AUTOMATED VISUALIZATION OF THE VERSION HISTORY OF A SOFTWARE SYSTEM IN THREE DIMENSIONS

RAMYA ASOKAN

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Information Systems

Dr. Denis Gracanin, Chair
Dr. Shawn Bohner
Dr. Athman Bouguettaya

September 21, 2003
Falls Church, Virginia

Keywords: three-dimensional visualization, software history visualization, version history visualization, visualizing CVS, visualizing software evolution, visualizing software releases.

AUTOMATIC VISUALIZATION OF THE VERSION HISTORY OF A SOFTWARE SYSTEM IN THREE DIMENSIONS

RAMYA ASOKAN

ABSTRACT

Software changes constantly and continuously. It is often beneficial to record the progressive changes made to software, so that when any problems arise, it is possible to identify the change that might have caused the problem. Also, recording these changes enables recovery of the software as it was at any point of time. A *version control system* is used to track modifications to software. Version control systems (VCS) display when and where a change was made. In the case of multiple developers working on the same software system, version control systems also record which developer was responsible for the change. RCS, SCCS and CVS are examples of such version control systems, and they usually have a command-line interface. The widespread use of CVS has however given rise to a host of 'CVS clients', which provide a two-dimensional graphical interface to CVS. While working with a version control system in two dimensions is a definite improvement over traditional command line interfaces, it is still not sufficient to display all the necessary information in a single view. Using three dimensions to display the information from a version control system like CVS is an effective and efficient way to represent multiple attributes in a single view. There are many advantages to using a third dimension for visualizing the version history and evolution of software. A three-dimensional visualization tool has been developed to provide insights into the structure and characteristics of the history of a software system. It demonstrates the benefits of three-dimensional visualization and illustrates a framework that can be used to automatically derive information from a version control system.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my academic advisor, Dr. Denis Gracanin, for his valuable guidance, support and patience throughout the period of my graduate studies at Virginia Tech. I would also like to thank my committee members, Dr. Shawn Bohner and Dr. Athman Bouguettaya, for taking time off their busy schedules to provide invaluable input towards the completion of my Masters thesis. I also appreciate all the support provided by the faculty, staff and my colleagues in the Department of Computer Science, Virginia Tech at the Northern Virginia Center

DEDICATIONS

This work is dedicated to my dad who has been a role model throughout my life. I hope to achieve at least a fraction of the dedication, sincerity and diligence he applies to his profession as a Neurologist. I couldn't have completed my Masters without the understanding and support from my family. Thank you dad, mom and Charan. A special thanks to my grandmother who has been a pillar of strength when I most needed it. And of course, none of this would have been possible without my fiancé, Vikram, whose love and encouragement has helped me reach farther than I thought I could.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 PROBLEM STATEMENT

It is no secret that software changes constantly and continuously. Debugging and maintenance costs associated with software increase in proportion to the size of the software and the number of people who developed it or are developing it. At any point during the evolution of a software system, it should be possible to retrieve documents associated with the system as they were on a particular date or at a particular time. In his book 'Software Configuration Management – Coordination for Team Productivity' [41], Wayne A. Babich states that 'real-world software exists in alternate forms or versions, successive releases, or custom-tailored adaptations for different environments'. So, as software changes, there should be some mechanism available to store these variations to allow retrieval of the software as it used to be before it was changed. As it evolves, software can be seen as consisting of incremental *revisions*. A software system is typically a collection of packages or subsystems or modules, at the highest level of abstraction, and files consisting of source code at the lowest level. Tracking the revisions a software system makes during its lifetime is equivalent to tracking the various *versions* of the files it comprises.

A version control system can be used for purposes of tracking changes and modifications made to files in a software system. While existing version control systems provide an excellent command-line or two-dimensional graphical interface to the characteristics of changes made to files in a system, there is much more useful information that can be derived from these version control systems and effectively presented to the end user using three dimensions. Three-dimensional interfaces pack more details in the same space as a two-dimensional interface, and enable faster comprehension of the displayed information. This thesis emphasizes the use of three dimensions for effective and efficient visualization of data from a version control system, and details the design and implementation of a visualization tool that provides useful views of a version control system in three dimensions.

1.2 RESEARCH OBJECTIVES AND APPROACH

To understand the concept of a version, consider this simple example: A software project consists of files a, b, c and d. A developer could modify any of the files at any time. Every time a file is modified, it results in a new version of that file. So, a file could consist of versions numbered 1,2,3…n when n modifications have been made to it. Section 3 talks more in detail about revisions, the relationship between revisions and a software release and how information about a modification to a file is recorded. These revisions made to a file have to be reported to some entity so that they can be tracked. That entity is a tool commonly used in software engineering, called a *version control system*. Again, Section 3 details the workings of a version control system and some commercial examples of these tools, like CVS, RCS and SCCS. At a glance, a version control system (VCS) should provide the following information about the software system it is tracking:

- The structure of the system, which includes the number of files under version control, and the directory each file resides in. Any software system has a hierarchical structure, much like a traditional file system.
- The time and date of creation of each file, the directory it belongs to, and, in the case of multiple developers working on the same system, the creator of the file.
- The time and date a modification to a file was committed, the directory it resides in, and the person responsible for the modification.
- A log where developers can record the changes they made to a particular version of a particular file; in other words, the reasons for the modification.
- A *software release*, stated simply, is a group of files that 'denote a significant point in the software life-cycle' [43]. A system can have a number of releases as it evolves. The version control system should be able to record the versions of files that comprise a particular release.
- When multiple developers work on the same files, they might make changes simultaneously. The version control system must allow for proper *branching* and *merging*, so that the changes are not lost.

Although a version control system provides a lot of information to the end-user, there is still some more information that can be unearthed from its history and log files. There are a number of *software metrics* that could be used to query the VCS and provide information on the evolution of the system. These queries on the VCS can provide more insight on the software system. Some useful information that can be derived has been compiled by researchers over the years and presented in [28], [35], [36], [40] and [44]. It includes:

- Changes in the structure of the system as files are added or modified.
- The growth rate in the size of the system.
- The percentage of change in the system, and the module or file where this change is concentrated. Percentage of change can be derived at any level of abstraction of the system, in order to clearly identify the regions that change most frequently.
- Areas of high maintenance in the system, based on the growth rate and changes in code size.
- Areas that haven't been touched in a long time, based on the time of the last modification made to them. These, along with the high maintenance areas may require code restructuring.
- A number of files may be modified together as part of a change to the system. These files are said to *cluster* together [38] and they may denote dependencies in the software system.
- A developer's contribution to the system, based on the number of files he/she modified, the number of lines of code he/she modified, and the time spent for a modification by him/her.

These metrics can be studied at various intervals as the software system develops. Also, all this information can be presented as text, but a graphical interface allowing user interaction provides a semantically richer experience. A user should be able to state the metrics he/she needs information about, and then should be provided with visualizations

satisfying the request. This visualization can be two or three dimensional, and the focus of this thesis is the use of three dimensions for effective and automatic visualization of a version control system, specifically, CVS. Figure 1.1 shows a simplified architecture for such a visualization tool.

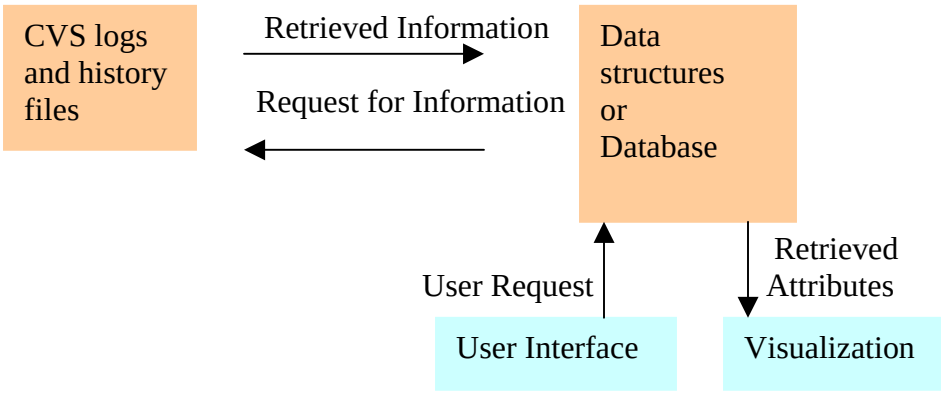


Figure 1.1: Simple Architecture for a Visualization Tool

The user states the attributes he/she needs to visualize. If the requested information is not already present in the databases or the data structures used by the visualization tool, it is retrieved from CVS. These attributes are then used for the visualization. The shape the attributes take in the visualization is determined by the *visual metaphor*. The user can select the metaphor he/she wants or the metaphor can be hard-coded into the visualization tool – it is decided by the user interface design.

1.3 THESIS CONTRIBUTIONS

This thesis outlines a visualization tool that can interface to CVS and demonstrate the usefulness of three-dimensional visualizations, by visualizing multiple attributes of a version control system in a single view. At a glance, the visualization states the following information:

- The size of the system under version control, and its structure.
- The number of versions of each file, and hence the number of changes made to each file in the system.
- Point and click interaction to retrieve information about the version number assigned to each version, the date of creation of a version, and the user responsible for the version.
- Point and click interaction to view the source code of any version of any file.
- The use of color to indicate the files that have been changed most frequently. When the number of versions for a file crosses one thousand, it will be seen that the visualization is highly dense and may indicate a need for restructuring the system.

This work differs from previous work by being three-dimensional and completely automatic. It can work for any module on any CVS repository on a Windows machine. The visualization tool needs only the path to the *repository* where the root directory for the software system is housed, and the name of the *root module* that needs to be visualized. It is entirely automatic, three-dimensional, and built using C++ and the GLUT API for OpenGL. References for GLUT and OpenGL can be found in [45].

1.4 REPORT STRUCTURE

Providing a graphical interface to shapeless software forms the concept of *software visualization.* Visualizing the version history of a software system is but a kind of software visualization. This document is organized as follows: Section 2 provides a review of literature in the field of software visualization, and then a review of literature in the related field of version history visualization. Section 2 also states the advantages of three-dimensional visualization over two-dimensional visualization and concludes with a review of the concept of a visual metaphor for software visualization. Section 3 discusses version control systems and the functionality they offer in more detail, with a focus on CVS. Some two-dimensional interfaces to CVS are also discussed in Section 3. Section 4 details the design implementation of the visualization tool and the views it provides into the version control system. Section 5 illustrates the working of the tool with examples and case studies. Finally, Section 6 concludes with suggestions for future work.

CHAPTER 2: LITERATURE REVIEW

2.1 INTRODUCTION

The review of existing literature is done in two parts. The first part, which is Section 2.2, deals with work done in the field of software visualization, while the second part, Section 2.3, narrows down specifically on work done in the field of visualization and extraction of information pertaining to the evolution/version history of a system. The literature review concludes with an overview of the concept of visual metaphors for visualization in Section 2.4.

2.2 SOFTWARE VISUALIZATION

Software Visualization, stated in the simplest of terms, is representing software using graphics. Software Visualization has its roots in the oft-repeated adage: 'A picture is worth a thousand words', and has been around since the concept of flow-charting evolved. Software Visualization has been defined in a number of ways. The most frequently quoted definition is that by Price et al. [1] which states: 'Software Visualization is the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software'. A more general definition found in [2] and [3] states: 'Software Visualization is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration.'

The key point expressed in both definitions, which can also be found in other work related to software visualization is the fact that software visualization attempts to give physical shape to shapeless or intangible software that 'disappears into disks' for better comprehension and understanding [4]. This is the same concept expressed by Storey et al in [5]. They state the need for *software exploration* tools that link graphical representations of software to corresponding textual representations of source code and aid program comprehension. Their paper also states cognitive design elements for software exploration, improving program comprehension and reducing the maintainer's cognitive overload. Here the maintainer is the person who is attempting to comprehend the software system.

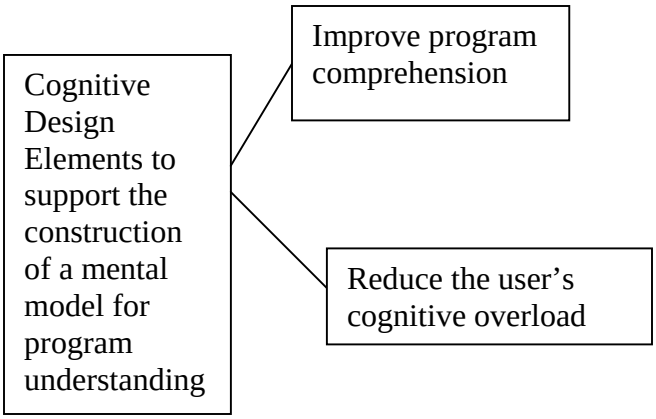


Figure 2.1: Cognitive Elements for Software Exploration

Software Visualization raises the question 'What can be visualized?' Research over the years has visualized different aspects of source code – the code itself, data flow and run-time behavior. Software Visualization has been applied for various disciplines like algorithm animation, software engineering, concurrent program execution and static and dynamic visualizations of object oriented code, to name a few. An extensive compilation of research work relating to these fields can be found in [6, 7].

One major view about software visualization is that the programmer or designer has a mental model while creating the system and visualization is effective only if it recreates this mental model as closely as possible in the minds of the users trying to comprehend the system [5, 8]. Also, a number of taxonomies have been stated that characterize the properties of an effective software visualization system [1, 9]. They discuss properties like *scope* and *content* (what is the aspect of the program being visualized?), *abstraction* (what kind of information is conveyed by the visualization?), *form* and *technique* (how is the graphical information being conveyed?), *method* (how is the visualization specified?) and *interaction* (how can the user interact with the visualization?). Stasko and Patterson [11] identify an additional characteristic, the level of *automation* provided for developing the software visualization system.

It is apparent that there are many faces to software visualization. The following sections discuss related work in 2D and 3D software visualization. Then follows a more specific discussion of work relating to the visualization of software changes and software version history. The literature review concludes with an overview of visual metaphors, design criteria for effective metaphors and a look at some of the metaphors and attributes that have been suggested for comprehending changes in software.

2.2.1 Visualizing in 2D, 3D and Virtual Environments

Software Visualization has progressed from using simple two-dimensional graphs to three-dimensional representations and more recently, virtual reality. The following sections discuss systems in each of these categories and justify why virtual environments are better for program comprehension.

2.2.1.1 Two-dimensional Software Visualization

Two-dimensional visualization of software typically involves a lot of nodes and arcs. A complex software system might include thousands of such nodes and arcs. To make conceptualization and comprehension easy for the user, visualizations of such systems present pieces of the graph in different views or different windows so that the user can focus on the level of detail he desires. Previous work that has focused on two-dimensional visualizations of software systems is therefore represented in multiple windows that present to the observer different characteristics of the system under consideration. Some examples of such visualization systems are SeeSoft, Shrimp, GROOVE and FIELD. A brief summary of these systems is presented below.

SeeSoft

As stated in the paper, SeeSoft – A Tool for Visualizing Line Oriented Software Statistics [12], the system can analyze 50,000 lines of code simultaneously. SeeSoft analyzes 'C' source code and two-dimensionally displays (in color) statistics associated with each line of source code like age, programmer, feature, type of line, number of times the line was executed and so on. Files in SeeSoft are displayed as columns and lines of code as thin rows. The color of each row is determined by a statistic associated with the line of code that it represents. One drawback of SeeSoft is that it cannot scale more than 50000 lines of code.

Shrimp

Shrimp [13] was developed as a visualization technique to explore software structure and browse source code. Shrimp moved away from the statistics-based approaches to visualization proposed in systems like SeeSoft (Section 2.1.1) and SeeSys [14] and used large nested graphs to provide better visibility into software structure. More specifically, Shrimp advocated the use of pan+zoom and context+detail or fisheye views of the graphs of the software system. These views enable a user to zoom out for better overview of structure and zoom in or enlarge nodes in the graph for better detail. Shrimp tried to avoid having multiple cascading windows open while understanding software structure because the it was felt that the user might have difficulty 'accurately conceptualizing and integrating the implicit relationships among the contents of the individual windows'.

GROOVE

In their technical report, Stasko et al. [15] emphasize the need for visualization of run-time or execution time behavior of object-oriented code. Their report states that static representation of object-oriented code does nothing more than represent class hierarchies, class visibility and interfaces and that it does not specify clearly how instances of classes collaborate at run-time to implement tasks not specific to a single class. They specify a system called GROOVE (Graphical Object-Oriented Visualization Environment) that can visualize both static code and run-time dynamics.

GROOVE uses geometrical shapes to represent static properties of code like inverted triangles and arrows between these shapes to represent inheritance and methods of a class. The triangles representing classes can be either compressed or expanded to show

necessary level of detail. The table below shows the 'metaphors' or visual mappings from source code to representation used in GROOVE for components of C++ code.

Table 2.1: Graphical representations for C++ program entities in GROOVE [15]

Current focus	***Related Entity***	***Entity's depiction (visual attribute)***
Class	Itself	Dark color, bold outline
	Base class(es)	Light color, inheritance connection arrows
	Derived class(es)	Light color, inheritance connection arrows
	Instance of the class	Light color, bold outline
	Instance of a derived class	Light color
	Friend class or function	Hand icon
Instance	Itself	Light color, bold outline
	Its class	Light color
	Classes it inherits from visible instances	Double, broken outline
Function	Itself	Bold outline

The table represents static entities in C++ code. For visualization of program dynamics, the GROOVE system uses animations that indicate the creation of an instance of a class or message/data flow between instances.

FIELD

FIELD (Friendly Integrated Environment for Learning and Development) [6] was developed as a framework for program visualization and programming support. FIELD had multiple views for displaying a textual and visual front end, a data structure displayer, displays of current debugger state (eg. stack display) and display of information from a cross-reference database. FIELD could be used to display call graphs, class hierarchy displays (for C++), make and build dependencies in a software system and data structures, to name a few.

2.2.1.2 Three Dimensions and Virtual Environments for Software Visualization

Evidently, two-dimensional visualizations attempt to clutter a plethora of information on a flat plane. Even though pan+zoom and fisheye views have been explored (as in

Shrimp), visualizing software in two-dimensions does introduce a cognitive overload by presenting too much information. Stasko et al. [17] identify the need for an extra spatial dimension in visualizations when they state that 'by adding an extra spatial dimension, we supply visualization designers with one more possibility for describing some aspect of a program or system'. As an example of the advantages of using three-dimensional visualizations, the Cone Tree concept developed at Xerox Parc [17, 18] can be considered. It has been claimed that the cone tree can display up to one thousand nodes without visual clutter, which is far beyond the capabilities of a two-dimensional visualization. The 3D visualization developed at Xerox Parc presents structured information such as computer directories and project plans. The system's designers noted that the 3D displays help shift the viewing process from being a 'cognition task to a perception task'. In line with representing the execution time behavior of object-oriented code [15] in two dimensions, Stasko et al. [17] discuss the development of a system called POLKA-3D to represent the same as a three-dimensional animation.

Ware et al. [18] developed a system called GraphVisualzer3D to visualize object-oriented code in three dimensions. They suggest that perception is less error-prone if software objects are mapped to visual objects, as there is a natural mapping from the former to the latter. Their paper presents the results of experiments that analyzed perception in two dimensions and three dimensions and concludes that there is encouraging empirical evidence that error rates in perception are less in three-dimensional visualization. One major advantage of three-dimensional visualization is that it allows a user to perceive the depth of a presented structure. While representing the design of a system, it is no longer necessary to stick to planar, flat depiction. With 3D, users can zoom-in or walk around structures or choose another angle (by rotating the design). It is possible that hidden structures in a software system become evident when 3D is used for visualization. The hierarchy of relations and dependencies in design or source code would also become more readily apparent with 3D, because of the added depth. Also, the world we inhabit is three-dimensional, so the added sense of familiarity while browsing three-dimensional structures might help develop the 'mental model' (Section 2.2) faster in the mind of the user.

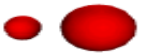
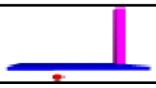
Another example of visualizing large nested graphs in 3D is the NV3D system [20] that has been tested with graphs containing more than 35,000 nodes and 100,000 relationships. The NV3D system uses techniques like rapid zooming, elision and 3D interactive visualization to display nested graphs in 3D. The NV3D system and the POLKA-3D system [17] analyze issues like spatial navigation, layout, semiotics and the common uses of the third dimension to represent characteristics like value, structure position, history of computation, state of computation and aesthetics to refine the appearance of a three-dimensional visualization.

3D visualizations have been explored for all areas where 2D visualizations were used, including metrics based visualization of object oriented programs and visualization to track software errors, isolate problems and monitor development progress [21, 22, 23]. Three-dimensional UMLs have also been researched. The next step for richer, more natural visualizations is virtual environments. Virtual environments open possibilities of

'immersion' and 'navigation' that might help to explore software structure better. Once again, the concept of 'worlds' in a virtual environment can be mapped to 'entities' or 'components' in object-oriented code or a software system. It is possible that all software artifacts from requirements to source code can be represented in a virtual environment to improve comprehension. It might be possible to link specifications in the requirements and design documents of a software system to the source code. Virtual environments would enable users to navigate through these links faster and in a more intuitive manner than 2D representations or even 3D structures. Research into the usage of virtual environments and virtual reality is at the inception stage. Again, as in 3D, virtual environments enable the user to interact with a representation of something familiar, namely a world with familiar objects that he/she can interact with. It might be worth investigating the possibilities of better program comprehension with virtual environments, especially if the end-user can be provided with a high-quality representation.

Examples of software visualization systems that use virtual environments for representing object-oriented software systems are ImsoVision [27] and Software World [26]. The former represents C++ code in an immersive virtual reality environment while the latter does the same for static Java code. A major characteristic of both systems is the mapping of static properties of object-oriented code to objects in the virtual environment. ImsoVision uses geometrical 3D shapes like platforms, spheres, horizontal and vertical columns as visual metaphors for the characteristics of C++ code, while Software World uses real-world metaphors like a world, countries, districts and buildings as visual metaphors for the various parts of Java code. Tables 2.2 and 2.3 show the visual metaphors used for the mappings in both these systems.

Table 2.2: Mapping of entities to visual metaphors in ImsoVision [27]

Name	Visualization	Meaning
Platform		Class
Platform Size		Number of methods plus the number of attributes
Sphere		Attribute
Sphere Size		Type of Attribute
White Column		Constructor Member Function
Green Column		Accessor Member Function
Purple Column		Modifier Member Function
Column Size		Logical Lines of Code per Method
Sphere/ Column Location		Information Hiding

© 2001 IEEE

Table 2.3: Depicting Relationships between entities in ImsoVision [27]

Name	Visualization	Meaning
Adjacency with Shading		Inheritance
Yellow Stacks		Overloaded Element
Aqua Flat Link		Dependency Relationship
White Flat Link		Aggregation Relationship

© 2001 IEEE

It should be observed that both visualization systems visualize only static properties of code. They cannot be used to characterize the run-time behavior of an object-oriented system, which has been emphasized repeatedly for better comprehension [15, 17]. Table 2.4 shows the visual metaphors for Java code in Software World.

Table 2.4: Mapping Java source code to real world metaphors [26]

Visualisation Level	Code Element
World	The software system as a whole.
Country	Directory structure, which maps to the packages in Java.
City	A file from the software system.
District	Class (contained within the specific file and hence city in the visualisation).
Building	Methods.

To summarize, object oriented systems are large complex systems composed of multiple components. To effectively comprehend these systems, it is necessary to provide varying levels of detail. Any user attempting to understand the system must be able to zoom-out and zoom-in to each level of detail as necessary. Three-dimensional visualizations and more recently, virtual environments allow a user to concentrate on one aspect of the world in detail while providing a distant view of other aspects that are situated farther away. As the user moves close to each entity or visual component, it comes to 'life' or presents a higher level of detail. This is similar to the real world we inhabit, where objects and their structure come into focus only when we are within certain viewing distance. This technique, called elision is a major property of virtual environments, which abstracts distant objects and details closer objects. The user can move back and forth between objects or structures in this world, and rotate them around to view information that might be hidden from normal view. An example of this can be seen in the Imsovision system [27], which hides the private attributes of an object under the platform that represents the object. The private attributes are visible only when the user rotates the platform around. While it is evident that virtual environments and using the third dimension provide a far richer experience than 2D visualizations for a user attempting to comprehend a software system, it is necessary to further investigate metaphors and representations that allow us to move beyond visualizing static code.

The major focus of this thesis is the automatic visualization in three dimensions of the software history recorded by a version control system such as CVS. A detailed discussion of work related to visualizing evolution and version history of software systems in both two and three dimensions follows.

2.3 VISUALIZING THE VERSION HISTORY OF SOFTWARE

Visualizing the version history of a software system can be seen as a special case of software visualization. We are still attempting to give shape to intangible software and deduce information from the visualization to enhance our comprehension of the software system. Visualizing the version history of a software system has also been labeled as the

visualization of the evolution of the system. As was stated earlier in Section 1, visualizing version histories typically involves visualizing metrics such as number of lines of code in a particular version of a file included in the system, the percentage of growth and the percentage of change in the system, defect density and change complexity measures [35, 36]. This section discusses some important work in the field of version history visualization.

The work discussed focuses on the software of large switching systems for the most part, written in C/C++ and consisting of approximately thirteen million – twenty million lines of code [35, 36, 38, 39]. An important concept in most of the papers discussed below is that of a ***Modification Request*** or ***Maintenance Request*** (MR). A software system is assumed to consist of *subsystems*. Each of these subsystems has a number of *modules*. The modules include the *program elements*, which may be a collection of one or more source files, and an MR is the information representing work to be done to each module. ***Deltas*** are part of a MR, representing editing changes made to individual files in order to complete an MR. A file can be checked out, edited and then checked in [38, 39]. This terminology works well with version control systems like SCCS and ECMS, which can record the parent MR for each delta list, along with the number of lines added, deleted and modified by that change. In the case of CVS, used in the current work, there is no concept of a MR. Changes made to files are recorded as part of a checkout or update of modules.

The forerunner to most of the attempts at version history visualization can be seen in ***SeeSoft***, developed by Eick et al. [37]. SeeSoft is a tool for visualization of line oriented software statistics. SeeSoft can visualize up to 50,000 lines of code and provides information about various statistics like the number of files under version control, the age of each line of code in a file, the number of lines of code in each file, the MR that touched a particular line of code in a file and the number of times the line was executed during testing.

SeeSoft uses a row-column metaphor. Each column represents a file and the rows in each column represent the number of lines of code in the file. SeeSoft allows user interaction to decipher interesting patterns in the version history. It provides information about the dates of changes, the reasons for changes, the developer who changed the code, etc. For example, if the cursor is positioned over one MR on the screen, all the lines of code touched by that MR are visible. Similarly, activating a line of code activates the MR associated with that line and hence all the lines of code affected by that MR. Also, activating a file activates all the lines of code in the file and hence all the MRs used to create them. Screenshots of SeeSoft can be seen in Figure 2.2 and Figure 2.3. In Figure 2.2, SeeSoft shows code age. The newest code is in red and the oldest is in blue. Figure 2.3 shows execution profile results. In the color spectrum on the left, frequency of execution can be seen, with red showing the hot spots. Non-executed lines are gray and non-executable lines (like comments) are black.

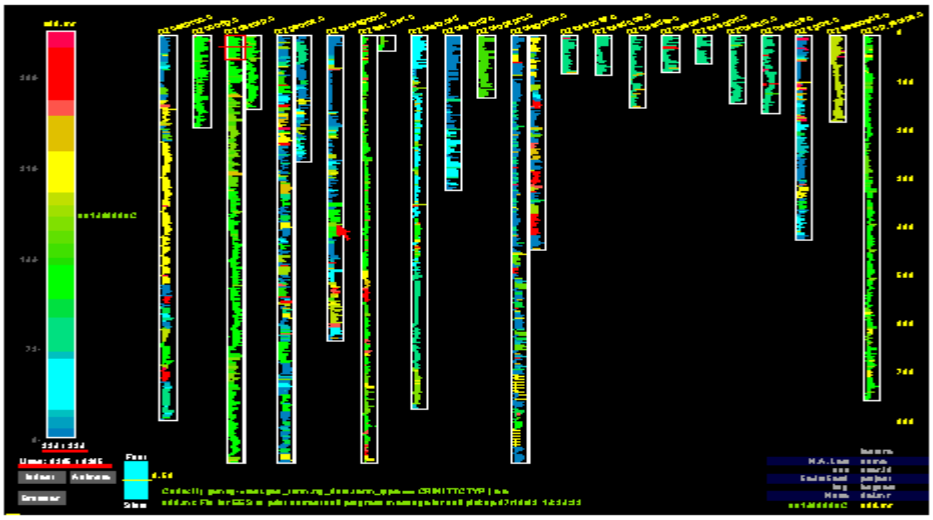


Figure 2.2: SeeSoft showing the age of code [37]

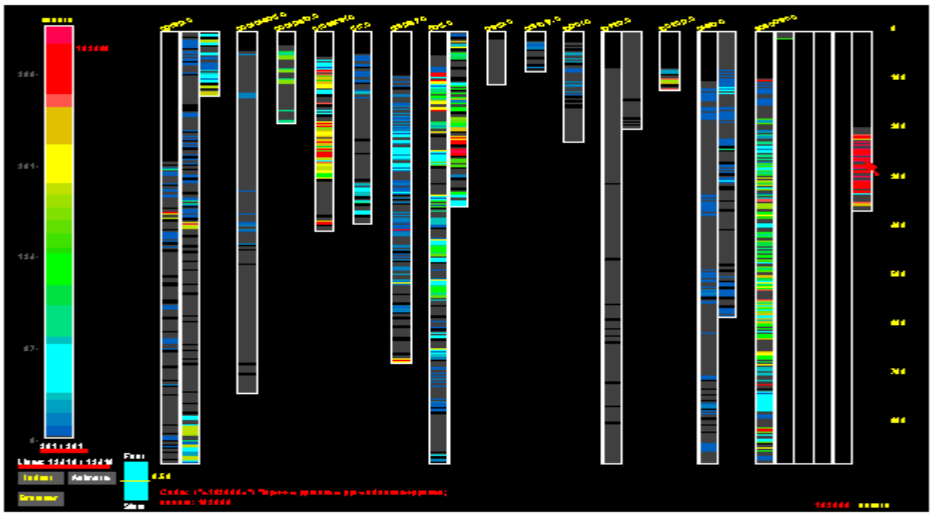


Figure 2.3: SeeSoft – execution profile [37]

Figure 2.4 shows more screenshots of SeeSoft, with interaction information. Figure 2.4 represents a SeeSoft view of a directory with 20 files. The color spectrum in this case represents the MRs applied to the system, with the oldest in blue and the newest in red. It then follows that the MR that created the line determines the color of each line of a file. As the cursor is positioned over a MR, the lines of code associated with that MR are also activated. It is also possible to identify the number of changes that apply to a file out of the total number of changes in a directory, and when these changes were made. SeeSoft also provides a code window to view source code of a file selected by the user. SeeSoft also provides information about the developers touching a file, the clustering of files into

groups based on the MRs, the files that have changed continuously and the types of MRs responsible for the changes to the files.

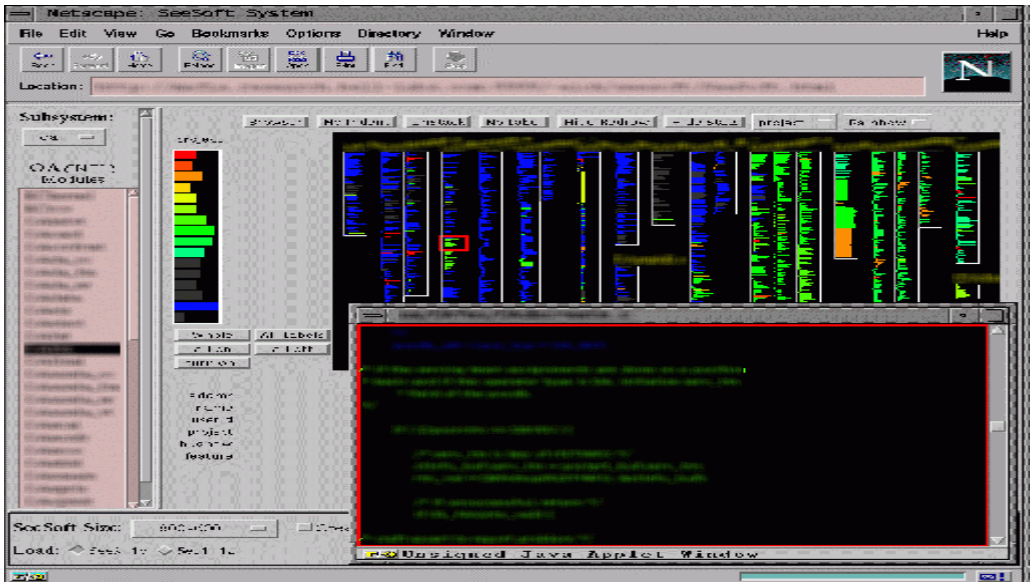


Figure 2.4: SeeSoft with user interaction [37]

The visualization also provides animation to visualize progressively the changes made to a system.

According to the creators of SeeSoft, the major goals of the visualization are:

- Code discovery.
- New developer training - Any new programmer has to comprehend the changes made to files in the system, why these changes were made, when these changes were made, what parts of the system were affected by the change, what parts of the system change frequently, and who made the change.
- Project management – to keep track of the age of code, the time taken to complete a change, and developer effort.
- Quality assurance and testing.
- Software/Code analysis.
- Code execution optimization.

Closely related to this thesis is the work presented by Gall et al. in [35]. Their paper uses color and the third dimension effectively to visualize software release histories. They summarize their contributions as:

- The use of information visualization to study the release of a software system.
- The combination of three dimensional views and color for the study of abstract software structures.

- Presentation of a particular industrial system's visualization to show the kinds of analyses and observations that are possible from release histories.

As mentioned above, Gall et al. visualize the evolution of an industrial software system for twenty versions over a two – year period. In their words, the metrics that they visualize include size of the system in terms of lines of code, age in terms of version numbers and error-proneness in terms of defect density. Their Software Release History visualization is composed of three entities:

- ***Time:*** The visualization is grouped according to the Release Sequence Number (RSN). A snapshot of the system at the time of each release enables the end-user to see the evolution of files between releases. Addition, deletion or modification of files between releases is also clearly visible.
- ***Structure:*** The system is decomposed into subsystems. Each subsystem is decomposed into modules and each module comprises the source code files.
- ***Attributes***: These include version number, size, change complexity and defect density.

If a program element or the lowest level of abstraction in the structure of the system changes only in release 2 and release 5 of the system, then its sequence number is represented as: <0 1 1 1 2>, assuming there are five releases of the system. Figure 2.5 shows the visualization by release sequence number. Color encodes the release number, as represented by the color scale in Figure 2.6. Figure 2.5 shows the files in three modules of the system. If a file is nonexistent in a release, a black cube represents it. The versions of the files also follow the same color scale as in Figure 2.6.

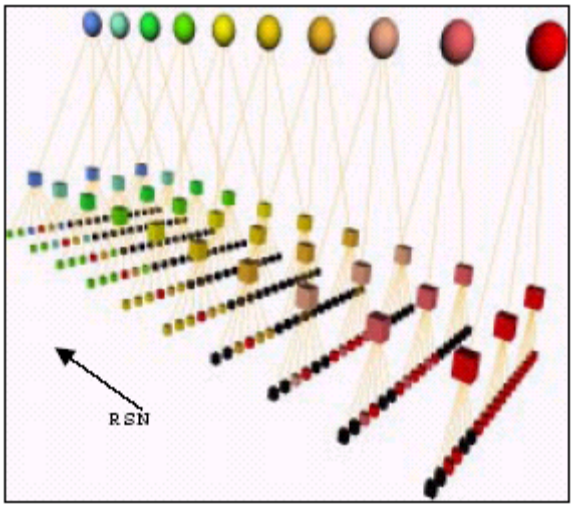


Figure 2.5: Visualizing the history by release [35]

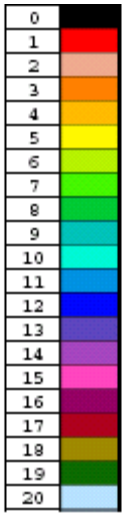


Figure 2.6: Color scale [35]

For example, release 3 has some files that were created in release 1, some that were created in release 2, some that were created in version 3 and some that are not yet created (the black cubes). It is acknowledged in this paper that any visualization, even if presented in 3D space can become incomprehensible while viewing large volumes of data. So percentage bars, as shown in Figure 2.7, are used for each module to represent the number of files that were created in that particular release of the system, the number of files that have not changed from the previous release and the number of files that are nonexistent (implying that they were deleted in a particular release, or will be created in future releases).

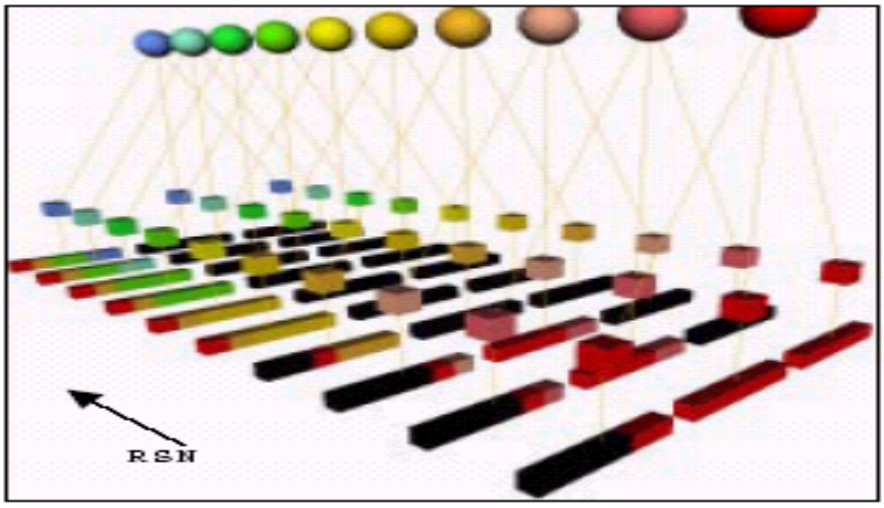


Figure 2.7: Visualizing the history using percentage bars [35]

The visualization was created in Java, using VRML to render and navigate the 3D spaces. The end user can navigate through the visualization and use the mouse to extract information about the structure of the entire system for a release, or focus on a particular subsystem and extract the values of the modules in the subsystem. The paper concludes with the suggestion that other metrics like lines of code, complexity measures and defect density can be visualized. It also suggested the detection automatic detection of change patterns to identify module dependencies. The type of change pattern to be investigated could be input by the user.

Gall et al. discuss another application of version history visualization in their paper '*Detection of Logical Coupling Based on Product Release History*'. They present an approach that 'uses information in the release history of a system to uncover logical dependencies and change patterns among modules.' They have developed a technique that automatically extracts information about the logical dependencies among the modules of a system. These logical dependencies are different from the syntactic dependencies that are evident through source code analysis. They propose the idea of *Change Sequence Analysis* and *Change Report Analysis* to identify logical dependencies. As a case study, they have considered a large software system consisting of subsystems, modules and programs. This software system consists of approximately 10 million lines of code. The dependency analysis is carried out over twenty releases of the system.

The Change Sequence Analysis lists the releases in which a module has been changed. Different modules can be compared on the basis of such change sequences and common change patterns can be identified. As an example, Table 2.5 lists the change sequence for program Pi. Since Pi was changed in releases 3 and 5, the change sequence is <3,5>**.**

Table 2.5: Program Pi's version numbers and change sequence

System Release	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Pi version no.			1.1	1.1	1.3	1.3	1.3	1.3	1.3
Pi change sequence			**3**		**5**				

To verify the dependencies deduced as part of the change sequence analysis, a change report analysis is proposed. The change reports describe the reasons, amount and type of change of a single program with regard to a particular version number. By looking at the change reports for programs with the same change sequence, the logical coupling process can be verified. The paper goes on to discuss coupling among subsystems based on the releases in which they were changed simultaneously. This logical coupling identifies the releases in which the maximum number of changes were made and which modules / programs were affected by these changes.

In '*If Your Version Control System Could Talk…*' by Ball et al. [39], understanding a program's development history from the information stored in a Version Control System is discussed. Their theory is based on the fact that a Version Control System should be able to track a group of related changes via a Modification Record (MR). Since the

changes as part of a MR are made for a specific purpose, they should be semantically related. The system analyzed here is a compiler written in C++, consisting of approximately 275 classes. The VCS used to record the changes is ECMS. The classes comprising he compiler are organized into five categories:

- The top-level classes used primarily as base classes. A diamond is used to represent these.
- Symbol table classes representing a language construct or type, represented by unfilled squares.
- Abstract syntax tree classes, represented by filled circles
- Classes that apply optimization transformations to the abstract syntax tree, represented by unfilled circles.
- Code generation classes represented by filled squares.

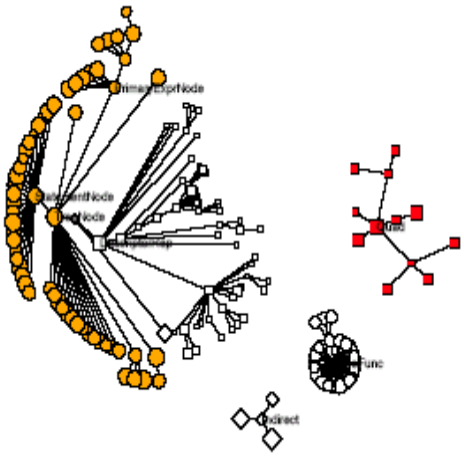


Figure 2.8: The classes of the compiler [39]

Figure 2.8 shows the classes of the compiler, with the area of the node representing a class representing the number of MRs that touched the class. The paper also presents a cluster analysis of classes based on the number of modification records that touched a class. It is suggested that if N_{mr} is the number of MRs that touch a class N and NP_{mr} is the number of MRs that touch both classes N and P, the ***link probability*** for N and P is:

$$NP_{mr} / \text{square\_root}(N_{mr}P_{mr})$$

If N and P are always changed together, the link probability will be 1. Using this method, the clustering of classes is shown in Figure 2.9. The clustering analysis has defined the five categories of classes identified for the compiler. The paper thus discusses the derivation of VCS-related metrics, like connection strength based on the probability that two classes are changed together.

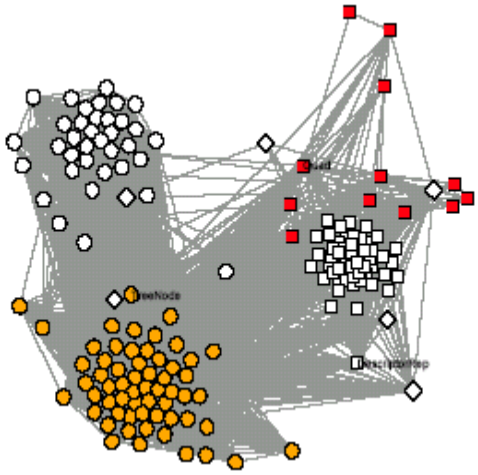


Figure 2.9: Cluster Analysis of classes based on MR [39]

Michele Lanza and Stéphane Ducasse, in '*Understanding software evolution using a combination of software visualization and software metrics*' [40], study the evolution of classes in a system using a combination of software visualization and software metrics. The visualization is two-dimensional, with rows representing the classes in a system and columns denoting the version of the system. So, the first column would represent version 1 of the system, the second version 2 and so on, as shown in Figure 2.10. The number of methods in the class decides the width of each rectangle representing a class, while the number of instance variables in the class decides the height of the rectangle. The author suggests that other metrics can also be used effectively to represent a class. This metaphor allows easy visualization of the number of class in the system, the most recent classes that have been added to the system and growth and stagnation phases in the evolution, as represented in Figure 2.11. An innovative technique here is the classification of classes based on the kind of changes made to them over the different versions of the system. These categories are:

- A *pulsar* class is one that grows and shrinks repeatedly during its lifetime. They can be seen as hotspots in the system.
- A *supernova* class suddenly explodes in size. They have to be closely watched as they might introduce new bugs because of the sudden growth.
- A *white dwarf* class is one that does not have any real functionality and may represent dead code.
- A *red giant* class is one that is large over several versions – it might implement too much functionality.
- A *stagnant* class does not change over many versions. It might represent dead code.
- A *dayfly* class has a very short lifetime – it might exist only in one version of the system.
- A *persistent* class exists from the conception of the system to the very end. It should be examined closely – it is possible that this class is not touched because nobody is sure of the functionality it implements.

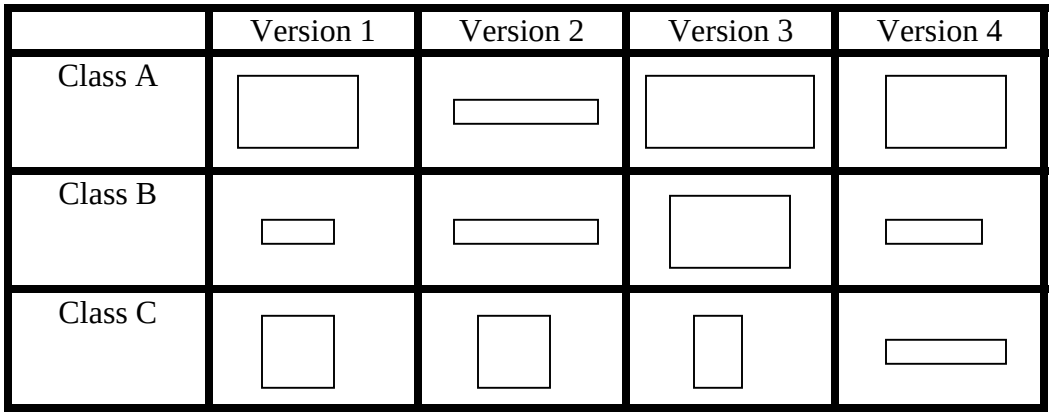


Figure 2.10: The evolution matrix, based on [40]

The paper also presents the evolution matrix of a system called MooseFinder as a case study. Growth and stagnation phases, versions of the system over time, the classes that belong to each version and pulsars and dayflies are identified.

The last paper in this literature review is by Hideki Koike et al., titled '*The Role of Another Spatial Dimension in Software Visualization*' [25]. Here, a 3-D visualization framework is presented, by means of which a user can interact with a version control system. Versions of the files in a system are represented as cubes arranged along the z-axis, ordered by time. Releases that link versions of various files together are represented as circles.

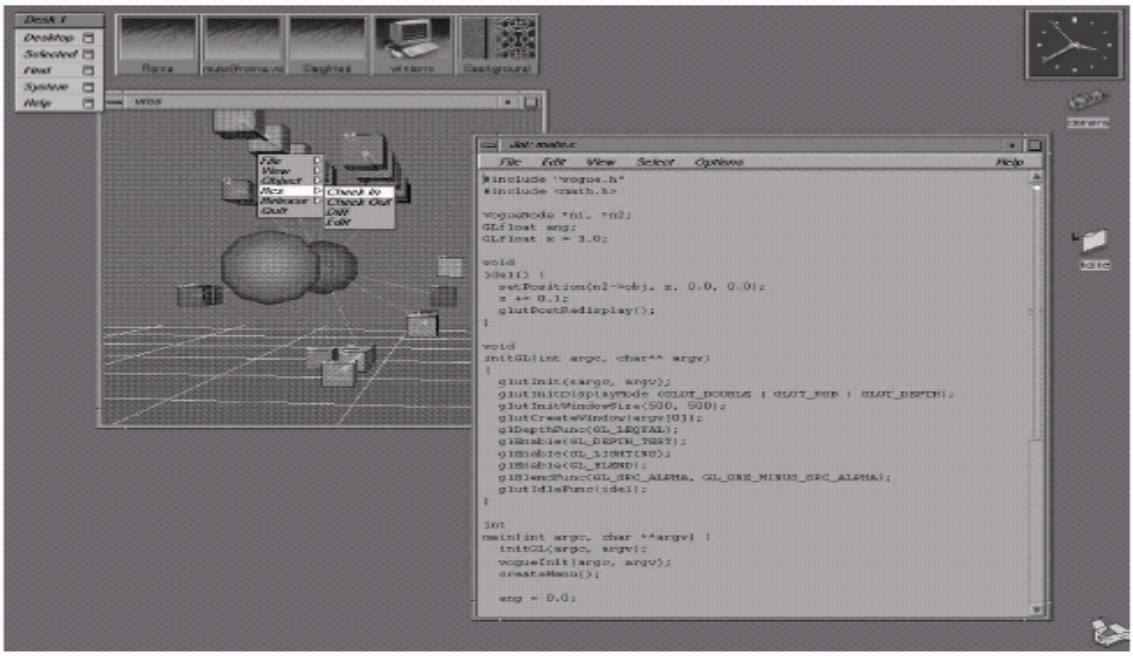


Figure 2.11: Visualizing version histories of multiple files [25]

VRCS has been implemented using OpenGL / C and serves as an interface to RCS. Users can 'check out', edit and 'check in' files, view differences between two cubes/versions of a file, retrieve all the files that compose a release and even build the executable file for a release. Figure 2.11 shows an example of VRCS. VRCS can only be applied to single-user systems. Also, the authors suggest some mechanism that enables the user to select the amount of graphical information presented.

2.4 VISUAL METAPHOR

Building on ideas stated in the previous sections, a metaphor can be defined as 'a rhetoric figure whose essence is understanding and experiencing one kind of thing in terms of another' [28, 29]. Averbukh defines a visualization metaphor as a mapping that provides correspondence between notions and objects of modeled application domain and a system of similarities and analogies [30]. This is clearly demonstrated in the systems that use virtual environments for software visualization. Direct mappings (Tables 2, 3, 4) of source code to objects in the virtual environment were witnessed. So, for the purposes of software visualization, a metaphor can be stated as the visual object in the virtual environment that provides a representation for an artifact in the software system being visualized.

Roman and Cox [9] represent the role of the visual metaphor in program visualization as shown in Figure 2.12.

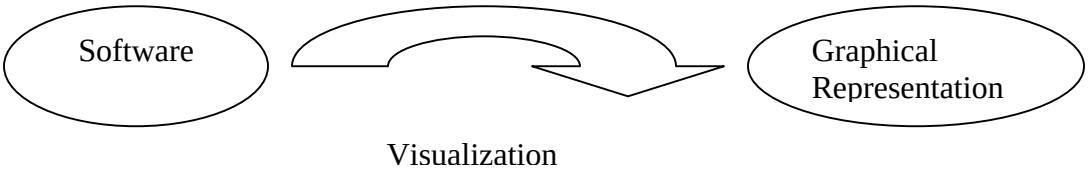


Figure 2.12: Mapping Software to a Graphical representation, based on [9]

Evidently, a metaphor is the entity that gives shape to the different faces of intangible software. Also, the power of the visual metaphor will affect the extent to which the visualization is effective. So, it becomes necessary to state the desirable properties of a metaphor for software visualization in two-dimensional, three-dimensional or virtual environments. While considering the properties of a metaphor, issues that arise regarding the characteristics of the software visualization system can also be discussed.

2.4.1 Design criteria for effective visualizations

In his paper 'Automating the Design of Graphical Presentations of Relational Information' [33], Mackinlay discusses graphical design issues on the basis of two criteria: *Expressiveness* and *Effectiveness*. Expressiveness is the medium used to express the graphical representation and effectiveness is the extent to which the representation is effective for comprehension of the visualized information. These two criteria form the

basis on which we propose our design issues for effective visualizations. To be effective and meaningful, any visualization system should consider the following key areas.

2.4.1.1 Scope of the representation

Visualizing complex, real-time systems can create chaos if the scope of the visualization is not defined. Scope, as identified by Price et al. [1], is isolating the characteristics of the system that the visualization will address. The visualization might choose to address static or dynamic features of the software. It might choose to represent control flow or data flow or dependencies or all three. For example, visualization of Java source code might address the classes in other packages that a particular class depends on or inheritance hierarchies for a class or interface dependencies for a class, to state a few.

2.4.1.2 Medium of representation

As was discussed in the previous sections, visualization can be two-dimensional, three-dimensional or virtual. The type of information being visualized and the level of detail required in the visualization are just two factors that dictate the type of output medium needed. If the system to be visualized is relatively small and if detail like complexity of the source code, version history, detailed dependency navigation or linking of the graphical representation to source code is not needed, then the most simple two-dimensional graph would be sufficient. If however, the system to be visualized should state in detail information like security vulnerabilities in the code and design or if the representation should present varying levels of information about the system without overwhelming the user, then three-dimensional visualizations might be considered.

2.4.1.3 The Visual Metaphor

Metaphors in the medium of representation affect the expressiveness of the visualization. Metaphors might be abstract geometrical shapes (as in ImsoVision, NV3D, GraphVisualizer3D and other two-dimensional representations) or they might be real-world entities (as in Software World). While it is true that a user would be more familiar with a real-world metaphor, the visual complexity of the metaphor should not affect the effectiveness of the visualization. However, if the medium of representation uses collaborative virtual environments [35], users might feel more comfortable interacting with their colleagues in a real-world immersive virtual environment rather than three-dimensional space.

Consistency of the Metaphor:
The metaphor or the mapping from software artifacts to the representations should be consistent throughout the visualization. If a triangle represents a class in an object-oriented system, it should be so throughout the visualization. Multiple software artifacts cannot be mapped to the same metaphor. Similarly, a software artifact cannot be mapped to multiple metaphors. Doing so will result in unnecessary complication and confusion. Also, in a virtual environment, the metaphor should be consistent with the world it is present in. For example, geometrical shapes that do not belong to the world when real-world metaphors are used will lead to inconsistencies.

Semantic Richness of the Metaphor and complexity:
The metaphor chosen should be rich enough to provide mappings for all aspects of the software that need to be visualized. The *scope* of the visualization (key area 1) determines to a certain extent the nature of the metaphor to be chosen. There should be enough objects or equivalent representations in the metaphor for the software entities that need to be visualized. Once again, the concept of low visual complexity but high semantic richness has to be stressed. The visualization should not be so mindless that the user is diverted from the information that the visualization system attempts to convey. If virtual environments are used for visualization, this becomes a critical issue. The virtual environment should provide pertinent representations without giving the user the impression that he is immersed in endless space. Similar views can be found in [26].

2.4.1.4 Abstractedness

The user of the visualization system should be able to focus away from certain parts of the representation and focus in detail on certain parts of the representation. This is the property of elision (used in NV3D [20]) that permits different users to focus on the level of detail they desire. For example, if a visualization system should aid an evaluator in discovering security vulnerabilities, the evaluator would look for different levels of detail (say, low-level representations that map to source code) as opposed to a user who will be interested only in visualizing if any security problems exist in the system. This ability to zoom-in and zoom-out is what makes navigation through a three-dimensional system easier than understanding a two-dimensional representation. Roman and Cox [10] identify different levels of abstractedness that a visualization system might have namely direct representation, structural representation, synthesized representation and analytical representation.

2.4.1.5 Ease of Navigation and Interaction

Ease of navigation is obviously a major design issue when constructing a visualization. The user should understand what he is looking at and what level of abstraction in the system he is currently at. It should be easy for the user to move back and forth between different views or different worlds (in the case of virtual environments). Also, the nature of the medium of representation would affect the level of navigation a user expects to have in the visualization. 3D visualizations should allow users to rotate the entities around for different angles of view. Users would also expect to interact with the visualization at some level. It should be possible to hide or ‘close’ objects that are not of interest by clicking on them or interacting with them in other ways.

2.4.1.6 Level of Automation

Automation specifies the level that the construction of the software visualization system is automatic. Both the virtual reality visualization systems discussed in the previous section are at present only partly automated. Effective visualizations would need to be fully automated for software visualization to be more widely used.

2.4.2 Examples of visual metaphors

Eick et al. [28] present a good collection of the various metaphors that can be used, as well as a discussion of some of the metrics that can be visualized. The authors state that

the principal metaphors are matrix views, cityscapes, bar and pie charts and that 'multiple views are combined to form perspectives that both enable discovery of high-level structure in software change data and allow effective access to details of those data'. The fundamental components of software change are stated as:

- *Time:* When a change was made.
- *Software Space:* Which files were changed; which lines were added and deleted.
- *Developer:* Who made the change
- *Size:* How many modules, files and lines were affected by the change
- *Effort:* The number of developer hours required to perform the change, and how long the change took, in calendar time

The metaphors discussed in this paper and the attributes they represent are presented below. The components of the software change data have been grouped into indices and responses.

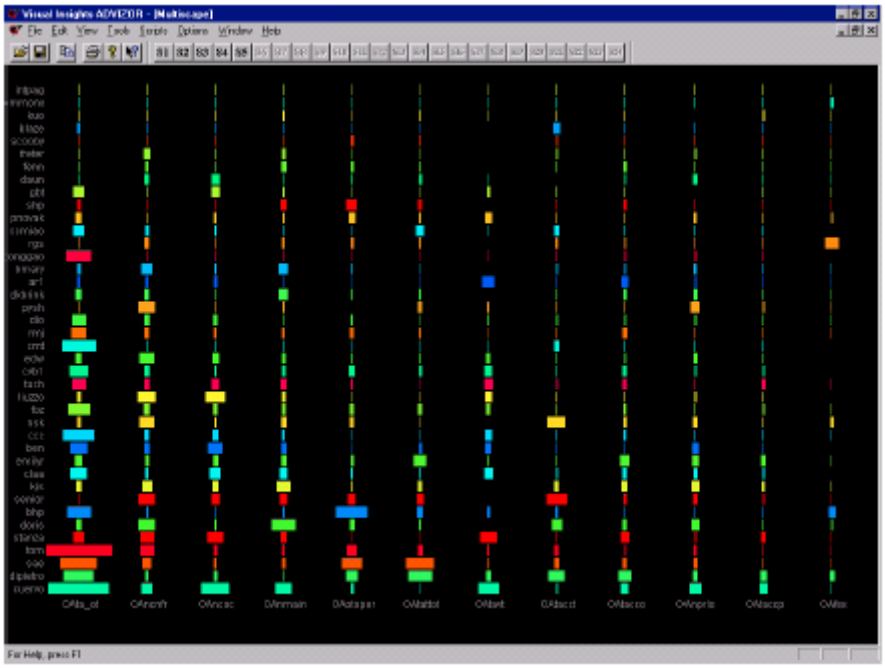


Figure 2.13: Example of a Matrix view [28]

Indices are the independent variables in the views like time, software space and developer. Responses are dependent variables in the views like size and effort.

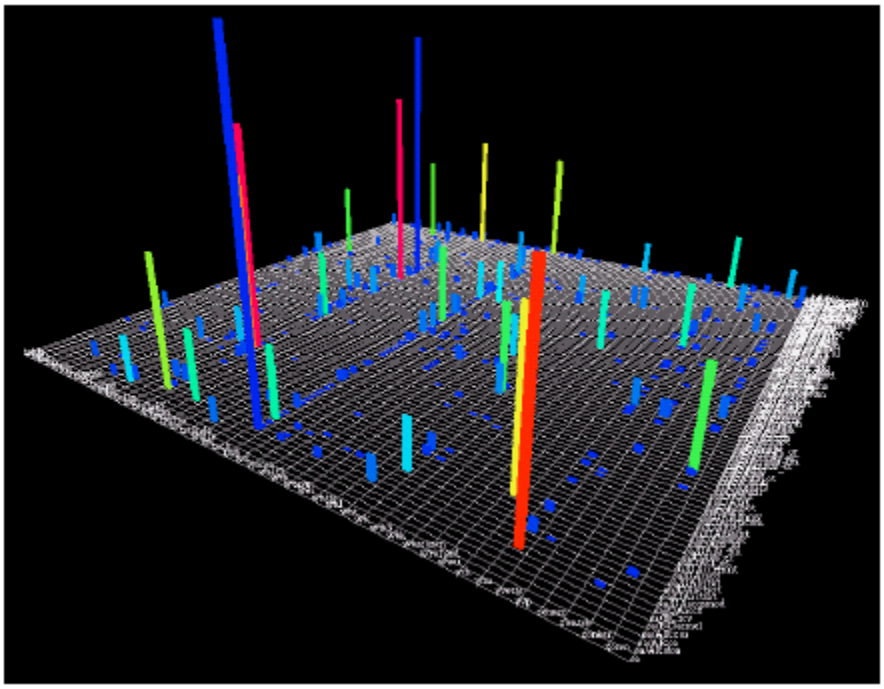


Figure 2.14: Example of a Cityscape view [28]

2.4.2.1 Matrix Views

Matrix views represent two-dimensional grids. Developers are represented as rows and software space as columns (indices). The response is the size of the changes, which is mapped onto the width of the bar in each cell.

2.4.2.2 Cityscapes

These are three-dimensional extensions of matrix views. The indices are again developer and software space, while bar color and height both redundantly encode number of changes to a module.

2.4.2.3 Bar and Pie Charts

These can be used effectively to illustrate the number of deltas per MR and the number of lines added and deleted, indexed by developer. A bar chart can be used to index the number of changes by year and a pie chart can be used to index the number of changes by file type. Bar and pie charts can also be used to interactively represent an interesting subset of the data.

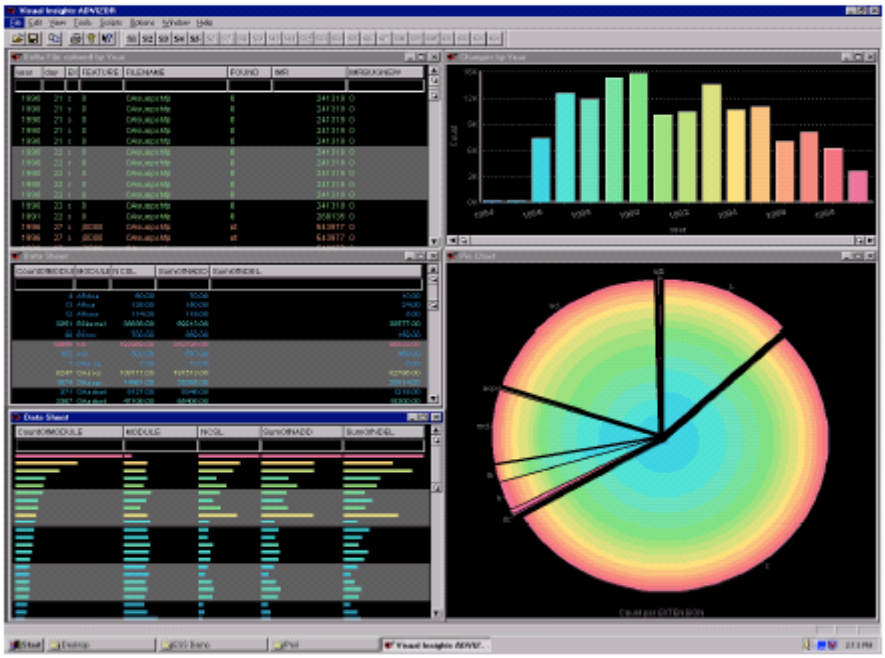


Figure 2.15: Perspective showing changes indexed by time [28]

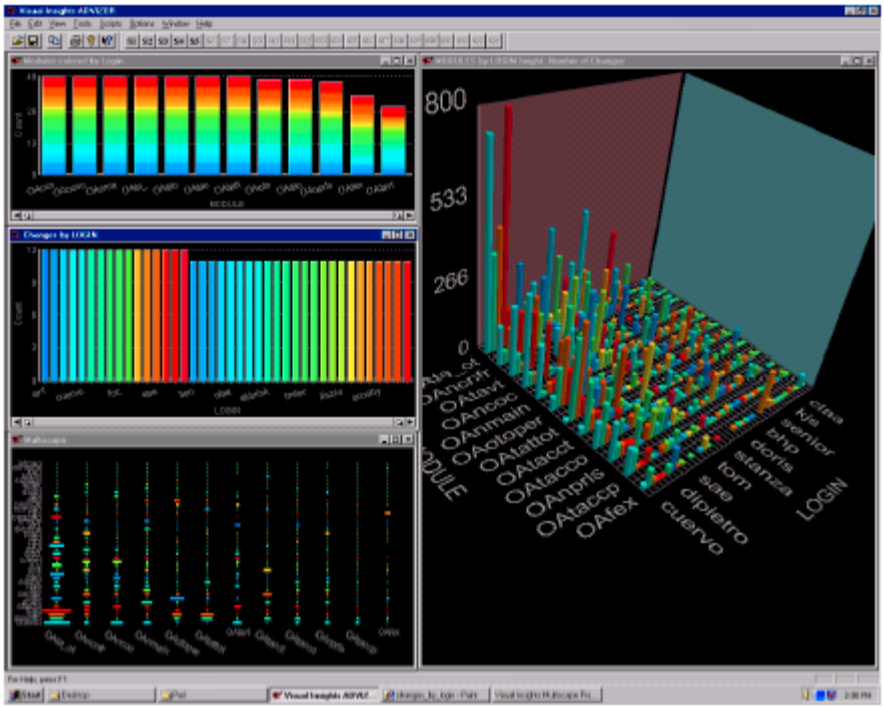


Figure 2.16: Number of changes indexed by developer and module [28]

Other useful views like number of changes indexed by developer, changes to the system sorted by size, changes to the system sorted by release number, clustering of files based on the MRs in which they change, perspective or multiple views showing the effects of a MR initiated by a particular developer, perspectives showing high severity MRs, etc. are discussed. Figures 2.13, 2.14, 2.15 and 2.16 show some of the metaphors presented in the paper. In Figure 2.16, color encodes year. The bar chart shows number of changes by year. Pie chart shows the number of changes by file type

2.5 A BRIEF SUMMARY OF SECTION 2

Section 2 provided a literature overview of software visualization, advantages of three-dimensional visualization and visualization of version history in Sections 2.2 and 2.3. It also presented related work in designing and using visual metaphors in Section 2.4. The next section will discuss version control systems and some two-dimensional interfaces to version control systems. It also take a detailed look at the features, functionality and commands of CVS, which is the version control system used to build the visualization tool presented in this thesis.

CHAPTER 3: VERSION CONTROL SYSTEMS

Section 3 provides an introduction to version control systems, their history, design and implementation. Section 3.1 compares three version control systems, SCCS, RCS and CVS, and states the differences as well as the similarities between them, based on their design and implementation. Since the major focus of this thesis is visualizing version control with CVS, Section 3.2 discusses the functionality of some two dimensional graphical tools for CVS. It also emphasizes the advantages of using three dimensions to visualize version control. Section 3.3 then details the features of CVS to explain why it was chosen to build the visualization tool.

3.1 INTRODUCTION

Walter Tichy in his paper 'RCS – A System for Version Control' defines version control as 'the task of keeping software systems consisting of many versions and configurations well organized' [42]. Version control primarily manages revisions to a document. That document can contain text, programs or documentation. So, any document under the control of a version control system should record the successive changes made to the document and when and where the change was made. In case of multiple users working on the same group of documents, it is also necessary to determine who made the change. The group of files under version control may form a software system. As the software system changes, the functionality of its various elements also changes. To understand why a change was made, it is first necessary to comprehend when and where the change was made. Version control systems record the history of source code. One of their many applications is to isolate when a bug or a feature was introduced into the system, and who was responsible for it. Whenever a file is modified, it results in a new version. Version Control Systems do not store all versions of all files – instead they store the differences between files, called *deltas*, so that any version can be recreated at any time. Some common version control systems are RCS [42], and SCCS [44]. A number of other version control systems by vendors like Microsoft are also commercially available, but the working principle behind all of them was derived from one of the above two.

SCCS stands for Source Code Control System and is the precursor to attempts at version control. It was released in 1975 by M.J. Rochkind of AT&T [44]. Prior to SCCS, every version of every file had to be stored entirely. SCCS introduced the concept of *merged deltas* that store information about the differences between various versions of a file. Using the information from these deltas, any version of any file can be recreated. RCS (Revision Control System) was developed by Walter Tichy at Purdue University around 1982 [42]. RCS corrected the mistakes of SCCS and used the concept of *forward* and *reverse deltas* to store the differences between versions. SCCS and RCS also introduced the concept of *locking* a file prior to using it, so that multiple users could access the same files, and *branching* of a version. RCS introduced the use of *symbols* for configuration management. Dick Grune, Brian Berliner and Jeff Polk developed CVS progressively from 1986 through 1989 [43]. CVS was built using the concepts of RCS. It is a more sophisticated form of RCS and provides additional features like painless addition, deletion and modification of hierarchical directory structures inside a main directory.

CVS also provides a number of administrative files that can be altered to customize version control for a user, and uses *tags* instead of RCS symbols. Tags and symbols have some minor differences and these are explained below. SCCS and RCS do have some differences when it comes to their implementation and these are discussed in the following paragraphs. Since CVS builds on RCS, only the additional enhancements that make it different have to be studied.

In both RCS and SCCS, whenever a file has to be edited, a copy of it is made in a directory for the user. This directory is called the 'working directory' and the file is 'checked out' into this working directory. Any modifications made to the checked out files have to be reported to the version control system. 'Checking in' the file from the working directory does this. Once the file is checked in, the version control system stores the differences between this version and the previous version as a delta. The algorithm used for delta computation and retrieval of a certain version of a file depends on the version control system. SCCS uses the concept of merged deltas [44] while RCS uses reverse and forward deltas [42]. To illustrate, consider Figure 3.1.

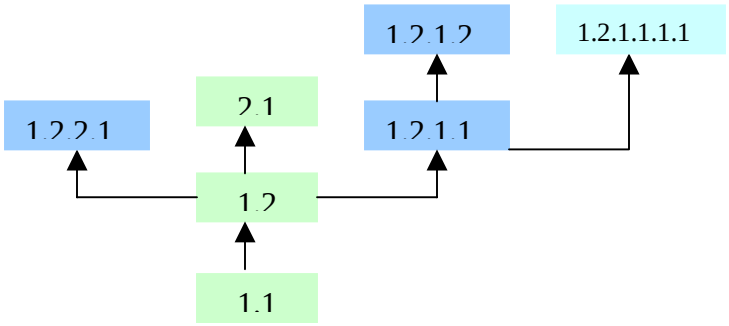


Figure 3.1: Revision tree for a file

The main revision tree has many side branches and each level is coded in a different color. In the case of merged or interleaved deltas used in SCCS, the delta information is stored in a sequential file. They hold information about lines inserted into a version, lines deleted from a version, and lines common to both versions. Whenever a particular revision is desired, the revision file must be scanned to regenerate that version from its delta information. SCCS takes the same amount of time to regenerate any revision or version, but merging a new delta is expensive. The old version has to be regenerated and the difference to the new version has to be computed. This tends to be messy and complicated, especially when overlapping changes and branches are considered [42]. RCS, on the other hand uses reverse deltas for the main revision tree. So, only the latest version of a file is stored, and to generate previous versions, the delta information is used. According to the figure, only version 2.1 would be stored completely. To regenerate version 1.1, reverse deltas are applied to 2.1 and 1.2 successively. Branches in RCS use forward deltas, where the initial revision is stored, and deltas are applied to it to regenerate successive versions. To get version 1.2.1.2 in the figure, reverse deltas are applied until the start of the branch (until 1.2) and then a forward delta is applied to version 1.2.1.1. RCS' delta scheme is faster and more efficient than SCCS [42]. Also, because of the reverse delta scheme, retrieving the latest version is faster in RCS than

SCCS. SCCS can determine when a specific line of code was added to a system [48]. The version numbers are usually sequential, like 1.1, 1.2, etc. The user can explicitly force the numbering to start with a different sequence; like 2.1, 2.2, or 3.1, 3.2, etc.

RCS stores the history information in files with a ',v' suffix, while SCCS stores history information in files with a 's.' prefix. Branches have to be explicitly created in SCCS while they are checked in as any other revision in RCS. RCS searches for a file either in the current directory or the RCS subdirectory. While using SCCS, the SCCS subdirectory should always be used. The major difference between RCS and SCCS is the use of symbols. In RCS, symbolic names can be given to a group of files so that they can be treated as one cohesive unit. The user has to write specific scripts for this in SCCS [48]. When it comes to conflict resolution, both RCS and SCCS allow strict locking of a file, so that multiple users editing the same file do not create inconsistencies. A user locks a file and then checks it out. Another user can checkout and edit the same file only if the lock on it is released.

CVS uses RCS 'behind the scenes'. The deltas and the branches are handled the same as in RCS. However, unlike RCS that supports only a flat file structure, CVS supports a hierarchical structure in the files under version control. This implies that subdirectories can also be part of the 'repository', or the root under which the files are placed. In the same directory as the repository root, there is a directory named CVSROOT that consists of all the administrative files. These files allow a user to customize CVS according to his/her needs. CVS provides more sophisticated locking or conflict resolution techniques than RCS, so that multiple users can checkout and edit the same files. Another distinguishing feature is the use of *tags* as opposed to *symbols* in CVS. Also, the CVS repository can be housed on a server and clients can connect to this server. More details about the features of CVS are provided in Section 3.3.

3.2 GUIs and Tools For CVS

CVS is the most commonly used version control system today. Its sophistication continues to grow with the creation of new tools that provide a GUI to CVS. These tools were intended to introduce a user to CVS commands, and make CVS easy to use for everyday version control. They present a user-friendly, powerful, two-dimensional interface that can be used to execute all the CVS commands and browse the revision history of the files in the CVS repository. Some examples of these tools are: WinCVS/MacCVS/gCVS, and CVSWeb. The features provided by these tools are summarized in [49] as:

- Complete support for all CVS commands, so that beginners and experts can use the GUI.
- Native look and feel for Windows/Mac/ UNIX machines.
- Ability to extend scripting, so that tasks can be customized.
- A browser-like window that allows selection and work on any file/subfolder in the local machine.
- A command line interface to allow execution of commands not supported by the GUI.

- A graphical view of the revision history of the file.
- Files can be added or imported from another source, and the type of the file (text, binary or Unicode) is automatically detected.
- Reserved edits help to organize teamwork, when multiple developers are involved.
- The entire source code is usually made available, so that any necessary modifications can be made.

Since a CVS repository is almost always stored on a server, a host of CVS clients that run on various platforms have evolved. Some of these clients provide a graphical interface, while others are command line. Examples are: MacCVSPro, MacCVSClient, jCVS, TkCVS and SmartCVS [49]. A host of tools for differencing and merging files also exist, but these are out of the scope of this thesis. This section illustrates in detail the features of WinCVS and CVSWeb. They have been selected because the interfaces and the functionality each provides are very different and most of the other GUIs / CVS clients are similar to one of the two.

3.2.1 WinCVS

WinCVS was first released in 1999 by Don Harper. Documentation and download details can be accessed at the website: http://www.wincvs.org [49]. WinCVS can be configured as a client for a CVS repository running on either a Windows NT server or a UNIX family server. It provides a highly detailed graphical interface for almost all the CVS commands. A command line interface is also provided to execute the few CVS commands that do not have graphical support in WinCVS. WinCVS provides a two dimensional interface for the following CVS features:

- Logging on to a server that houses the repository and setting up an authentication method to access the server.
- Checking out a module from the repository into the work area / Updating the work area.
- Editing the files that have been checked out into the work area.
- View text diffs (differences) between revisions of a file. WinCVS can also be used to view differences between revisions of a file using an external differencing and merging tool. Some examples of these tools are: Araxis Merge, WinMerge, CSDiff, Guiffy [49].
- Committing files after editing. Adding/Removing files and folders.
- Multiple Developer Coordination, using the 'Merging and Unreserved Checkout' model and the 'Locking and Reserved Checkout' model [49].
- Release Management by tagging a product release; checkout/update using tags.
- Editing the CVS administrative files.

The first step in setting up WinCVS is specifying the CVSROOT variable, or the path to the repository. The repository may be on the local machine or on a remote server. In addition to the path to the repository, it has to be specified whether the repository is local or remote. Also, an authentication mechanism has to be specified to access the repository, especially if it is password-protected. While installing WinCVS, Python and Tcl/Tk

should also be installed so that WinCVS can run its macros. The WinCVS interface looks as in Figure 3.2. The frame on the right shows the state of the checked-out module, listing the name of the file, the path of the file relative to the repository, the revision number, the status of the file (whether it has been modified after checkout), the tag associated with the file, if any, the timestamp for the file (time of checkout), and conflicts that may be caused by other users editing the same file at the same time.

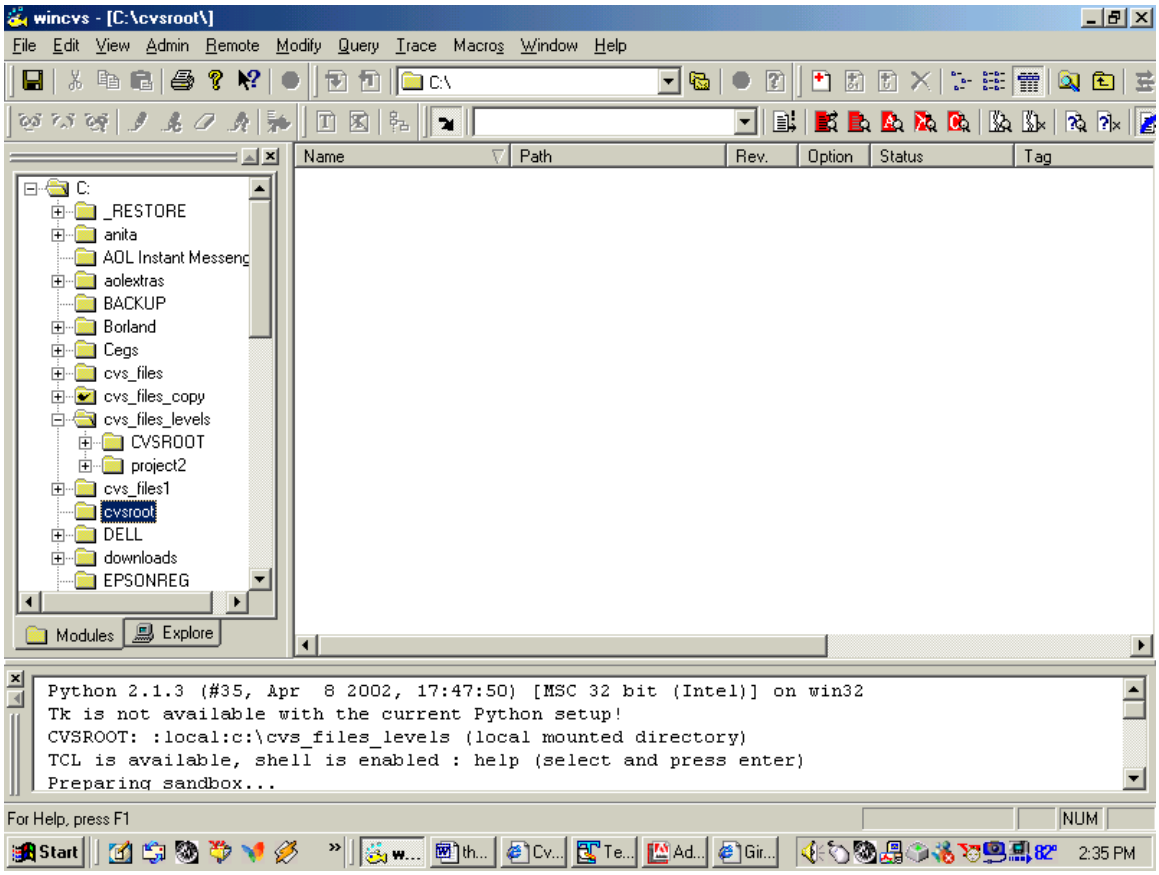


Figure 3.2: The WinCVS interface

3.2.1.1 Setting up the repository and access methods

The 'Admin' menu in Figure 3.2 is used to specify a local or remote server. As shown in Figure 3.3, an authentication method can be set up (local, pserver, ntserver, ssh, etc.). The various authentication mechanisms are detailed in the CVS documentation [43] and discussed more in Section 3.3. The path denotes the path where CVS is installed and CVSROOT is the environment variable that indicates the root of the repository. If the repository is on a remote server, the IP address of the server is indicated. If the repository is password protected (authentication by pserver), the login can also be specified. The other tabs under 'Admin' can be used to specify the format of the command line interface (for example, if using the '-d' option with the cvs command has to be enforced), to specify the external program to be used for displaying 'diffs' between versions of a file,

to specify the default editor for viewing/editing files, to list the cvs modules, and to list the contents of a selected module, to name a few.

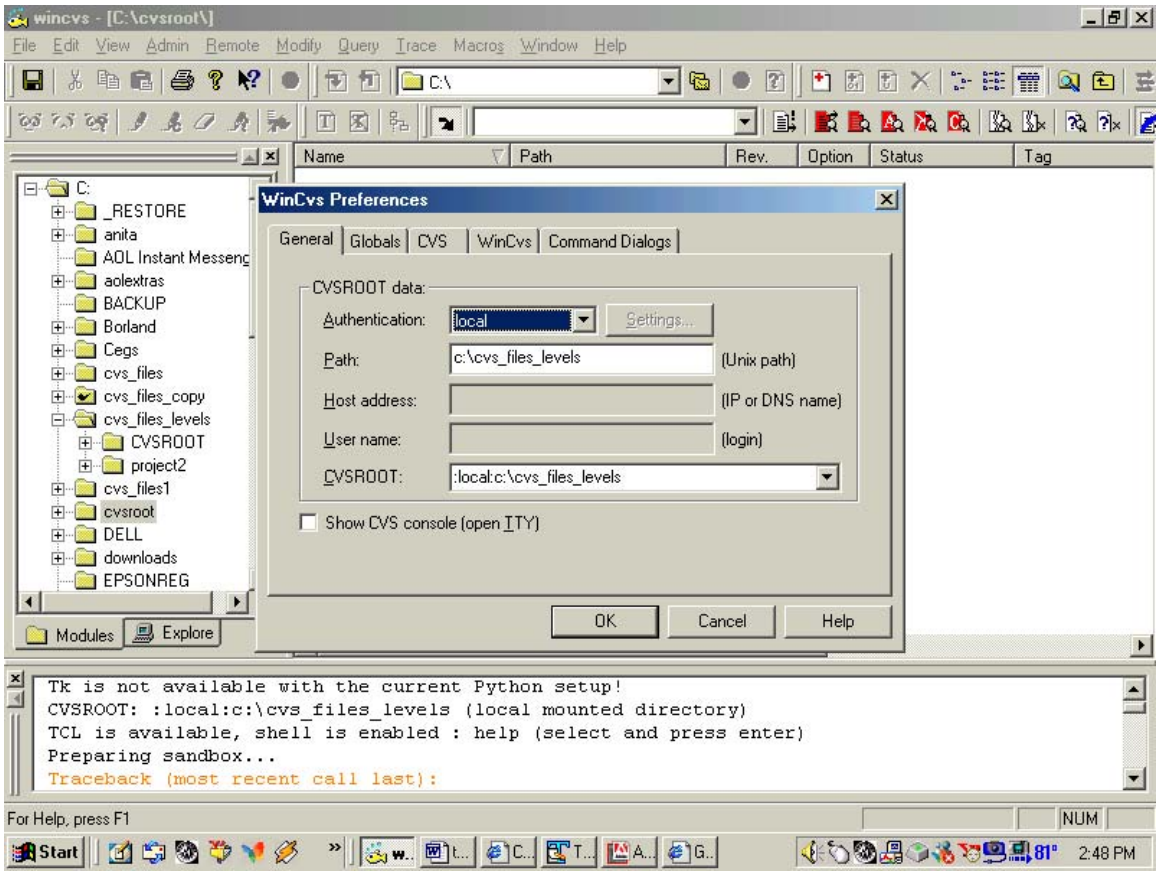


Figure 3.3: The WinCVS 'Admin' options

3.2.1.2 Checking out and committing files and modules

The modules available in the repository can be listed from the 'Admin' menu and the 'CVS Admin -> Checkout module' command can be used to check out a module. 'CVS Admin -> Checkout module' opens a checkout panel to select the local work area or the working directory. Once this directory is selected, the module to be checked out can be indicated. The checkout settings panel can also be used to specify a tag or branch to checkout, instead of a module. Figure 3 illustrates; the checked out module in this case is 'project2'. The CVSROOT has been set to c:\cvs_files_levels for the local machine, as shown in Figure 3.4. The checked out files, and the status of the checkout are displayed as in Figure 3.5. Right clicking on a file can open the file in notepad (or the specified default editor) for editing; it also provides options for committing the file. When a file has to be committed, a 'Commit Settings' panel is presented to the user to enter a log message for the commit. The panel also provides options to retrieve a template for log messages, and for viewing previous log messages associated with the same file.

3.2.1.3 Other features in WinCVS

Besides the basic setup for the administrative files, and the checking out/editing/committing of files, WinCVS provides similar graphical interfaces (as mentioned in Section 3.2.1) for:

- Adding/Importing/Updating files and folders
- Tagging a file or a group of files for release management
- Permitting multiple developers to work on the same files
- Viewing the differences between any two versions of a file in text format (default), or using an external differencing/merging tool to view the differences graphically.

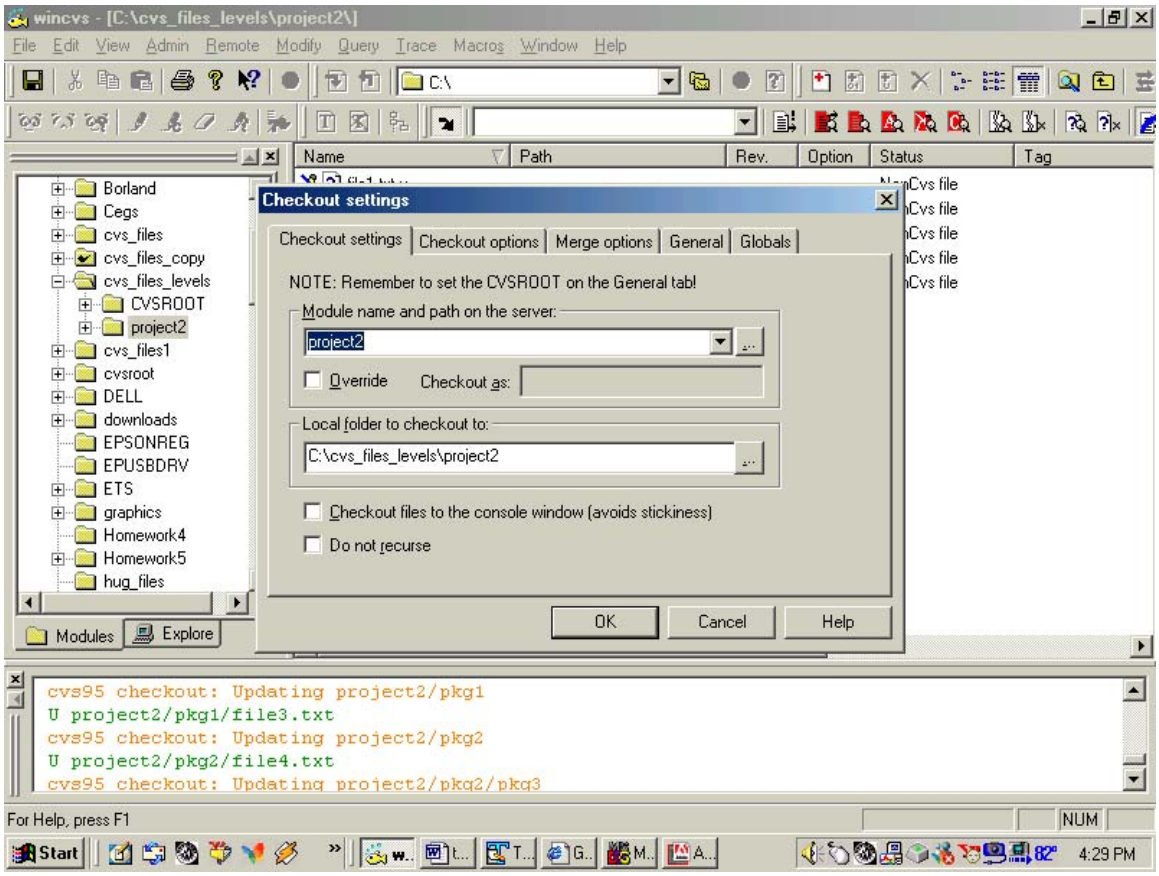


Figure 3.4: Screenshot of WinCVS with the checkout panel

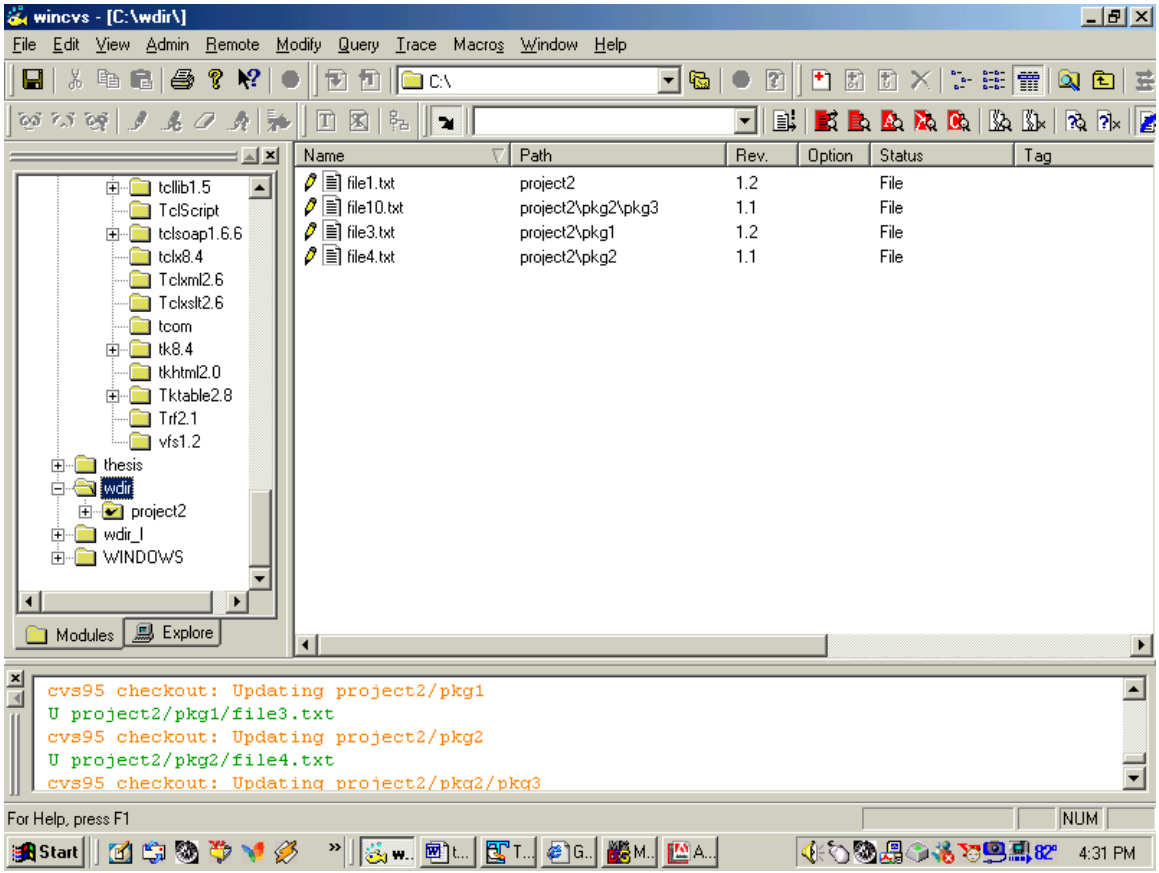


Figure 3.5: The checked out files in the module

More information on configuring and using WinCVS for all the features stated above can be found at http://www.wincvs.org [49].

3.2.2 CVSweb

CVSweb provides functionality that is very different from that of WinCVS. While WinCVS acts as a two dimensional graphical alternative to command line CVS, CVSweb is a 'WWW interface for CVS repositories with which you can browse a file hierarchy on your browser to view each file's revision history in a very handy manner' [47]. CVSweb was first developed by Bill Fenner for the FreeBSD project. The version of CVSweb available today is the FreeBSD-CVSweb, which is an enhanced version of the original CVSweb [47]. The other enhanced version of CVSweb available today is Henner Zeller's CVSweb [50].

CVSweb and its enhanced/advanced versions can be installed on a Web server, and can provide an interface to a file hierarchy in a module housed in a CVS repository. Installation scripts and downloads are accessible at [47]. An example to the interface provided by CVSweb was taken from http://www.freebsd.org/cgi/cvsweb.cgi/ and is illustrated in Figure 3.6. As shown in the figure, the CVSROOT can be selected from the drop down list (in this case, it is FreeBSD). The module path can be stated in the html

form provided, or the links can be used to navigate to the desired module. In this case, the modules under FreeBSD are listed as: CVSROOT, CVSROOT-doc, CVSROOT-ports, etc. Each of these module names is hyperlinked so that their file hierarchy can be explored.

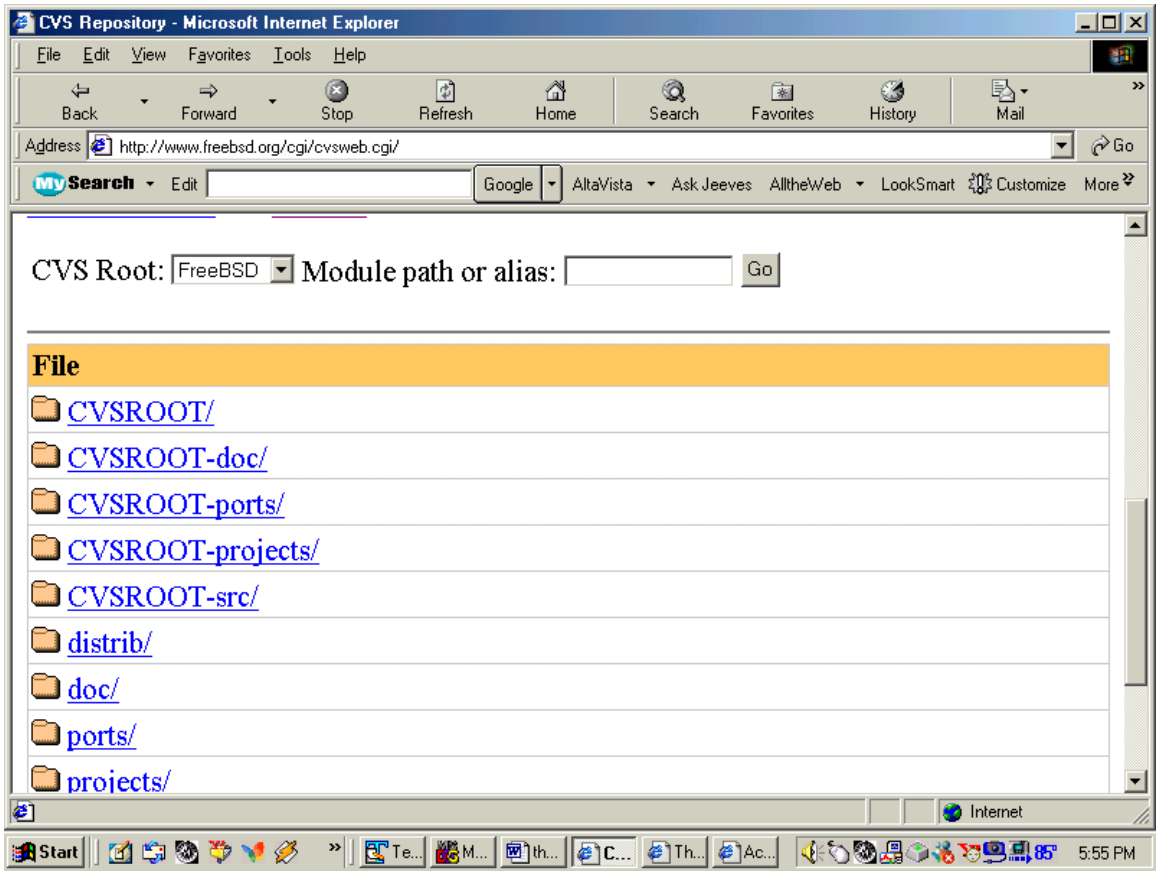


Figure 3.6: A web page using CVSweb from http://www.freebsd.org/cgi/cvsweb.cgi/

If the user clicks on a directory, it opens up the contents of the directory for further exploration. If the user clicks on a file, it opens up a web page that lists the version history for that file. As an example, Figure 3.7 shows the projects/ folder from Figure 3.6. This folder consists of other directories and a single file, README. Also from Figure 3.6, it can be seen that the current revision of README is 1.10. The page also shows the age of the README file, the author who modified the file and the last log entry made for the file. Figures 3.8 and 3.9 shows the revision history of the README file, with links to 'annotate' (see Section 3.3) the file, details about changes since the previous revision in a colored format and a form to request differences between arbitrary versions of the file. The form also has options to select a format to display the differences [47] between versions of a file. There are options to list the differences between a particular revision and each revision of the file (the 'select for diffs' link). Details about each revision include the log message that was entered when that revision was committed. Figure 3.10 shows a sample of a colored diff between version 1.1 and version 1.10 of README.

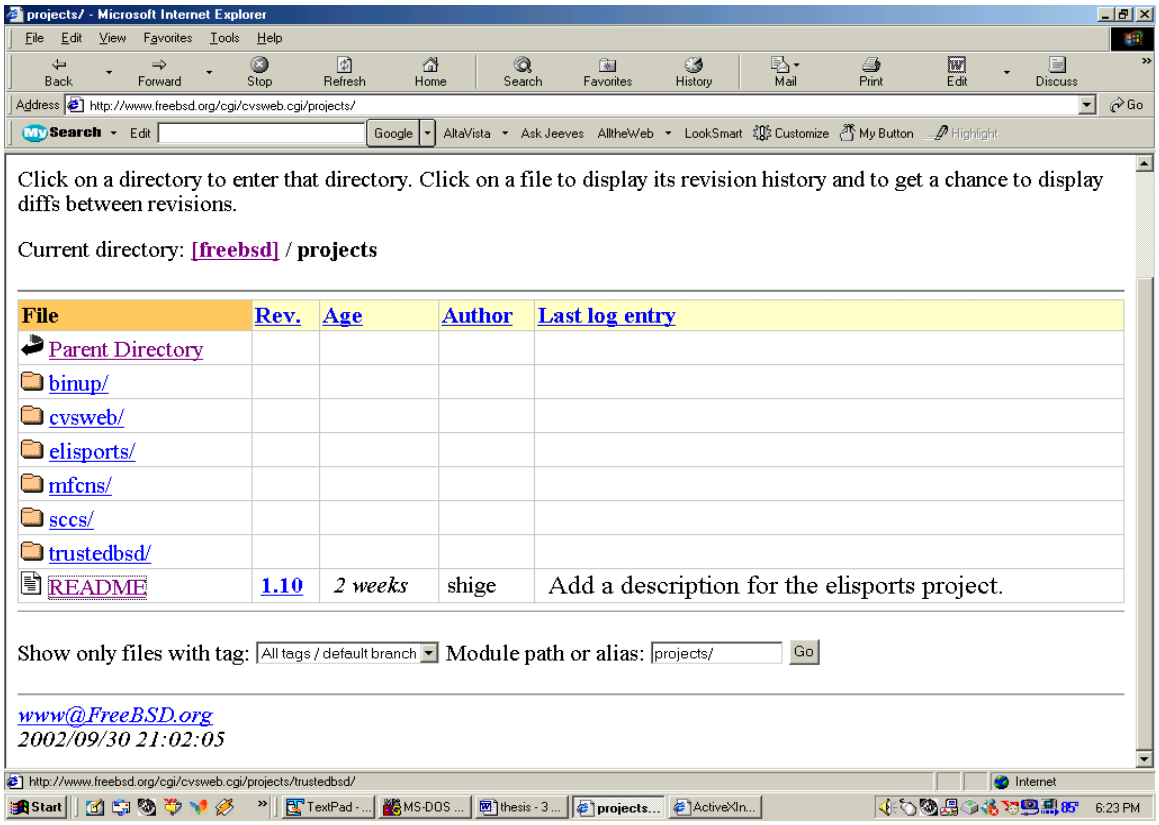


Figure 3.7: The projects/ folder from http://www.freebsd.org/cgi/cvsweb.cgi/

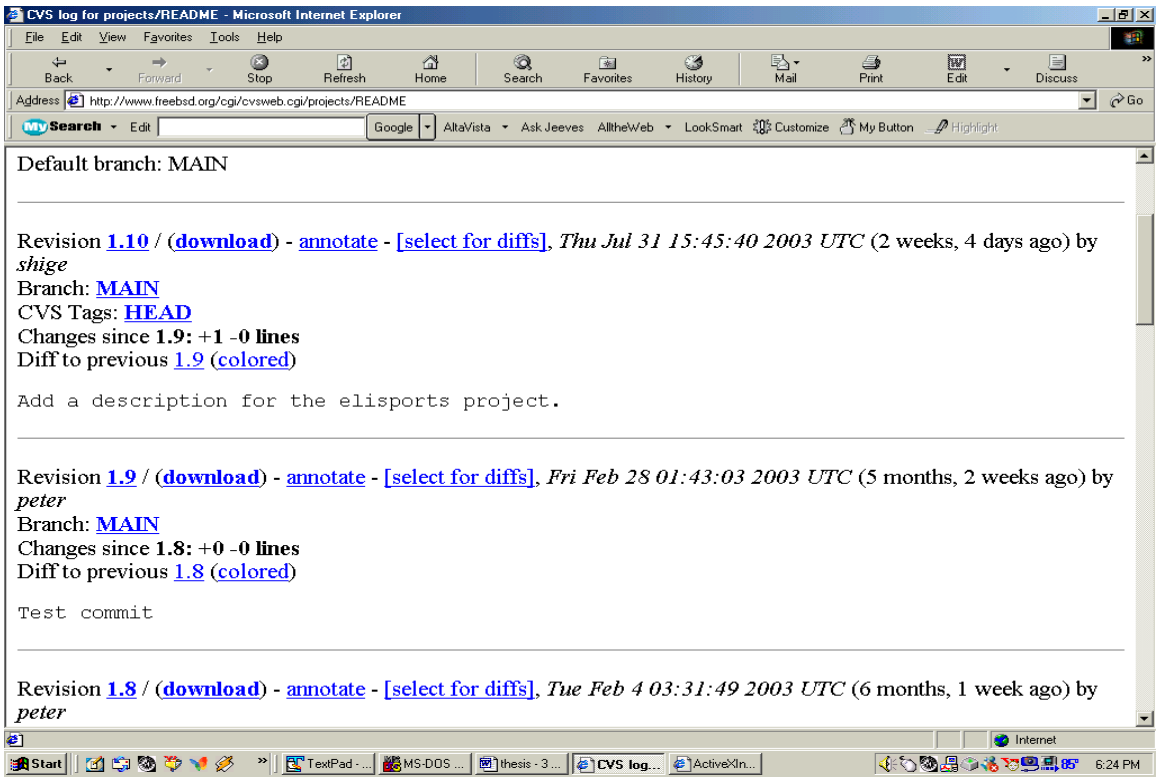


Figure 3.8: Part of the revision history of README from http://www.freebsd.org/cgi/cvsweb.cgi/

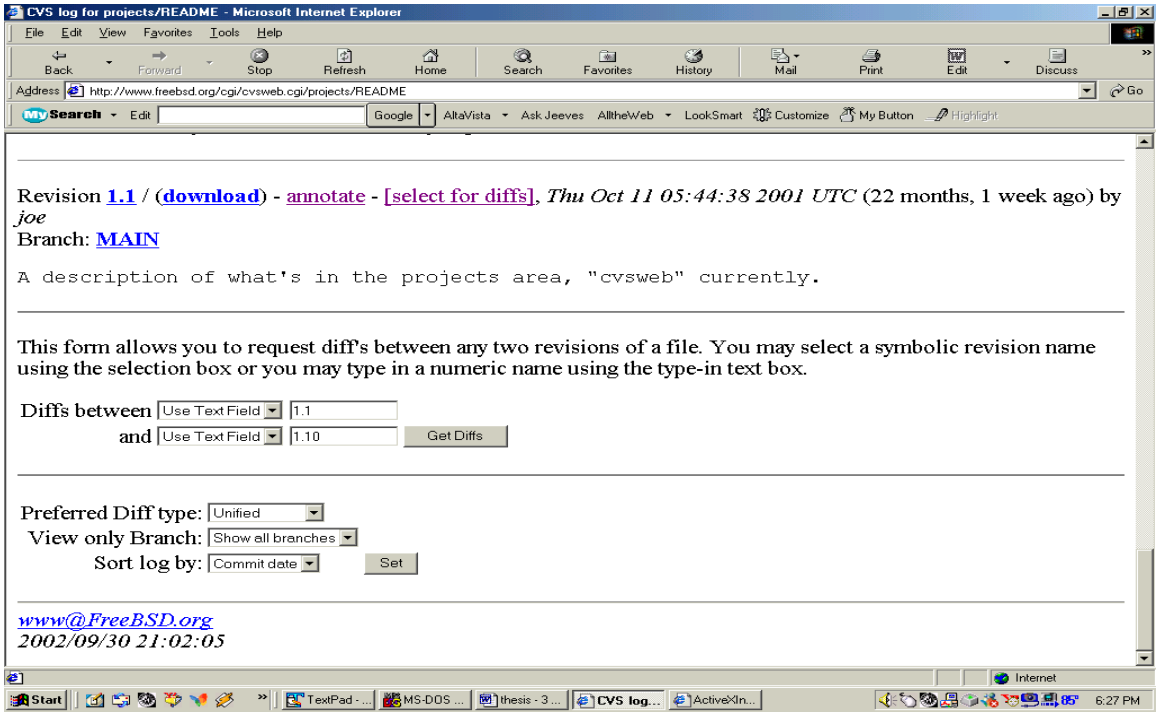


Figure 3.9: Part of the revision history of README from http://www.freebsd.org/cgi/cvsweb.cgi/

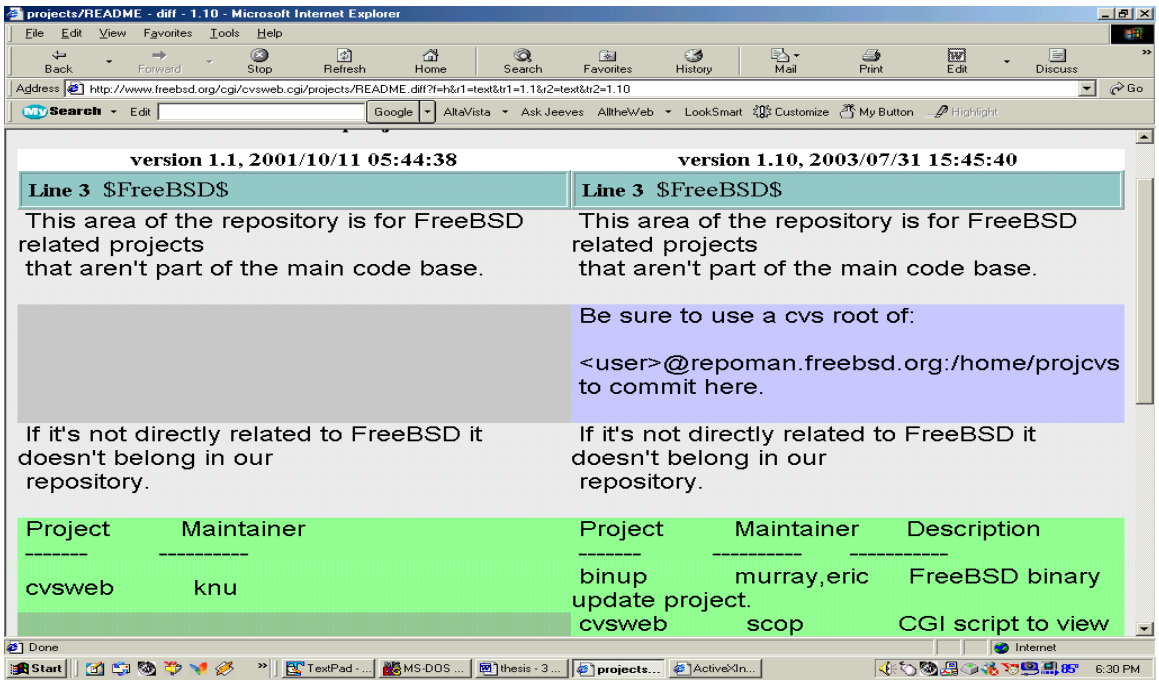


Figure 3.10: Colored Diffs from http://www.freebsd.org/cgi/cvsweb.cgi/

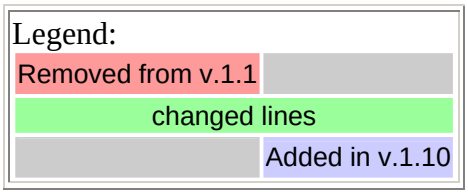


Figure 3.11: Legend for colored diffs in CVSweb from http://www.freebsd.org/cgi/cvsweb.cgi/

So, while WinCVS can be seen as a graphical, two-dimensional alternative to CVS, CVSweb provides a browser interface to the CVS repository through the WWW. CVSweb does not allow any modifications to the files of the module, like WinCVS. Rather, it displays as a web page the structure of the files in the repository and each file's version history, along with options to view differences between any two arbitrary versions of a file in multiple formats. WinCVS can also display diffs between file versions using an external differencing tool, but not through the Web. WinCVS and CVSweb are different beasts that provide different access methods to a repository and different degrees of information about the CVS repository.

3.2.3 Three-dimensional visualization for version control systems

WinCVS, CVSweb and other similar tools provide an excellent two-dimensional interface to CVS, with many and varied features. They represent all the information that a version control system was intended to provide in a user-friendly manner. These two-dimensional interfaces also minimize the need for learning obscure CVS commands, and provide a beginner with a comfortable introduction to the features, functionality and capabilities of CVS. However, information in two dimensions limits visibility into desired information from the version control system and does not always represent all the

necessary details 'at a single glance'. There is a wealth of information that can be derived from a version control system like CVS, and this information is not always apparent from the results of the CVS commands. Direct information from CVS (also outlined in Section 1) includes:

- The module under version control, and the structure of the files in the module.
- The number of files in the module, and the number of versions of each file in the module.
- The age of each version of each file in the module.
- Indicators as to when and where a change was made to a file, and which developer made the change.
- Differences between versions of a file.
- Tags associated with a file or a group of files.
-

However, as was seen in WinCVS and CVSweb, this information tends to be 'flat'. There is no way to visualize the structure of the files under version control (WinCVS just lists the path for the files, as shown in Figure 4 and CVSweb takes the user to a different page for each subdirectory, as shown in Figures 3.6, 3.7 and 3.8). Also, the user has to write additional scripts to visualize the order in which files were added to the directory (to analyze dependencies). If it is desired to view all files associated with all tags in the repository (maybe to view files associated with a release), it cannot be done in a single view in either WinCVS or CVSweb. While two-dimensional graphical interfaces are an improvement over command-line interfaces, the use of three dimensions allows the display of more relevant information. Three-dimensional views of version control systems fall under the broad category of software visualization, which was discussed in Section 2. Section 2 also listed the advantages of three-dimensional visualization, and illustrated some related work in the visualization of version control systems. This thesis focuses on the effective use of the third dimension to display relevant and useful information about a version control system for easier comprehension, specifically:

- A view of the entire hierarchical structure of the repository, with color shades that vary from light to dark as the number of changes to a file (or the number of versions of a file) increase. This view also shows the number of versions of each file in the repository. A cone tree representation is used to view the hierarchy of directories, files and version that make up the CVS repository.
- User interaction to view information about any file, directory or version in the CVS repository. This interaction also allows viewing the source of any selected version of a file.
- Since the depth of the hierarchy and the number of nodes represented can be so high that viewing all the information at the same time may overwhelm the user, toggling on or off of icons representing directory or file nodes is provided. This way, the user can concentrate on and explore only areas of interest, while other paths are hidden from view.

More details about the implementation and samples of each of these views for a version control system like CVS are provided in Chapters 4 and 5. The argument that three-dimensional views enhance visibility into the information housed by a version control system, as opposed to command-line and two-dimensional interfaces, will be

strengthened in these two chapters. More useful views that can be extracted and presented in three dimensions are discussed in Chapters 4 and 6 as suggestions for future work.

3.3 A LOOK AT CVS COMMANDS

This section outlines the most commonly CVS commands and the concept behind each of these commands. The information in this section was obtained from the CVS manual at http://www.cvshome.org [43]. The CVS manual organizes the CVS commands into the following subsections.

- The CVS repository
- Starting a project with CVS
- Revisions
- Branching and Merging
- Recursive Behavior
- Adding, removing and renaming files and directories
- History browsing

3.3.1 The CVS Repository

The repository is the root that houses the modules to be placed under version control. The CVSROOT environment variable is set to point to the repository. This repository can be local or remote. If the repository is remote, it is housed on either a Windows NT server or a server of the UNIX family (Linux, Solaris, etc.). A remote repository provides CVS as a service at a port number, usually 3129. Configuring CVS as a service on a remote server varies with the Operating System used, and is not discussed in detail here. The commands to access CVS as a service are however depicted. Changes to files are not made directly to the files in the repository. The files have to be checked out into a *working directory* before they can be edited or modified. To illustrate the process of 'checking out' a module, consider Figure 3.12.

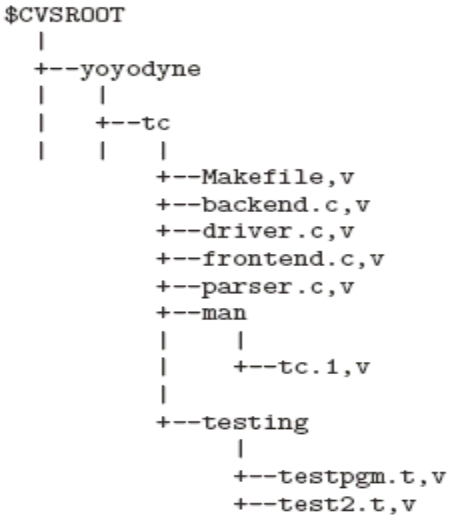


Figure 3.12: The structure of a sample CVS repository, from http://www.cvshome.org

The CVSROOT in this case, is /usr/local. The modules under version control are /gnu and /yoyodyne. Either of these modules can be checked out into a working directory using the command: ***cvs –d /usr/local checkout yoyodyne***

The '-d' forces /usr/local to be used as the repository root, regardless of whether CVSROOT is set or not. This command has to be executed from the working directory, which cannot be a subdirectory of /usr/local or /usr. Once the module has been checked out, any changes can be made to the files in the working directory and these changes can be 'committed' or reported to the repository using the commit command. As an example, the command ***cvs –d /usr/local commit yoyodyne/tc/file1*** can be used. A commit allows the repository to record information about the change such as when and where the change was made and who made the change. This information also allows CVS to compute the differences between the new version that resulted from the most recent change and the previous version in the repository. The difference or the *delta* eliminates the need for storing a version of a file entirely. As discussed in Section 3.1, deltas can be used for efficient storage of information about the differences between versions, so that any version of a file can be regenerated at any time.

Figure 3.12 shows each atomic file with a ',v' extension. This is how the files are stored in the repository and the ',v' extension indicates an RCS history file. So backend.c, v is the history file for backend.c and this file is consulted every time a particular version for backend.c is desired. Usually, if a user needs to check out a directory, the user should have write permissions on the directory. This is because CVS might need to create lock files in the directory to coordinate multiple developers using the same files. One way to provide write access to the same directory to multiple users in UNIX is to set up a group, grant write access on the directory to the group and make all users members of that group. Group access is similar in Windows. The CVSROOT directory found in the repository stores the CVS administrative files. All other directories are user-defined modules. The administrative files also have a ',v' extension, so they are also stored as RCS history files, with the most recent version being replaced every time a commit is made. For example, the 'loginfo,v' file changes every time a new log message is entered and is used to recreate the log messages for any version of a file. CVS sometimes creates an 'Attic' folder in the directory of its modules. So, backend.c,v which is usually found in /usr/local/yoyodyne/tc might be moved to /usr/local/yoyodyne/tc/Attic. The Attic is totally transparent to the user and used by CVS to store files that have been removed or never added for that revision. A repository can be created with the *init* command. If CVSROOT has not been specified, it can be specified with the '-d' option, like: ***cvs –d /usr/local init***.

3.3.1.1 Remote Repositories

The CVS repository need not be on the local machine. It can be housed on a remote server, and the client can connect to the remote server for various actions on the repository. This is called *client-server* CVS. The client can be command-line or customized with a graphical interface. The format of the command to connect to the remote repository is:

[:method] [[user][:password]@]hostname[[:port]] /path/to/repository
The client can connect to the repository using the rsh protocol, password authentication, direct connection with GSSAPI, direct connection with kerberos or using fork. The password is not usually specified in clear text as shown above. Instead password authentication with the login mechanism is used.

Connection via the rsh protocol works as follows. If a user m1 on machine1 wants to connect to the CVS repository on machine2 as user m2, then a .rhosts file should be created in the home directory of user m2 on machine2. This file should consist of the line *machine1 m1,* so that machine2 knows the machine from where a user will be connecting. The format of the command then looks like:
cvs –d :ext:m2@machine2:/usr/local checkout yoyodyne
To connect using a secure password, CVS should run as a service on the server at port 2401 (which is the default, it can be changed). To configure CVS as a service, the /etc/inetd.conf file should be updated on the server. The exact changes to be made are listed in [43]. A password file should be created as $CVSROOT/CVSROOT/passwd and should list the usernames and passwords to allow access to the CVS repository on the server. If the passwd file contains only the user name, but not an encrypted password, then any client can access the repository with that user name without providing a password. Password-less user names are generally used to provide read-only access to a repository. The passwd file also lets the administrator specify a different user name to be used on the server by the client. The format of the command to access a password-protected repository available as a service on a port is:
cvs –d :pserver:username@hostname:/usr/local login
This command prompts for a password. The password entered by the client is compared against the encrypted password listed for the client in the passwd file. If the passwords match, authentication is granted and the client can checkout/use modules. Other methods of connecting to a remote repository (GSSAPI, kerberos, ssh, fork) are outlined in [43].

3.3.2 Starting a project with CVS

It is easier to import an existing group of files into CVS rather than creating them from scratch. After setting the CVSROOT variable, the import command can be used for this purpose. If the files you want to import reside under the directory /temp and you want to import them to the CVS repository under the name impdir, then the command
cvs import –m "Imported files" yoyodyne/impdir yoyo start
can be used. 'yoyo' is a vendor tag and 'start' is the release tag. These attributes are required by CVS as part of the import command. The '-m' prevents CVS from requesting a log message for the import. Files can also be imported from other version control systems into CVS. Usually, most version control systems have a ',v' file for their files. These RCS history files can be move into the appropriate locations in the CVS module. The only constraint is that the RCS history files should not be locked while moving them to CVS. If the file to be moved under CVS control is a SCCS file, CVS has a script called 'sccs2rcs' which converts the SCCS history file to a RCS history file.

3.3.3 Revisions and tags

Every time changes to a file are committed, it results in a new revision. The revisions of a file are numbered sequentially, like 1.1, 1.2, 1.3 etc., as shown in Figure 3.13. The highest version number in the repository determines the version number of the latest revision to a file. For example, if the repository consists of the version numbers 1.2, 2.3 and 3.4, the new file would have version number 3.1. The first number of the new revision number is equal to the highest number in the repository and the second number is always 1.

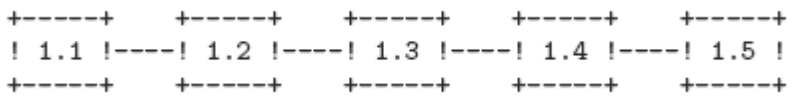


Figure 3.13: Revision numbers, from http://www.cvshome.org

Various revisions from different files are grouped together to form a software release. For example, let the repository consist of files a1, a2, a3 and a4 and each let file have a different number of revisions. Revision 1.2 from a1, revision 2.1 from a2, revision 1.12 from a3 and revision 3.2 from a4 may form a new software release. As a software system develops, it will have multiple releases. For convenience, each of these releases will be *tagged* with a name. The tag for a release can then be used to checkout all the files belonging to that particular release. It is not necessary that a tag should be applied to a group of revisions. A single file can also be tagged with a name that's easier to remember than file x, version 1.2. A file or group of files can be tagged with the *cvs tag* command. While the *cvs tag* command is applied to files in the working directory, the *cvs rtag* command can be used to tag files in the repository. Figure 3.14 shows the concept of tagging versions of different files as part of a software release.

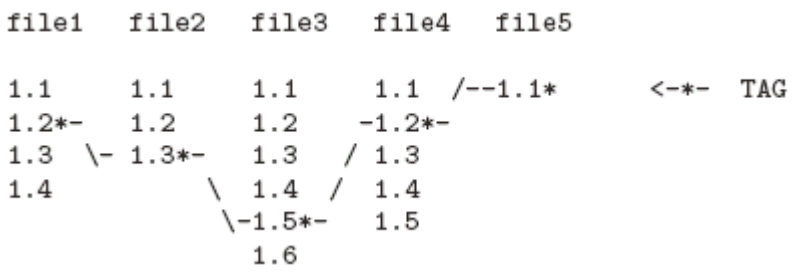


Figure 3.14: Tagging files, from http://www.cvshome.org

It is also possible to specify what to tag by date and revision number. While discussing tags the concept of 'sticky tags' has to be mentioned. These are used to identify the branch to which a file belongs to, and to allow a user to work in isolation without updating his/her working copy. More details are provided in [43]. Sticky tags for a file can be viewed by executing the *status* command for the file.

3.3.4 Branching and Merging

Figure 3.13 showed the revision tree of a single file. Let's assume that release 1 of the software has already been made. After the release, more revisions have been added to the files. Then, a bug is revealed in release one of the software. To correct this bug, the developers should go back to the files that form release one. However, the files have moved beyond these revisions. For example, in figure 12, if a bug is discovered in revision 1.2 after the development has moved to revision 1.5, a new revision has to be added between 1.2 and 1.3 to fix the bug. In order to do this, a new branch is created for the revision tree starting at revision 1.2. Necessary changes are made on this branch and later *merged* with the latest version on the main tree (maybe revision 1.5). The *tag –b* command can be used to create branches and the –j option can be used with the update or checkout commands to merge two revisions. Figure 3.15 illustrates branching and merging.

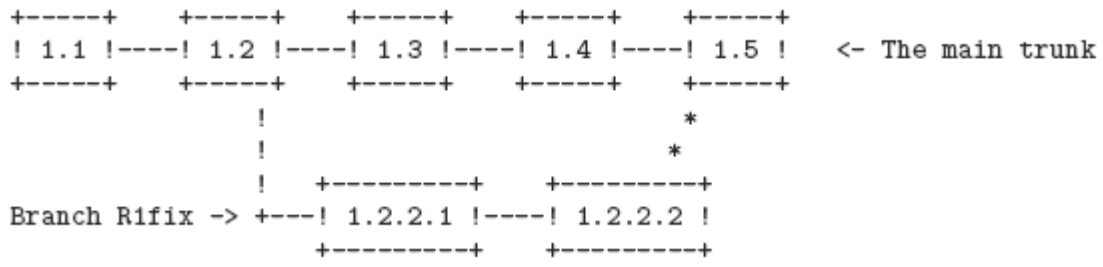


Figure 3.15: Branching and Merging, from http://www.cvshome.org

In Figure 3.15, the revisions on the branch starting at 1.2.2.1 are first merged with the latest revision to give revision 1.5. If a new revision is added to the branch, then it is possible to merge only the changes since version 1.2.2.2 with version 1.5 to yield a new version, so that the older merges are not lost.

3.3.5 Recursive Behavior and Adding/Removing/Renaming files

Most of the CVS commands are recursive, meaning that commands can be executed for files in a subdirectory at any level of nesting, from a parent directory. The add command is an exception. This command is used to add subdirectories and/or files to the CVS repository from the working directory. To add a file to CVS, it is necessary to be in the directory where the file is situated. The add command cannot be executed from a parent directory. After adding a file, it has to be committed, so that a new version is created. The remove command removes files from under the control of CVS. After remove is executed, a commit has to be executed on the file in the working directory, so that CVS knows that the file has been removed. Renaming files can be done by removing the old file, and adding the new file. The renaming causes the revision numbers to be assigned to the renamed file as if it were added to CVS, so the commit –r option can be used to override this. Removing a directory can be done after emptying it of all its files.

3.3.6 History Browsing

The visualization tool described as part of this thesis relies heavily on the CVS history file. By default, recording information in the history file is enabled, and the command cvs history can be used to display the history of the repository. The cvs history command comes with multiple options, all of which are described in detail in the CVS manual [43]. The options described here pertain to the visualization tool developed. The history file of CVS consists of records of different types, listed in the order each action was performed. An example history file is shown below, in Table 3.1.

Table 3.1: Sample output from the *cvs history* command

O	2003-07-09	03:17	+0000	default		Project2	Project2	C:\wd\p2
A	2003-07-09	03:19	+0000	default	1.1	File1	Project2	C:\wd*
A	2003-07-09	03:29	+0000	default	1.1	File2	Project2	C:\wd\p2
O	2003-07-09	03:30	+0000	default		Project2	Project2	C:\wd*
A	2003-07-09	03:33	+0000	default	1.1	File3	Project2	C:\wd\p2
A	2003-07-09	03:42	+0000	default	1.1	File4	Project2	C:\wd\p2
M	2003-07-09	03:45	+0000	default	1.2	File1	Project2	C:\wd\p2
O	2003-07-09	03:50	+0000	default		Project2	Project2	C:\wd\p2
M	2003-07-09	04:03	+0000	default	1.2	File2	Project2	C:\wd\p2
M	2003-07-09	04:10	+0000	default	1.3	File1	Project2	C:\wd\p2

The records in the history file belong to one of the following categories:

- O: The module or file was checked out
- A: The file was added to be placed under version control – this denotes the first available of the file.
- M: The file was modified and a new version of the file was created as a result of a commit.
- R: The file was removed from version control.
- U: Similar to a checkout; the file or module already checked out was updated.

It is obvious from Table 3.1 that CVS can also handle files in subdirectories nested to any level within the main repository. The history file provides information about which file was checked out or modified, when a file was checked out, when it was added, modified or removed, the relative location of the file within the repository, the working directory, and the user who was responsible for the modification. Some of the options that can be used with the history command include:

- The *cvs history –e* option, to list all the records
- The *cvs history –e filename* option, to list all records for filename
- The *cvs history –x recordtype [filename]* option, to list records of a particular record type for all files or for a particular file indicated by filename. The record type may be any combination of O, A, M, R and U

Other commands to browse history are: *cvs annotate*, *cvs log* and *cvs status.*

3.4 A BRIEF SUMMARY OF SECTION 3

This section introduced version control systems, and detailed some of the commonly used commands in CVS. Section 3.1 compared RCS, SCCS and CVS and stated similarities and differences between them. Section 3.2 illustrated some CVS GUI tools like CVSweb and WinCVS, and emphasized the need for three- dimensional visualization of version control systems. Section 3.3 demonstrated the CVS commands and the features provided by CVS in some detail. Section 4 will talk in detail about the implementation of the three-dimensional visualization tool to provide the three views stated in Section 3.2.3. Section 5 provides some case studies and evaluates the tool. Section 6 then concludes with suggestions for future work.

CHAPTER 4: VISUALIZING WITH CONE TREES

The first three chapters discussed version control systems and the features of three-dimensional visualization that make visualizing in three dimensions preferable over visualizing in two dimensions. Previous work on version control visualization was presented to emphasize this concept. This chapter illustrates and analyzes the algorithms used for developing a visualization tool than can automatically visualize in three dimensions, any module in a CVS repository. Section 4.1 talks about the proposed metaphor for three-dimensional visualization. Section 4.2 introduces the algorithms for automatic construction. Section 4.3 lists additional information that can be harnessed from the CVS logs, if necessary. Section 4.4 and 4.5 discuss the scalability and complexity respectively of the metaphor/algorithms presented in Sections 4.1 and 4.2. Section 4.6 concludes with a brief summary of Section 4 and suggestions for possible improvements.

4.1 PROPOSED METAPHOR

The information to be visualized is inherently hierarchical. It resembles the structure of a file system on UNIX or Windows machines, with a *root* and *subdirectories* and *files* under the directories. Obviously, the subdirectories can be nested to any level. Several three-dimensional structures have been proposed to visualize hierarchies, like cone trees [50], treemaps [51], hyperbolic trees [52] and disk trees [53]. They each have their own advantages and disadvantages, with more or less similar features and capabilities [54]. To illustrate the hierarchical structure of a CVS repository consisting of directories, files, and any number of *versions* per file, cone trees are used here. Traditional file systems are represented as inverted two-dimensional tree structures, with the root of the tree as the root of the hierarchy. Using cone trees to represent a hierarchy in three dimensions is a natural extension, because cone trees have been described as ‘a three-dimensional representation of hierarchical information in which one node is located at the apex of the cone, and all of its children are arranged around the circular base of the cone’ [50]. For the purposes of visualizing version control systems using cone trees, three types of nodes and three types of links between nodes are identified:

- Directory nodes
- File nodes
- Version nodes
- Link between a parent directory and its subdirectory
- Link between a parent directory and its file
- Link between a file and its version node

The original cone tree proposed by Xerox Parc is shown in Figure 4.1. It has nodes at different *levels* and a varying number of nodes at each level. When visualizing hierarchies with a large *depth* or hierarchies with a huge number of nodes at each level, the performance and effectiveness of the cone tree is not too beneficial. However, for visualizing *reasonably large* hierarchies, cone trees are an extremely good metaphor. The

questions of large depth, large number of nodes at each level and reasonably large hierarchies are discussed in Section 4.4 (scalability).

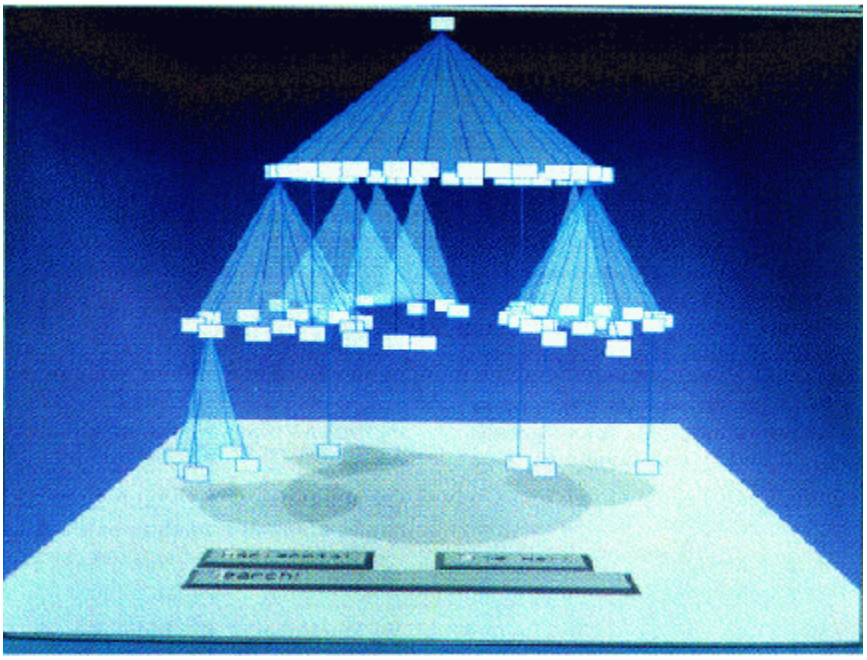
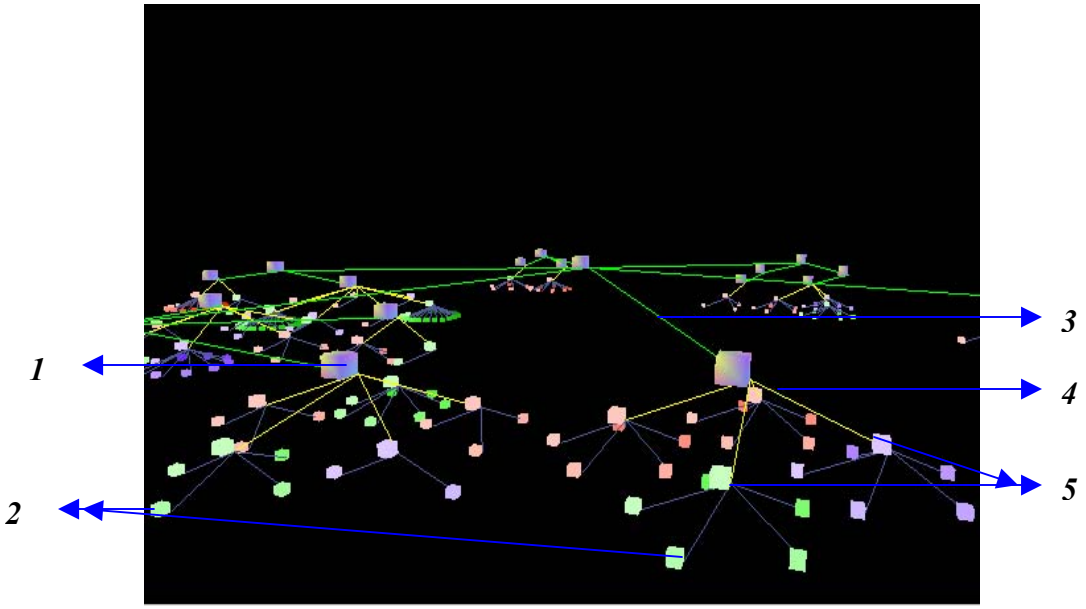


Figure 4.1: The original Xerox Parc cone tree [50]

In the cone tree constructed for visualization of a module in a CVS repository, cubes represent the directory, file, and version nodes. Links are the lines between the nodes. A directory node is a shaded cube. A file node is a simple red, green or blue cube and a version node is a cube with varying shades of red, green and blue. These shades range from light to dark, and represent the versions of a file from version 1 to version n. So, version 1 is a light-colored cube and version n is a darker cube of the same color. The leaves of the cone tree are always the versions of a file. A subdirectory is represented in the cone tree only if it has a file with at least a single version. A sample cone tree illustrating the metaphors for a CVS repository is shown in Figure 4.2.



1: Directory node
2: Version History Nodes
3: Link between directory nodes
4: Link between a directory node and a file node
5: File Nodes

Figure 4.2: Mapping the data structures to the visual representation

4.2 ALGORITHMS FOR AUTOMATIC CONSTRUCTION

First, information about the CVS repository is gathered into data structures. After this information is collected, it can be visualized using a cone tree. The cone tree algorithm assumes there are three types of nodes in the hierarchy

- The directory nodes
- The file nodes
- The version nodes

Files are present in a particular directory and may have one or more versions. The version nodes for a file are always the leaf nodes in the cone tree. File nodes are parents of leaf nodes and directory nodes are parents of file nodes or other directory nodes. The cone tree algorithm starts by fixing the radius of each file or directory node. The radius of each of the children of the node determines the radius of a node. So, the radius should first be determined for nodes at any level i-1, before it can be calculated for the parent node at level i. Also, before determining the radius of a node at level i, the number of children for that node should be determined. From the radius of a node at any level and the number of children at that node, the coordinates for the children of the node can be determined.

Building the number of children at each node starts at the lowest level. The number of versions of a file determines the number of children of a file node. The number of files and subdirectories of a directory determines the number of children of a directory node. The radius of each node is then determined as follows, based on the cone tree algorithm presented by Carriere and Kazman [54]. The algorithm presented here differs from the one in [54] in the determination of arc length for a child and the angle at which a child is placed. The algorithm is as follows:

- The radius of a file node is directly node is directly proportional to the number of versions that are available for the file node. This determines the radius of a node at the lowest level.
- At all the levels above, the circumference for a cone at level n-1 is estimated by

$$C_{n-1} = 2\sum_i r_{i,n} \quad \text{---------- Equation 4.1}$$

$r_{i,n}$ represents the radius of node i at level n. The radius of a node at level n is the calculated by:

$$r_n = C_n / 2\prod \quad \text{---------- Equation 4.2}$$

Once the radius for each node has been calculated, the location of the node in (x,y,z) coordinates can be computed. The circumference of a cone at level n determines the area available for the placing of the child nodes of the cone. Since the base of a cone is a circle, information about the number of children at a node and the radius of the node can be used to determine the position of each child of the node on the circle. A circle is a two-dimensional entity, so the parametric form shown in Equation 4.3 is used to generate points on the circle. 'r' is the radius of the cone at level n, and 'angle' is the angle at which a child is placed around a node. The number of children for a node and the fact that a circle's arc is 360 degrees determines the values for 'angle'. So, if a node had four children, these children would be placed around the node at angles: 90, 180, 270 and 360 degrees respectively.

$$x = r \cos(\text{angle})$$
$$z = r \sin(\text{angle}) \quad \text{----------Equation 4.3}$$

In the three-dimensional cone tree, the *x* and *z* coordinates represent the position of each directory, file, or history node. The *y* coordinate is used to represent the level of the node. So, directories and files at level n are *higher* along the y coordinate while directories and files at level n-1 are *lower*. The radius of a file node is adjusted according to the number of versions it possesses. If the same fixed radius were used for all files, visual clutter would be impossible to avoid when a file has more than a hundred versions. Similarly, the spacing between children of a node can be increased to prevent overlap.

4.3 ACCESSING OTHER INFORMATION FROM THE CONE TREE

The cone tree coveys information about the size of the system under version control, meaning the number of directories, files and versions for each file. It also shows the structure of the system under version control, by representing the way the files and directories are organized. The files that have undergone the maximum changes can be easily identified, because their cones are much denser than the others'. Again, the number of nodes that a cone can have without leading to clutter is discussed in Section 4.4. The features provided by the cone tree to visualize a version control system in three dimensions are:

- *Zooming in / Zooming out*: It is possible to use the keyboard and zoom-in to view any particular node of interest. The files contained by the node and the number of versions of each file in the node can be clearly visualized. Once the details for a node are visualized, the keyboard can be used to zoom out and view the overall structure. Examples are provided in Section 5.

- *Rotation and Navigation*: The keyboard can be used to rotate the visualization around and view it from different angles. This way, it is easier to bring parts of the tree into focus, instead of being overwhelmed by the entire structure. The concepts of zooming in and out, and navigation and rotation are the main features of any effective software visualization, as discussed in Section 2. These features aid faster comprehension and selective understanding of the presented material by allowing the user to move to and focus on areas of interest.

- *User interaction*: The user can click on any node in the visualization with the mouse to view information about that node. A left-click on a directory or file node displays the name of the directory or the file relative to the root module being visualized. In the above example, this is the *implementations* module. A left-click on a node representing a version of a file displays information about the name of the file, the version number that was assigned to it, the date of creation and whether the version resulted from adding a new file, or modifying/removing an existing file. A right-click on a version node opens the source code of the corresponding version of the file in *notepad*. It is also possible to toggle a directory node *on* or *off*, by a right-click. If a directory node is toggled *off*, no child nodes are displayed for that directory node. If a directory node at level n is toggled *on*, the nodes at level n-1 are displayed, but not the nodes below level n-1. Toggling reduces visual clutter caused by the presence of innumerable nodes by allowing the viewer to decide what he/she wants to see. It also avoids *cognitive overload* on the user by conveying only the required information. File and version information is available only for selected nodes. The node representing the root of the module cannot be toggled *off*. Again, examples are provided in Section 5.

- *Identification of areas that change constantly*: From the cone tree, the areas of the module that have changed more frequently than other areas can be identified. When a file has more versions, the base of the cone representing the file has many *points* generated for these versions, i.e., the file's cone is much denser. With the three-dimensional visualization, identifying the areas of constant change is much

faster than using a command-line interface. In the latter case, the logs for each file in the hierarchy have to be analyzed before identifying the areas of maximum change. These areas may have to be identified for a new user to learn about the functionality of the system. He/she would know where to concentrate while understanding a module or software system, because the same area is more likely to change in future than the others.

4.4 SCALABILITY

It is important to determine the maximum number of nodes, branches and levels that can be handled by the visualization. When the size of the repository exceeds these thresholds, the advantages of three-dimensional visualization are lost, and the representation is no longer clear and distinct. Since the cone tree algorithm presented above determines the radius of a node at level i based on the radii of its children at level i+1, the scalability of the visualization depends on the radius / circumference of a node at any level. The sums of the radii of all nodes at any level i should not exceed 1000. These nodes may be the children of a single node or the children of sibling nodes. Also, the number of versions of a file that can be represented without clutter is 500. This number also applies to the maximum children that a node can have without causing fuzziness in the display. Even though the radius of a file node is linearly proportional to the number of versions of the file, when the number of versions for any single file exceeds 500, the cubes that represent the versions are no longer distinctly visible. These numbers were derived from testing the quality of the visualization with an increasing number of nodes, as well as the *refresh rate* for the visualization, determined by the complexity of the cone tree algorithm. More about complexity is discussed in Section 4.6.

Some CVS repositories do have around 700 – 1000 versions for a single file, though such a high number of versions for all files are rare. Repositories that contain the source of web pages are more likely to have a large number of versions for each file. For example, the index.html file of the htdocs module of the NetBSD project has around 900 versions [54]. Similarly, most of the html files in the www module of the OpenBSD project [55] have around 300 – 800 versions. This might not prove that html files have more revisions than other file types, but it is a possibility because web pages are being constantly changed to reflect new updates or releases or convey the most recent news to users. In the case of files that have an extremely large number of versions, or even directories that have a large number of files and subdirectories, other metaphors can be used to indicate the fact that a node is larger than average. The visualization can also provide options to view information about the node in parts, or only display changes made in a particular period of time.

Since the CVS repository can be housed on a central server, and clients can connect to the server, the possibility of visualizing the history of the CVS repository on multiple client machines can be explored. In this type of networked virtual environment, the number of clients that can connect to the Linux or Windows NT server hosting the repository limits the number of clients that can connect to the CVS server. The CVS server only sees multiple clients requesting a check out or update of the same module.

Once the check out or update is complete, each client requests the CVS repository over the network for its history information. The delays that will be created by multiple clients requesting the same files have not been studied in this thesis. Once the history file is received, building the visualization is local to each client and depends on the client's memory capacity and processing speed. The only other situation that can lead to delays is when multiple clients request the same file over the network to view source code. Again, if this file is extremely large, it might lead to longer delays. Every time the visualization is rotated, or some node is toggled *on/off*, the *refresh rate* for the display depends only on the client machine's processing power and resources, because there is no need to connect to the CVS server over the network.

4.5 COMPLEXITY

The pseudocode for the cone tree algorithm used for the visualization starts with the root module and proceeds recursively in a depth first manner, drawing nodes for all the children of the current directory and then for all the siblings of the current directory.

```
begin drawnode
n = no_of_children for current node
temp1 = pointer_to_files_list of current node
temp2 = pointer_to_child_directories_list of current node
for(angle = 0;angle<2*PI;angle+=2*PI/n)
{
 generate x,z coordinates
get current node's y coordinates
   if(temp1!=NULL)
   {
     draw file node for temp1 at x,y,z;
     get next file;
  }
  if(temp2!=NULL)
 {
   get temp2's y coordinates (lower level than parent directory)
  draw directory node for temp2 at x,y,z;
 get sibling pointer of temp2;
  }
}
repeat procedure drawnode with child pointer;
repeat procedure drawnode with sibling pointer;
end drawnode
```

It can be shown that this algorithm is of order O ($m_1 + m_2 + \ldots + m_k$), when the cone tree has k levels, and m_i is the total number of children at level i. So the cone tree algorithm is of complexity O(n), where n is the total number of nodes in the cone tree. It is not possible to classify the complexity as quadratic, or cubic, or exponential, or logarithmic, because the number of nodes at each level varies. As a simple example, consider a root

module with 5 subdirectories. Each subdirectory has 100 children, and the first subtree of the second level has 3 children in each branch. Each node at each level of the tree has a varying number of children, with a total of 300+500+5 nodes. Each node or cube is represented by six polygons, so the total number of polygons required to render the cone tree are: 4830. The number of polygons that has to be rendered obviously increases with the number of nodes required to represent the version control system.

The processing capabilities and graphics hardware on the machine visualizing the version control system will determine the maximum number of nodes that can be rendered. Though this number can be very high, the structure of the system being visualized becomes incomprehensible when the radius of any node exceeds 1000 or the number of children of a single node exceeds 500 or the total number of nodes at any level exceeds 3000.

4.6 POSSIBLE IMPROVEMENTS

The three-dimensional cone tree discussed above is useful in visualizing huge hierarchies like a module structure in a CVS repository and present useful information about the evolution of the module. While scalability issues do arise for extremely large hierarchies, it is still much better than a flat two-dimensional representation in terms of space, clarity, and aiding faster understanding. Different metaphors can be explored for representation of a version control system, and the effectiveness of the metaphors can be studied by exploring their effect on comprehension. Other useful information about the version control system can also be represented using three dimensions. Multiple attributes like code churn for a particular period of time, lines of code per developer in the development of the system, developer productivity at any point in time and the changes in the structure of the system over time can be represented using three dimensions. It might be useful to study the usefulness of three-dimensional representations by attempting to visualize the same information in two dimensions, and exploring the effects on the end-user. Studying files that are changed together most of the time can help explore *clustering* effects. The clustering effect can expose dependencies among files and help new developers identify files that have to be explored when any file is changed.

Other improvements can be made in the user interface design. The metaphors or shapes in the cone tree and their colors are fixed. The user should be able to choose what he/she wants to see in the visualization, by selecting shapes, textures and colors for the nodes from a list. Also, the user should be able to select what attributes of the history of the module he/she wishes to see. Also, a user should be able to specify the age of the versions he/she wants to see. For example, the user can request to see only the changes made ‘two weeks ago’ or ‘six months ago’. The user interface design should be flexible enough to customize the visualization according to the end-user’s needs.

So, future work can concentrate on visualizing multiple attributes according to the end-user’s requirements, and on emphasizing the benefits of three-dimensional over two-dimensional visualization for version control systems.

CHAPTER 5: CASE STUDIES / EXAMPLES

This chapter presents screenshots of the three-dimensional visualization, whose implementation was discussed in Chapter 4. Two sample modules from a CVS repository are used as case studies, in Section 5.1 and Section 5.2 respectively. These screenshots illustrate the features discussed in Chapter 4. For each case study, an overview of the cone tree layout (the 'big picture') is first provided, followed by illustrations of zooming in on subtrees, pruning of subtrees (toggling), rotation/navigation of the cone tree and user interaction with the cone tree. The areas that have changed the most are also identified for each case study.

5.1 EXAMPLE 1

The case study for the first example is the *implementations* module of the OpenEAI project, available on the World Wide Web at http://www.openeai.org/cgi-bin/cvsweb.cgi/project/java/source/org/openeai/. The history of the module is published on the website mentioned above using CVSWeb. With CVSWeb, the *depth* of a module can only be determined by browsing each subdirectory at each level, but with the three-dimensional visualization, the size of the module can be determined instantly. Figure 5.1 shows the cone tree layout of the *implementations* module.

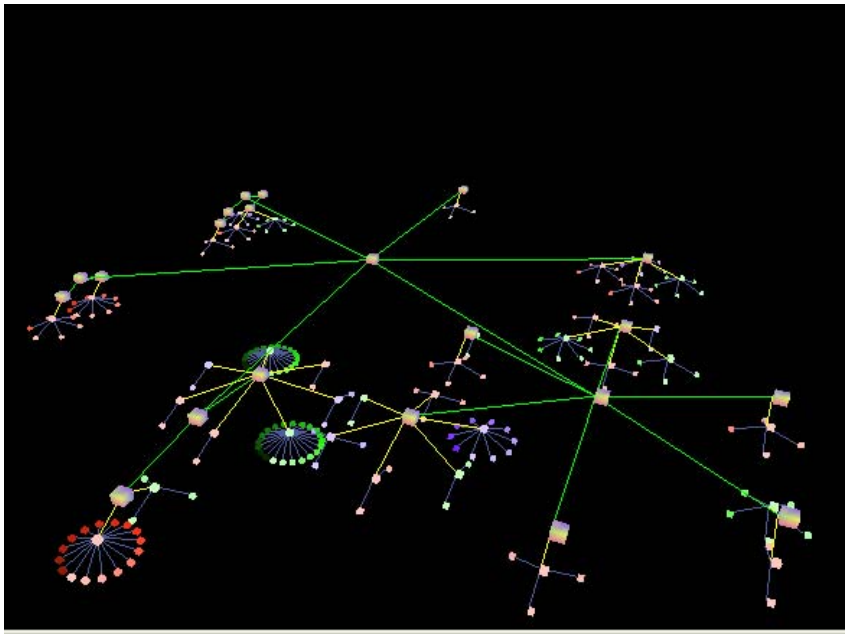


Figure 5.1: An overview of the *implementations* module

The directory, file and version nodes are visible from the metaphors provided in Section 4. The children of each node are also visible from this overview. The cones of files that have been changed the most have a broader base, indicating more versions.

Figure 5.2 zooms in on the *implementations/services* subtree. The cone tree can be rotated about the x, y or z axis and the camera can be moved along any of the three axes to focus on the point of interest. The subtree of interest in Figure 5.2 is small, with a few files. Each of these files also has a lesser number of versions than other files in the root module. Subtrees other than the one of interest hover on the horizon.

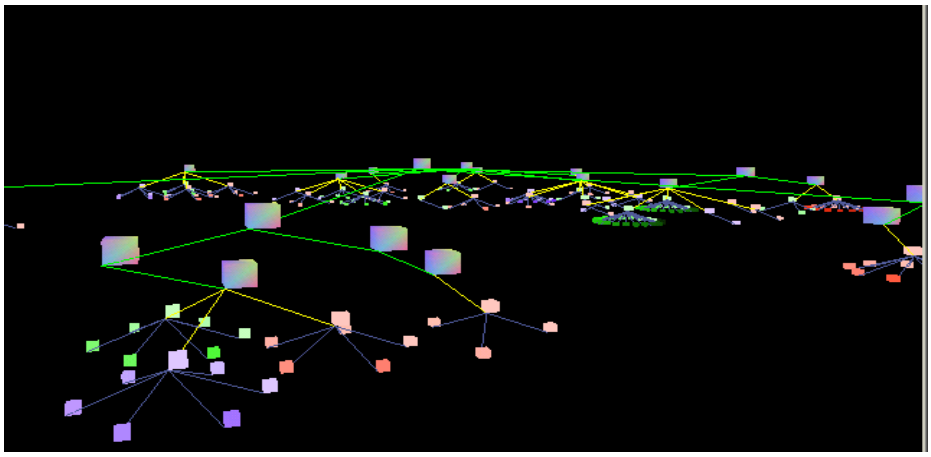


Figure 5.2: A closer look at the *implementations/services* subtree

Figure 5.3 shows a pruned cone tree. The directories at each level are now clearly visible without any clutter, providing a high-level abstraction of each level.

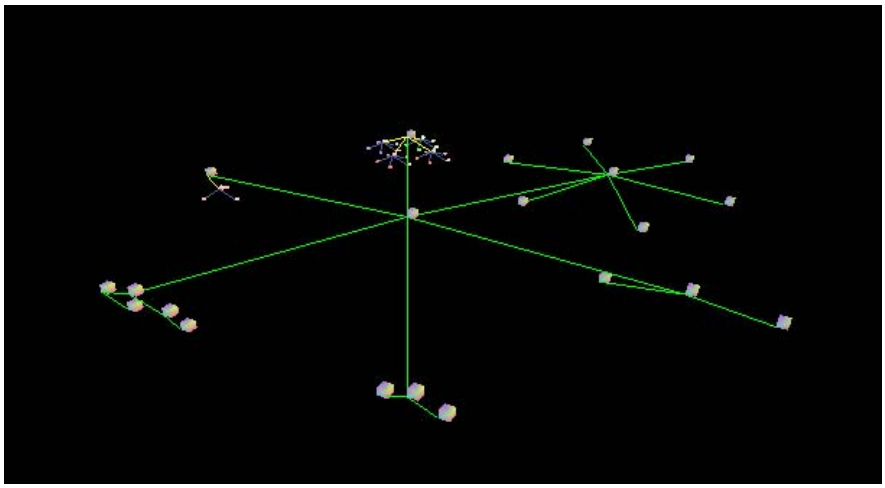


Figure 5.3: Pruning of the cone tree

Figure 5.4 expands one of the pruned subtrees in Figure 5.3. The expanded subtree is brought into focus, while the other subtrees remain collapsed.

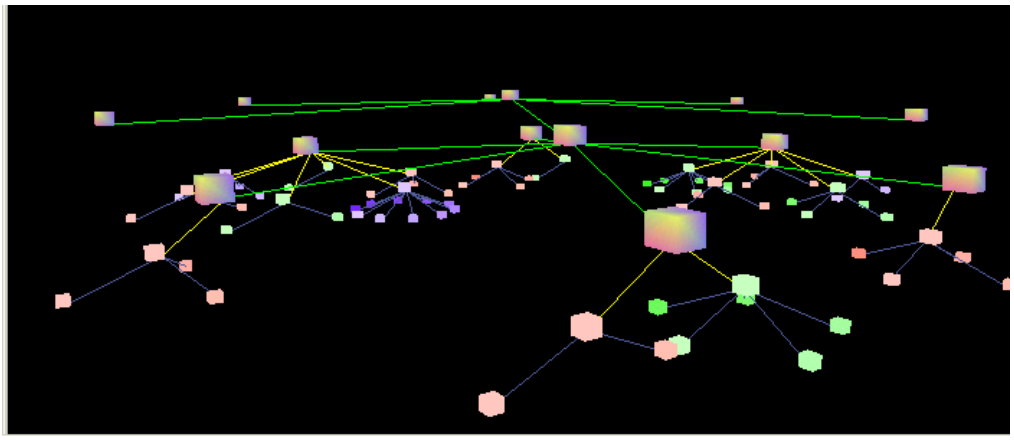


Figure 5.4: Expanding and exploring the *implementations/gateways* subtree

Figures 5.5, 5.6 and 5.7 show examples of user interaction with the cone tree. A directory or file node can be selected to view information about it, or a version node can be selected to view the source code corresponding to that version.

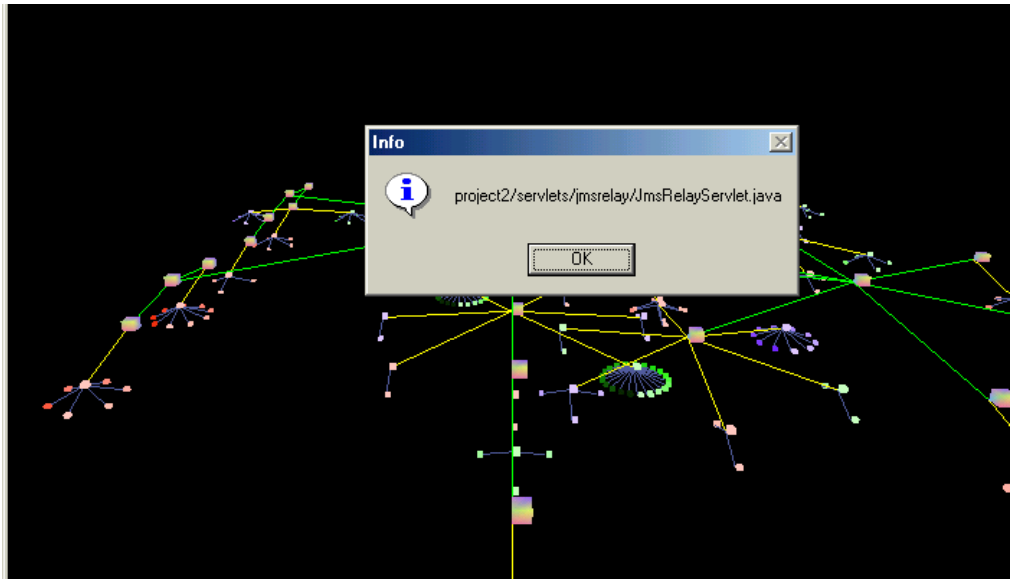


Figure 5.5: User Interaction for a directory/file node

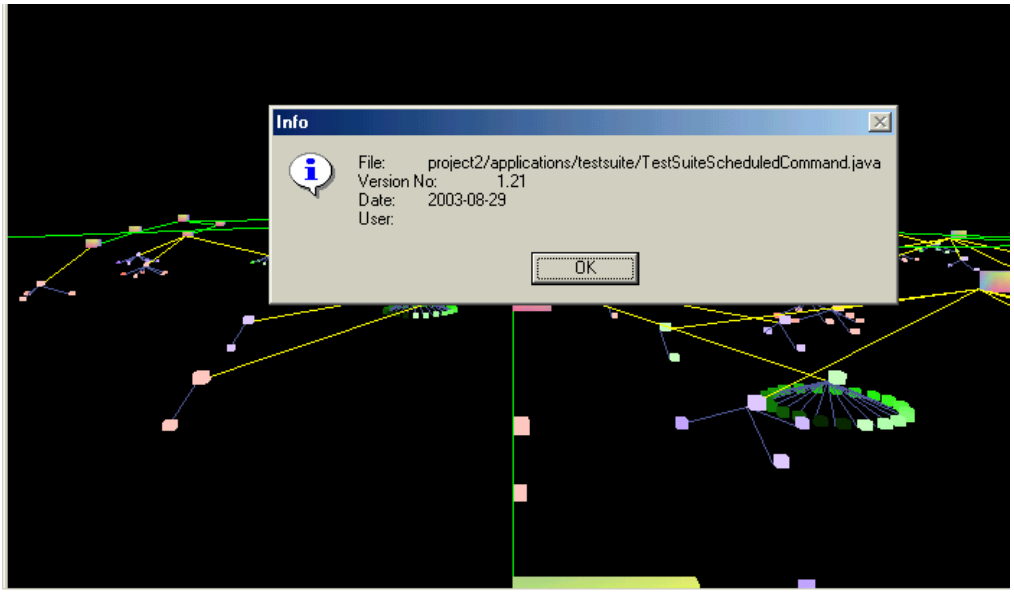


Figure 5.6: User Interaction for a version node

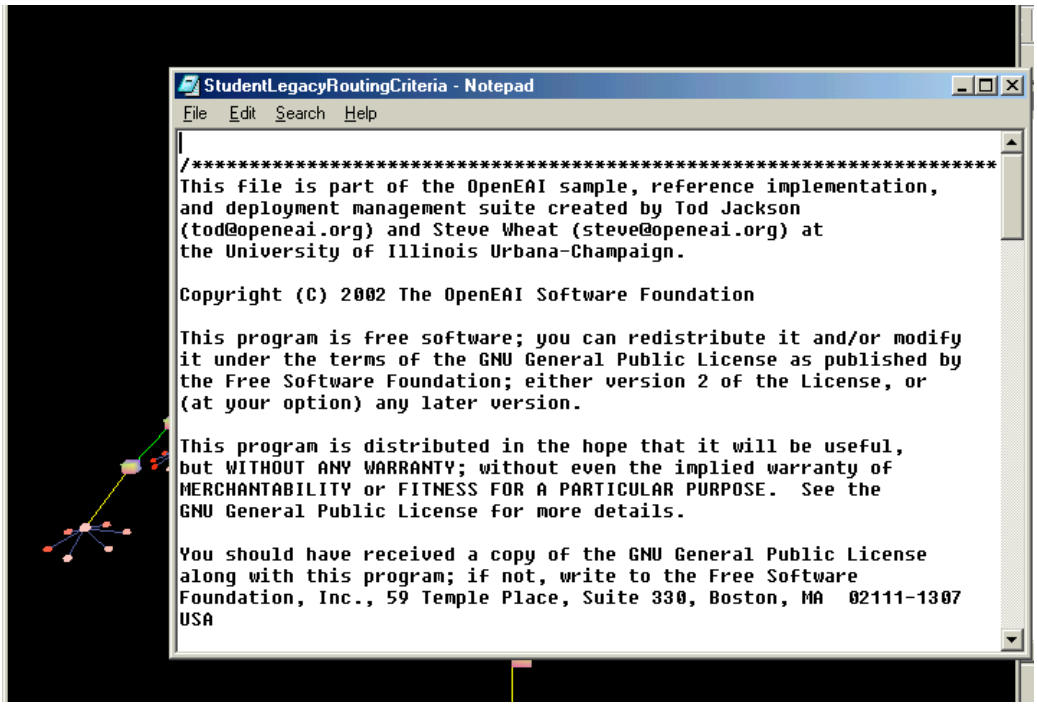


Figure 5.7: Viewing the source code of a version

The information presented for any node type can be modified to include information like the number of lines a particular version consists, and whether that version was the result of adding / modifying / removing a file. The visualization tool collects this information from the history and log files during construction of the cone tree. Also, this information can be represented visually by the height of the version nodes or the color of the version nodes.

5.2 EXAMPLE 2

The second case study is the *org* module from the website: http://dev.w3.org/cvsweb/java/classes/. It is available from the WWW interface to the W3C public CVS tree, available under the W3C Software License. The cone tree layout for the *org* module is shown in Figure 5.8. It can be seen at once that the organization of this module is different from that of the *implementations* module in Section 5.1. Some of the files in this module have been modified regularly, while others have very few versions. It can be speculated that the unchanged files are mostly classes defined once and used/referenced many times by the files that have changed constantly, but this avenue has not been explored.

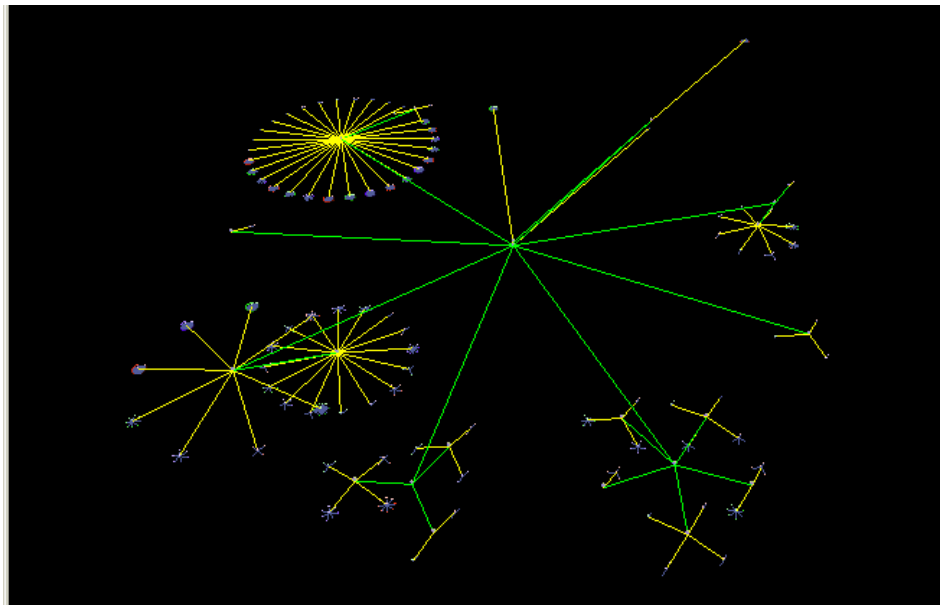


Figure 5.8: The cone tree for the *org* module

Figure 5.9 takes a closer look at the *IsaViz* subtree, by rotation of the cone tree and navigation along the z-axis. This subtree has a large number of files, with some files being more constantly changed than others. The files that are changed constantly are on the inner rim of the cone while the files that have very few versions are seen on the outer rim.

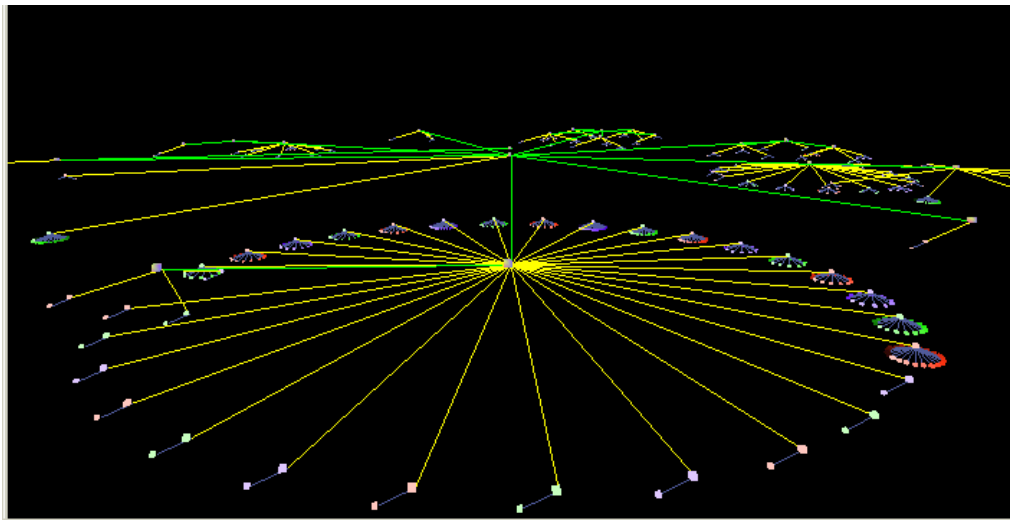


Figure 5.9: The huge *IsaViz* subtree

Figure 5.10 shows a pruned cone tree, with only a few files visible. Again, pruning gives an idea of the overall structure of the module, without any clutter. Subtrees of interest can now be explored with ease.

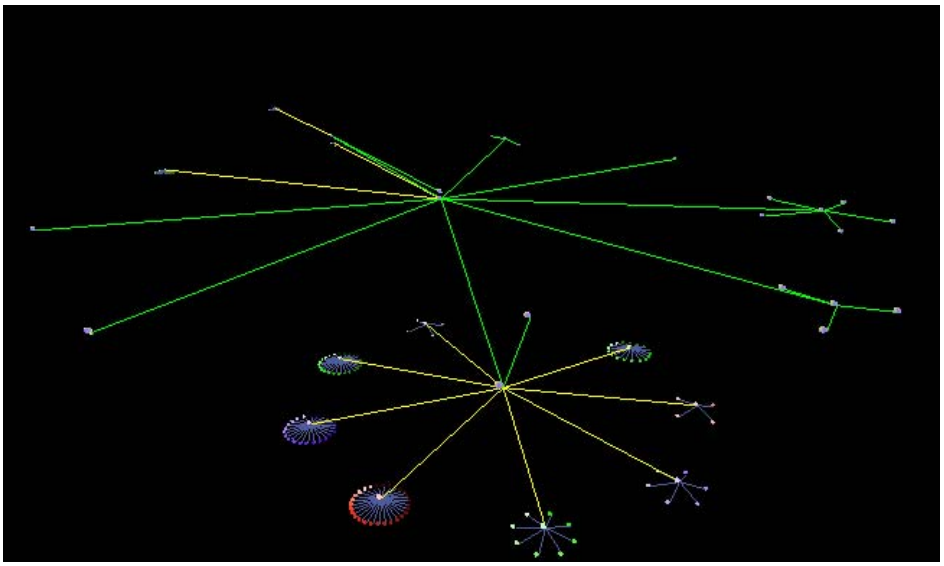


Figure 5.10: The benefits of pruning

Figure 5.11 moves closer to the expanded *w3c/amaya* subtree. Due to the radius of this subtree, all its nodes are not immediately visible, and some rotation and navigation has to be done to view all nodes.

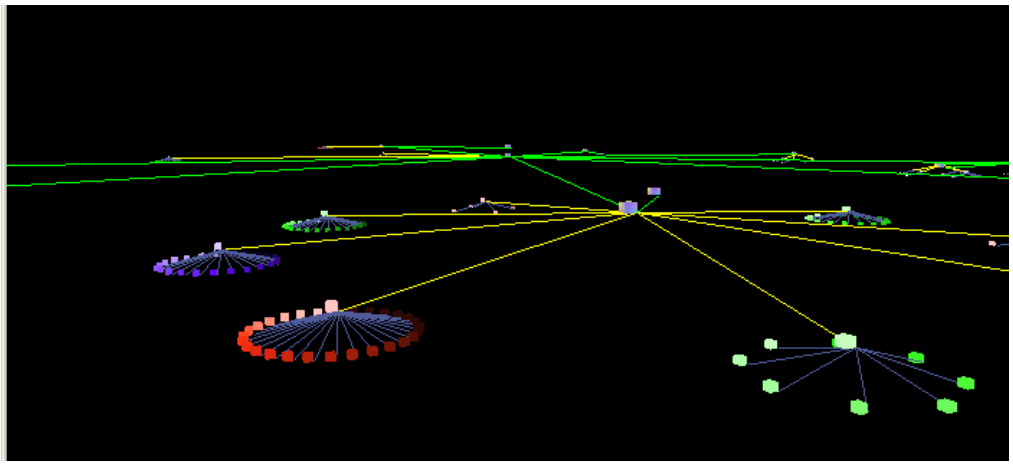


Figure 5.11: Expanding the *w3c/amaya* subtree

Figures 5.12, 5.13 and 5.14 show some examples of user interaction for the *org* module.

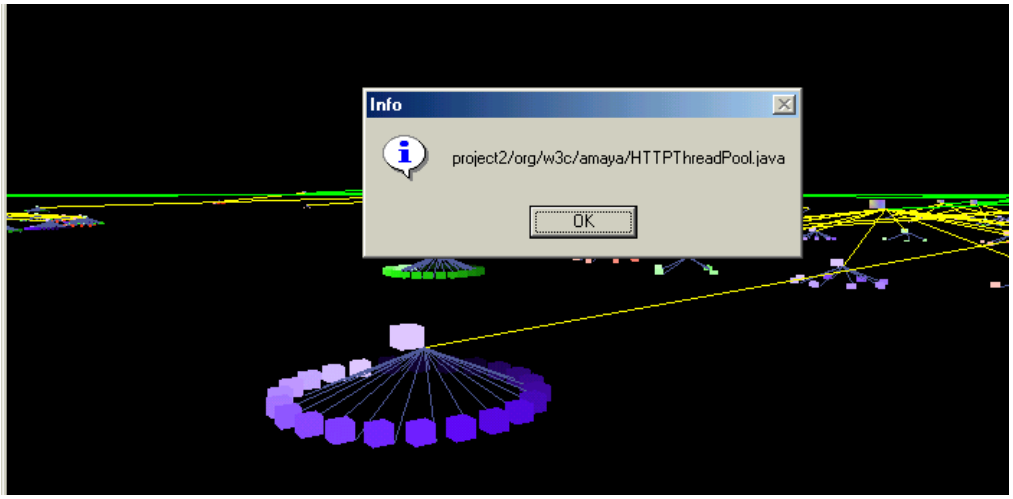


Figure 5.12: User Interaction with rotation and navigation

In Figure 5.13, the green cluster behind the blue cluster in Figure 5.12 is focused upon, and the source code for one of its version nodes is displayed. As an improvement, the selected node can be highlighted, so that it is clear which node has been selected.

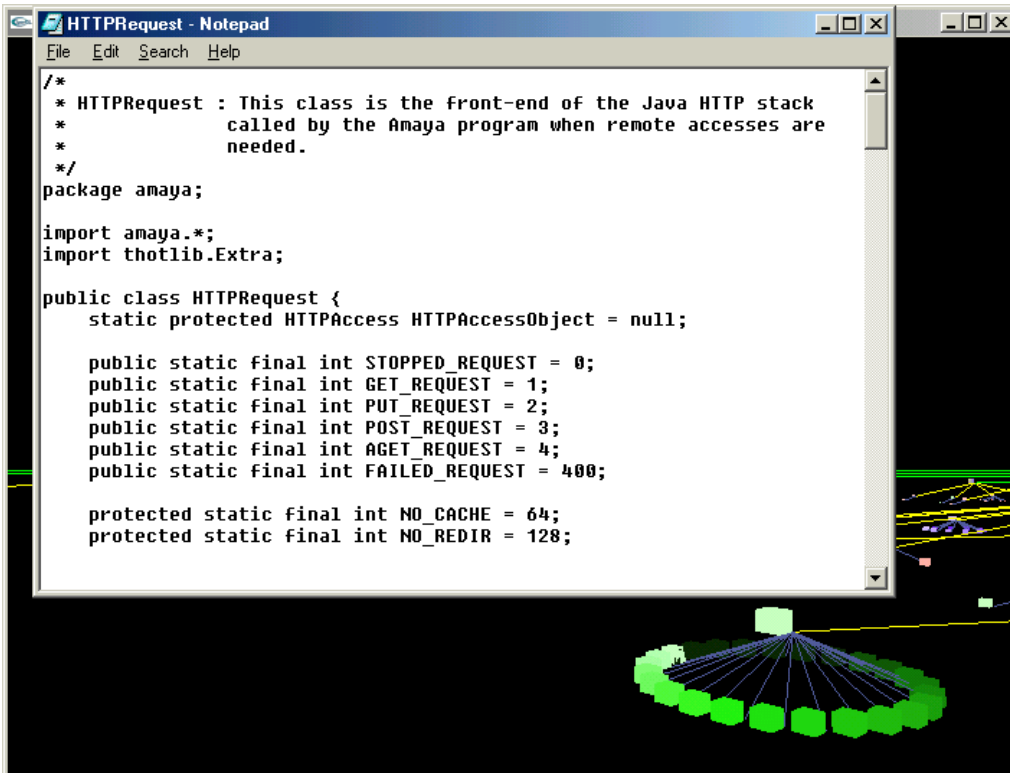


Figure 5.13: Viewing source code

This chapter has thus illustrated the version history of two modules in three dimensions. There are several possible improvements and extensions, which will be discussed in Chapter 6, but it can be seen from the screenshots that three-dimensional views of a system can contain a substantial amount of information in the same space as a two-dimensional visualization. Also, the third dimension allows the user to literally pick up an object and rotate it with six degrees of freedom (along three axes, bi-directionally). Since the visualization is automated, it requires little effort from the user other than specifying the module to be visualized. The power of three-dimensional visualizations can now be applied to other useful information extracted from a version control system, like the number of changes in a particular time period, and the region that changed frequently during a particular time slice. This temporal quantity may be in terms of hours, weeks, days or months. The productivity of developers in a multi-developer environment can also be analyzed, and the number of changes made by a developer to a file can be studied.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

The thesis objectives outlined in Chapter 1 were met by the design and implementation of the visualization tool presented in Chapters 4 and 5. It cannot be denied that the third dimension allows the representation of at least one more attribute than two dimensions, when visualizing a software system. The main objective of this thesis was to demonstrate the usefulness of three-dimensional visualizations for an easy grasp of the structure of a software system and the concentration of changes during its evolution. The cone tree layout and the coloring of the version nodes achieved this. Exploring the software structure was also made easier by providing user interaction to rotate the cone tree, navigate around the three-dimensional world, and pruning/expansion of the tree.

The visualization shows the number of versions of each and every file in the system, and also lists the date of creation of the version, but it does not make obvious the order in which changes were made to the system. It is not immediately apparent whether version a of file 1 was created before version a of file 2. It might be useful to study this temporal aspect, as information about files that change in the same time slice, or files that change together, can provide some idea of the dependencies that exist between them. In the same vein, it would also be useful to visualize the code churn for any particular period of time, and the developers responsible for the maximum changes. This can be studied by analyzing the number of lines of source code in a file (excluding blank lines and comments) that were affected or created or removed by a change. The CVS logs can be used to extract values for most of these attributes.

Other improvements can be made in the design of the user interface. The person requesting the visualization should be able to control what he/she wants to see. A choice of metaphors can be provided, along with options to set the color and texture of each metaphor. Ideally, the user should select the attributes to be visualized, and the necessary information can then be extracted from the version control system. The visualization tool should hide the operation of the version control system and provide the end-user with useful information from the version control system. Three-dimensional views can thus be used to present multiple attributes of a software system to any user requesting information about the software system.

REFERENCES

[1] Price, B. A., Baecker, R. M., and Small, I. S. A Principled Taxonomy of Software Visualization. Journal of Visual Languages and Computing, vol. 4, no. 2, 1993, 211-266.

[2] Knight, C, and Munro, M. Comprehension with[in] Virtual Environment Visualisations. Proceedings of the IEEE 7th International Workshop on Program Comprehension, Pittsburgh, PA, May 5-7, 1999.

[3]Knight, C, and Munro, M. Visualising Software -- A Key Research Area. University of Durham, Computer Science Technical Report 5/99. http://www.dur.ac.uk/~dcs3crk/workfiles/documents/te ch-report-5-99.ps.gz

[4] Ball, T.A., Eick, S.G. Software visualization in the large. IEEE Computer, 29(4):33-43, April 1996

[5] Storey, M.-A.D., Fracchia, F.D., Muller, H.A. Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization. Proceedings of the Fifth International Workshop on Program Comprehension, Dearborn, MI, May 28-30, 1997, 17-28.

[6] Stasko, J., Domingue, J., Brown, M.H., Price, B.A. (eds.) Software Visualization, The MIT Press, 1998, ISBN: 0-262-19395-7.

[7] Diehl, S.(ed): Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures, ISBN: 3-540-43323-6.

[8] Jerding, D.F., Stasko, J.T. Using Visualization to Foster Object-Oriented Program Understanding. Georgia Institute of Technology Technical Report GIT-GVU-94-33, July 1994. file://ftp.cc.gatech.edu/pub/gvu/tech-reports/94-33.ps.gz

[9] Roman, G.-C. & Cox, K. C. A taxonomy of program visualization systems. IEEE Computer pp. 11—24, 1993

[10] G.-C. Roman and K. C. Cox. Program Visualization: The Art of Mapping Programs to Pictures. Proceedings of the 14th International Conference on Software Engineering, May 1992.

[11] Stasko, J., and Patterson, C. Understanding and characterizing software visualization systems. Proceedings of IEEE Workshop on Visual Languages: pp 3-10, 1992.

[12] Eick, S. G., Steffen, J. L., and Sumner, E. E. SeeSoft -- a tool for visualizing line oriented software statistics. IEEE Transactions in Software Engineering, 18(11):957-68, 1992.

[13] M. Storey, K. Wong, F. D. Fracchia, and H. A. Müller. On Integrating Visualization Techniques for Effective Software Exploration. Proceedings of IEEE Symposium on Information Visualization, October 20-21, 1997.

[14] Marla J. Baker, Stephen G. Eick. Visualizing Software Systems. ICSE, pages 59-67, 1994.

[15] Jerding, D.F., Stasko, J.T. Visualizing Message Patterns in Object-Oriented Program Executions. Georgia Institute of Technology Technical Report GIT-GVU-96-15, May 1996. file://ftp.cc.gatech.edu/pub/gvu/tech-reports/96-15.ps.gz.

[16] V Haarslev and R. Moeller. A framework for visualizing object-oriented systems. Proceedings of ECOOP/OOPSLA, pages 237--244, October 1990.

[17] Stasko, John T. and Wehrli, Joseph F. Three-Dimensional Computation Visualization. Proceedings of the IEEE Symposium on Visual Languages, Bergen, Norway, pages 100-107, August 1993.

[18] C. Ware, D. Hui, G. Franck. Visualizing Object Oriented Software in Three Dimensions. CASCON (IBM Centre for Advanced Studies), Conference Proceedings, pages 612-620, Toronto, Canada, October 1993.

[19] Loe Feijs and Roel De Jong. 3D visualization of software architectures. Communications of the ACM, 41(12):73-78, December 1998.

[20] G. Parker, G. Franck, C. Ware. Visualization of Large Nested Graphs in 3D: Navigation and Interaction. J. Visual Languages and Computing, 9(3):299-317, 1998.

[21] Claus Lewerentz, Frank Simon. Metrics-Based 3D Visualization of Large Object-Oriented Programs. First International Workshop on Visualizing Software for Understanding and Analysis, June 26 - 26, 2002.

[22] Tim Dwyer. Three Dimensional UML Using Force Directed Layout. Australian Symposium on Information Visualisation, Sydney, Australia, December 2001.

[23] Chuah M.C. and Eick S.G. Glyphs for Software Visualization. International Workshop on Program Comprehension, pages 183-191, May 1997.

[24] Jonathan I. Maletic, Andrian Marcus and Michael L. Collard. A Task Oriented View of Software Visualization. First International Workshop on Visualizing Software for Understanding and Analysis, June 2002.

[25] Hideki Koike, Hui-Chu Chu. VRCS: Integrating Version Control and Module Management uisng Interactive Three-Dimensional Graphics, Proceedings of 1997 IEEE Symposium on Visual Languages (VL'97), pp.170-175, 1997.

[26] P. Young and M. Munro. Visualising Software in Virtual Reality. Proceedings of the IEEE 6th International Workshop on Program Comprehension, pages 19-26, June 24-26, 1998.

[27] Jonathan I. Maletic, Jason Leigh, Andrian Marcus and Greg Dunlap. Visualizing Object-Oriented Software in Virtual Reality. Ninth International Workshop on Program Comprehension (IWPC'01), Toronto, Canada, May 12 - 13, 2001.

[28] Stephen G. Eick, Todd L. Graves, Alan F. Karr, Audris Mockus and Paul Schuster. Visualizing Software Changes. Journal of Software Engineering, 28(4):396-412, 2002.

[29] Lakoff , G., Johnson. M. Metaphors we live by, University of Chicago Press, July 1982, ISBN: 0-226-46801-1.

[30] Vladimir L. Averbukh. Toward Formal Definition of Conception "Adequacy in Visualization". IEEE Symposium on Visual Languages, Italy, 1997.

[31] J. Mackinlay. Automating the design of graphical presentations of relational information. ACM Transactions on Graphics, 5, pages 110—141, 1986.

[32] Grundy, J.C. and Hosking, J.G. High-level Static and Dynamic Visualisation of Software Architectures, accepted to 2000 IEEE Symposium on Visual Languages, Seattle, Washington, Sept. 14-18 2000.

[33] Mulholland, P. Using a Fine-Grained Comparative Evaluation Technique to Understand and Design Software Visualization Tools. Empirical Studies of Programmers: Seventh Workshop. New York: ACM Press, 1997.

[34] Sarita Bassil, Rudolf K. Keller. Software Visualization Tools: Survey and Analysis. Ninth International Workshop on Program Comprehension (IWPC'01), Toronto, Canada, May 12-13, 2001.

[35] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM99), 1999.

[36] A. Mockus, S. G. Eick, T. L. Graves, and A. F. Karr. On measurement and analysis of software changes. Tech. Rep., National Institute of Statistical Sciences, 1999.

[37] H. Gall, M. Jazayeri, R. G. Klsch, G. Trausmuth. Software Evolution Observations Based on Product Release History. Proceedings of the Conference on Software Maintenance, pages 160-166, 1997.

[38] H. Gall, K. Hajek and M. Jazayeri. Detection of Logical Coupling Based on Product Release History. Proceedings of the International Conference on Software Maintenance, ICSM98, November 1998, Washington D.C.

[39] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk... In ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering, May 1997.

[40] Michele Lanza, Stephane Ducasse. LMO 2002 Proceedings (Languages et Modeles a Objets), pp. 135 - 149, Hermes Publications, 2002.

[41] Babich, Wayne A., Software Configuration Management – Coordination for Team Productivity, ISBN: 0-201-10161-0, Addison-Wesley Publishing Company, 1986.

[42] W. Tichy, RCS: a system for version control, Software-Practice & Experience, 15(7):637--654, July 1985.

[43] Concurrent Versions System, http://www.cvshome.org/.

[44] M. J. Rochkind, The Source Code Control System, IEEE Transactions on Software Engineering, SE-1, 4, Dec. 1975, pp. 364-370.

[45] OpenGL – High Performance 2D/3D Graphics, http://www.opengl.org.

[46] Collberg, Kobourov, Nagra, Pitts, Wampler, A System for graph-based visualization of the evolution of software, Proceedings of the 2003 ACM Symposium on Software Visualization, pages 77-ff, 2003.

[47] FreeBSD CVSWeb Project, http://www.freebsd.org/projects/cvsweb.html.

[48] Unix – Frequently Asked Questions (7/7), http://www.faqs.org/faqs/unix-faq/faq/part7/.

[49] A set of GUI front-ends for CVS written in C++ and distributed under the GNU General Public License (GPL), http://www.wincvs.org.

[50] G. Robertson, J. Mackinlay, and S. Card, Cone Trees: Animated 3D visualizations of hierarchical information, Proceedings of ACM SIGCHI conference on Human Factors in Computing Systems, pages 189 - 194, 1991.

[51] Tanaka, Okada, Niijima, Treecube: Visualization Tool for Browsing 3D Multimedia Data, Seventh International Conference on Information Visualization, p 427, 2003.

[52] Lamping and R. Rao, Laying out and Visualizing Large Trees Using a Hyperbolic Space, Proceedings of UIST'94, November 1994, pages 13-14.

[53] E. Chi, J. Mackinlay, P. Pirolli, R. Gossweiler, and S. Card, Visualizing the evolution of web ecologies, Proceedings of ACM Conference on Human Factors in Computing Systems (CHI), 1998.

[54] Carriere J.,and Kazman R, Interacting with Huge Hierarchies: Beyond Cone Trees, Proceedings of IEEE Symposium on Information Visualization (Atlanta, Georgia, 30-31 October 1995), IEEE Computer Society Press, pages 74-78.