Glyphs for Software Visualization

Mei C. Chuah
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-2145
mei+@cs.cmu.edu

Stephen G. Eick
Bell Laboratories
Room 1G-351
Naperville, IL 60566
eick@research.bell-labs.com

Abstract

Producing large software systems is an extremely challenging engineering task. The main reason is the difficulty of managing the enormous amounts of code and the great numbers of people involved in the effort. We have developed three novel interactive glyphs specifically tuned for visualizing software data. Our glyphs enable users to track software errors, isolate problems, and monitor development progress. To demonstrate our glyphs' functionality, we apply them to visualizing statistics from a multi-million line software project.

Keywords: Glyphs, information visualization, software visualization, large-scale systems, team-productivity.

1: Introduction

Software is a huge industry producing the most complicated data-driven systems ever created. Developing large software systems is an extremely complex, team-oriented, engineering activity. One aspect that makes software production particularly difficult is that software is invisible. It has no physical shape or form. By making software visible we can help software engineers cope with the inherent complexity, help managers better understand the software process, and thereby improve productivity of the production process. The fundamental research problem in displaying software, as in Information Visualization, involves inventing visual metaphors for representing abstract data.

Over the last several years many novel techniques and systems have appeared for visualizing algorithms, code text, and the artifacts associated with software such as code author, file sizes, file changes, and execution times[1, 2, 3, 4, 5]. Much of this research has focused on improving individual programmer productivity. Unfortunately, less research has been directed to visualizing software data associated with team productivity. In large scale systems with thousands of programmers, the team-oriented aspects of the process and management decisions dominate the effects of individuals, no matter how talented and productive individuals are. Our focus is visualizing project-oriented software data.

The main components of a software project are shown in Figure 1 with software releases at the apex of the tree. A release may be viewed from three different perspectives: functional, organizational and structural. Each of these perspectives appears as a column in Figure 1.

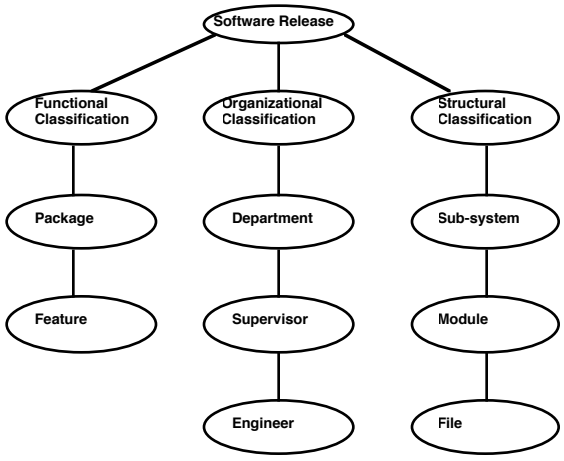

Figure 1: Software project components

The functional perspective views a release based on the various capabilities it provides to customers. A set of software capabilities are collected together and packaged into a release for customers. This categorization is useful because it defines a customer delivery against which deadlines are cast. In addition, it is commonly used as an organizing factor for the development team producing the release.

The organizational perspective focuses on the people involved in a software production process. The development team is usually divided into

departments. Each department in turn has its own hierarchy of groups and engineers. This categorization is useful for providing human links for resolving or following up on problems.

Finally the structural perspective views the software structure of the project. This refers to how the code in the project is grouped, i.e. the directory structure. At the highest level the code is divided into subsystems. Within each subsystem are modules, and within each module are source code files.

A multi-million line software project may be partitioned into tens to hundreds of subsystems, hundreds to thousands of modules, thousands to hundreds of thousands of files. For each file the version management system maintains the complete history, including the date and time of every change, lines affected, reason for the change, functionality added by the change (for new code additions), the fault repaired (for error correcting code), etc.

These databases associated with software projects are a rich, underutilized resource for understanding software production. The challenge, however, is to extract information from the database and present it to software engineers and managers in a useful and actionable form. This is difficult because the data is nontraditional, unstructured, and noisy. The volume of information (hundreds of potentially interesting statistics on hundreds of thousands of files) makes looking at printouts infeasible. Furthermore, by looking at fine-grain details it is impossible to obtain an overall perspective. The traditional way to analyze large datasets involves statistical analysis. Unfortunately, for software data, the many missing values and non-numeric nature of the data frustrate statistical analysis software.

To overcome these problems we have developed three glyphs, *infoBUG*, *timeWheel*, and *3D-wheel*, that we believe to be novel. Each of the glyphs presents software data from a new perspective. We believe the glyphs are useful for visualizing software project data for the following reasons:

Use of prior established visualizations: We combine established visualization views (time series, histogram, rose-diagram) to form our glyphs and have tailored them to the software domain. This allows users to more easily interpret the glyph by using prior graphic knowledge. In addition, our glyphs allow us to effectively show both continuous (e.g. time) and discrete (e.g. code author, file type) data, unlike previous glyph work[6] which can only encode discrete attributes.

Maintenance of object coherence: We use glyphs to view many dimensions of a data object simultaneously. We do this by clustering together simple graphical artifacts (e.g. marks, bars, lines). In infoBUG clustering is achieved by forming a familiar shape, namely an insect. In timeWheel and 3D-wheel the graphical components are clustered by using a circular layout technique.

Another method for viewing such multi-dimensional objects uses linked scatterplots [7]. The advantage of glyphs over linked plots is that glyphs preserve the "objectness" of the data elements (i.e. all properties of a particular software component are grouped together spatially). This is important in analyzing software systems because the data elements within that domain conform to real programs, people, or concepts. On the other hand it is much harder to look for general trends across objects using glyphs. For this task, we provide users with the hierarchy of scatterplots (Figure 9).

Ability to show multiple different object types: The glyphs are versatile and can show data for many different software components (e.g. releases, engineers, features). Thus users do not need to relearn new visualization structures for each component type.

In the following three sections we present each of the glyphs in detail and apply them to visualizing software data associated with a large real-time software system. This software system, developed over the last two decades by thousands of programmers, has gone through many releases and is deployed world-wide. Our glyphs focus on showing the various components present in this software project, and how these components change through time.

2: InfoBUG

The InfoBUG glyph provides an overview of the software components and facilitates comparisons across these components. It is so named because it resembles an insect, consisting of four main sections: wings, head, tail, and body. Figure 2 shows this decomposition and Figure 3 shows 16 releases of the software project using the infoBUG interface. The advantage of an InfoBUG over tiled single-variable plots is that it permits the user to simultaneously compare objects across many dimensions.

InfoBUG Wing: Each insect wing is a time series with time being encoded from top to bottom. The x-axis on the left bug wing encodes the relative number of code lines in the software object *(loc)* and the x-axis on the right bug wing encodes the relative number of errors *(err)*.

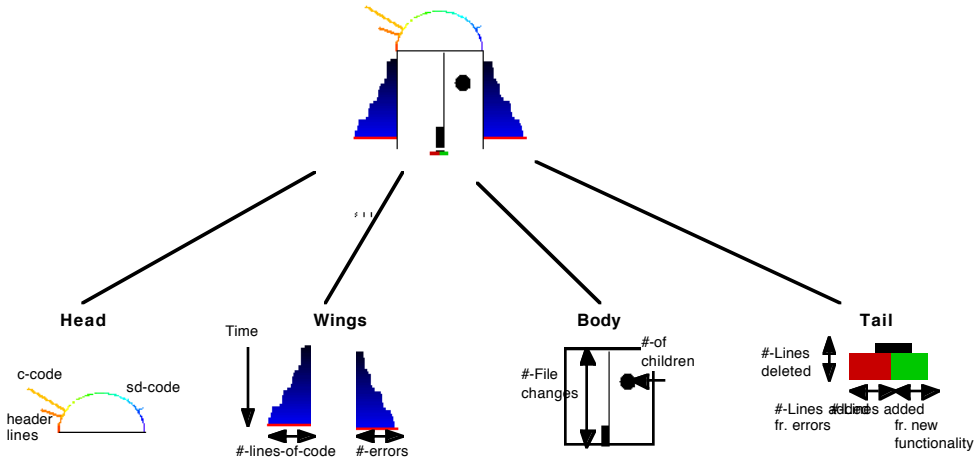

Figure 2: Decomposition of InfoBUG into its four main components

By comparing the shape of the right and left wings we can determine whether increases in *loc* bring about similar increases in *err*. Usually we would expect increases in *loc* to bring about comparable increases in *err* and vice versa. If *loc* increases are not accompanied by increases in *err*, then the particular object is possibly not being well tested. On the other hand, if increases in *err* are not caused by similar increases in *loc*, then the existing code could be inherently difficult, have architectural problems that are causing problems, or be poorly written and in need of re-engineering. In such cases the wings of the infoBUG would be non-symmetrical, and thus relatively easy to identify in the representation that we have chosen. From Figure 3, all the bug wings appear symmetrical, indicating that there are no *loc*/*err* growth problems at the

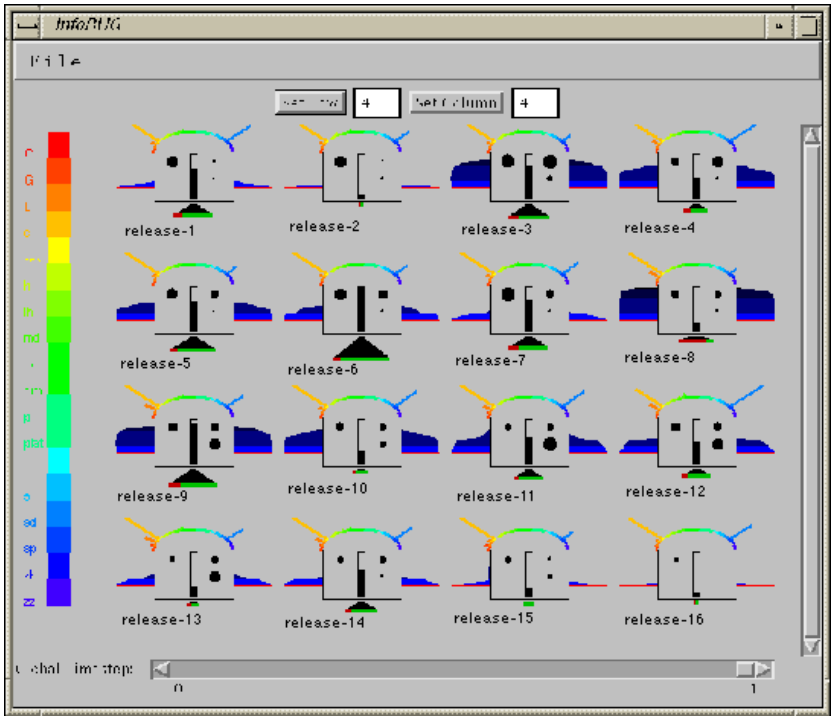

Figure 3: infoBUG interface showing 16 software releases

release level.

The position of the wings (whether at the bottom or to the top) indicates whether an object was worked on early or late in the project. For example, we can tell that *release-1, release-2, release-7, release-13, release-14, release-15* and *release-16* are all new releases because their wings are placed at the bottom of their bodies. *Release-3*, *release-4*, *release-8*, *release-9* and *release-10* are all older releases, indicated by their wing positions close to the top.

The InfoBUG glyph is interactive. Clicking on the wings, will cause the selected *time* slice to be shown in the head, body and tail of the infoBUG. The current time slice shown is indicated with a red band. The *time* component for all infoBUG glyphs can be changed simultaneously by using the slider at the bottom of the interface (shown at the bottom of Figure 3). Through interactive manipulation, the analyst can determine how the different data attributes of an object change through time. Stroking the mouse through the wing(s) of objects shows the changes in file type consistency (head), in number of file changes (body) and in the number of lines added and deleted (tail).

InfoBUG Head: Within a software object (such as a release, module, or a feature package) there are many different types of code. For example, the release shown in Figure 2 has *C*-language code, *SD*-language (State Definition) code, header lines, preprocessor directives, and debugging code. The antennae on the infoBUG head show, for a given time, the relative code sizes by type. The code type is color coded and the color scale for it is shown at the left of the display (Figure 3).

This encoding allows the analyst to compare the composition of different objects. For example, Figure 3 tells us that most of the releases are made up of *C* code (shown in light gray) and *SD* code (shown in dark gray). Certain releases have significantly more *C* code than *SD* code (e.g. *release-5*, *release-6*), while others have about equal amounts of both (e.g. *release-4*, *release-1*). All of the releases have some small amount of *L* type code (headers).

By interactively changing the *time* component we are able to obtain information on how the different code types evolve. An interesting example is *release-10* which started off only having *L* type code (header code). Shortly after, *C* code was added and then more slowly, *SD* code. Currently *C* code is most prevalent and *SD* code next most with about two thirds as much. Such changes give us hints to development practices and changes in the demands(or functionality) of a software component.

InfoBUG Tail: The bug tail is triangle-shaped. Its base encodes the number of code lines added (*add*) and its height encodes the number of code lines deleted (*del*). The tail base (i.e. total number of code lines added) is further divided into two parts: code added due to error fixing (left, color coded in dark gray) and code added for new functionality (right, color coded in light gray). Figure 3 shows that most of the releases consist of code added for new functionality, except for *release-8* which is a bug-fixing release.

By looking at the shape of the tail we can determine the ratio of *add* to *del*. A short squat triangle like the one for *release-8* shows a high *add/delete* ratio. The shapes of the triangles for most of the other releases are less squat indicating a lower *add/delete* ratio. A triangle that is higher than it is wide has more deleted lines than added lines. This could be an indication of a serious problem in the release. None of the releases in Figure 3 show this property.

Apart from the shape of the tail, the size of the tail indicates the amount of activity in a particular object. Objects with larger tails (e.g. *release-6*, *release-9*) have more activity than those with smaller tails (e.g. *release-11*, *release-10*). It is interesting to note that *release-6* appears to be the largest release but it is a fairly new effort that only became active relatively late in the development process.

InfoBUG Body: The infoBUG body encodes number of file changes (*fchg*) in the bar at its center. From Figure 3 we can tell that *release-6* has the most number of file changes (*fchg*). This is not surprising as *release-6* also has the most lines of code added and deleted as indicated by its larger tail.

The InfoBUG body also shows the number of child objects contained within the current parent object. This is encoded as the size of the black circles on the insect body. The type of child objects encoded depends on the software hierarchy of the system being analyzed. In our system for example, a release object will have as its children modules, supervisors, and packages. A supervisor object on the other hand has developer and feature objects as children. The size of the children groups helps us gauge whether a software object is "wide" (i.e. related to many other components) or "narrow" (i.e. related to only a few other components). A software object may be wide in certain respects and narrow in others. For example *release-1* and *release-2* in Figure 3 are spread out over many modules (top left body circle) but affects very few supervisors and packages. This indicates that the releases are specific to a small set of packages but the changes made affected large portions of those

packages. On the other hand, *release-11* affects many packages (lower right body circle) but the effects within each package are relatively small as indicated by the small module circle (top left circle).

3: Time-wheel

The timeWheel allows the user to view multiple different properties of a software component through time. Each property is represented as a time series and all of them are laid out circularly. Figure 4 shows a timeWheel for developer *userId1*. The properties encoded by the height of each of the time series in the timeWheel is also shown in Figure 4.

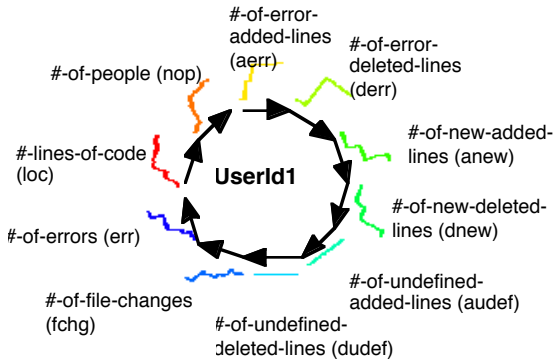

Figure 4: Variables represented in timeWheel

The number of undefined added code lines (*audef*), and undefined deleted code lines (*dudef*) refer to code changes that have missing data in its purpose field. The number of people attribute (*nop*) for developer objects refers to the number of people that worked on the errors owned by the current developer. Each of the variables on the timeWheel are color coded according to the color at the left of the timeWheel interface (Figure 6). The direction of the arrows in Figure 4 indicates the direction of time increment for each series.

Two main trends that can be easily identified using the timeWheel are: tapering trends (Figure 5-left) and increasing trends (Figure 5-right). Tapering trends have high activity at the outset which slowly tapers off with time. Increasing trends, on the other hand, have little activity at the outset followed by increasing activity towards the end.

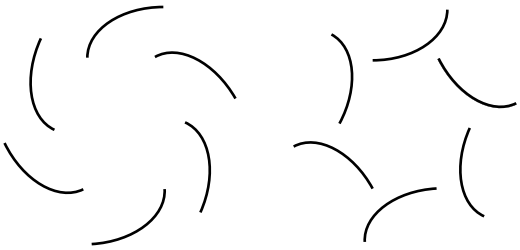

Figure 5: (left) tapering trend (right) increasing trend

Figure 6 shows the timeWheel display of all the

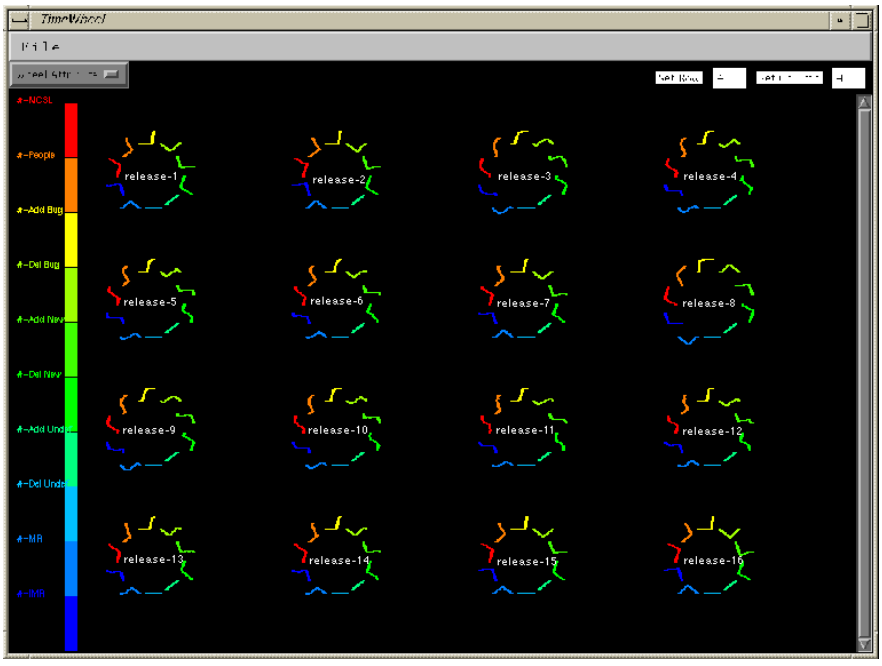

Figure 6: timeWheel interface showing 16 software releases

releases shown in Figure 3. From Figure 6 we can pick out which releases have increasing and which have decreasing activity. Increasing trends can be seen on *release-1, release-2, release-7, release-13, release-14, release-15* and *release-16*. Not surprisingly these are all the later releases and thus most of the effort came late in the development process. On the other hand, *release-3, release-4, release-8, release-9*, and *release-10* exhibit decreasing trends. These are the older releases that were developed earlier on in the development process and then completed.

From Figure 6 we also see that all the time trends within an object have approximately the same shape (except for the two flat time series which indicates that there is missing data). This indicates that there are no deviations from the dominant trend of progress.

Some objects, however, may have mixed trends. For example in Figure 4, *userId-1* has an overall tapering trend, but there are divergent variables. The interesting information to derive from *userId-1*'s timeWheel display is that the *aerr* and *derr* attributes have tapering trends while the *anew* and *dnew* attributes have increasing trends. Because the *loc* trend (dark gray) is tapering, we can deduce that most of the code added were from error fixes. In addition, we can tell that there are two clear phases for developer *userId-1*. First, *userId-1* did error fixes but later moved on to developing new code. We can also deduce that error fixing accounted for a more important portion of *userId-1*'s activities because it corresponds to the dominant trend. All this information would have been lost in the other visualizations because detailed time information is not shown.

4: 3D-wheel

The three dimensional wheel encodes the same data attributes as the timeWheel but using the height dimension to encode time. Each object variable is encoded as an equal slice of a circle and the radius of the slice encodes the size of the variable just as in a *rose-diagram*. Each variable is also color coded as in the timeWheel display.

An object that has a sharp apex has an increasing trend through time and an object that balloons out has a tapering trend. Figure 7 shows the same 16 releases of Figures 3 and 6 as 3D-wheels. It is easier to perceive the overall time trends from the 3D-wheel glyphs (Figure 7) than the timeWheel glyphs (Figure 6). This is because in the 3D-wheel, the trend is clearly expressed by the shape of the wheel whereas in the timeWheel the trend has to be derived from the wheel pattern of each software component.

While it is easier to identify trends using the 3D-wheel, it is harder to identify divergences because of occlusion and perspective. Users must rotate the 3D-objects to see all of the data and at any one time only a subset of the data can be seen. Figure 8 shows the 3D-wheel representation of *userId-1*. Note that unlike the timeWheel display (Figure 4) it is harder to see

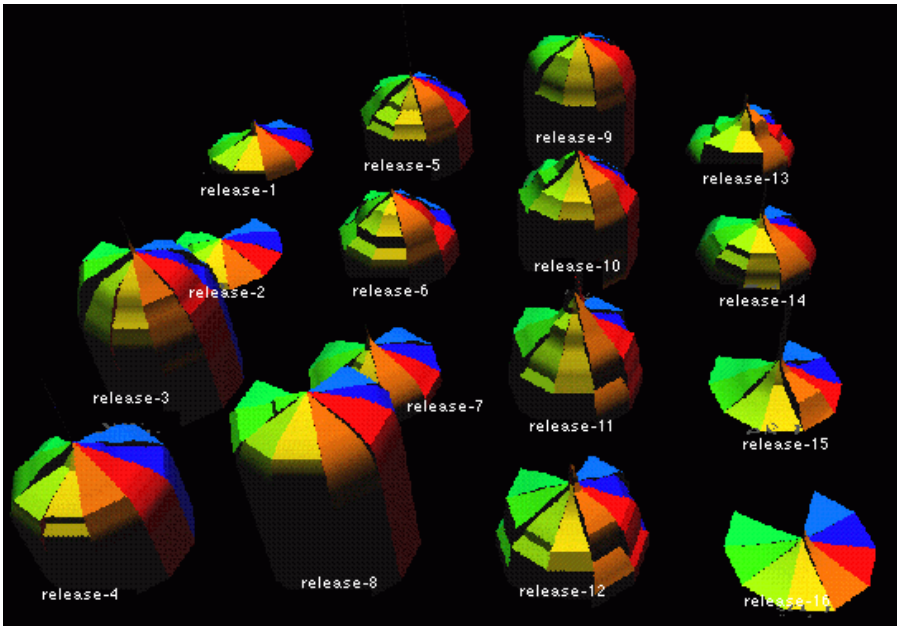

Figure 7: 3D-Wheel interface showing 16 software releases

the difference in trends between the *anew, dnew* attributes (which have increasing trends) compared to the other attributes (which have decreasing trends). This is because part of the glyph is occluded.

Figure 8: 3D wheel display of UserId-1

5: Scenario

We have integrated our different glyphs using a hierarchical sequence of tiled scatterplots (Figure 9). This hierarchy is based on the software hierarchy described in Figure 1 but with some modification. At the x-axis of each scatterplot we encoded the number of code lines (*loc*) and at the y-axis we encoded the number of errors (*err*). This encoding is made because the ratio of *err/loc* helps determine the quality of a software component.

In this scenario, a software manager owning a software subsystem is exploring its components for possible problems. First of all, the manager loads all the relevant software releases into the scatterplot hierarchy (top row Figure 9). To get an overall summary of the releases, she views them using the InfoBUG glyphs. This is achieved by selecting all objects in the scatterplot with a bounding box and then pressing the *BUG* button on the interface. The infoBUG view on all 16 releases is shown in Figure 3. An object that stands out in Figure 3 is *release-8* which has significantly more bug-fixes than the other releases.

The manager selects this release and displays all of its child components at level two of the hierarchy (middle row Figure 9). This level groups objects according to *module*, *supervisor*, and *feature*. In this view, she looks at the supervisor scatterplot and picks the supervisors that were most involved with this particular release (i.e. has high number of code lines), namely *sup-1* and *sup-2*. She then hands the problem over to them.

The supervisors examine the *module* and *feature* information for *release-8* and finds that in particular, changes were made to *moduleY* and *moduleX*. Thus they display the child components of these modules at level three of the hierarchy (bottom row Figure 9). This level shows developers and low-level features. Based on these scatterplots, the primary developer

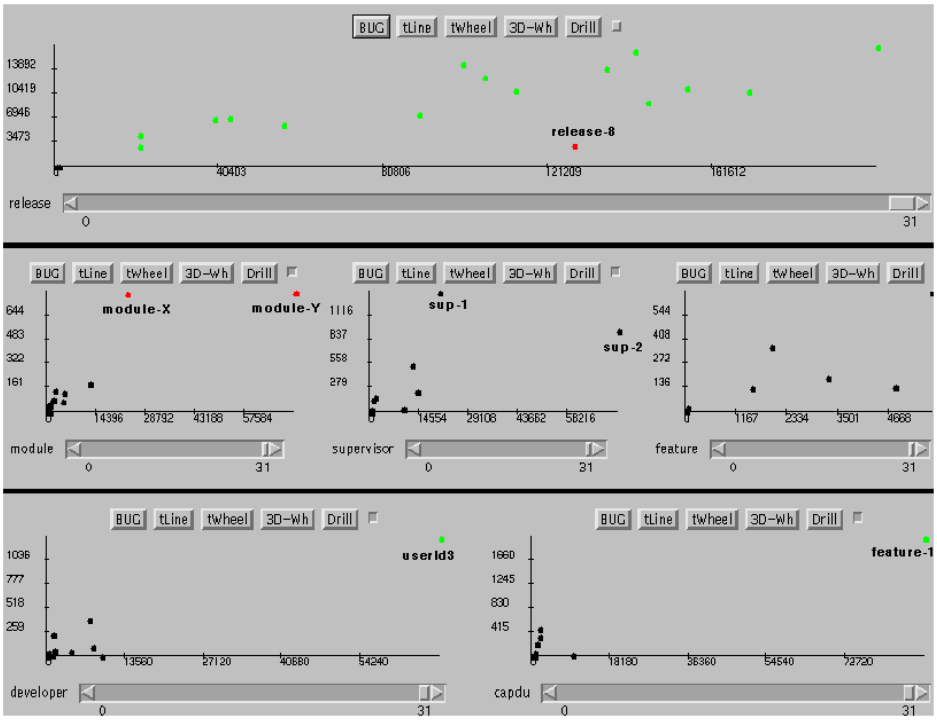

Figure 9: Hierarchy of scatterplots

involved in changing *moduleX* and *moduleY* for *release-8* is *userId3*. The main low-level feature is *feature-1*.

To get more information, the supervisor looks at the global profile (statistics collected from all releases) of *feature-1* to see if the problem is special to this particular release or is common across releases. The global data of *feature-1*, viewed as an infoBUG glyph (not shown), show that this particular problem is present globally. The supervisor makes note of this and moves on. There is a possibility that the high *err/loc* ratio may cause the release to be delayed. To check on lateness the supervisor brings up a timeLine display for *release-8* (Figure 10).

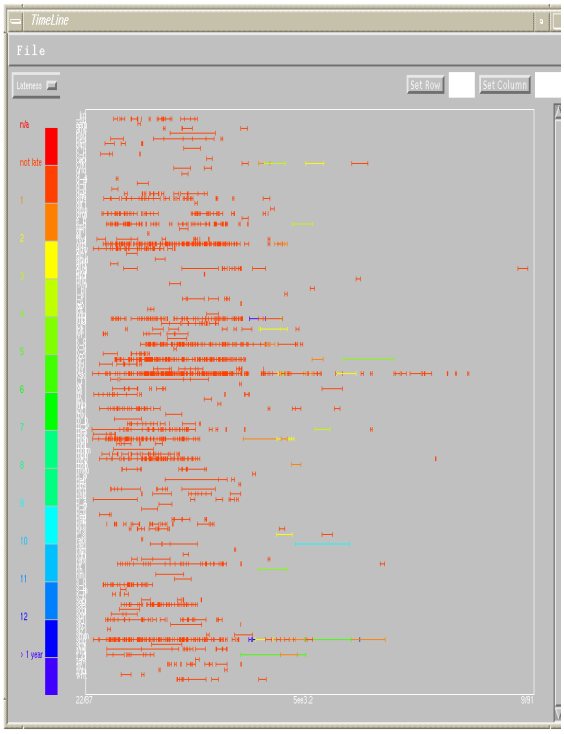

Figure 10: TimeLine display of release-8 with lateness attribute encoded as color

The timeLine display (Figure 10) is used to view detailed information about each error. Each timeLine encodes time in the x-axis and developers on the y-axis. The start of each line represents the time at which the error was opened for fixes and the end of each line encodes when the error was fixed. Other properties of the error(e.g. priority, severity, lateness) may be mapped to the color of the lines. The color encoding is shown by the axis on the left. By setting the color of the timeline to encode lateness (Figure 10), the supervisor discovers that there are no serious lateness problems in this release.

The supervisor then examines the main developer involved in modules *moduleX* and *moduleY*, namely *userId3*. From the infoBUG glyph of the global profile of *userId3* (Figure 11), it appears that *userId3* is a bug fixer. This is indicated by the infoBUG glyph tail base which is predominantly dark gray. The timeLine display of *userId3* (not shown) further shows that he is conscientious of fixing his bugs promptly. At this point the supervisor contacts developer *userId3* and asks him for information on the problem.

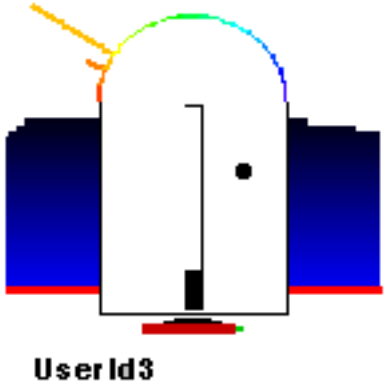

Figure 11: InfoBUG global profile for userId-3

UserId3 is currently working on multiple different releases, and is unable to recall the situation surrounding this problem. Therefore, he runs the glyph system and examines the information provided by *sup-1* and *sup-2*. He then loads in the files associated with the problem and examines them using SeeSoft[8]. He also uses a timeLine display (not shown) to view the actual bug concepts that were involved and summarizes his findings for his supervisor, providing visual aids where necessary.

This scenario demonstrates that it is important to have a system that spans different levels of data abstraction (e.g. from releases to pakages to low-level features) because different levels of the organization are interested in different parts of the data. By having all this within the same system, the people involved can communicate through a common base and share their findings.

6: Conclusions

We present three novel glyphs for visualizing software data. These glyphs are used to analyze different aspects of a large software project, including isolating problems, uncovering their effects, and finding possible solutions. The glyphs are innovative in that:

- They integrate established visualization techniques thereby capitalizing on known graphical skills.
- They maintain the "objectness" of software components, which is important in the software data domain.
- They are versatile and able to show many different software data types.

One particularly exciting aspect of this research involves our company Intranet. We are using the corporate WEB as a distribution mechanism to provide access to our visualizations. We built our glyphs using Java and VRML and have them running on top of a Netscape browser. Now anyone inside the corporate firewall can access our software visualization glyphs and display software project data. In the past we have built many innovative tools that were not widely used because of platform and database obstacles. By centralizing the databases and building on top of a ubiquitous platform, we can connect with a much wider user base.

References

1. M.H. Brown, "Algorithm Animation," *in ACM Distinguished Dissertations*, MIT Press, New York, 1988.

2. R.M. Baecker and A. Marcus, *Human Factors and Typography for More Readable* Programs, Addison-Wesley, Reading, Mass., 1990.

3. B.A. Price, I.S. Small, and R.M. Baecker, "A Taxonomy of Software Visualization*," J. Visual Languages and Computing,* No. 3, Vol. 4, 1993.

4. T. Ball and S.G. Eick, "Software Visualization in the Large," *IEEE Computer*, No. 4, Vol. 29, 1996, pp. 33-42.

5. E. Kraemer and J.T. Stasko, "The Visualization of Parallel Systems: An Overview*", J. of Parallel and Distributed Computing*, Vol. 18, 1993, pp. 105-117.

6. R.M. Pickett and G. G. Grinstein, "Iconographic Displays for Visualizing Multidimensional Data," *Proceedings IEEE Conference on Systems, Man and Cybernetics*, 1988, pp. 514-519.

7. R.A. Becker and W.S. Cleveland, "Brushing Scatterplots,", Technometrics, Vol 29, 1987, pp. 127-142.

8. S.G. Eick, J.L. Steffen, and E.E. Sumner, Jr., "SeeSoft—A Tool for Visualizing Line-Oriented Software Statistics," IEEE Trans. Software Eng., Vol 18, No. 11, 1992, pp. 957-968.