Space-Filling Software Visualization

Marla J. Baker and Stephen G. Eick*
AT&T Bell Laboratories

3 August 1994

Abstract

SeeSys™ is a system embodying a technique for visualizing statistics associated with code that is divided hierarchically into subsystems, directories, and files. This technique can display the relative sizes of the components in the system, the relative stability of the components, the location of new functionality, and the location of error-prone code with many bug fixes. Using animation, it can display the historical evolution of the code. Applying this technique, the source code from a multi-million line production software product is visualized.

To appear in *Journal of Visual Languages and Computing.*

*Technical Contact: AT&T Bell Laboratories - Rm 1G-351, 1000 East Warrenville Road, Naperville, IL 60566, email: eick@research.att.com

1 Introduction

The software industry produces the most complicated data-driven systems ever created. A single software system may contain millions of lines written by thousands of programmers. Now, as larger and larger systems are built, the problems of understanding their increasing complexity threaten to set a limit on system size.

One solution employs data visualizations [Tuf83] [Tuf90] to help software engineers understand the code they are working on. Making data or graphical representations of software, known as software visualization, is well-known and has produced useful tools for developing code [Ebe92]. Previous work in software visualization had its roots in academia and has focused on data structure and algorithm visualization. Notable examples include Baecker's seminal work on sorting [BM88], Brown's work on algorithm animation [BH92], North et al.'s displays of software graphs [GKNV93], Price et al.'s taxonomy of software visualization [PSB93], Stasko's research in visualizing parallel computation [KS93], and Roman and Cox's visualizations of specifications [RCWP92].

Unfortunately, current graphical techniques are ineffective for large software systems. When applied to production-sized systems, routines for producing flow charts, function call graphs, and structure diagrams [You89] often break because the display is too complicated. Or they produce displays that contain too much information and are completely illegible. For many existing systems, even system structure diagrams are impossible to create because the original abstractions have been forgotten or are inapplicable after years of software modifications. And for large software projects, management issues (involving the human aspects of the project rather than the execution of the code), are often more challenging than the technical issues. For example, project managers need to know where new development activity is occurring and which modules are error prone so they can assign them to their best engineers.

To deal with these problems we have developed a space-filling technique for displaying source code related software statistics. We have focused on visualizing program source code in files, directories, and subsystems. In previous work, based on a reduced representation, Eick et al. [ESS92] and [Eic94] presented a method for visualizing statistics associated with lines of code in files. The technique, implemented in a system called *SeeSoft*™, fits up to

50,000 lines of code on a high-resolution (1280x1024) display using a reduced representation. In Seesoft each line of code is displayed as a color-coded row within columns corresponding to the files, with the row indentation and length tracking the actual code. Therefore, to visualize larger systems using that technique, the code must be divided into 50,000 line chunks. The presented work generalizes the previous technique from displaying lines within files to files in directories within subsystems. This generalization is based on Johnson and Shneiderman's fundamental work using treemaps to show hierarchical data [JS91].

Our motivation for studying source code visualization came from the source code database for a large communications software system at AT&T. This software system is continually being modified for customers world-wide. Several million lines of code are organized hierarchically into tens of subsystems, several thousand directories, and hundreds of thousands of files. Software releases occur about once a year, each built on the previous release by making changes such as adding new functionality or hardware support. The database for each software release contains detailed information on all files, directories, and subsystems, including

- non-commentary source lines (NCSL),
- software complexity metrics,
- number and scope of modifications (e.g. which files are affected),
- number of programmers making modifications, and
- number and type of bugs (a subcategory of modifications),

and other statistics. This information raises five questions for project managers:

1. Which subsystems are the largest? Where is the new development activity?

2. Where are the large directories? Are the large and small directories distributed evenly throughout the subsystems? Is the new development distributed evenly between directories? Which directories are stable and which have the most activity?

3. Which subsystems are unusually complex or error-prone? Are there any problematic directories? How is the development activity apportioned between fixing bugs and adding new functionality?

4. Are bug fixes a reoccurring problem? What components would make good candidates for code restructuring projects?

5. What were the dates of the major releases? Have any subsystems shrunk or disappeared? What is the rate for growth of different subsystems?

   Which subsystems are growing at the fastest rates? Have any subsystems remained at the same size or grown at a steady rate? Where has the development work been done historically?

Statistical methods for analyzing the distributions of subsystem, directory, and file statistics are well-known [CM88]. For software engineering, however, it is equally important to understand the context in which the statistics relate to the code. By itself, the bug rate for a directory may not look like a sign of trouble. Only when we can see most of the other directories do we realize that something may be wrong in the subsystem. Similarly, scatterplots are useful for identifying unusual files and directories. But using this information is difficult. The unusual directories must be related to the subsystem to which they belong, and for a system with hundreds of thousands of files, even relating only the worst 1% of them to their respective directories involves correlating thousands of files.

Another problem with methods of statistical analysis to study software involves incorporating domain knowledge. Experienced software engineers often have detailed knowledge and explanations that must be included in the analysis to properly interpret the data. For example, one-time events with known explanations may not indicate general trends.

To address these problems, we have developed a graphical technique that visualizes subsystem, directory, and file statistics, but does so in context. Our method, implemented in a system called *SeeSys*™, preserves the hierarchical relationships in the code, making it easy to relate the statistics to the components. SeeSys accomplishes all this by means of space-filling representations. On a single computer monitor, it can easily show directory level statistics for a system with thousands of directories.

The approach we took in SeeSys is discussed below in Section 2, applications of SeeSys in Section 3, and factors that make SeeSys a powerful visualization tool in Section 4.

2 Approach

The visualization technique used in SeeSys is based on the idea that a software system can be decomposed into its individual components. For example, the NCSL for a directory is the sum of the NCSL's for the individual files in the directory, and the NCSL for a subsystem is the sum of the NCSL's for its individual directories.

Figure 1 shows the visual technique used in SeeSys. The left pane represents the entire software system, and *X*, *Y*, and *Z* represent its three subsystems. The area of each subsystem is based on some subsystem statistic, such as NCSL. The subsystems are each partitioned vertically to show their internal directories. In the left pane of Figure 1, the rectangles labeled *1*, *2*, *3*, *4*, and *5* represent the directories in subsystem *X*. Each vertical rectangle's area is proportional to the directory's NCSL, and so the sum of the areas over the directories equals the area of the subsystem.

This technique allows for a straight-forward visual comparison of directories within a subsystem because area of each visual component is always proportional to the statistic for the corresponding software component.

The middle pane in Figure 1 shows additional information by vertically filling each rectangle. The fill might indicate the NCSL which correspond to newly developed code. The fill provides a visually pleasing display of a second statistic, and a convenient method for cross directory comparisons. It is clear from the middle pane in Figure 1 that directory *5* has had the highest percentage of new code added to subsystem *X*, although not necessarily the the largest amount of new NCSL. In fact, directory *1* has the most new lines of code because it has the largest area.

The right pane in Figure 1 shows a zoom view of the *Y* subsystem from the middle pane in Figure 1. Each directory now has been partitioned horizontally to represent its internal files. Again, as with the directories, the rectangle sizes are proportional to the file-level statistics. The leftmost directory contains files *file1*, *file2*, *file3*, *file4*, and *file5*, and the average fill for the files reflects the fill for the directory level. As in the previous example,

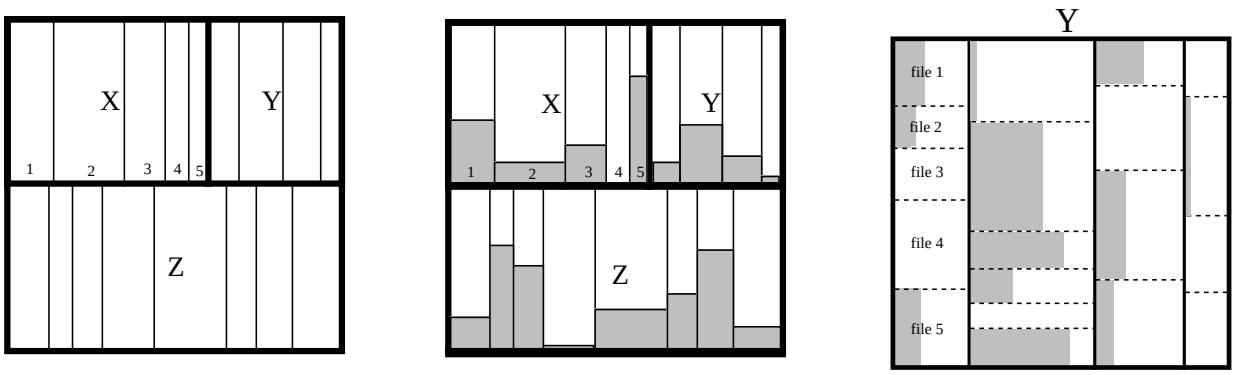

Figure 1: Left pane: subsystem and directory statistics. Middle pane: a fill statistic for directories. Right pane: a zoomed view on subsystem *Y* showing file level statistics.

at the directory level, the fill might display newly developed code in a recent release.

This technique is effective for showing software statistics that accumulate from files to directories to subsystems. These include NCSL, number of bugs, complexity metrics, fix-on-fix rates, and so on. The hierarchical decomposition immediately relates the files to their directories and the directories to their subsystems, making cross unit comparisons easy. The fill provides a visual representation of percentages, enabling quick identification of outlyers, such as directory *5* in Figure 1. The fill, for example, could be the NCSL fixing bugs, or the number of comments in the code, or in general, a percentage associated with each code unit. We also use color and interaction to overlay additional information onto the display.

3 Applications

The following illustrates the effectiveness of SeeSys by showing how it can be used to answer the questions listed in Section 1 for the software system.

3.1 Subsystem Information

"Which subsystems are the largest?" "Where is the new development activity?" In a SeeSys display that shows NCSL for subsystems (see Figure 2), the fill shows new NCSL added in a recent release. The rectangle forming the outmost boundary represents the size of the entire system in NCSL. Each subsystem is labeled with a code for its name. The rectangles contained within the boundary represent the size of individual subsystems in NCSL. Color is used here to redundantly encode size according to the color scheme in the slider at the bottom of the screen. Two medium sized subsystems, *A* and *B*, appear in the upper left-hand corner. Note that subsystems are shown in alphabetical order so they can be located easily.

What does the figure show? It is immediately clear that the three largest subsystems are *D*, *d*, and *n*, because their rectangles are visually the most prominent and their colors are towards the brighter end of the spectrum. The fill (light gray rectangles at the bottom of each subsystem rectangle) shows that *t*, *k* and *E* had the largest percentage increases in code size during the release. *t* is a new subsystem that was created in release 14. The number of NCSL [1] for *t* for recent releases is shown in the lower left-hand corner, *74,279* in release 15, *17,131* in release 14, and *0* in all previous releases. The percentage increase, *74%*, is shown in the color scale.

3.2 Directory Information

"Where are the large directories?" "Are the large and small directories distributed evenly throughout the subsystem?" "Is the new development distributed evenly between directories?" "Which directories are stable and which have the most activity?" Figure 3 shows directory statistics added to the display. The area of the subsystem rectangles is the same as in Figure 2, but each subsystem is partitioned vertically to display these internal directories. The area of each directory rectangle is proportional to its size in NCSL. Again, the fill is proportional to new development while the color redundantly encodes the size of the directory.

The largest directories are highlighted according to a color spectrum at the bottom of the display: pink for the biggest, through yellow, green, and finally blue for the smallest. We can see that the two largest directories are

[1] Data values, names, and release numbers have been altered.

in subsystems *e* and *Z*, and that subsystems *n* and *D* have no large directories since all of their directories show colors towards the blue end of the spectrum.

We refer to Figure 3 as the "software" skyline. It shows the sizes of the subsystems and directories, the new development, and even which directories are new or unchanged. The fill shows the NCSL added in release 15. For new directories the entire directory space is filled in gray; for directories with 50% new code, the gray area fills half of the directory space. In *t*, for example, we can see that almost all the directories are being actively developed, but that in *X* there has been almost no new development.

By zooming, we can obtain more information about any particular subsystem. In Figure 4, a zoom view of subsystem *t* makes the directory divisions more clear. The darker gray fill in each directory represents the new development. Thus we can see only one directory, *t7*, has been stable (1% new code), while all of the others have had significant activity during this software release.

3.3 Error-Prone Code

"Which subsystems and directories have the most bugs?" "Are there any problematic directories?" "How is the development activity apportioned between fixing bugs and adding new functionality?" In Figure 5, the size of each subsystem and directory is the new NCSL and the fill is the new NCSL added to fix bugs (directory color is uniform, no longer encoding any information). The largest subsystems are *k*, *d*, *Z*, and *t*. These subsystems have had the most new development. The light gray rectangle within each subsystem (obscured by the directory fill spikes) represents the bug fixing NCSL for the subsystem, while the directory spikes represent directory bug fixing NCSL detail. The *t* subsystem now takes up a larger portion of the display because it has had so much new development. The small light gray rectangle for *t* shows that almost all of the new development has added new functionality. Only a small percentage of the new NCSL fixed bugs in most of the *t* directories.

In contrast, subsystem *g* has a very high bug rate. The light gray subsystem rectangle shows that about 50% of the development activity in *g* has been fixing bugs. In several of the directories in *g*, the blue directory spikes that extend to the top of the rectangle indicate that all of the new NCSL involved bug fixes.

3.4 Recurring Problems

"Are bug fixes a recurring problem?" "What components would make good candidates for code restructuring projects?" A *fix-on-fix* bug is a software bug correction that modifies an earlier bug fix. A high fix-on-fix rate may indicate unstable or unmaintainable code. In Figure 6, the area representing each subsystem and directory is proportional to the number of bugs, while the fill area is proportional to the number of fix-on-fix bugs in each subsystem and directory. Again, the light gray rectangle represents the subsystem average while the spikes across the subsystems represent directory level detail. Subsystems i and K have the highest fix-on-fix rates. Some directories in these subsystems have relatively high fix-on-fix rates, making them good candidates for code restructuring projects.

3.5 System Evolution

"Which were the major software releases?" "Have any subsystems shrunk or disappeared?" "What is the rate of growth for different subsystems?" "Which subsystems are growing at the fastest rates?" "Have any subsystems remained at the same size or grown at a steady rate?" "Where has the development work been done historically?" To address these questions, SeeSys animates the display over the historical evolution of the code.

Although it is impossible to get the full effect of the animation from a static display, some ideas can be conveyed. Figures 7, 8 and 9 shows three frames from an animation. The animation depicts the NCSL growth through various software releases of the software. The bounding rectangle for each subsystem represents the maximum size of the subsystem across all software releases in the dataset, while the filled portion of the display represents the size of the system in the software release, as seen in the frame slider on the left of the screen. The frame slider controls which frame (or software release) is displayed. It will be described in more detail later. Color is again redundantly encoding the size of each subsystem, changing in relation to the size of the subsystem for each software release in the dataset. Figure 7 shows the earliest software release in the dataset. Note that subsystem t was not yet in existence while other subsystems, such as d and W, had almost reached their maximum sizes. Figure 8 shows a frame from the middle of the dataset, and Figure 9 shows the most recent software release in the dataset. At the

end of the animation, most of the subsystems have reached their maximum size. A few subsystems (e.g., W and f) have decreased in size and others have disappeared completely (e.g., O).

From watching the animation we have discovered the following about changes in the size of subsystems:

- several of the subsystems are stable;
- subsystem O grew and shrank before it finally disappeared;
- subsystems F and J have shrunk;
- subsystem D grew slowly at first but fast recently;
- subsystems t, k, and Z are currently growing fastest and at similar rates.

4 SeeSys Visualization System

SeeSys is a visualization system for displaying software metrics. There are two requirements that must hold for a metric to be displayable. It must be a quantitative measure, and it must be additive (the sum of the measure for the parts must equal the measure for the whole). In the examples above, the software metrics used were NCSL, newly developed NCSL, number of bugs, and number of fix-on-fix bugs. But SeeSys lends itself to the use of other software metrics as well. For example, the number of modifications is another valid measure. In that case, the area of the rectangles would be proportional to the number of modifications in each component, and the bounding rectangle would represent the total number for the whole software system.

The SeeSys visualization system may be extended to display software complexity metrics such as Halstead's program volume measure [Hal77] or McCabe's cyclomatic complexity [McC76]. The color or texture of each rectangle could encode the complexity of the corresponding subunit, with red indicating the most complex units and blue indicating the least complex units.

4.1 User Interaction

User interaction via a mouse is a powerful means of retrieving information visually. SeeSys tracks mouse movements over the display area, conveying additional information about whichever component the mouse is touching at that time. This is called the *active component* and is indicated by a red highlighted boundary. The active component may be either a subsystem or a directory and a subsystem. In Figure 3, subsystem *I* and directory *I17* are the active components. The directory's name and current statistic appear above the slider at the bottom of the screen, with a vertical bar marking its place in the slider according to the current statistic. Components can also be activated by placing the mouse in this slider.

The available statistics are shown on the lower left side of the screen. Clicking the mouse on one of these statistics causes the display to be redrawn with that statistic. On the display in Figure 3 the user may call up information on the NCSL, on newly developed NCSL, and on software faults or bugs.

There are also five buttons (*No Mod Lines*, *No Mod Names*, *Colors*, *Fill*, and *Zoom*) that appear on the upper left corner of the screen. The first button turns directory bounding lines *on* and *off* (turned *on* in Figure 4). The second button toggles the directory colors (turned *off* in Figure 5 and Figure 6). The third button turns on and off the fill for the directory level. The fourth button activates a zoom feature which allows the user to zoom in on a particular subsystem (see Figure 4). The user clicks on the zoom button and then on one of the subsystems to bring up a zoom window. Zoom windows and the main display are linked so that operations on either one will affect the other. The buttons can make the display simpler or more customized to user preferences.

The slider labeled *ROWS* is used to control the number of rows in the display. One may click at any point in the slider to change to the corresponding number of rows or drag through the slider to find the optimal layout.

Figures 7, 8, and 9 have two additional sliders, used to control the animation, on the left side of the screen. The *speed slider* controls the animation speed (the delay in milliseconds between frames) when the animation is active, and the *frame slider* controls which frame is displayed. Clicking the mouse anywhere in the frame slider causes the display to change to the corresponding frame and, as a result, the display space is redrawn. The *play*

button activates the animation and may be used interactively to stop at places of interest. Dragging the mouse through the slider shows the animation at the drag speed.

An effective interactive technique during animation is to put the mouse on one subsystem and watch the active bar move in the slider below. Each bar represents a subsystem, and is placed in the slider according to that subsystem's value. Since the bar moves across the slider during the animation, a user can see the active subsystem's evolution both in the color scale and in relative size, providing two different perspectives. In Figures 7, 8, and 9 the active subsystem, K, can be viewed in two ways throughout the animation. In the display area, one can see that K is about 50% of its maximum size in the first software release of the dataset, 75% its maximum size in the middle release, and finally, K reaches its maximum size in the last release. Looking at the color slider at the bottom of the screen offers another perspective. Here K begins towards the lower end of the spectrum (at 203,902 NCSL), along with most of the other subsystems. It then moves to the middle of the spectrum, with an increase to 270,653 NCSL. Finally, in the last software release, K moves towards the top end of the spectrum (356,070 NCSL), distinguishing itself as a relatively large subsystem. Although it is difficult to see in only three frames from the animation, K grows at a relatively constant rate.

4.2 Display Principles

The visualization approach used in SeeSys is based on three principles:

1. The individual components can be assembled to form the whole. This allows the user to easily see relationships between them.

2. Pairs of components can be compared to understand how they differ. In Figure 2, it is obvious that subsystem D is much larger than subsystem t.

3. The components can be disassembled into smaller components. In Figure 3, the vertical bars clearly show the division of the subsystems into directories. This important feature of the components allows the structure of the display to reflect the structure of the software.

4.3 Screen Real-Estate

In visualizing large software systems, it is important to utilize screen real-estate efficiently. Objects placed on the screen must be large enough to convey information, but small enough to allow room for many other objects. SeeSys does this by placing rectangles next to one another so that 100% of the display area is utilized.

When using area, components with large statistics become visually dominant, making the technique effective when the components representing the larger directories are the most interesting while de-emphasizing the small components. For example, in Figure 3 it is difficult to see the small directories. SeeSys overcomes this difficulty with a zoom feature that allows the user to zoom in on any subsystem (see Figure 4).

4.4 Spatial Relationships

It is important that the screen real-estate technique used lends itself to the rapid transfer of information. Simply packing data onto the screen is not necessarily helpful. The viewer must be able to process the information rapidly for the visualization technique to be useful. SeeSys speeds the transfer of information by taking advantage of the human ability to recognize spatial relationships. People naturally make inferences from the configuration of components and can relate each component to the whole.

It is easier to see relationships between components if the heights of the rectangles are approximately equal. We use an algorithm to equalize the heights while retaining the easy comprehensibility of the display (for example, by preserving the alphabetical ordering). If the heights were exactly even, it would be necessary to split subsystems between rows, leave ragged "margins," or have spaces of variable width between the subsystems. In Figure 2, subsystem D might have to be partially split between the first and second row. This would undermine the basis of the approach because all individual parts would not always represent one component in the system, making it harder to see relationships between individual components. To deal with this problem, we introduced a slider that allows the user to control the number of rows in the display. By changing the aspect ratio, the user can find the optimal display for gaining insight into a particular characteristic. In Figure 10, the number of rows has been reduced to just one so the user can

easily compare the percentage of new development across the subsystems.

Although we have chosen to order components alphabetically to facilitate rapid location of specific components, one can foresee other useful ordering techniques such as by area, complexity metric, percentage changes, or other statistic values. Future version of SeeSys will include controls to reorder the components alphabetically, by area, and according to a specified statistic.

4.5 Color

Color[2] is another visualization technique used to convey yet more information. In the previous examples, color was used to redundantly encode size, with the brighter colors denoting larger components. We use a perceptually uniform color spectrum [LH92] [LHMR92] to encode information, but other spectrums with fewer, greater, or differing colors could be used to make distinctions between components. Color can also be used to encode other information, such as age, complexity, activity (code churn), or number of programmers.

4.6 Implementation

SeeSys is implemented in C++ using AT&T Bell Laboratories' Vz graphics library™[Gro94]. Vz is a cross-platform class library forming a computational infrastructure for producing innovative, novel, production-quality visualizations of large datasets. Vz uses C++'s object-oriented capabilities, factors out common code, embodies direct manipulation, and data abstractions in a selective manner for building highly-interactive visualizations of large datasets based on linked views.

SeeSys consists of four linked views of data: the colorful space-filling display, the leftspace containing controls, buttons, and sliders, the bottom space containing the color scale and statistics, and the zoom view showing details of a particular subsystem. Each mouse-sensitive view is implemented as Vz class, is tied together though Vz's linking, and united through the use of a perceptually uniform color scale. Each view is highly-interactive and continuously responds to mouse and keyboard input in any of the views. On

[2]In the black and white version of this paper color is mapped to gray level.

a Silicon Graphics Iris workstation screen updates occur in real-time and appear continuous.

In total SeeSys is about 3,000 lines of delicate C++ code that runs on workstations supporting X11 and Motif or Silicon Graphics's GL graphics language [Boa92].

5 Summary

SeeSys is a system implementing a graphics technique for displaying large volumes of information about software. The SeeSys visualization technique can display code for any existing software system that is organized hierarchically into subsystems, directories, and files by representing the whole system as a rectangle and recursively representing the various subunits with interior rectangles. The area of each rectangle is proportional to a statistic (for example, NCSL) associated with its subunit. SeeSys then uses color, zooming, a fill statistic, and interactive techniques to layer additional information onto the base display.

Applying SeeSys to display the source code from the system we can

- show the sizes of the subsystems and directories and where the recent development activity has been;
- zoom in on particularly active subsystems;
- discover how much of the development activity involves bug fixes and new functionality;
- identify directories and subsystems with high fix-on-fix rates;
- locate the historically active subsystems and find subsystems that have shrunk and even disappeared.

The SeeSys visualization technique can display source code statistics for an entire software system in terms of the code structure, complexity, and evolution. Three principles should be adhered to when creating visualizations of large software systems:

1. The structure of the display should reflect the structure of the software.

2. The individual components should be visually comparable and decomposable.

3. Animation can be used to depict the evolution of the software.

Potential users of SeeSys include Project Managers, Feature Engineers, and Software Developers. Project Managers can use SeeSys to help track the evolution of the code and understand how each subsystem, directory, and file fits into the whole system. Feature Engineers may use SeeSys to help determine which components of the system will be impacted by planned enhancements. And, finally, Software Developers can use SeeSys to identify error-prone subsystems that are candidates for reengineering.

Acknowledgments

We gratefully acknowledge Haim Levkowitz for providing the perceptually uniform color scales, the many helpful suggestions made by David Weiss, Brian Johnson's careful proofreading, and suggestions from two anonymous referees. An early version of this paper appears in the ICSE16 proceedings. This research was conducted while Marla Baker was a summer employee at AT&T Bell Laboratories.

References

[BH92] Marc H. Brown and John Hershberger. Color and sound in algorithm animation. *IEEE Computer*, 25(12):52–63, December 1992.

[BM88] R. M. Baecker and A. Marcus. *Human Factors and Typography for More Readable Programs.* Addison-Wesley, Reading, Massachusetts, 1988.

[Boa92] OpenGL Architecture Review Board. *OpenGL Reference Manual.* Addison-Wesley, Reading,Massachusetts, 1992.

[CM88] William S. Cleveland and Marylyn E. McGill, editors. *Dynamic Graphics for Statistics*. Wadsworth & Brooks/Cole, Pacific Grove, California, 1988.

[Ebe92] Christof Ebert. Visualization techniques for analyzing and evaluating software measures. *IEEE Transactions on Software Engineering*, 11(18):1029–1034, 1992.

[Eic94] Stephen G. Eick. A graphical technique to display ordered text. *Journal of Computational and Graphical Statistics*, 3(2):127–142, June 1994.

[ESS92] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner. Seesoft™-tool for visualizing line oriented software software. *IEEE Transactions on Software Engineering*, 11(18):957–968, 1992.

[GKNV93] Emden R. Gansner, Eleftherios E. Koutsofios, Stephen C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[Gro94] Visualization Group. *Vz Visualization Library: User's Guide Version 1.0*. AT&T Bell Laboratories, 1000 E. Warrenville Road, Naperville, Illinois 60565, 1994.

[Hal77] M. H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

[JS91] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *IEEE Visualization '91 Conference Proceedings*, pages 284–291, San Diego, California, October 1991.

[KS93] Eileen Kraemer and John T. Stasko. The visualiztion of parallel systems: An overview. *Journal of Parallel and Distributed Computing*, 18:105–117, 1993.

[LH92] Haim Levkowitz and G. T. Herman. Color scales for image data. *IEEE Computer Graphics and Applications*, 12(1):78–80, 1992.

[LHMR92] Haim Levkowitz, Richard A. Holub, Gary W. Meyer, and Philip K. Robertson. Color vs black and white in visuzliation. *IEEE Computer Graphics and Applications*, 12(4):20–22, 1992.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 1(3):312–327, 1976.

[PSB93] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3), 1993.

[RCWP92] Gruia-Catalin Roman, Kenneth C. Cox, C. Donald Wilcox, and Jerome Y. Plun. Pavane: a system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3(2):161–193, 1992.

[Tuf83] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, Connecticut, 1983.

[Tuf90] Edward R. Tufte. *Envisioning Information.* Graphics Press, Cheshire, Connecticut, 1990.

[You89] Edward Yourdon. *Modern Structured Analysis.* Yourdon Press/Prentice Hall, 1989.

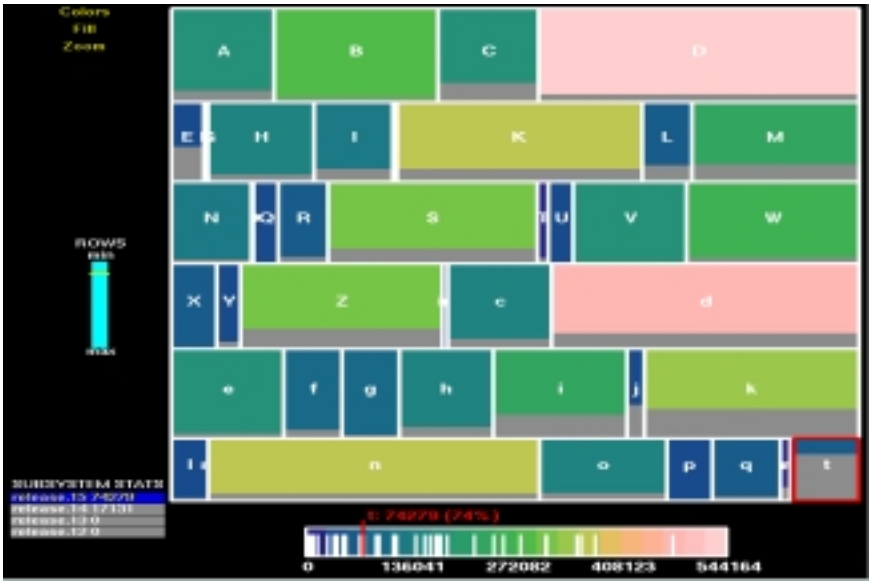

Figure 2: NCSL and new development by subsystem in a recent release.

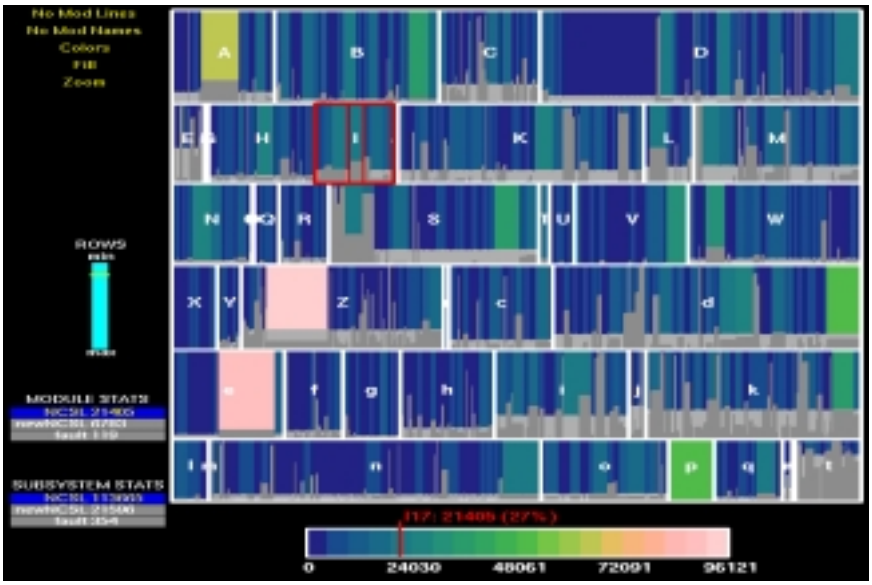

Figure 3: Directory level detail.

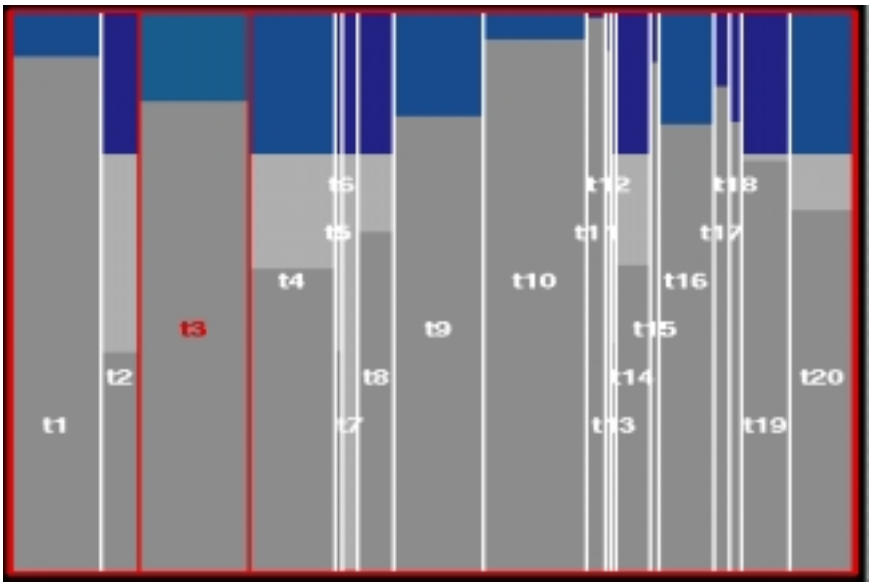

Figure 4: Zoomed view of subsystem *t*.

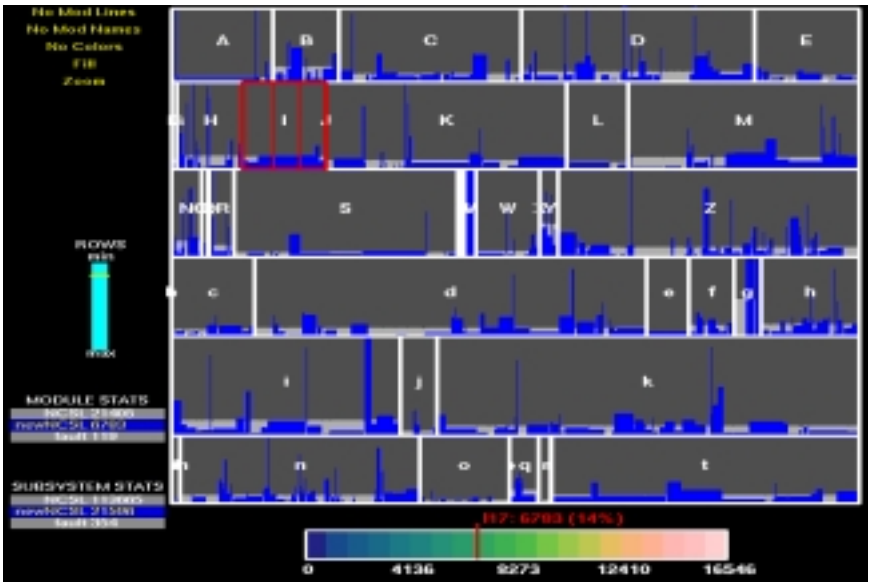

Figure 5: Bug rates by subsystem and directory.

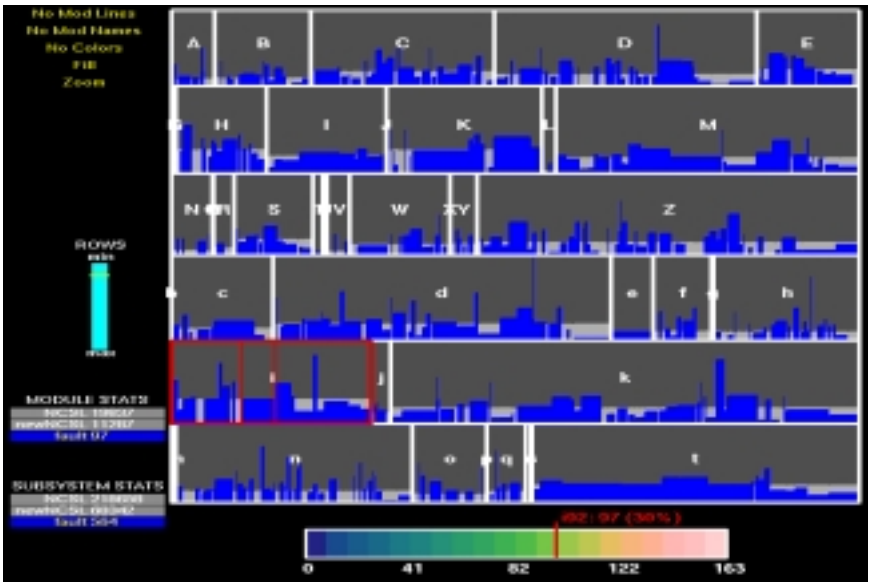

Figure 6: Fix-On-Fix rates.

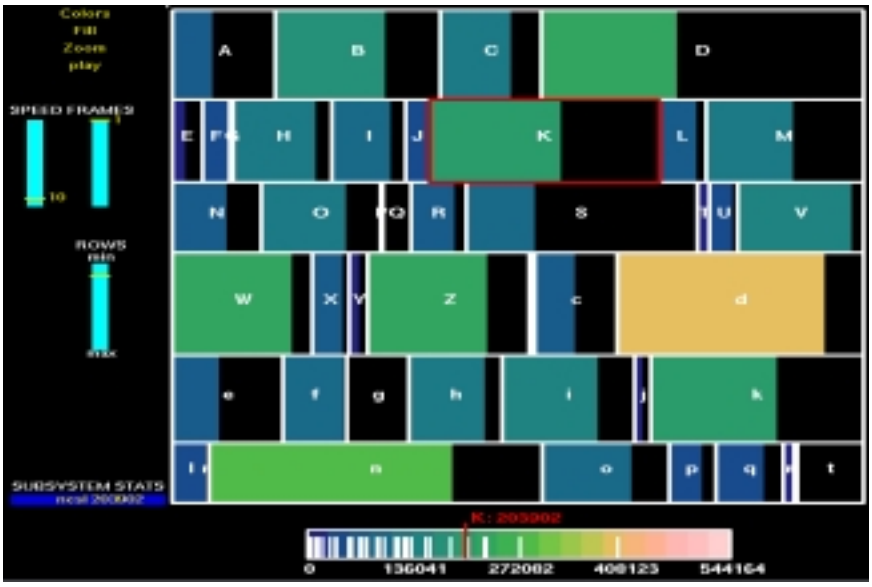

Figure 7: Frame 1 from an animation showing code growth.

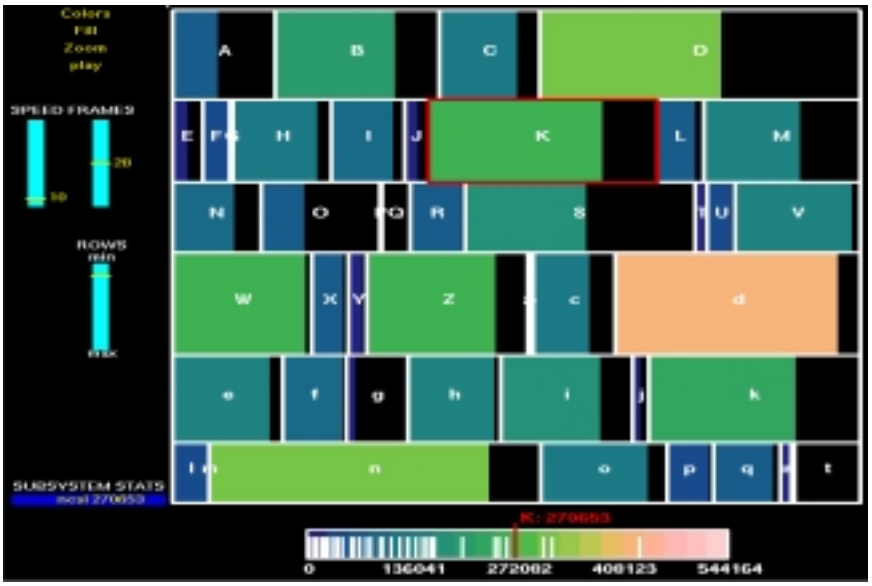

Figure 8: Frame 2 from an animation showing code growth.

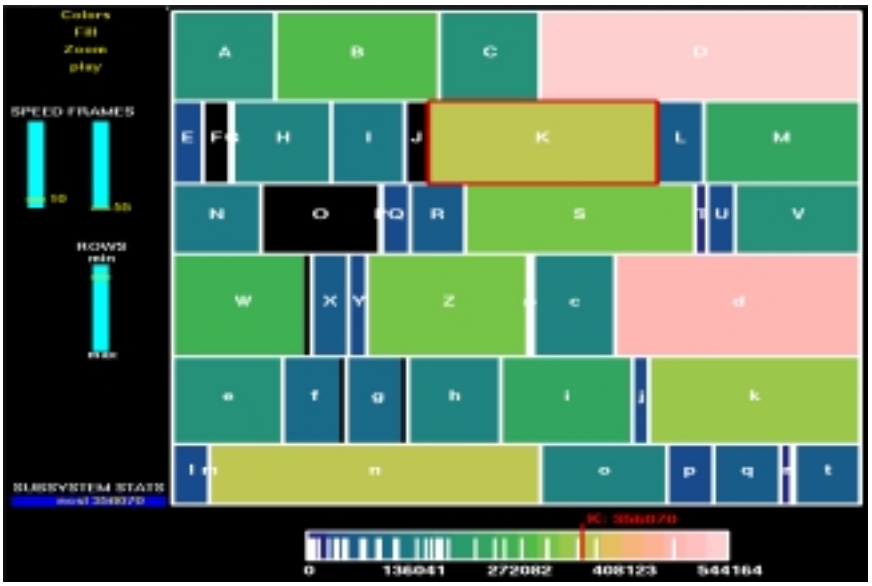

Figure 9: Frame 3 from an animation showing code growth.

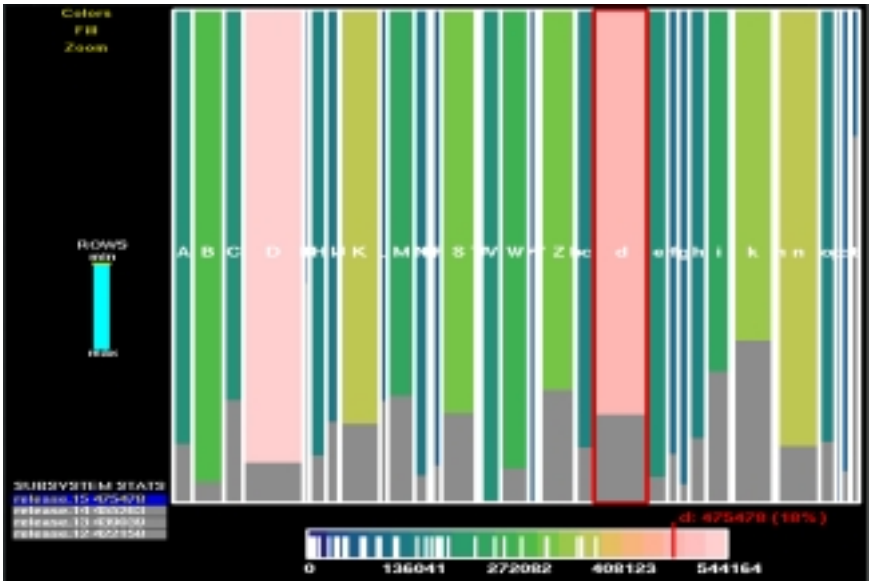

Figure 10: New development by subsystem.