interface and its implementing class StoredEventStream allow for playback of event data at various speeds and stepping through events backwards and forwards. StoredEventStream makes use of the classes EventStoreReader and EventStoreWriter, which handle the reading and writing of the event data to disk.

An additional component of the Laboratory Engine Layer, the Java Hierarchy Extractor (JAH), resides in its own package. This package contains a modified Java parser, created specifically for this project, which is the tool used for the extraction and creation of the structure maps which are passed into each visualization module.

4.3.3 Visualization Layer

The Visualization Layer is a GUI involving more than 20 classes. It provides dialogs for the creation of new streams, for the storage of event data to disk, and for the loading of a visualization module. Additional facilities are provided for the saving and recovery of state information, including the state of visualizations.

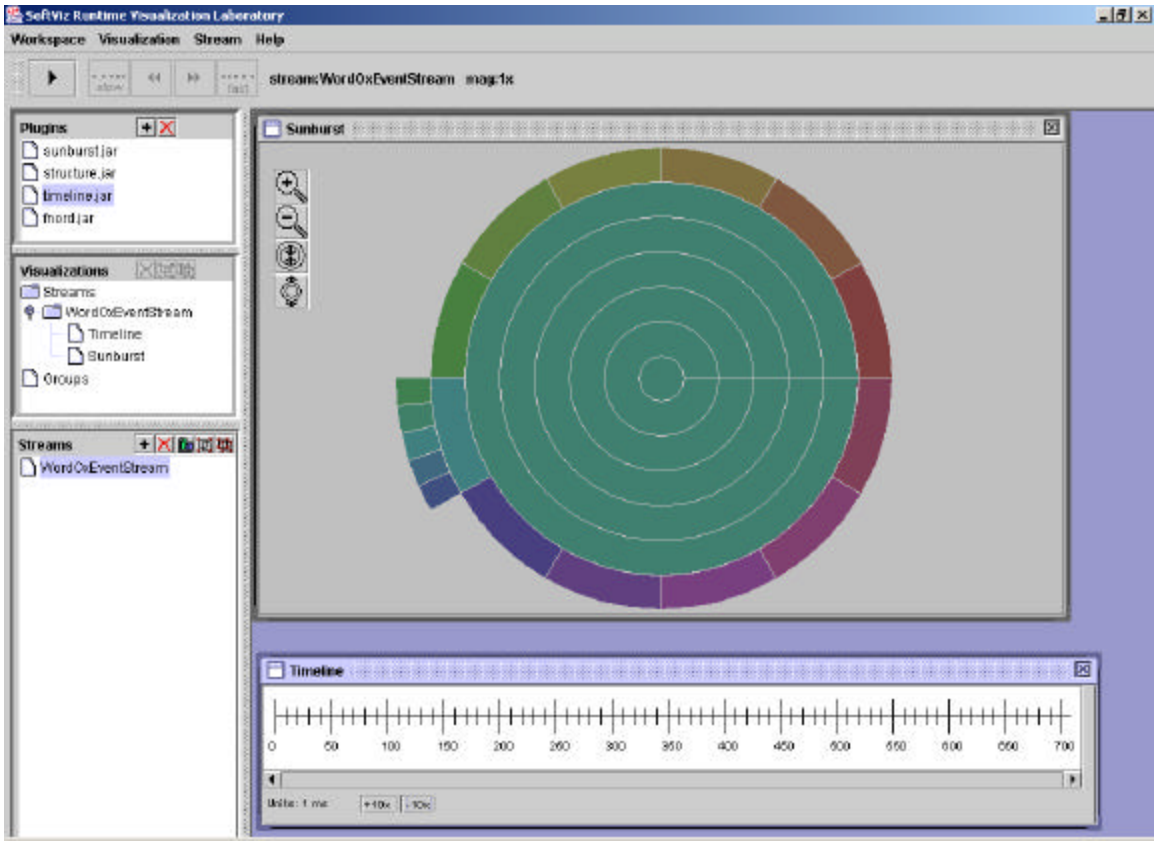


Figure 4.2 – Screenshot of a Prototype of *SoftViz* in Action

This layer also provides a widget resembling a VCR control. This widget, shown in the upper-left of Figure 4.2, provides the user the ability to control the flow of event data. Another widget, the Stream Selector, allows the user to select which stream the VCR controller is affecting. The Stream Selector is in the lower left of Figure 4.2. If a live stream is selected, the only options available on the VCR control are play and stop, due to communication with the Event-Substrate Layer via the `EventStream` interface. If

a stored stream is selected, the VCR control will allow additional options provided by the AugmentedEventStream interface in the Laboratory Engine Layer.

The Visualization Layer also presents the user with a list of registered visualization modules, shown in below the VCR controller in Figure 4.2. These visualization modules can be developed using the provided API which allows connectivity with the *SoftViz* environment. A diagram describing the class structure of the visualization module API is shown below.

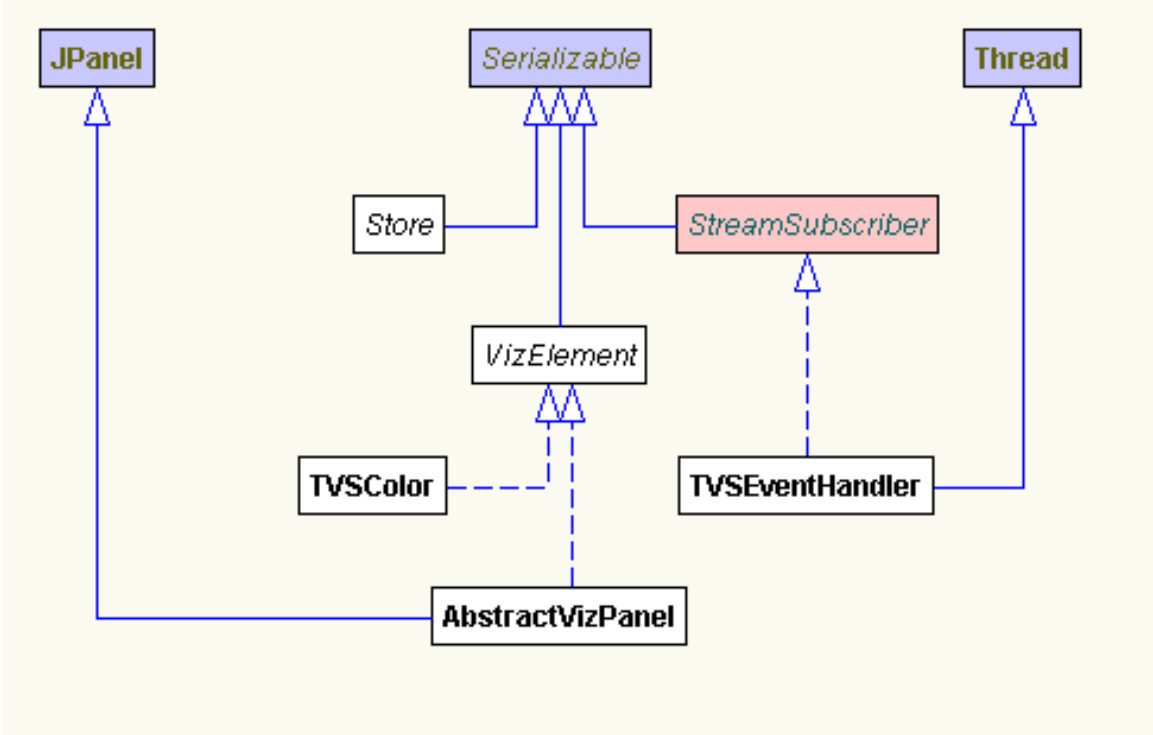


Figure 4.3 – UML Diagram of the Visualization Module API

The above diagram shows various elements of the API and their relationship to the rest of the environment. The key classes in the visualization module API are AbstractVizPanel and TVSEventHandler, both of which are abstract base classes. To create a custom visualization module, all that is required is the implementation of both these abstract classes. AbstractVizPanel is a subclass of the Java Swing class JPanel, and provides the drawing surface for the visualization and handles user input. TVSEventHandler implements the StreamSubscriber interface, which provides

connectivity to the Laboratory Engine Layer. The TVSEventHandler is responsible for handling incoming events from the instrumented software program that the visualization is targeting.

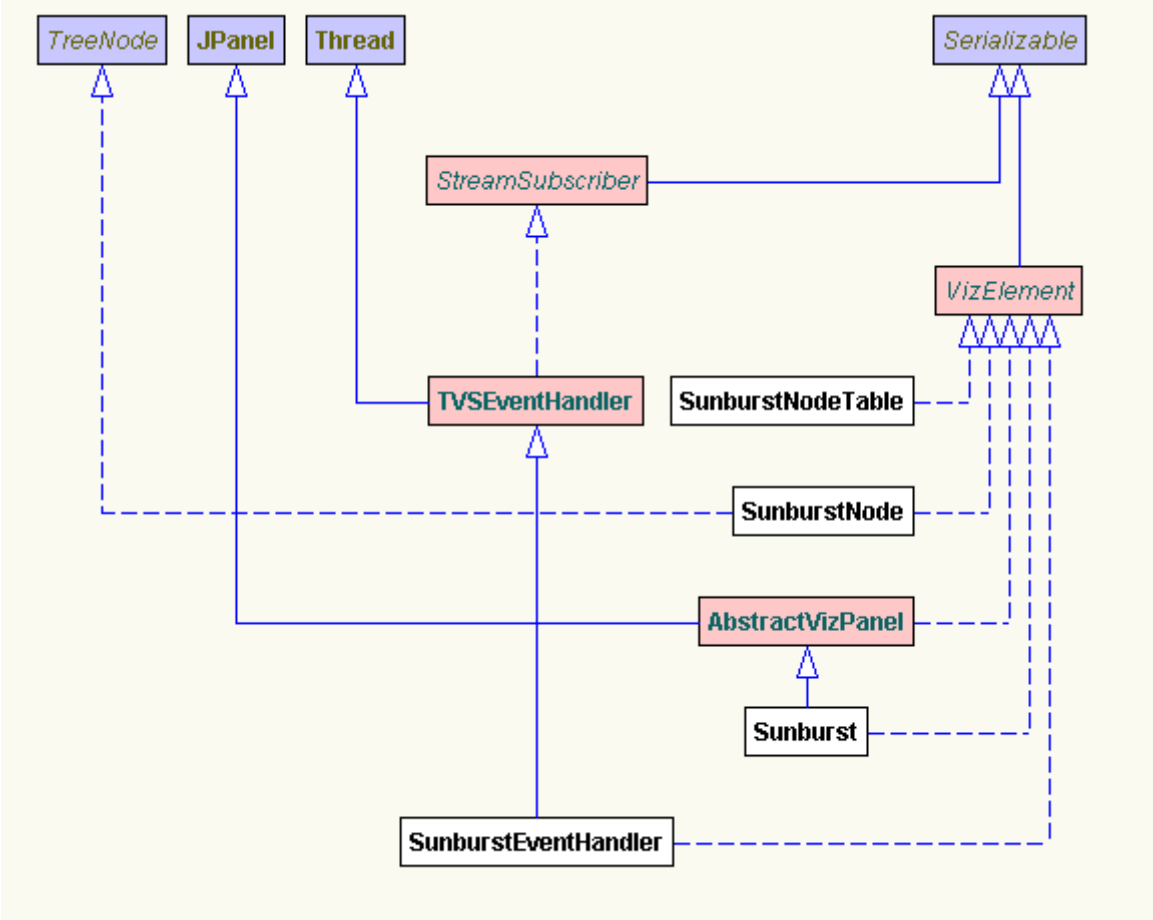


Figure 4.4 – UML Diagram of the Sunburst Visualization Module

Figure 4.4 shows the classes of an actual visualization module and their interaction with the API. The Sunburst class provides an implementation of AbstractVizPanel, and SunburstEventHandler implements TVSEventHandler. The Sunburst module makes use of two more classes: SunburstNodeTable and SunburstNode. These classes are required to implement the VizElement interface, which ensures that they will be properly serialized when the user saves the state of the environment.

An additional facility provided by the Visualization Layer is the grouping and communication of visualizations. The Group Selector widget allows the user to group visualizations together. Once these visualizations are grouped, user-generated events from one of the grouped visualizations propagate to all the members of the group.

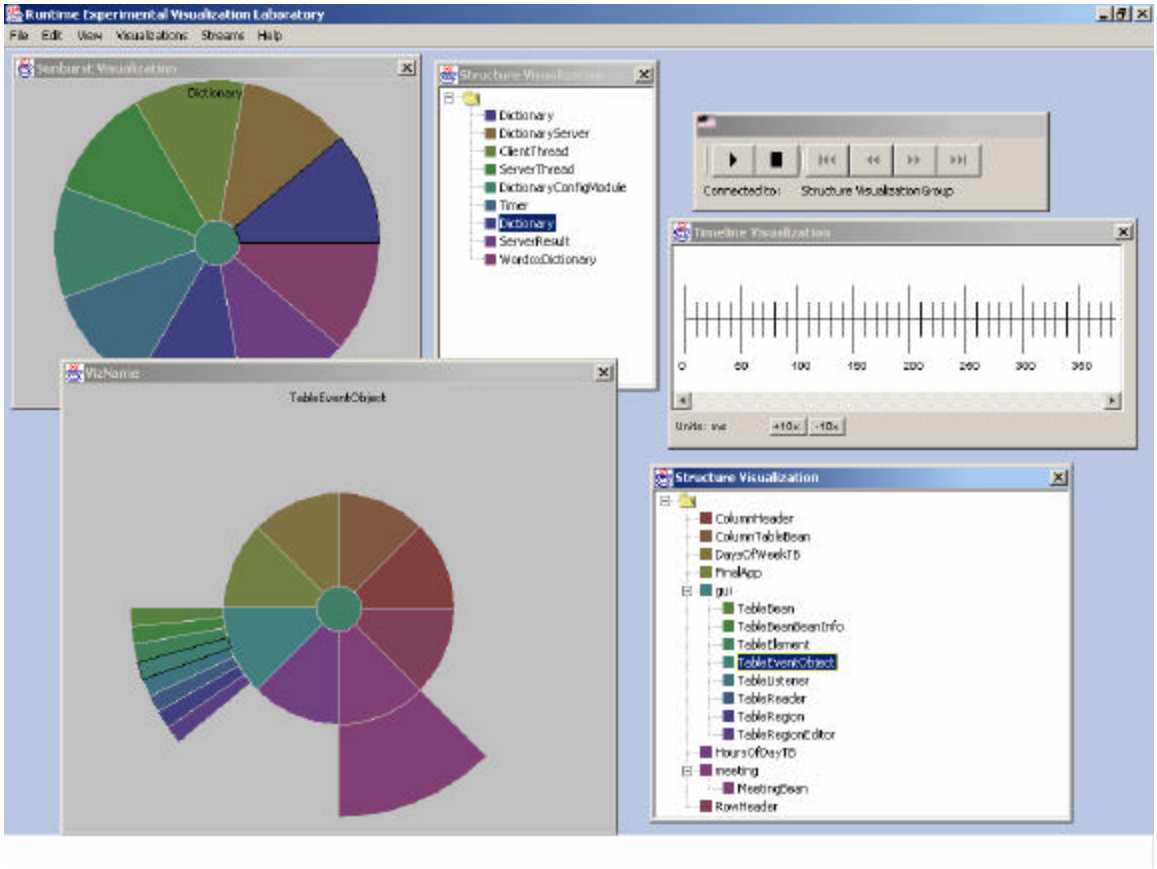


Figure 4.5 – Screenshot of an Early Prototype of *SoftViz* Demonstrating Visualization Events

The above screenshot of the *SoftViz* environment shows two separate groups of visualizations. One group occupies the upper region of the screen, and the second occupies the lower region. A user-generated selection on each Structure visualization has propagated to the other member of its visualization group, the associated Sunburst visualization. Selection is denoted on the Sunburst with a black outline, and on the Structure visualization with a box of inverted coloring.

This screenshot also provides an example of how the *SoftViz* environment maintains color consistency across different visualizations. As shown in Figure 4.5, the selected elements in each visualization group are decorated with an identical hue.

When a user-generated event, such as selection, occurs in a grouped visualization, a *Visualization Event* is propagated to the other visualizations in the group. Grouped visualization modules may use different data attributes to generate their visualizations. Some may use just structural elements and some may use only thread information. As a result, not all visualizations will be able to make use of a selection event generated by another.

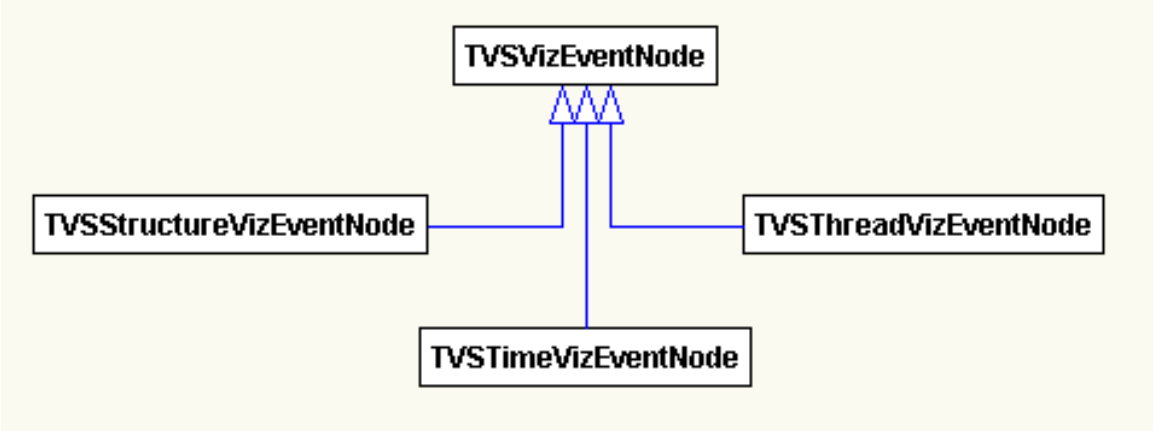


Figure 4.6 – UML Diagram of the Visualization Event Data Structure

TVSEventNode, shown in Figure 4.6, provides the common superclass that allows different visualization modules to communicate with each other. Its subclasses represent the different types of data a visualization may allow a user to select by, including time, thread id, and structural element.

4.4 Visualization Modules

Each visualization module defines a class of visualizations that can be constructed with the data provided by the structure map and the event stream. Provided with this implementation of *SoftViz* are three implemented visualization modules: Sunburst, Timeline, and Structure. Additional design has been performed on two additional

modules: Node-Link and Thread Tables. In the following sections, we discuss the principles and implementation of each module.

4.4.1 The Sunburst Visualization

The Sunburst visualization [49] represents a class of visualizations capable of representing a hierarchical data set in an intuitive and space-efficient manner.

4.4.1.1 Principles

For this module, we used the Sunburst visualization to convey the layered, hierarchal aspects of object-oriented source code through spatial representations. By using animation, the Sunburst module provides the user with dynamic information about what elements are currently executing, and allows for user interaction, including elision and selection operations.

The goal of the Sunburst module is to provide the user with a way of understanding both the static, hierarchal structure of a software system and the dynamic, control-flow of the system as it runs. Once the static hierarchy of the system is represented with a Sunburst, the flow of control between these elements is shown by raising the brightness and saturation of the element which is currently executing. The varying of brightness and saturation happens gradually across several seconds, creating a pulse effect. This allows the viewer of the visualization to follow the flow of the program.

To prevent confusion caused by screen clutter, the module should provide for the elision of sections of the hierarchy. This capacity for elision can also provide the user with a more abstract view of the hierarchy and control-flow of the target system.

4.4.1.2 Implementation

Sunburst is a radial display of segments of concentric circles, with the innermost circle representing the root node of the hierarchy. This implementation of the Sunburst visualization is based directly off the InterRing implementation developed by Jing Yang

at WPI [52]. An example Sunburst visualization taken from the *SoftViz* environment is shown below.

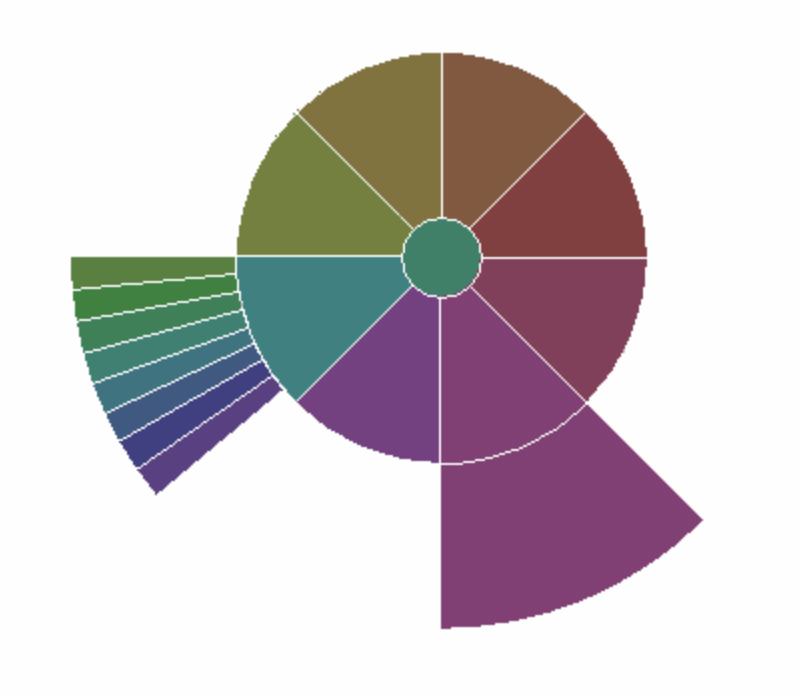


Figure 4.7 – Sunburst Visualization Example from *SoftViz*

The example shown in Figure 4.7 demonstrates a hierarchical set of 18 elements. The center circle represents the root node, and the surrounding 8 segments represent its children. The children of a node are initially given an equal portion of the radial space occupied by their parent. Although *SoftViz* uses the Sunburst visualization to represent the inheritance and packages hierarchies of object-oriented software, this class of visualization could be equally useful for any hierarchical data set.

As with all *SoftViz* visualization modules, the color for each structural element is calculated and preserved by the visualization environment, in order to ensure color consistency of corresponding elements across multiple visualizations. To maintain this consistency, visualization modules are not allowed to change the hue of a structure element. Sunburst avoids changing the hue of an element by the use of outlining, shading, and varying levels of brightness and saturation.

User interaction with the Sunburst module is performed with the mouse. Users can click on a particular structural element to select it. When an element is selected, it is outlined in black on screen.

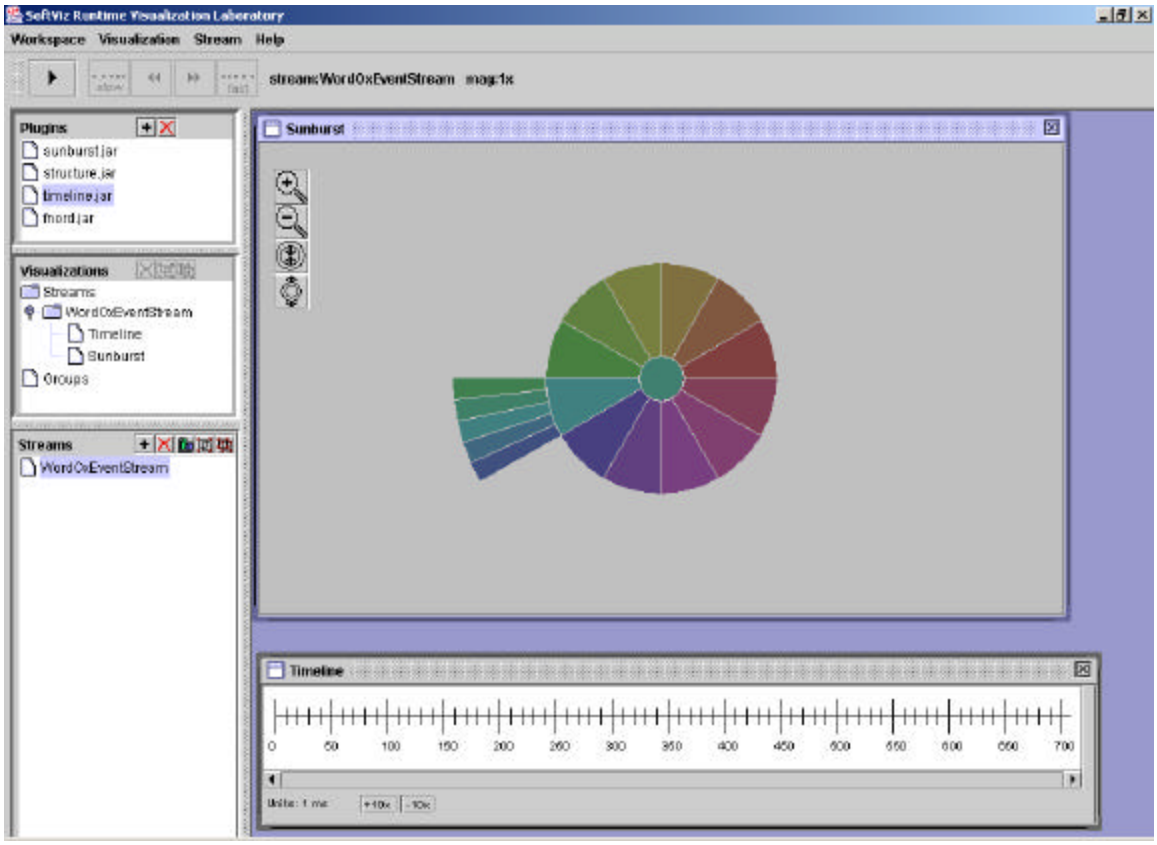


Figure 4.8 – Screenshot of a Prototype of *SoftViz* using Elision

The Sunburst visualization module provides four internal buttons, which allow the user to zoom in and out and to collapse and expand direct children of the root node that have no siblings. For example, the above figure is the same visualization as Figure 4.2, but with its central nodes, which have no siblings, collapsed.

Additional elision is allowed through a double-click on a parent node in the Sunburst. This feature allows for both the hiding of unnecessary data and for "zooming out" and getting a more abstract view of the behavior of a system. When this happens, all children of that node are hidden and the parent node is shaded. While a parent node is compressed, its brightness and saturation are pulsed instead of those of its children. For example, a user could collapse all leafs of a Sunburst and see how the control flows

between packages, super-classes, or interfaces. As part of the design of the *SoftViz* environment, the selection and elision operations are propagated to all other visualizations that are grouped together by the user.

4.4.2 TimeLine

The TimeLine visualization shows dynamic events on a timeline, colored according to the associated structural element.

4.4.2.1 Principles

The TimeLine module conveys the relationship of events along a temporal axis and contains additional information about the source of each event in the source code. As each event occurs, it will appear on the timeline as a colored bar. The color of the bar corresponds to the structural element from which the event originated.

This visualization module provides no information about the static structure of the system in itself, but provides a dynamic insight into temporal relationships of events, including the time between events and their ordering.

4.4.2.2 Implementation

A screenshot of the TimeLine visualization is shown below in Figure 4.9.

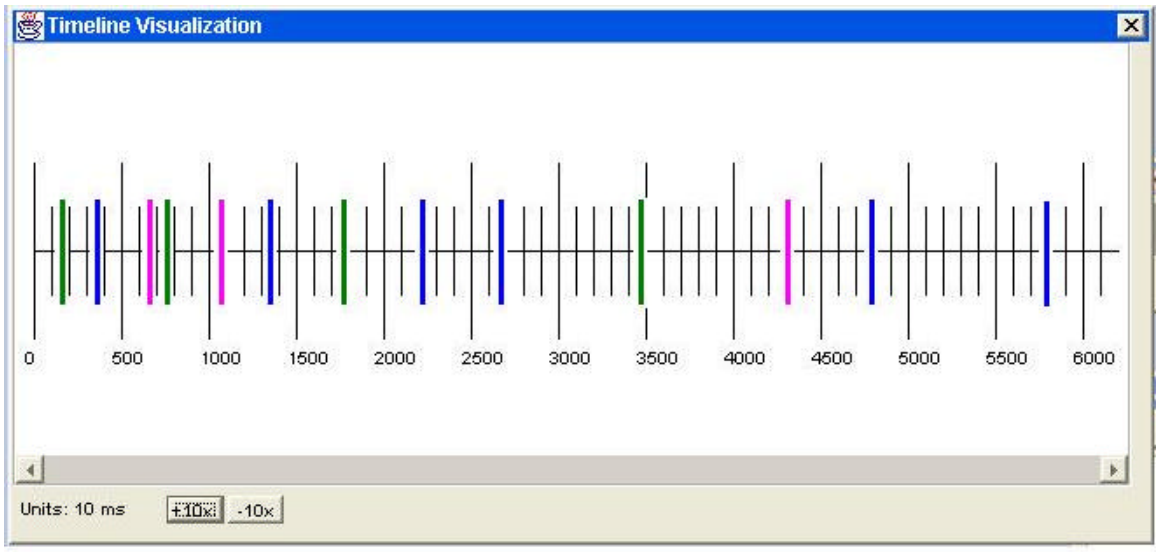


Figure 4.9 – The Timeline Visualization

This visualization module provides the user with a sense of the order in which events occur. The horizontal positioning of the bars indicates when they happened in time. The vertical height and the width of the bars are not used to convey information in this implementation. The color of the bars indicates which structural element is the source of the event. The above example shows three sources of events, represented by green, blue, and pink. This visualization provides no information about the control flow, however. In the above example, the first event is green, the second blue, and the third pink. The user cannot be certain if blue called pink or if blue returned and green called pink.

This module provides for the selection of a single event or a group of events on the timeline with a mouse drag. Selected regions are represented by inverting all of the colors in the selected region. To invert colors, the RGB values of the colors inside the selected region are XOR'ed with the RGB value for the color white. This selection event is propagated to all other visualizations that are in the same group.

Elision is provided by "zoom in" and "zoom out" buttons which can increase or reduce the scale of the timeline. The "zoom in" button, labeled "-10x" in Figure 4.9, decreases the scale of the timeline by a factor of ten. Correspondingly, the "zoom out" button, labeled "+10x", increases the scale of the timeline by a factor of 10. When this happens, the module moves all past events to their appropriate locations on the newly

resized timeline. The current level of time magnification is displayed onscreen as "Units:" and is measured in milliseconds. The timeline scales from units of 1 ms to units of millions of seconds.

4.4.3 Structure

The Structure visualization shows the name of each structural unit alongside its associated color.

4.4.3.1 Principles

The Structure module is a display of an expandable tree containing all of the structural units of a software system associated with their assigned color. This module is intended to allow the user to inspect the Object-Oriented structure of the target system and to provide a method of quickly locating the name of a particular structural element based on its assigned color.

4.4.3.2 Implementation

A screenshot of the Structure visualization in use is shown below in Figure 4.10.

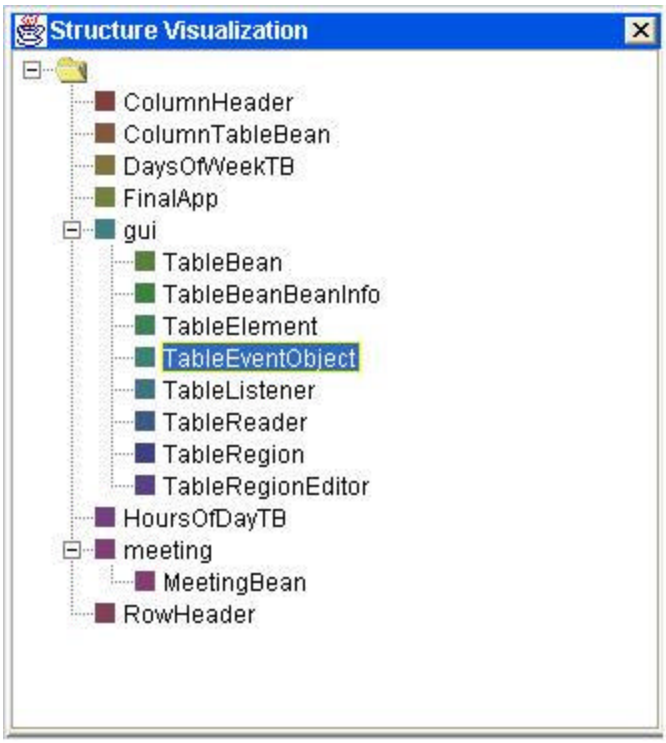


Figure 4.10 – The Structure Visualization

The above instance of the Structure visualization shows the package hierarchy of the instrumented program. Structure visualizations can also display the hierarchy of classes or interfaces. This module provides for elision by allowing every parent node to be collapsed or expanded.

As in the other modules, user selection operations are propagated to all other grouped visualizations. This is a particularly useful feature of this module, because it shows the mapping between the name of a structural element and its color. If another visualization module uses color to identify structural elements and does not provide the associated name of each element, the user can simply create a new instance of a Structure visualization and add it to the same group. When a structural element is selected from the Structure visualization, it will also become selected on the other visualization, and vice versa. Grouping any visualization with a Structure visualization adds the capacity to locate the name of a structure element with a selection operation on any visualization in the group.

This module also provides an onscreen legend for aid in identifying structural elements in other visualizations.

4.4.4 Node-Link

The Node-Link visualization provides information about the number and frequency of method calls between selected classes.

4.4.4.1 Principles

This visualization shows a user-selected set of interesting classes from the targeted software system. The visualization must be limited to only a set of interesting classes for screen space considerations. Each class is shown as a colored rectangle, or *node*, bordered by colored regions representing its superclass and all the interfaces that the class implements, as shown in Figure 4.11. The colors for the classes, superclasses, and interfaces are assigned by the visualization environment for consistency.

When the visualization is started, each node of the visualization is disconnected. When an event is received indicating that a class method has called a method of a different class, a colored arrow, or *link*, is drawn, connecting the two nodes of the visualization. If the method being called was provided by the superclass or an interface, the link points to the outer region of the node representing that superclass or interface. Additionally, if the calling method was provided by a superclass or interface, the link originates from the corresponding region of the source node.

The links themselves are colored according to the frequency of method calls. The color of a link can range from blue, representing infrequent calls, to red, representing very frequent calls. The width of a link corresponds to the total number of method calls.

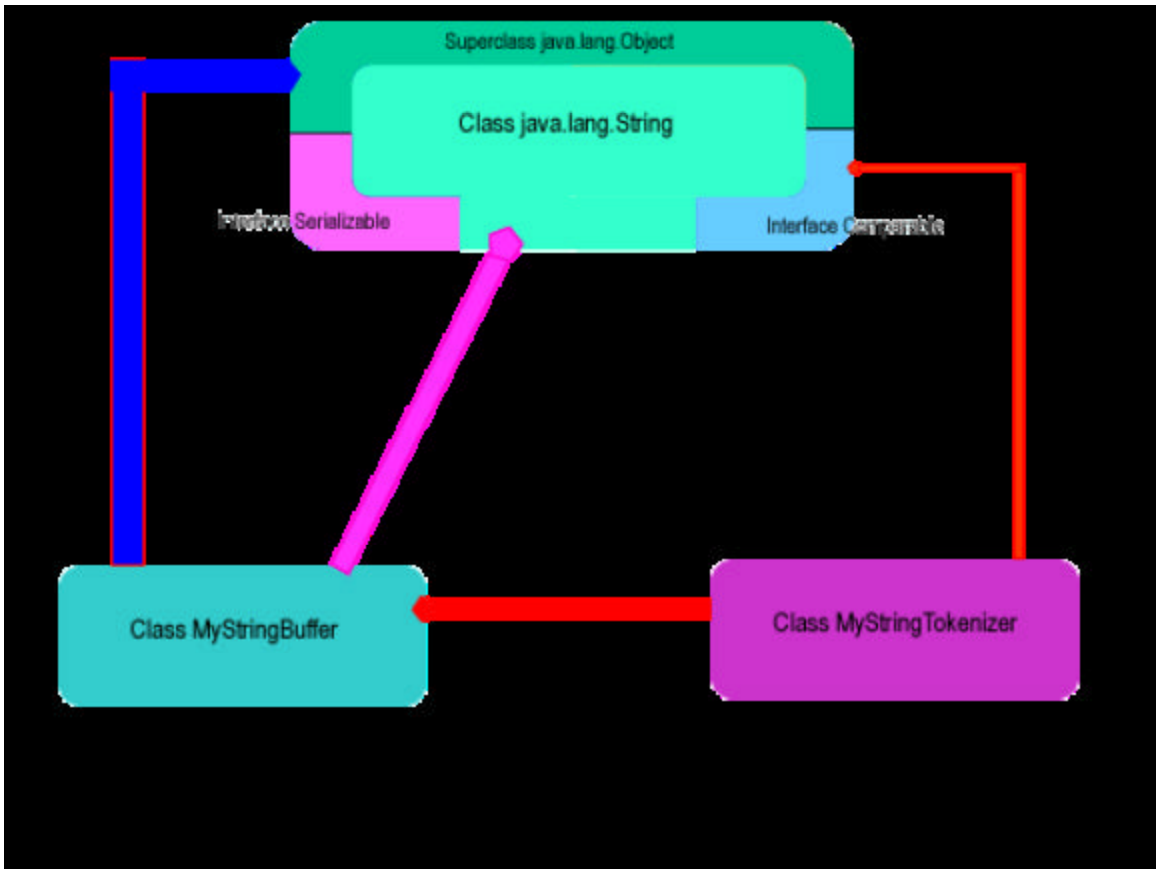


Figure 4.11 – The Node-Link Visualization

In this example, three classes are being visualized: String, MyStringBuffer, and MyStringTokenizer. The String class has a superclass, Object, which is represented by the upper half of the outer shell of the node. String also implements two interfaces: Serializable and Comparable, which are assigned regions of the lower half of the outer shell.

In the above example, four links are visible. The first link, from MyStringBuffer to String, indicates method calls made by class methods of MyStringBuffer to a method of the class String which was provided by it's superclass, Object. MyStringBuffer could be calling the toString() method, for example. This link is wide, indicating that many of these calls have been made, and blue, indicating that these calls have not been made frequently in the recent past.

The next link, from MyStringBuffer to String, indicates that several calls have been made relatively recently from class methods of MyStringBuffer to a class method of String, such as substring(). Another link, from MyStringTokenizer to

MyStringBuffer, indicates a number of recent method calls. Finally, the link from MyStringTokenizer to the Comparable interface of the String class indicates a small number of very recent calls to the compareTo() method, which is provided to the String class by the Comparable interface.

This visualization provides the user with information about class inheritance, interfaces, and the traffic between nodes. Due to the large amount of screen space required for each node, the Node-Link visualization requires sophisticated elision support. The first method of elision requires the used to choose which classes will be included in this visualization. Limiting the visualization to only interesting classes will avoid screen clutter.

The second method of elision is allowing the user the capacity to collapse and expand nodes in the visualization with a double-click. When a node is expanded, all of its interfaces and its superclass are visible. Links to the expanded node will target the superclass region, an interface region, or the class itself. The String node in Figure 4.11 is an example of an expanded node. When a node is collapsed, its superclass and interfaces will not be visible and all links targeting these regions will be aggregated into one link targeting the node. The MyStringTokenizer node in Figure 4.11 is an example of a collapsed node, as its superclass Object is not visible. This method of aggregation provides the user with the ability to choose whether or not to view a particular class in the context of its superclass and interfaces.

Although this visualization provides an interesting view of the structure and behavior of a target system, the implementation involves a constraint-solving engine capable of artificially intelligent layout for the automated handling of node and link placement.

4.4.5 Thread Table

The Thread Table visualization provides information about the activity level of the various threads running in a concurrent software system.

4.4.5.1 Principles

An example of the Thread Table visualization is shown below in Figure 4.12.

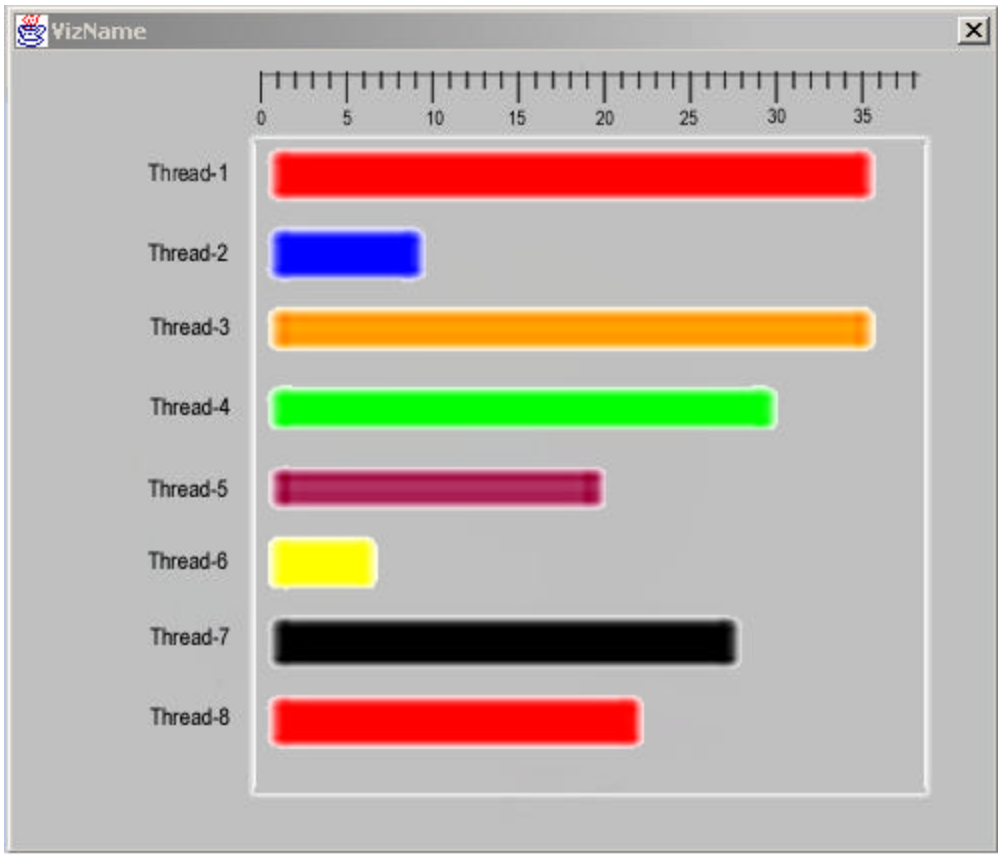


Figure 4.12 – Thread Tables Visualization

This visualization is a table representing each thread of the targeted software system and the number of events each thread has generated. In this example, color is used only to distinguish each bar from each other. In future implementations, it might be desirable to color each segment of the bar according to the assigned color of the structure element generating the event.

4.5 Summary

In this chapter, we described an implementation of the *SoftViz* environment, including the key classes in each layer and developed visualization modules.

5 Results

SoftViz provides an interactive environment combining visualizations of both the static structure and the dynamic behavior of complex systems, while fulfilling the design requirements.

5.1 Evaluation

As a part of the DARPA DASADA (Dynamic Assembly for System Adaptability, Dependability, and Assurance) program, *SoftViz* focuses on issues arising from the complexity of dynamically configurable architectures and incorporation with other DASADA technologies. A proof of concept demonstration of the *SoftViz* environment running on a dynamically reconfigurable client and server was performed at DARPA DASADA Demo Days in Baltimore, Maryland in July 2002. Additionally, the further development and evaluation of the *SoftViz* environment has evolved into an ongoing project at the Worcester Polytechnic Institute.

For serious consideration as a production software development tool, *SoftViz* requires both empirical evaluation as well as user studies. The invasive data-gathering technique used by *SoftViz* actually changes the source code of the system it is visualizing, and therefore has some effect on the behavior of the system it is visualizing. The extent of this effect can be examined by comparing the performance of otherwise identical systems instrumented with varying numbers of probes.

In addition to studying the performance cost of various numbers of probes, the scalability of the visualization system could be tested by finding the largest systems and largest number of probes that the visualization system can reasonably handle. This could be additionally tested by running the visualization system on several benchmark platforms and finding the maximum number of events that can be reasonably handled per second. None of the visualization systems that use invasive probing surveyed herein, including *SoftViz*, have been subjected to an empirical evaluation.

During the development and debugging of this implementation of the *SoftViz*, the environment was tested on several programs ranging in size from 200 to 2000 lines of code with no noticeable loss of performance. The number of probes inserted and activated during this testing was relatively small, however. The execution time of the probes was minimal, and it is expected that at least two hundred probes could be inserted and activated before the degradation of performance becomes noticeable on a Pentium III/400 Mhz. It is expected that the exact number of probes that can be inserted before the loss of performance becomes noticeable will vary with the computational power of the hardware. When running the environment on the same computer as the target system, performance of both programs can be degraded due to page faults if the computer does not have enough available resources.

Since the *SoftViz* environment is intended to help developers understand the software they work with, user studies of the environment and its visualization plug-ins are necessary. The ease of constructing new visualization modules, of using the pre-constructed visualizations, of instrumenting source code, and of using the environment itself should be evaluated. Several groups of developers, with varying levels of experience with the visualization system, could be given an unknown complex software system and asked to derive as much information from using the visualization system as possible, or to attempt several tasks of varying complexity. An additional control group with no access to the visualization system should be asked to perform the same tasks, to see if the system can actually reduce the amount of time it takes a user to comprehend a complex software system.

5.2 Future Work

The extensible nature of the *SoftViz* environment lends itself easily to future work, both on the environment itself and on its supporting technologies.

First, many additional visualization plug-ins can be developed to create a larger library of pre-built plug-ins. Providing a large set of general-purpose visualization modules will go a long way towards making *SoftViz* a useful development tool. Some of these visualization modules could incorporate three-dimensional graphics and sound,

using the Java 3D and the Java Sound API's. Although use of the Java 3D API has already been provided for, additional support for sound could be included in the Plug-in API, making it easier to use sound in a plug-in.

Additional facilities for the aggregation of event data and its compilation into "meta-events" could be developed. Currently, the aggregation of event data is performed by each visualization independently. There may be some advantage to more facilities in the laboratory engine layer for the pre-processing and compilation of events. The creation of new types of events, such as meta-events, would require additions to the Visualization Module API for the specification of what types of events each visualization module can handle.

At the moment, we assume that our event-generating probes are imbedded in a software system. We can remove that assumption with the creation of a system for codifying types of events by the data types of their fields, and a way of generalizing visualization modules to accept a variety of different types of events, or to accept event data by the type of data provided. Visualizations would be required to provide additional meta-data specifying what types of event data they require and additional, optional data types that the visualization can handle. The environment could keep track of a registry of visualization modules and what types of events they can visualize. When an event stream is created, the environment could provide the user with a list of available visualization modules that can accept the type of events produced by the stream.

In some cases, the environment may be able to automatically match the data fields of an event of arbitrary type to a published expected input of the visualization module. Timestamps, for example, would be a very common event data type that could be matched automatically. If an event contains several data fields of the same type, the user may be called upon to specify which event data field will be paired with a certain expected input of a visualization module. Consider a visualization module which draws a 3-dimensional bar graph based on incoming events. This module can only handle events which have at least three data fields: one of type <TimeStamp> and two of type <Integer>. The module may assume automatically that it will always use the value of <TimeStamp> for the position on the X axis, but will not be able to decide automatically which <Integer> corresponds to height and which to depth.

Generalization of event data and visualizations would allow *SoftViz* to use its set of visualizations to visualize the static structure and dynamic behavior of any system of remote probes. With the creation of probes for different types of systems, *SoftViz* could be used as a tool for the visualization of the structure and behavior of networks, file systems, or even events generated by computer-controlled monitoring of an external system such as a rat brain instrumented with electrodes or the stock market.

To better augment the *SoftViz* environment's usefulness to software developers, versions of the AIDE compiler could be developed for the instrumentation of programs written in other languages, such as C++ or C#. This would allow the visualization of programs written in these languages without making any changes to *SoftViz*.

Another possibility would be a new version of AIDE that could instrument compiled Java byte code with probes. This would allow the visualization of Java programs to which the source code was not available and the development of a complete back-end solution for the *SoftViz* environment. This package would combine the Java byte-code instrumenting program with the probe-monitoring infrastructure and a SIENA server. Once installed on a machine, a remote operator could connect to this machine using *SoftViz*, instrument already compiled programs on the machine with a variety of probes, activate or inactivate any of these probes at will, and collect the events generated by this system for runtime remote visualization. The back-end package's design should allow for the possibility of future uses of this environment for visualization of systems other than software.

Simpler optimizations can be made to the *SoftViz* environment by adding support for the JDBC to the Laboratory Engine Layer. The use of this Java API would allow the use of commercial databases, such as Oracle or SQL, for persistent storage of event data. Additionally, the Event Transport Bus can be further optimized by using binary XML instead of plain text, and by switching to a C implementation.

5.3 Conclusions

By combining streams of event data with extracted structural information with generalized visualization templates, this environment provides a powerful tool for both

professionals and students of software engineering. Such a system should provide a means for identifying performance problems, illuminating incorrect behaviors, and revealing flawed designs that is not provided by today's development tools.

Appendix A

A Programmer's Guide to SoftViz Module Development

This implementation features an API which is provided as a toolkit for the developer of new visualization modules. This section describes in detail the process of creating a new visualization module by using the provided API.

A.1 Getting Started

A visualization module for *SoftViz* requires the implementation of two abstract classes, which are provided in the tvs.api package: TVSEventHandler and AbstractVizPanel.

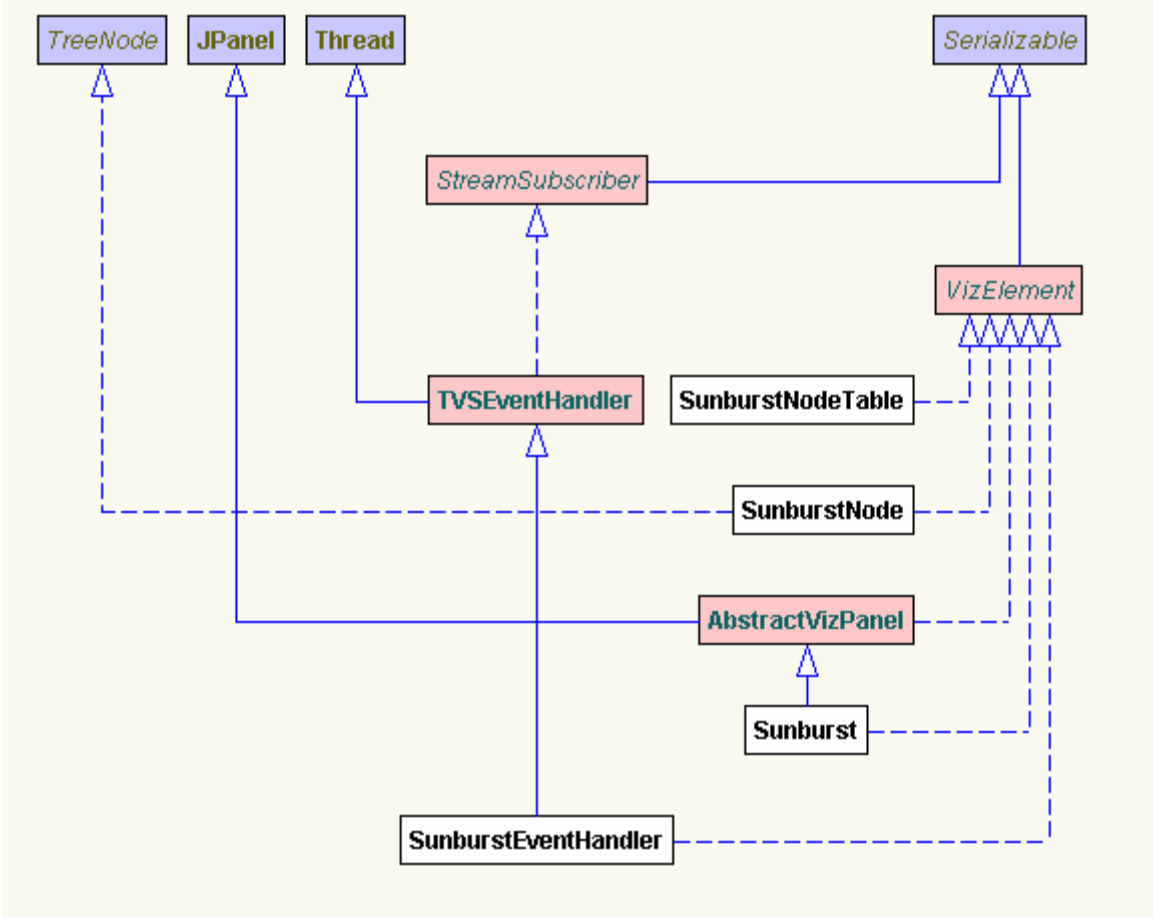


Figure A.13 – UML Diagram of the Sunburst Visualization Module

The above figure shows a UML class diagram of the Sunburst visualization module. The main class of a visualization module, which implements tvs.api.AbstractVizPanel, is responsible for drawing the visualization to the screen. The other required class, which implements tvs.api.TVSEventHandler, is a the Thread which is responsible for periodically redrawing the visualization and receiving events generated from the target program. All other classes used by the module must also implement the tvs.api.VizElement interface, which insures that they will be serializable by the system when the user saves state.

The main class of the module is required to have at least two constructors. The first constructor is called by *SoftViz* when the module is loaded, and is responsible for the initial setup of the visualization. This constructor has the following signature:

```
public Sunburst(String name, javax.swing.tree.TreeNode root, tvs.api.TVSColor colors)
```

The constructor receives three arguments: a String which is the name of this instance of the visualization, a TreeNode which is the root of a tree describing the hierarchical structure of the target program, and a TVSColor object which provides a lookup table containing the color of every structural object.

The following pseudo-code provides the basic behavior of this constructor:

```
public Sunburst(String name, javax.swing.tree.TreeNode root, tvs.api.TVSColor colors) {
        super(name, root, colors);
        processTree(root);  // Converts the tree to a usable form
        colorSchema = colors;  // Saves a handle to the color table locally
        // Any other setup required
        eventhandler = new SunburstEventHandler(this);
        eventhandler.start();
}
```

The last two lines of this constructor create a new TVSEventHandler and start its thread. This is important if you plan on your visualization module receiving dynamic events. If your visualization is static, this is not necessary.

Next, a second constructor is required, which has the following signature:

```
public Sunburst()
```

In Java, this is usually referred to as the "default constructor". This constructor must be overloaded if your module uses any objects which do not implement the Serializable interface. You will know if this is the case if you are required to use the "transient" keyword on a field in order to get the module to compile. If you have declared any fields transient, you must overload the default constructor and include code that restores these non-transient fields to their original state, as it is this constructor which will be called when *SoftViz* loads a module whose state has been saved by the user. For example, you may have to create a new TVSEventHandler and start it up again.

A.2 Graphics

The routine which is primarily responsible for drawing to the graphics context is the paintComponent method in the class which extends AbstractVizPanel. By overriding the behavior of this class, you can utilize the standard Java Graphics routines. It is also possible to use the Java2D and Java3D libraries as follows:

```
public void paintComponent(java.awt.Graphics g) {
        this.paintComponent((java.awt.Graphics3D) g);
}
public void paintComponent(java.awt.Graphics3D g) {
        // insert 3D Graphics code here
}
```

Throughout the rest of your module code, calling repaint() will result in the execution of paintComponent() as soon as possible.

When writing your graphics code, remember that you are essentially drawing to a JPanel (a standard Java Swing type) and not a Frame. The Frame your module's panel will be placed in is owned by the *SoftViz* program.

A.3 Stream-Based Event Handling

Incoming events from the instrumented program will be handled by the following method in your class which extends TVSEventHandler:

```
public void handleEvent(tvs.stream.ITVSEvent n) {
        String className = n.getClassName();  // Gets name of structure element
        // insert Event handling code
}
```

The methods that provide access to the event data are described in tvs.stream.ITVSEvent, including getClassName().

A.4 Visualization Message Passing

Visualization modules can also pass messages to each other, including user-generated selection events. These messages are passed only between visualizations that have been grouped together by the user. If you wanted to send a SELECT event when the user clicked on an on-screen structure object, add the following code to your mousePressed method:

```
// generate SELECT event (as well as de-select events).
TVSVizEvent ve = new TVSVizEvent(TVSVizEvent.SELECT);
// keyname is the String description of the structural element selected
ve.addNode(new TVSStructureVizEventNode(node.keyname));
// Add as many nodes to this event as you like!
notify(ve);
```

The notify() method is fully implemented by AbstractVizPanel, and will publish your event to all other visualizations in the same group.

To respond to events generated by other visualizations, you must implement the following method, found in the class of your module which extends AbstractVizPanel:

```
public void handleVizEvent(tvs.api.TVSVizEvent event) {
   switch (event.getType()) {
      case tvs.api.TVSVizEvent.SELECT :
         {
            java.util.Iterator iter = event.nodes();
            while (iter.hasNext()) {
               TVSVizEventNode en = (TVSVizEventNode) iter.next();
               if (!(en instanceof TVSStructureVizEventNode))
                  continue;
                              // Insert code to handle a single node of a SELECT event here
                              }
                              }
}
```

Visualization events are explained in more detail in the file tvs.api.TVSVizEvent.java.

A.5 Deployment

The *SoftViz* program requires that visualization modules be packaged as JAR files and that they have a special entry in the JAR Manifest file. In a standard deployment, this Manifest file is located in a subdirectory of the main package directory called META-INF. The JAR utility can be found in the bin/ directory of your Java SDK installation.

To package your visualization module for deployment, create a directory called META-INF off of the base directory of your module code. For the Sunburst visualization, the base directory listing looks like:

```
drwxr-xr-x    2 root     root         4096 Apr 17 14:30 META-INF
drwxr-xr-x    3 root     root         4096 Apr 17 14:30 tvs
```

Now, create a file called Manifest.mf in the META-INF directory. At a minimum, this file should contain the following two lines:

```
Manifest-Version: 1.0
Main-Class: tvs.sunburst.Sunburst
```

The Main-Class attribute of the Manifest file tells *SoftViz* where to begin the execution of the visualization module. For your visualization, change tvs.sunburst.Sunburst to the location of the class of your visualization module that extends tvs.api.AbstractVizPanel. Notice that the Manifest file uses package notation rather than directory notation, and that there is no ".class" extension.

Finally, return to the base directory of your visualization module code. Make sure that everything is compiled, and run the JAR utility:

```
jar cvfm sunburst.jar META-INF/Manifest.mf tvs
```

This will create a JAR file called sunburst.jar which incorporates the Manifest file in META-INF and the classes found in the tvs directory.

Now, load the *SoftViz* program, load your new module, and try it out!

References

[1] Baecker, R. M. (1981). *Sorting Out Sorting*. Narrated color videotape, 30 minutes, presented at ACMSIGGRAPH '81 and excerpted in ACM SIGGRAPH Video Review #7, 1983. Los Altos, CA: Morgan Kaufmann.

[2] Bentley, J. L. & Kernighan, B. W. (1991a). "A System for Algorithm Animation." *Computing Systems*,**4**(1): 5-30.

[3] Bentley, J. L. & Kernighan, B. W. (1991b). *A System for Algorithm Animation (Tutorial and User Manual)* (Computing Science Technical Report No. 132). AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

[4] Brown, M. H. (1988a). *Algorithm Animation*. New York: MIT Press.

[5] Brown, M. H. (1988b). "Exploring Algorithms Using Balsa II." *IEEE Computer*, **21**(5): 14-36.

[6] Brown, M. H. (1991). "Zeus: A System for Algorithm Animation and Multi-View Editing. "*Proceedings of IEEE Workshop on Visual Languages*, New York: IEEE Computer Society Press: 4-9.

[7] Brown, M. H. (1992). *Zeus: A System for Algorithm Animation and Multi-view Editing* (Research Report No. 75). DEC Systems Research Center, Palo Alto, CA.

[8] Brown, M. H. & Hershberger, J. (1991). "Color and Sound in Algorithm Animation." *Computer*, **25**(12):52-63.

[9] Brown, M. H., Myrowitz, N., & van Dam, A. (1983). "Personal Computer Networks and Graphical Animation: Rationale and Practice for Education." *ACM SIGCSE Bulletin*, **15**(1): 296-307.

[10] Brown, M. H. & Najork, M. A. (1993). *Algorithm Animations Using 3D Interactive Graphics* (Technical Report). DEC Systems Research Center, Palo Alto, CA.

[11] Brown, M. H. & Sedgewick, R. (1984a). "Progress Report: Brown University Instructional Computing Laboratory." *ACM SIGCSE Bulletin*, **16**(1): 91-101.

[12] Brown, M. H. & Sedgewick, R. (1984b). "A System for Algorithm Animation." *Proceedings of ACM SIGGRAPH '84*. New York, ACM Press: 177-186.

[13] Brown, M. H. & Sedgewick, R. (1985). "Techniques for Algorithm Animation." *IEEE Software*, **2**(1):28-39.

[14] DARPA, (2002). "Dynamic Assembly for Systems Adaptability, Dependability, and Assurance" (DASADA) **http://www.rl.af.mil/tech/programs/dasada.**

[15] De Pauw, W., Mitchell, N., Robillard, M., Sevitsky, G., and Srinivasan, H. (2001). "Drive-by Analysis of Running Programs," *Proceedings for Workshop on Software Visualization, International Conference on Software Engineering*, Toronto.

[16] De Pauw, W., Helm, R., Kimelman, D., and Vlissides, J. (1993). "Visualizing the Behavior of Object-Oriented Systems", *OOPSLA '93 Conference Proceedings*, Washington, D.C.: 326-337.

[17] Eagan, J., Harrold, M., Jones, J., and Stasko, J. (2001). "Visually encoding program test information to find faults in software," Georgia Institute of Technology Technical Report No. GIT-GVU-01-09.

[18] Fua, Y., Ward, M.O., and Rundensteiner, E.A. (1999). "Navigating hierarchies with structure-based brushes," *Proc. Information Visualization:* 58-64.

[19] Fua, Y., Ward, M.O., and Rundensteiner, E.A. (2000). "Structure-based brushes: A mechanism for navigating hierarchically organized data and information spaces," *IEEE Trans. Visualization and Computer Graphics,* Vol. 6, No. 2: 150-159.

[20] Georgia Institute of Technology, (2002). *Polka, Samba and XTANGO,* **http://www.cc.gatech.edu/gvu/softviz/**.

[21] Gill, P.W. (2001). "Probing for a Continual Validation Prototype," Master's Thesis, Computer Science Department, Worcester Polytechnic Institute.

[22] Goldstein, H. H. & von Neumann, J. (1947). "Planning and Coding Problems of an Electronic Computing Instrument." A. H. Taub (Eds.), *von Neumann, J., Collected Works.* New York: McMillan: 80-151.

[23] Haibt, L. M. (1959). "A Program to Draw Multi-Level Flow Charts." *Proceedings of the Western Joint Computer Conference*, 15. San Francisco, CA: 131-137.

[24] Heineman, G. (2002). "Coping with Complexity: A standards-based kinesthetic approach to monitoring non-standard component-based systems," **http://www.cs.wpi.edu/~heineman/dasada/.**

[25] Henry, R. R., Whaley, K. M., and Forstall, B. (1990). "The University of Washington Program Illustrator (UWPI)." *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*: 223-233.

[26] IBM Corporation, (2002). *Jinsight,* **http://www.research.ibm.com/jinsight/**.

[27] IBM Corporation, (2002). *PV*, **http://www.research.ibm.com/pv/**.

[28] Jerding, D., and Stasko, J. (1998). "The information mural: a technique for displaying and navigating large information spaces," *IEEE Trans. Visualization and Computer Graphics,* Vol. 4, No. 3: 257-271.

[29] Kimelman, D., Rosenburg, B., and Roth, T. (1994). "Strata-Various: multi-layer visualizations of dynamics in software system behavior," *Proc. Visualization '94*: 172-178.

[30] Knowlton, K. C. (1966a). *L [6]: Bell Telephone Laboratories Low-Level Linked List Language*. 16 mm black and white sound film, 16 minutes. Murray Hill, NJ: Technical Information Libraries, Bell Laboratories, Inc.

[31] Knowlton, K. C. (1966b). *L [6]: Part II. An Example of L [6] Programming*. 16 mm black and white sound film, 30 minutes. Murray Hill, NJ: Technical Information Libraries, Bell Laboratories, Inc.

[32] Knowlton, K. C. (1966c). "A Programmer's Description of L [6]." *Communications of the ACM*, **9**(8):616-625.

[33] Knuth, D. E. (1963). "Computer-Drawn Flowcharts." *Communications of the ACM*, **6**(9): 555-563.

[34] Ledgard, H. F. (1975). *Programming Proverbs.* Rochell Park, NJ: Hayden.

[35] Myers, B. A. (1986). "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy." M. Mantei & P. Orbeton (Ed.), *Proceedings of Human Factors in Computing Systems (CHI '86)*. New York: ACM Press: 59-66.

[36] Myers, B. A. (1988). *The State of the Art in Visual Programming and Program Visualization* (Technical Report No. CMU-CS-88-114). Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA.

[37] Myers, B. A. (1990). "Taxonomies of Visual Programming and Program Visualization." *Journal of Visual Languages and Computing*, **1**(1): 97-123.

[38] Nassi, I. & Shneiderman, B. (1973). "Flowchart Technique for Structured Programming." *ACM SIGPLAN Notices*, **8**(8): 12-26.

[39] Price, B., Baecker, R., and Small, I. (1993). "A principled taxonomy of software visualization," *Journal of Visual Languages and Computing*, Vol. 4: 211-266.

[40] Princeton University, (2002). *WordNet*, **http://www.cogsci.princeton.edu/~wn/**.

[41] Sevitsky, G., De Pauw, W., Konuru, R. (2001). "An Information Exploration Tool for Performance Analysis of Java Programs", *TOOLS Europe 2001*, Zurich, Switzerland.

[42] SmartDraw.com, (2002). *SmartDraw*, **http://www.smartdraw.com/resources/centers/software/nassi.htm**.

[43] Software Engineering Research Laboratory, University of Colorado, (2002). *Siena*, **http://www.cs.colorado.edu/serl/siena/**.

[44] Stasko, J. (1990). "TANGO: A framework and system for algorithm animation." *IEEE Computer*, 23(9): 27-39.

[45] Stasko, J. (1997). "Using student-built algorithm animations as learning aids." *Proceedings of the SIGCSE Technical Symposium on Computer Science Education*. New York, NY: ACM Press: 25-29.

[46] Stasko, J., Badre, A., & Lewis, C. (1993). "Do algorithm animations assist learning? An empirical study and analysis." *Proceedings of the INTERCHI'93 Conference*. New York, NY: ACM Press: 61-66.

[47] Stasko, J., and Patterson, C. (1992). "Understanding and characterizing software visualization systems," *Proc. IEEE Workshop on Visual Languages:* 3-10.

[48] Stasko, J. T. & Whrli, J. F. (1992). *Three-Dimensional Computation Visualization* (Technical Report No. GIT-GVU-92-20). Graphics, Visualization, and Usability Center, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280.

[49] Stasko, J., and Zhang, E. (2000). "Focus and context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations," *Proc. Information Visualization*: 57-65.

[50] Tufte, Edward R. (1983). *The Visual Display of Quantitative Information*. Graphics Press.

[51] Ward, M.O., and Heineman, G. (2001). "A Framework for Visualizing the Behavior of Component-Based Software Systems," *Software Visualization Workshop at OOPSLA 2001*, Tampa, Florida.

[52] Yang, J., Ward, M.O., and Rundensteiner, E.A. (2002). "InterRing: an interactive tool for visually navigating and manipulating hierarchical structures," *Proc. IEEE Symposium on Information Visualization*: 77-84