Managing Software with New Visual Representations

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

Managing large projects is a very challenging task requiring the tracking and scheduling of many resources. Although new technologies have made it possible to automatically collect data on project resources, it is very difficult to access this data because of its size and lack of structure. We present three novel glyphs for simplifying this process and apply them to visualizing statistics from a multi-million line software project. These glyphs address four important needs in project management: 1) Viewing time dependent data; 2) Managing large data volumes; 3) Dealing with diverse data types and 4) Correspondence of data to "real-world" concepts.

1 Introduction

Visualization can support software engineering at many levels. For code production, which includes code writing, code understanding, debugging, and feature modifications, visualizations are used to highlight functions, identify the differences between releases, examine function execution times, and to understand code change history and authorship [1,2,6,11]. For algorithm understanding, animation can easily communicate the underlying mathematical concepts [4,8]. There is, however, an aspect of the software process that visualization has yet to impact: Project management, the overall task of efficiently managing and processing resources, both human and machine, involved in a software project.

Managing a large software project is time intensive. Any reasonable-sized project will have many different classes of resources (lab equipment, staff time, machine cycles, disk resources, interim deliverables, customer commitments) that must be scheduled and tracked. Inevitably problems will arise and solutions must be found. To support the management process, information systems collect and maintain large status databases. Our aim is to support and improve this process through visualization.

Although large volumes of data are collected, much of it remains underutilized. The size of the data volumes make them unfeasible to read textually, and their lack of structure frustrates statistical analysis tools. New visual metaphors can simplify the process of extracting information and presenting it to users in an actionable form. This paper presents three novel glyphs for exactly this purpose.

Managing a large software project is a specific instance of the project management problem. Project management is *"the art of directing and coordinating human and material resources throughout the life of a project by using modern management techniques to achieve predetermined objectives of scope, cost, time, quality, and participation satisfaction"* [7]. Besides software, project management is involved in many other industries including construction, manufacturing, and transportation. Thus even though we will only show the application of these glyphs to software production management, they address issues that are common to project management domains in general.

There are four interesting issues in project management data:

1. *Time*: Project management is time-oriented. Each project has a "life" or a time in which it must be completed (deadline). To properly meet deadlines it is important to track milestones, monitor resource usage patterns, and anticipate delays.

For Color Plate See Page 118

0-8186-8189-6/97 $10.00 © 1997 IEEE

2. *Large Data Volumes*: Large projects have a lot of data associated with them. For example, a multi-million line software project may be partitioned into tens to hundreds of subsystems, hundreds to thousands of modules, and thousands to hundreds of thousands of files. Our experience is that much of this data is unstructured, so mining information from it is difficult. Our approach, as is typical in large projects, is to partition the data hierarchically. For example, a software project will have a high-level manager with overall responsibility. This manager may have several supervisors under her and each supervisor will lead a group of engineers.

3. *Diversity/Variety*: It is common for projects to have a diverse group of resources as well as resource attributes. Expressing different types of resources (e.g. engineers, computers) as well as their attributes (e.g. number of code lines, number of errors) requires that the visual representations be flexible enough to convey meaningfully information about a set of diverse data. Our glyphs are designed to be versatile so that they can show data for many different software artifacts. Users do not need to relearn new visualization structures for each object type. Flexibility is achieved by enabling our representations to show many different data types, including both discrete and continuous domains. This is unlike previous glyphs [5,10] that focus on a narrower set of data types.

4. *Correspondence to "real world" concepts*: In a project database, data elements usually correspond to "real world" entities or concepts. For example, a *userID-1* data element in a software database represents an actual person and the element *file-125* corresponds to a source code file. By using glyphs, we maintain the "objectness" of the data elements because all the properties of a data element are grouped together visually. Visual grouping is achieved in our glyphs in two ways: 1) by bringing together various graphical artifacts to form a familiar shape, namely an insect; 2) by arranging the graphical elements according to a common geometric shape (circle). Another method for viewing multi-dimensional objects is through linked scatterplots [3], however, this method does not preserve the "objectness" of the software components.

Glyphs are not a new concept, being first developed for multi-dimensional data by Chernoff in 1973 [5]. Our work, however, is different from previous efforts because it combines established visualization views (time series, histogram, rose-diagram) to form glyphs. This allows users to more easily interpret the glyph by using prior graphic knowledge.

Above we have described how our glyphs deal with the issues of *diversity* and *"real-world" correspondence.* In the next two sections we describe how our visual representations can be applied to view time information and deal with large data sets.

2 Viewing time-oriented information

Visualizing time-oriented information is challenging because it is unclear what representation will best show the salient information. Animation, a traditional method, uses a symbol to represent the information at one slice in time. This symbol is then rapidly varied to show the data for subsequent time slices. Rapid changes representing outliers are jarring and easily perceived [3]. Although effective for identifying outliers, animations are less effective than traditional time-series plots for determining overall time patterns. A time-series plot has time on the x-axis and a variable on the y-axis. For viewing time information we present two types of visual representations: timeWheel and 3D-Wheel. These representations are a variation of the time series plots.

2.1 TimeWheel

In a timeWheel each object attribute is represented as a time series and the time series are laid out around a circle. The goal of this display is to be able to quickly or even preattentively pick out objects based on their time trends. Figure 1 shows a timeWheel and the different attributes that are mapped onto it. The direction of the arrows indicates the direction of time increment for each series.

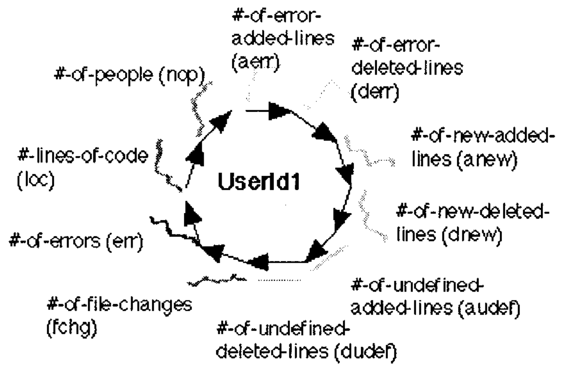

#-of-error-added-lines (aerr)
#-of-error-deleted-lines (derr)
#-of-people (nop)
#-lines-of-code (loc)
UserId1
#-of-new-added-lines (anew)
#-of-new-deleted-lines (dnew)
#-of-errors (err)
#-of-undefined-added-lines (audef)
#-of-file-changes (fchg)
#-of-undefined-deleted-lines (dudef)

Figure 1: TimeWheel Glyph

We encode attributes in two ways, by their position on the circle and by a rainbow-hue colormap. Color, or hue, simplifies the process of identifying object attributes. The rainbow colormap is appropriate for encoding the attribute types because it is a perceptually discrete dimension [12]. Nevertheless, with a careful choice of scale, hue may also be used to encode continuous variables [9].

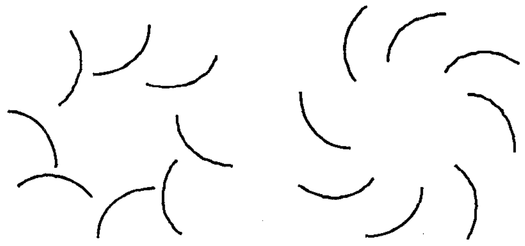


Figure 2: Left - increasing trend timeWheel (prickly fruit); Right- decreasing trend timeWheel (hairy fruit)

TimeWheel glyphs show two major trends: the increasing trend and the decreasing or tapering trend. The increasing trend glyph looks like a "prickly fruit" (Figure 2-left) and it indicates objects which have very little activity at the outset but increasing activity towards the end. The decreasing trend glyph on the other hand looks like a hairy fruit, e.g. a coconut husk (Figure 2-right). Decreasing trends indicate high activity at the outset but declining activity through time.

Figure 3 (Plate 1-a) shows 16 software releases using the timeWheel glyph interface. Looking at this figure we can partition releases into three major classes: 1) objects with increasing trends, which are new releases that have only been worked on later in the project (outlined in light gray in Figure 3 and white in Plate 1-a); 2) objects with decreasing trends which are the older releases and activity has since slowed to a crawl (outlined in black in Figure 3 and red in Plate 1-a), and 3) in-between objects which represent releases worked on in the middle of the project (non-outlined objects in Figure 3 or Plate 1-a).

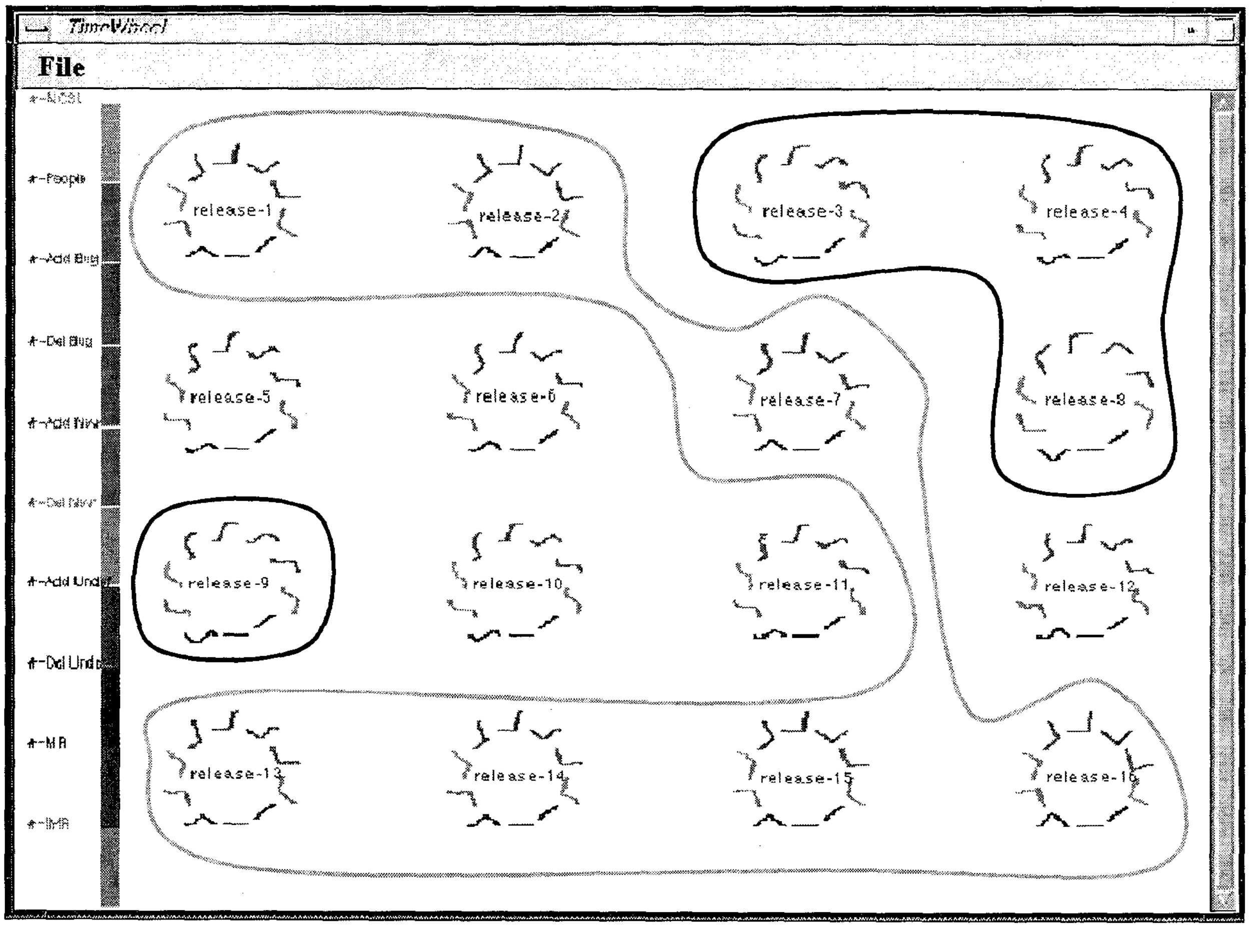


Figure 3: timeWheel interface for 16 software releases (Plate 1-a)

Grouping the time series into a glyph allows us to identify the dominant time trend, however, having the individual time series available allows us to examine divergences from that trend. For example in Figure 1, *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 (colored in red) 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.

An obvious and traditional way to arrange a set of time series on a two dimensional plane would be to lay them out linearly as in Figure 4. For tasks involving browsing or searching for gestalt patterns, the circular layout may be more effective than a linear layout for four reasons: eye movement, local pattern perception, reading order, and information density.

Reduce the number of eye movements per object: Cropper & Evans as well as Danchak found that the visual angle over which the eye is most sensitive is 0.088 radians (5 degrees) [14]. Cropper and Evans subsequently stated that "*the presentation of information in `chunks' ... which can be taken in one fixation will help to overcome the limitations in the human input system in searching tasks*". Laying out the time series in a circular fashion as in Figure 5 allows all of the time plots to be taken in with one eye fixation. However, laying them out in a linear fashion as in Figure 4 requires more than one eye fixation.

There is, however, a limit to the number of object attributes that can be displayed in the timeWheel for it to fit within the area of an eye fixation. As a rough approximation, for a viewer that is 15 inches from the screen, a visual angle of 5 degrees translates into a circular area on the screen with a radius of 0.65 inches. Our experience is that we can comfortably encode 10 variables in that area. It may be possible to encode as many as 15 variables before the display becomes too dense to interpret.

An alternative layout scheme arranges the time series in rows. This reduces the number of eye fixations, however, the user might begin to cluster the data by rows because we are conditioned to it from reading text. This would adversely affect the users' ability to sense the overall time patterns in the glyphs. Another possibility positions the time series out in 3D space and encode properties along the z-axis. The drawback to this 3D layout is occlusion: The first few series occlude the others.

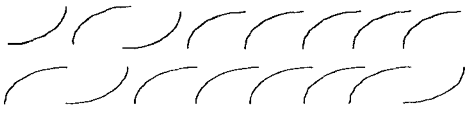


Figure 4: Linear layout of *object-1* and *object-2*

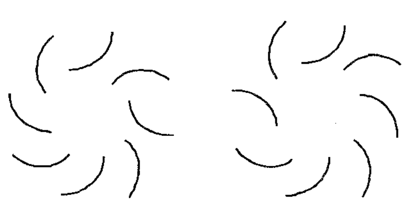


Figure 5: Circular layout of *object-1* and *object-2*

Less susceptible to local patterns: Linear ordering highlights local patterns. For example, in Figure 4 a dominant visual impression is the cyclic local pattern shown in Figure 6. The local pattern is formed here because our perceptual system groups the two time series based on the gestalt principle of closure [13]. This grouping however is spurious because the object attributes have no ordering.

The perceptual differences between the two rows in Figure 4 are emphasized due to local grouping effects. By comparison, the circular placement suffers less because the symbols are not placed directly next to each other. As an example Figure 5 shows that the rows in Figure 4 are in fact quite similar -- one is merely a rotation of the other. This pattern is masked in Figure 4 because of distracting local patterns.

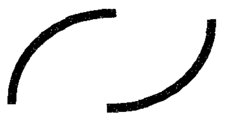


Figure 6: Cyclic local pattern

Reading order: A linear layout encourages users to read the plots from left to right. Since the attribute types are unordered, this may cause false impressions. For example, more importance could be placed on the series at the start or end. Unlike linear ordering, a circular layout positions each time series at the same distance from the glyph center. In this way, the time series position has a much weaker ordering implication. Reading order is another reason why the two rows in Figure 4 appear to be different. The bottom object has cyclic patterns at its start and end while the top object has two opposing patterns namely and .

Less separation and therefore higher information density: The circular layout creates a strong gestalt pattern out of individual time series. We recognize the circular pattern because it is a common geometric shape. On the other hand, the linear layout ties the time series together only through spatial proximity. As a result, for us to see the boundaries between objects, we have to leave a lot more whitespace between the series than in the timeWheel case.

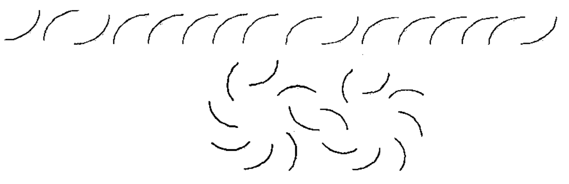


Figure 7: The objects in Figure 4 and Figure 5 placed close to each other

For example the top row in Figure 7 contains the same information as the bottom row and is divided by the same amount of whitespace, however it is hard to see the division between the top objects while it is much easier to see the division between the bottom objects. Instead of whitespace we could use a bounding box to indicate object boundaries for the linear layouts, however this adds to the density of the display and may distract the user[14].

2.2 3D-Wheel

The 3D wheel encodes the same data attributes as the timeWheel but using the height dimension to encode time. Each variable is encoded as an equal slice of a base circle and the radius of the slice encodes the size of the variable as in a *rose diagram*. Each variable is also colored in its own discrete, shaded color. An object that has a sharp apex as in Figure 8-left has an increasing trend through time and an object that balloons out as in Figure 8-right has a tapering trend.

Figure 9 (Plate 1-b) shows the 16 releases from Figure 3 using 3D wheel glyphs. The 3D wheel shares the advantages of the timeWheel over linear ordering methods. However, unlike the timeWheel where the dominant trend is perceived through the global pattern formed by the series, the 3D wheel allows users to perceive the dominant time trend through its shape. As a result it is easier to identify overall time trends using the 3D wheel. It is however harder to identify divergences because of occlusion and perspective. Even though there is occlusion in the 3D wheel, it is still a lot less than if we were to lay out the time series over the z-axis.

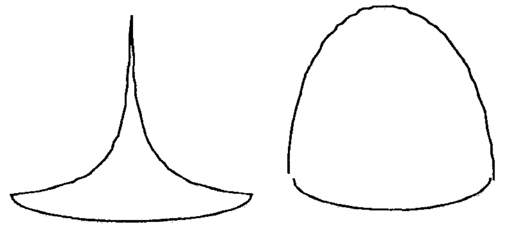


Figure 8: Left - increasing trend (sharp apex); Right - decreasing/tapering trend (balloon)

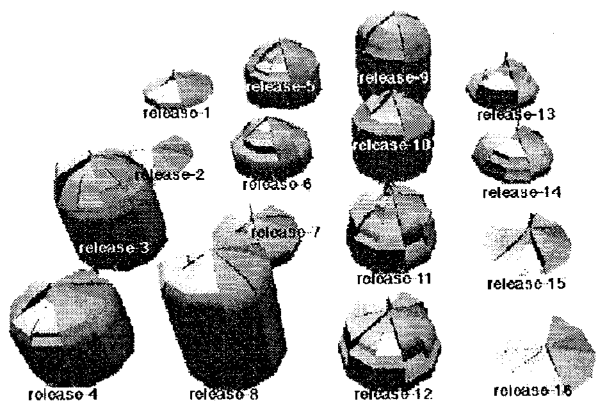


Figure 9: 3D wheel interface of the 16 software releases shown in Figure 3 (Plate 1-b)

3 Viewing summaries with INFOBUG

Associated with any reasonably-sized software project are diverse data sets involving the developers, files, software releases, etc.

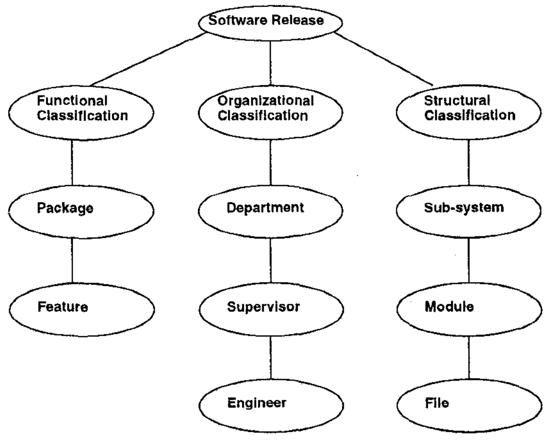

Software Release
Functional Classification
Organizational Classification
Structural Classification
Package
Department
Sub-system
Feature
Supervisor
Module
Engineer
File

Figure 10: Software Hierarchy

Physical constraints (e.g. screen space) and cognitive constraints (e.g. short-term memory) make it unfeasible to view all this data at once. To address this problem, we partitioned the data hierarchically into three classes of software artifacts (Figure 10) and used it to construct a scatterplot interface (Figure 11).

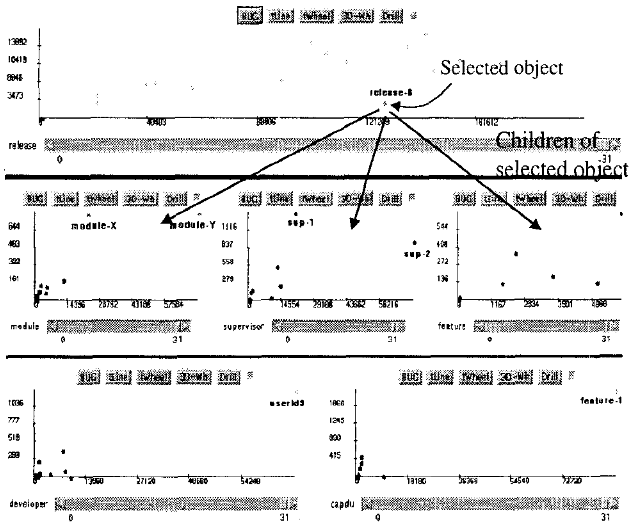


Figure 11: Scatterplot interface

On the x-axis of each scatterplot we encode the number of code lines (*loc*) and on 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. Objects may be selected in the scatterplot by using a bounding box swept out by the mouse. Once selected, the children of the objects may be viewed at a lower level in the scatterplot hierarchy (Figure 11).

Viewing objects hierarchically helps alleviate some of the data scale issues. Even so, it is often the case that within each level of the hierarchy summarization is still needed. The InfoBUG glyph shows many properties simultaneously in a small footprint such that patterns preattentively "jump out" at the user.

Four important classes of software data are represented by the infoBUG head, tail, wings, and body. As is shown in Figure 12 the number of code lines and number of errors are assigned to the bug wings, the code type consistency to the head, the number of changes to the bug tail, and the component size to the body.

The infoBUG glyph is interactive and through animation can show the information at different times within the project. Clicking on the wings selects a time-slice causing the head, body, and tail to update. The selected time slice is indicated with a red band on the infoBUG wing. The time component for all infoBUG glyphs can be changed simultaneously by using the slider at the bottom of the interface (Figure 13).

Lines of code vs. number of errors ratio: This is an important measure for determining problems within project components and is encoded within the wings of the infoBUG. Each wing is a time series with time running from top to bottom. The x-axis on the left bug wing encodes the number of code lines and the x-axis of the right bug wing encodes the number of errors. Usually increases in code bring about comparable increases in number of errors. This results in symmetrical insect wings.

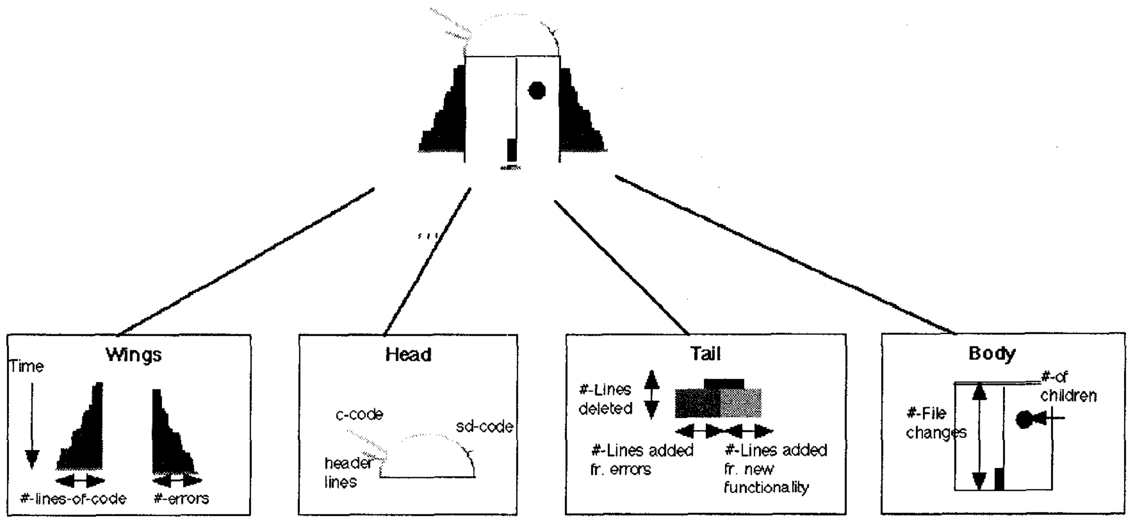


Figure 12: InfoBUG glyph

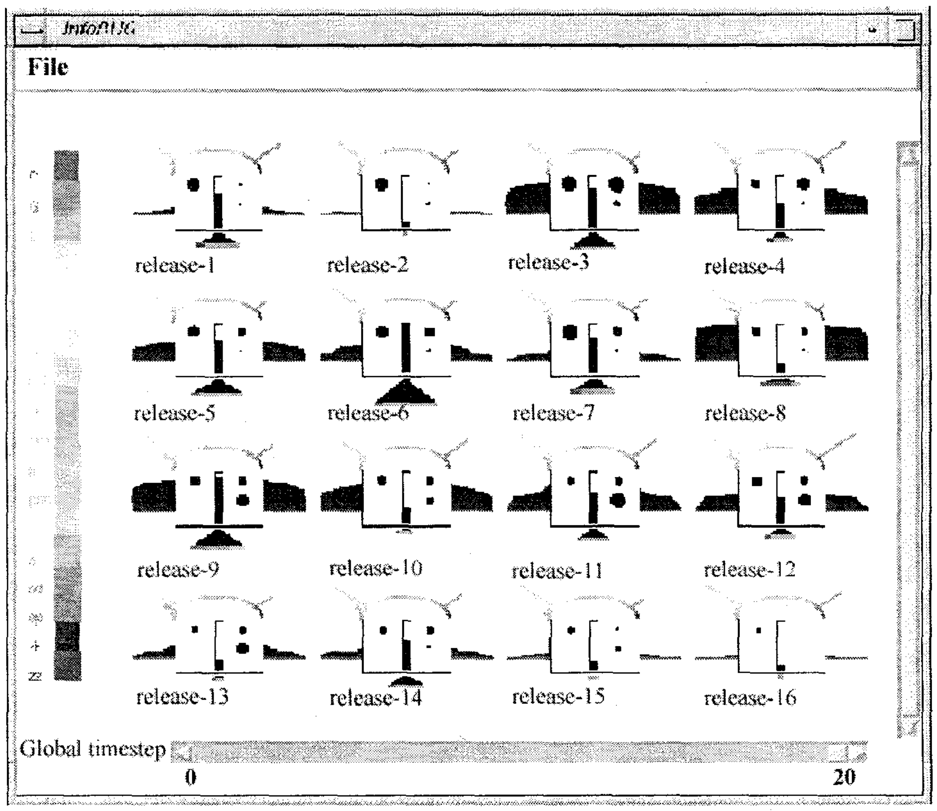


Figure 13: infoBUG interface of the 16 software releases shown in Figure 3 and Figure 9 (Plate 1-c)

Increases in code that are not accompanied by similar increases in errors may imply that the component is not being well tested. On the other hand increases in the number of errors that are not accompanied by similar increases in code could mean that the existing code is inherently difficult, has architectural problems, or is poorly written and in need of re-engineering. These cases are represented visually by non-symmetrical wings which are easy to identify using infoBUGs.

The position of the wings (whether starting at the top or bottom) indicates the time at which the project was started while the shape of the wings shows whether the number of code lines and the number of errors found are increasing, decreasing, or static with time.

Code type consistency: A particular software component may consist of several different types of code. For example to implement the glyph systems described in this paper (Figure 3, Figure 9, Figure 11, and Figure 13) we used Java to specify and control the interfaces, Perl to extract and process the data, and VRML to render the three dimensional wheel glyphs.

The code consistency of software components indicate the components' capabilities and purposes. For example knowing that VRML is used suggests three dimensional representations. Examining such data might also show changes in development practices and in the requirements for a software component.

Code type consistency is encoded by the infoBUG head. The head shows, for a given time slice, 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 interface in Figure 13 (Plate 1-c) For example the software component in Figure 12 is made up of C code, SD (State Definition) code, and header lines.

By interactively changing the time component we are able to obtain information on how the different code types evolve. Such changes give us hints about the changing needs of a software component.

Information on changes made: The bug tail is triangle-shaped. Its base encodes the number of code lines added and its height encodes the number of code lines deleted. The tail base is further divided into two parts: code added due to error fixing (color coded in red) and code added for new functionality (color coded in green). Figure 13 (Plate 1-c) 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 number of code lines added to lines deleted. A short squat triangle like the one for *release-8* shows a high added to deleted ratio. The shapes of the triangles for most of the other releases are less squat indicating a lower added to deleted 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 13 show this property.

Size of components: The size of a component is often important as it reflects the extent to which a component affects the project. Component size is encoded in two ways: through the number of altered files and through the number of child objects a software component contains. The bar in the middle of the infoBUG body shows the absolute number of file changes. The size of the black circles on the body encode the number of child components that are contained within the current objects. The type of child objects encoded depends on the software hierarchy of the system being analyzed. Our system for example, is based on the software hierarchy shown in Figure 10.

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 component may be wide in certain respects and narrow in others. For example *release-1* and *release-2* in Figure 13 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).

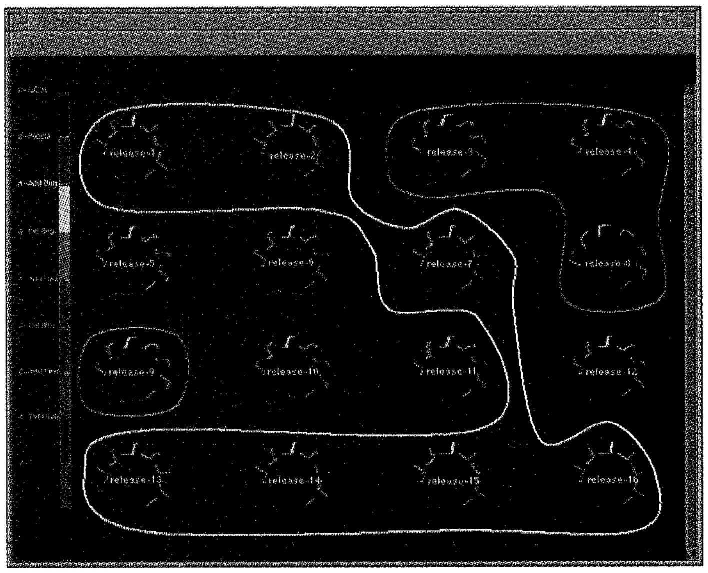
Conclusion

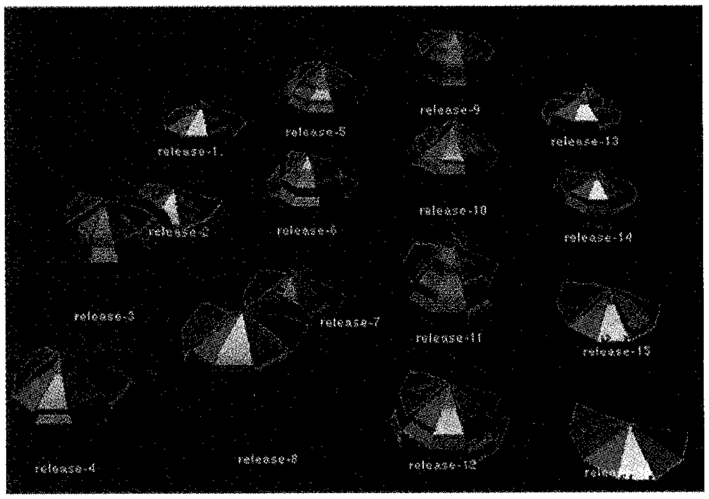
We have developed three novel representations for dealing with project management data. These representations address important issues in project management namely: time dependent data, large data sets, diversity, and correspondence to real world entities.

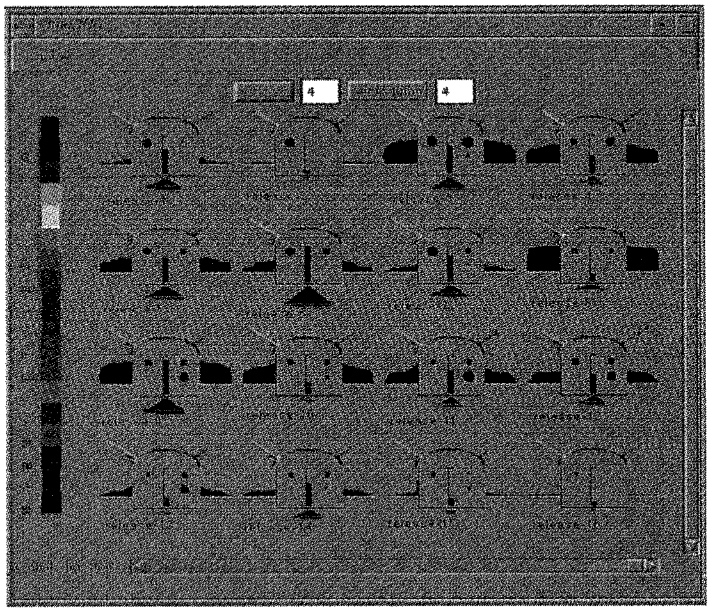
One particularly exciting aspect of this research involves the company Intranet. We are using the corporate WEB as a distribution mechanism to provide access to our visualization. We built our glyphs using Java and VRML and have them running on top of a Netscape browser. Now anyone inside the corporate 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. R.M. Baecker and A. Marcus, *Human Factors and Typography for More Readable Programs*, Addison-Wesley, Reading, Mass., 1990.
2. T. Ball and S.G. Eick, "Software Visualization in the Large," *IEEE Computer*, No. 4, Vol. 29, 1996, pp. 33-42.
3. R.A. Becker and W.S. Cleveland, "Brushing Scatterplots,", *Technometrics*, Vol 29, 1987, pp. 127-142.
4. M.H. Brown, "Algorithm Animation," *in ACM Distinguished Dissertations*, MIT Press, New York, 1988.
5. H. Chernoff, "The Use of Faces to Represent Points in k-Dimensional Space Graphically", *Journal of the American Statistical Association*, 1973, pp. 361-368.
6. 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.
7. C. Hendrickson, T. Au, *Project Management for Construction,* Prentice Hall, 1989.
8. 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.
9. H. Lefkowitz and G.T. Herman, "Color Scales for Image Data", *IEEE Computer Graphics and Applications*, Vol. 12, No. 1, January 1992, pp.72-80
10. 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.
11. 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.
12. B.E. Rogowitz and L. A. Treinish, "An Architecture for Rule-Based Visualization", *Proceedings IEEE Visualization '93*, pp.236-243.
13. Spoehr, K.T., and Lehmkuhle, S.W., *Visual Information Processing*, W.H. Freeman and Company.
14. T.S. Tullis, "The Formatting of Alphanumeric Displays: A Review and Analysis", *Journal of Human Factors*, Vol. 25, No. 6, 1983, pp.657-682.


release-1
release-2
release-3
release-4
release-5
release-6
release-7
release-8
release-9
release-10
release-11
release-12
release-13
release-14
release-15
release-16


release-9
release-5
release-13
release-1
release-10
release-6
release-14
release-2
release-3
release-7
release-11
release-15
release-4
release-8
release-12


4
4