Donald A. Zaremba

Adding a Data Visualization Tool to DEC FUSE

Digital's Data Visualizer tool uses condensed file views to display thousands of lines of source code. These displays can include the output of many other tools. As part of the DEC FUSE programming environment, the tool helps software developers by providing capabilities for displaying large bodies of text with associated events or statistics. The Data Visualizer tool combines the results of other tools into a single display, keeps track of work items, and scales up to support large software projects.

In January 1993, Digital began research on a tool for visualizing large sets of data. The design of the Data Visualizer tool was complete in March 1995, and the tool is scheduled for inclusion with the next major version of the DEC FUSE software. DEC FUSE is a programming environment for UNIX that provides an integrated suite of graphically oriented tools built on the commonly used UNIX programming tools. For more information on the DEC FUSE environment, see the paper "DEC FUSE: Building a Graphical Software Development Environment from UNIX Tools" in this issue.[1]

In this paper, we focus on the technology that was used in the data visualization tool and the process by which this tool was taken from an advanced development project to become a part of an existing product. We start with a discussion of the problems encountered when visualizing large sets of data, the various graphical techniques that are used to solve these problems, and the implementation of these techniques in a demonstration tool. We then describe the design of the final tool, its evolution from the prototype into a product, and its integration with the otherDEC FUSE tools. We then give a functional overview of the tool and scenarios of how it can be used. We conclude with comments on the process from advanced development work into final product.

Development of a Data Visualization Tool

Software development of even a moderately sized project typically involves working with many files and hundreds of thousands of lines of source code. Working with so much data in so many files is difficult because most software tools are written to work on a single file at a time (like a compiler or an editor). Those tools that do operate on multiple files (like agrep tool used with wildcards) produce a stream of output that can be large and that can only be associated with the source code by identifying a line number or by displaying a single line of source in context. Although these tools do provide the requested answer, they provide little of the context that would help the user see how this answer relates to the source code or how it would relate

to other answers. It is often hard to see how these detailed answers fit into the large picture.

One technique for solving this problem is to use computer graphics in the display portion of software development tools. Graphics are used to display information such as build dependencies, cross-reference data, call tree data, and class hierarchies.

Unfortunately, when the application becomes large, the graphic displays become too dense to provide any real insight into the relationships between the components in the application. The screen is simply not large enough to display all the information. The layout of nodes on a two-dimensional display is often inadequate to effectively represent the complexity of the underlying structure and relationships in the code. The common use of overlapping windows of data actually hides data, preventing users from seeing important relationships among the windows or even knowing which windows contain relevant data. In effect, programmers who must work on today's complex software applications are confronted with a situation similar to entering a large dark room with a complicated piece of machinery in it. Current technology hands the engineers a penlight and says figure out what the machine is, how its parts work, and then make enhancements to it.

The Data Visualizer tool addresses some of these problems by providing a condensed view of source code; the tool is capable of displaying thousands of lines of code in a single view. This condensed display is used as a backdrop for showing the output from tools and how it relates to the source code. Figure 1 is a sample screen output from the Data Visualizer tool being used in conjunction with a search tool to find occurrences of a particular string. This simple example shows many of the features of the Data Visualizer. The rendering of each file in the view shows the indentation of the source code. Source code is colored to show comments in green, the beginning of functions or procedures in red, and the actual code in gray. The sizes of files and functions are readily apparent. The results of the search inquiry are highlighted.

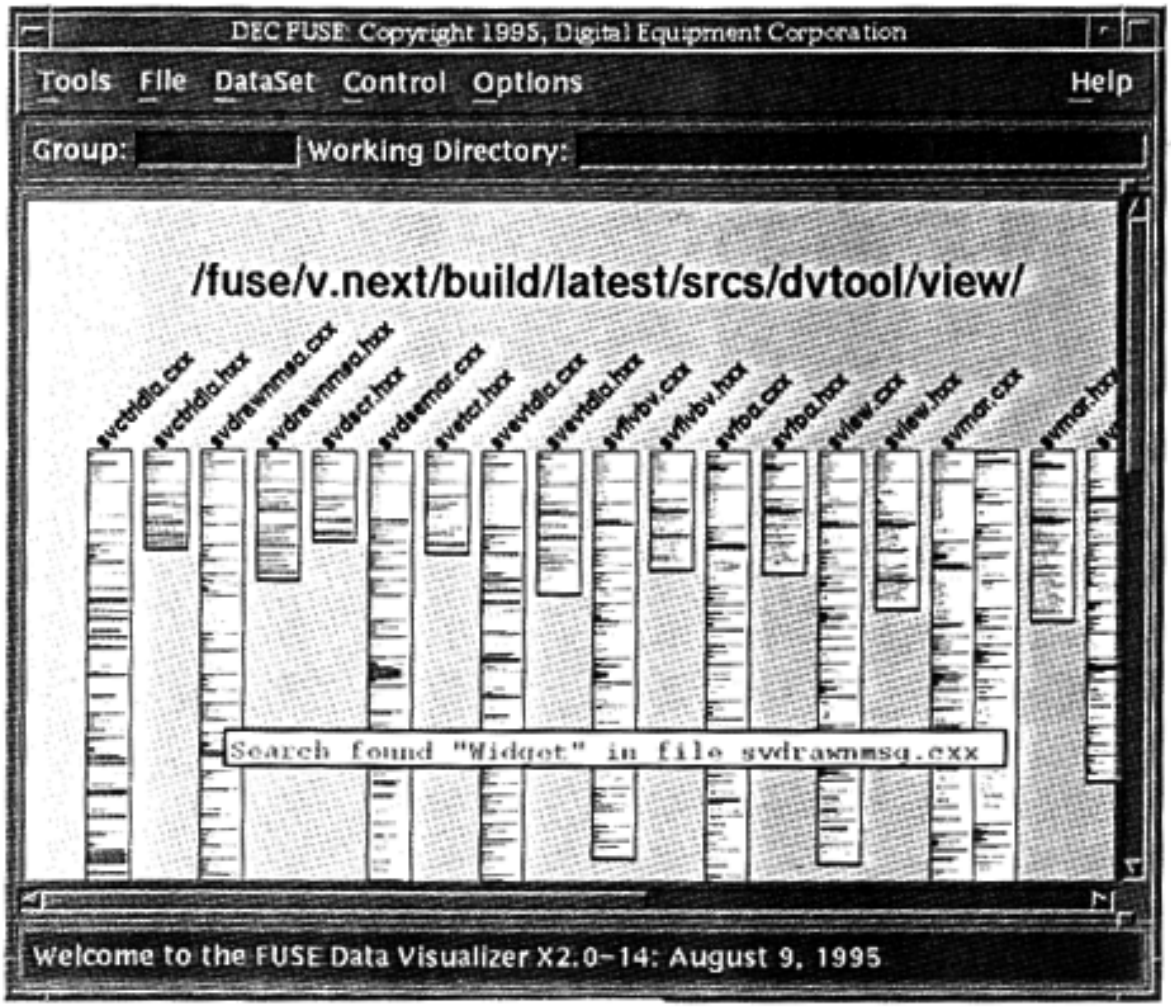


Figure 1
Main Window of the Data Visualizer

Graphical Techniques

During the early phases of this work, research was done to find appropriate graphical techniques. This section describes in detail three techniques that influenced our design and appear in some form in the Data Visualizer tool. It also gives references to related work.

Condensed File View

One technique that looked promising from the very beginning was the condensed file representation done by Stephen Eick in 1993. In his paper "Graphically Displaying Text," he describes a program called SeeSoft that is used to display statistics associated with lines of text.[2,3] He has used this technique to show statistics about lines of program source code and other text files, such as text from the Bible or revision history of text paper. He also uses the technique to analyze computer log files and describes that work in a separate paper.[4]

The idea behind the SeeSoft program is to create small pictures of files that reveal information about a file in a nontextual manner. The size of the rectangle is scaled to the number of lines in the file. Each line of text is shown with the correct indentation and length. In addition, lines can be color-coded either to reveal program structure or to highlight some point of interest. As an example, green lines could be used for comments, red lines to indicate the start of each function, and gray lines for executable code. As can be seen in Figure 2, the information reveals the size of each file and some information about the file contents. It is easy to see where function definitions begin, because the red lines stand out. Also, the indentation of the code helps the viewer recognize programming structures like if then else statements or case statements.

One of the appeals of this method was the ability to display many lines of source code. (Eick's SeeSoft tool claims to display as many as 50,000 lines of code.) Programmers can get a clear and complete overview of their code. From the simple view shown in Figure 2, with no additional data, we can see the size of each file, the relative size of individual functions in a file, and the frequency and distribution of comments.

Multiple Levels of Details

We investigated a second technique that seemed appropriate: the drawing of objects in multiple sizes and in multiple levels of details. The concept of adjusting the amount of detail presented to the user as a function of the apparent size of an object is a technique developed in a unique computer interface model called Pad.[5] Pad provides an infinite two-dimensional information plane that the user can browse using portals (analogous to magnifying glasses) to zoom into the data.

The larger the object, the more details are revealed. This corresponds to the notion that things that interest us are the ones we bring closest to us; they require the greatest amount of detail. Those items of lesser interest are placed in the background and drawn smaller. As can be seen from the pictures in Figure 3, as the size of the file increases, more details are shown about the file. The smallest picture reveals only the major structural parts of the file; we call this *chunk* level. Each chunk is drawn as a colored rectangle and represents either a group of comments (green), the start of a function (red), or lines of executable source code (gray). The next picture shows line-level detail like that shown in Figure 2, and the last picture shows each line large enough to be drawn as readable text. Note also that the largest picture begins to look like a text editor and that the scroll bar on the right is a chunk-level rendering of the file.

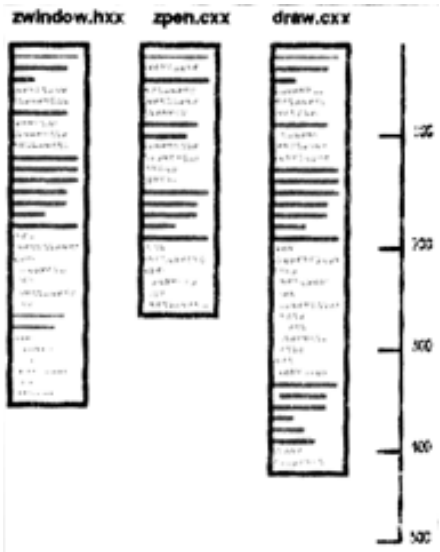


Figure 2
Condensed File View

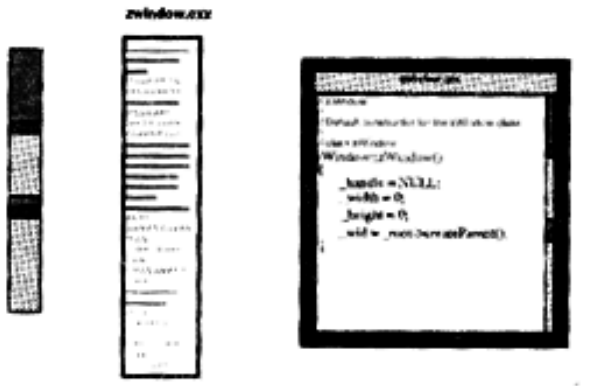


Figure 3
Multiple Sizes of Files

The Use of the Third Dimension

We also chose to investigate the use of the third dimension for ways to better visualize large, dense graphs. We did not pursue this work for several reasons, which we describe later in this paper.

We did find a simple use of three-dimensional (3-D) viewing that was beneficial when trying to visualize certain types of data. We converted the condensed file pictures into 3-D views by adding a small side to each picture. We could use that area to show line-related data as in Figure 4. This example shows a numeric value (the blue lines) associated with a line of source code. The horizontal dotted line is a threshold, and values that exceed the threshold are drawn in red. We use this type of graphic to show source code profiling data, like execution counts and CPU time. Even though it is a simple drawing, it uses a 3-D effect that helps the user visually organize a great deal of information. It is relatively easy for a user to look at the front data at one moment and put the side data off into the background, and then change focus and examine the side data. The effect is even more noticeable and useful when many of these 3-D file pictures appear in the same display. An example of this is given later in the section on the SoftVis Program.

The Advanced Development Project

This section describes the advanced development phase of the project. It discusses the process used, the software prototypes produced, and the major design decisions made during this phase.

The Advanced Development Process at Digital

The type of work done in Digital's Advanced Development Group, working with new technologies and implementing new ideas, is difficult to do within a schedule-constrained product development organization. Although the goals of advanced development work may be well specified, only a vague idea of a possible solution and of the time needed to find the solution is known. These two facts make it impossible to schedule advanced development work in a product's project plan. At Digital, the Advanced Development Group is a separate organization that operates outside the product schedule constraints of other groups. It is staffed by engineers from the development groups, who rotate into the Advanced Development Group, perform their work, and then return to their sponsoring group to transfer the technology into a product.

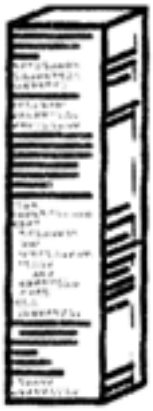


Figure 4
3-D File Picture

The stated goal at the beginning of our project was to enhance the software browsers available in the DEC FUSE product by adapting the results of current research in visualization techniques. Of particular interest was the ability to browse large software systems containing large amounts of source code. We were also looking for techniques that would provide new information about source code and new ways of looking at source code. Our objective was to add features to DEC FUSE that were not currently available in other products.

The process we used was to research as many different techniques as possible and select those that appeared most promising for prototyping. The prototypes gave us experience in the technology and helped us in our evaluation. We then sought input from our sponsoring group to determine which prototypes were feasible to add to the product, and we continued to develop and refine these.

Using 3-D Computer Graphics

At the beginning of the project, we wanted to explore the 3-D graphics technique. For this research, we used a DECstation 5000/20 workstation with a 3-D graphics accelerator option installed. The code was written in C++. We used the Motif standard to build the windows and menu part of the user interface and the programmers hierarchical interactive graphical standard (PHIGS) to write the 3-D graphics code.

We quickly built three demonstration programs to gain experience in 3-D graphics programming. The first program was an instrumented C++ class library that created and destroyed color-coded cubes in 3-D space as constructors and destructors were called. Message passing was shown by connections between the cubes. The *z*-axis was used for time: the older an object became, the farther back it would appear on the *z*-axis. The second demonstration drew hierarchies in 3-D space and gave the user limited capabilities for manipulation in 3-D. The third demonstration visualized a C++ class as a cube in 3-D space, with different sides being assigned different types of data. One side

contained a class inheritance graph, another contained a condensed view of the interface to the class, and the third side contained a window into the source code of the class.

After a short period, for several reasons, we stopped working with 3-D graphics. We realized that the types of visualizations we were doing would require 3-D accelerators on users' workstations, and we knew that would not be acceptable. In addition, development of this technology would take a great deal of time, and we felt we could make better progress working on other graphics techniques.

Early Prototypes

Having seen the work done by Stephen Eick, we decided to experiment using his technique. We also started to think about the concept of building a framework that we could use to build prototypes of different techniques. Eventually, this evolved into the design we describe later in this paper. At this time, we also considered what platform to use. Our sponsoring group had developed the DEC FUSE product for the UNIX environment, but other groups were starting to work on the Windows NT operating system for personal computers. Since we were interested in learning more about the Windows programming environment, we decided to produce code that would work on either platform and to build prototypes on both platforms. In hindsight, our decision to support multiple windowing systems was a diversion that did not directly contribute to the project goals, but it was a valuable learning experience.

To achieve cross-window system portability, we developed a class library that encapsulated parts of the programming interfaces on the MS Windows system and the X Window System. We decided to restrict our class library, collectively referred to as the "ZWindow" or "ZWIN component," to encapsulate only the low-level graphics drawing routines (e.g., line and rectangle) and avoid trying to encapsulate all the graphical interface components like windows, icons, and menus. We encapsulated at the level of the graphics device interface (GDI) on MS Windows and the X library interface (Xlib) on the X Window System. This worked well; we achieved portability of our graphics drawing code, which was our area of concentration. The fact that we had to do separate implementations for the remainder of our user interface (that is, the menus, toolbars, and dialog boxes) was not a hindrance since the bulk of our code was still portable.

Designing the ZWIN interface was fairly straightforward. The line and shape drawing routines were easy to encapsulate because they existed on both platforms. The drawing contexts were different. The MS Windows system has color pens and brushes to control drawing attributes; but on the X Window System, all drawing attributes are defined in a single data structure, the graphics context (GC). We decided to create classes for pens and brushes and to handle the X Window System implementation by encapsulating an appropriate GC in the pen and brush classes. The largest class in the ZWIN component was the canvas class. It encompassed a DrawingArea Widget on the X Window System and a window on MS Windows. It had member functions that provided all the drawing functions available (e.g., line or rectangle), as well as functions to select the appropriate drawing object (pen or brush).

The condensed file view was implemented in two sets of classes. A set of file-type–dependent scanner classes was developed to handle the parsing of C, C++, Ada, makefiles, etc. Once scanned, a single file visualization class could perform the rendering of the object on the display. Speed was a concern since we wanted to be able to visualize an entire directory of files very quickly. To do this, we wrote a small, efficient scanner for each type of file that could pick out only the relevant information as quickly as possible. Throughout our work on all the prototypes and into the final product, we found that we could always fill a complete display without any noticeable delay to the user.

Figure 5 shows part of the first prototype. It displays a condensed file view of all the text files in the default directory. Files were sized to fit within the size of the window, with an appropriate level of detail shown. Files could also be individually selected and resized. Files are shown in the three different levels of detail described in Figure 3. Most of the files are drawn at the chunk level and reveal only the relative size and location of each function in the file. Two of the files have been enlarged to show line-level details, and one file has been fully enlarged to be a readable size.

Later prototypes improved upon the design of this condensed file view. We also implemented other views that we thought would be useful. The C++ class view rendered a condensed picture of a C++ class with its member functions and data members. It is described later in this section.

SoftVis Program

Throughout the process of creating the first few prototypes, we kept in mind the concept of building a framework that we could use to speed up the delivery of new graphical techniques. The SoftVis demonstration program used that design. Based on a View-Object-Tool architecture, its concept was that a view would set the backdrop and style for the display, such as the condensed file view. We would render objects into that view style and support many different types of objects per view. Tools would then be written to

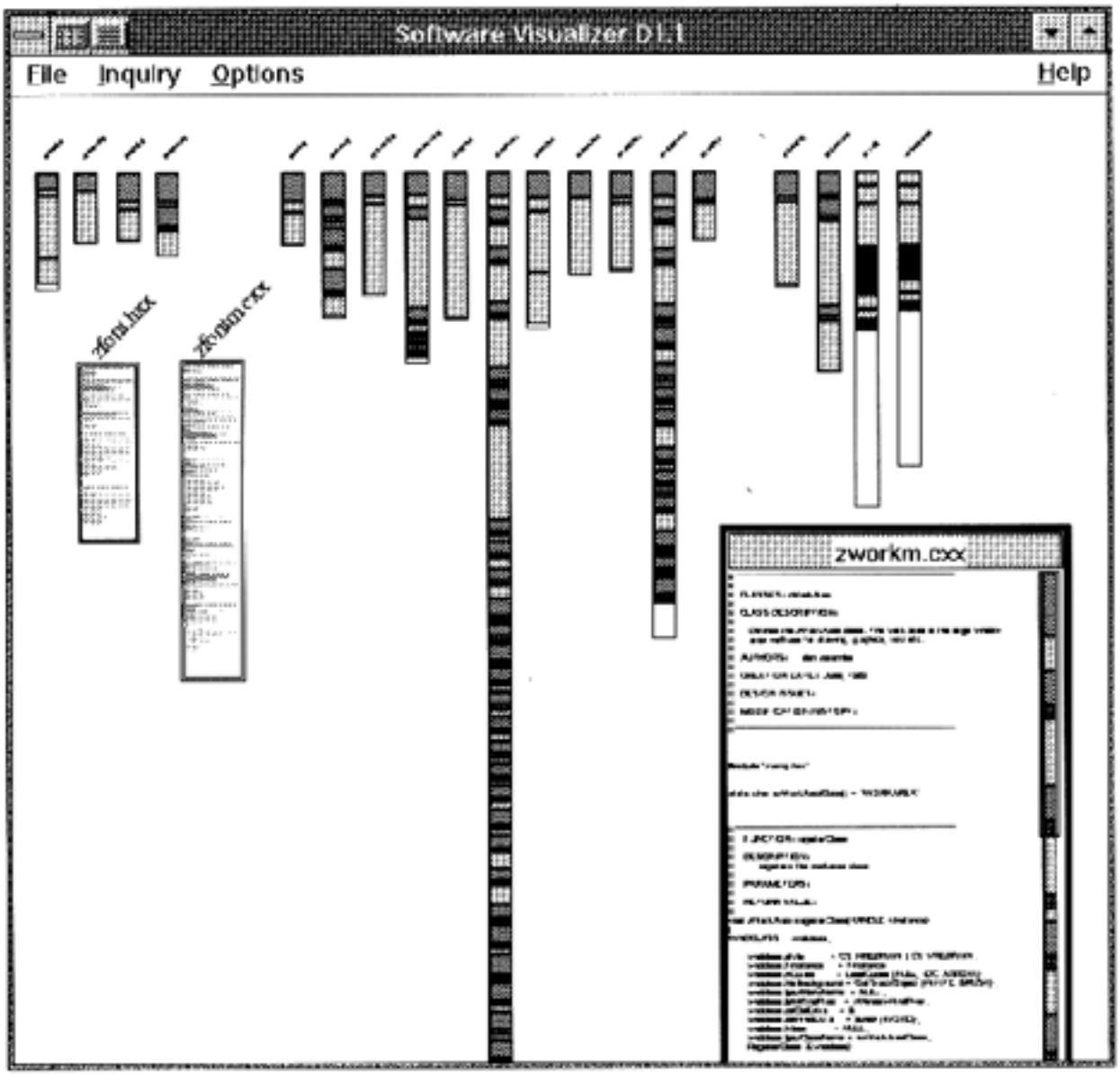


Figure 5
First Demonstration Program

interact with the objects in the view. Our objective was to develop a "plug-and-play" architecture that supported the following:

- View
  - Condensed file view
  - Condensed file 3-D view
  - C++ class view
- Object
  - C++ source code
  - C source
  - Ada source
  - .o (object files)
  - .a (library files)
  - executable files
- Tool
  - Magnify tool
  - Probe tool
  - Cross-reference tool
  - Search tool
  - IF-DEF lens tool

The goal was to be able to create a view containing all the files in a directory and displaying an appropriate visualization for each of the file types (either a text file or a binary file), and to enable the tools to operate on all the objects in the view. For example, the magnify tool would show a readable view of the text in a source file; however, when used on a binary object file, it would show information about the size, address, and type of segments in the file.

Figures 6 and 7 are screen captures from the prototype. Figure 6 shows a cross-reference tool being used on C++ source files. The list box shows functions from all the source programs, and the highlighted function color-coded lines point to where that function is first declared, implemented, and called. Figure 7 shows the magnify tool used in the 3-D file view to show source code details and profiling data. In this case, the profiling data is a mock-up of line execution counts; the real tool will use this space to report actual data.

Figure 8, also a screen shot from the prototype, shows the C++ class view. This view uses a condensed representation of a C++ class. Each line in the class corresponds to either a member function or a data attribute of the class. These are grouped together as public, protected, and private members. Member functions are shown in red; data elements are shown in blue. Inheritance is shown by connected arcs.

SoftVis Design

The system is divided into several components. Each component can be built separately; has its own makefile; and, in most cases, its own test programs. Table 1 gives an overview of these components and their relative sizes as of the latest base level.

The SoftVis design begins by supporting the desired prototype architecture of View-Object-Tool. A component was developed for each of these; it contained a base class, derived classes, and supporting classes.

From Advanced Development to End Product

This section describes the effort required to turn parts of the final advanced development prototype into a product-quality tool for release withDEC FUSE.

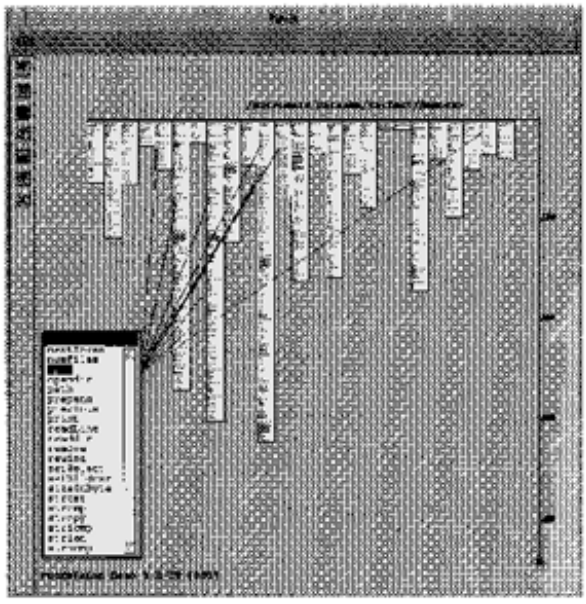


Figure 6
Demonstration of the Cross-reference Tool

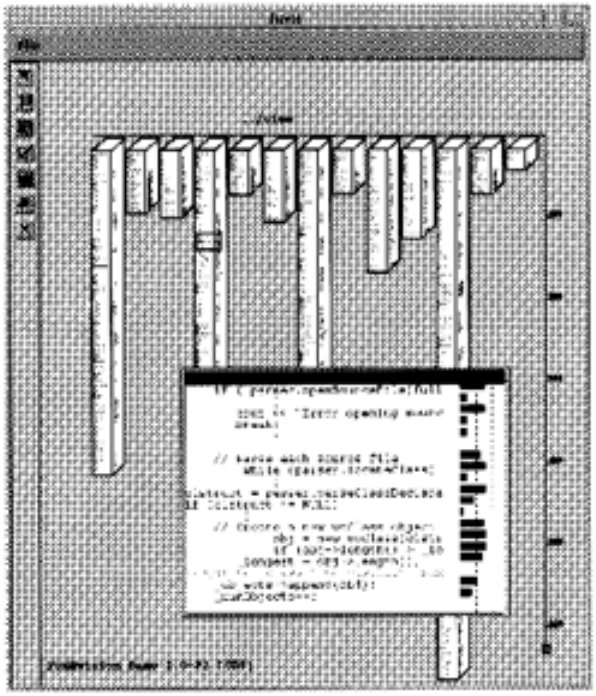


Figure 7
Demonstration of the 3-D View with Profiling Data

Finding a Place for the Work

At the conclusion of the advanced development project, we returned to our sponsoring group and attempted to introduce the data visualization technology into the product. A number of obstacles had to be overcome: The SoftVis program was written in C++, and DEC FUSE had been written almost entirely in C. The requirements for the next release ofDEC FUSE had been gathered, and the goals were set. Where exactly would the new data visualization technology fit into the DEC FUSE product set?

At first we tried to build a class of reusable software components that DEC FUSE tools could use to incorporate the new technology. This would be a set of Motif widgets that encompassed the techniques prototyped in the SoftVis program. Although progress was made on building the widgets, no progress was made incorporating these into any of theDEC FUSE tools. Their incorporation would have required major changes to the user interfaces of these tools, and it was not clear that the benefits would justify these changes.

In hindsight, we realize that the plug-and-play design we used for the prototype did not match the DEC FUSE design of loosely coupled separate tools that passed data by means of simple messages. Although the plug-and-play approach made it easy to add new components into the model, its tightly coupled design made it difficult for us to take parts of that design and use them in theDEC FUSE product.

The proposal that was finally accepted was to develop a new, separate tool, called the Data Visualizer, that

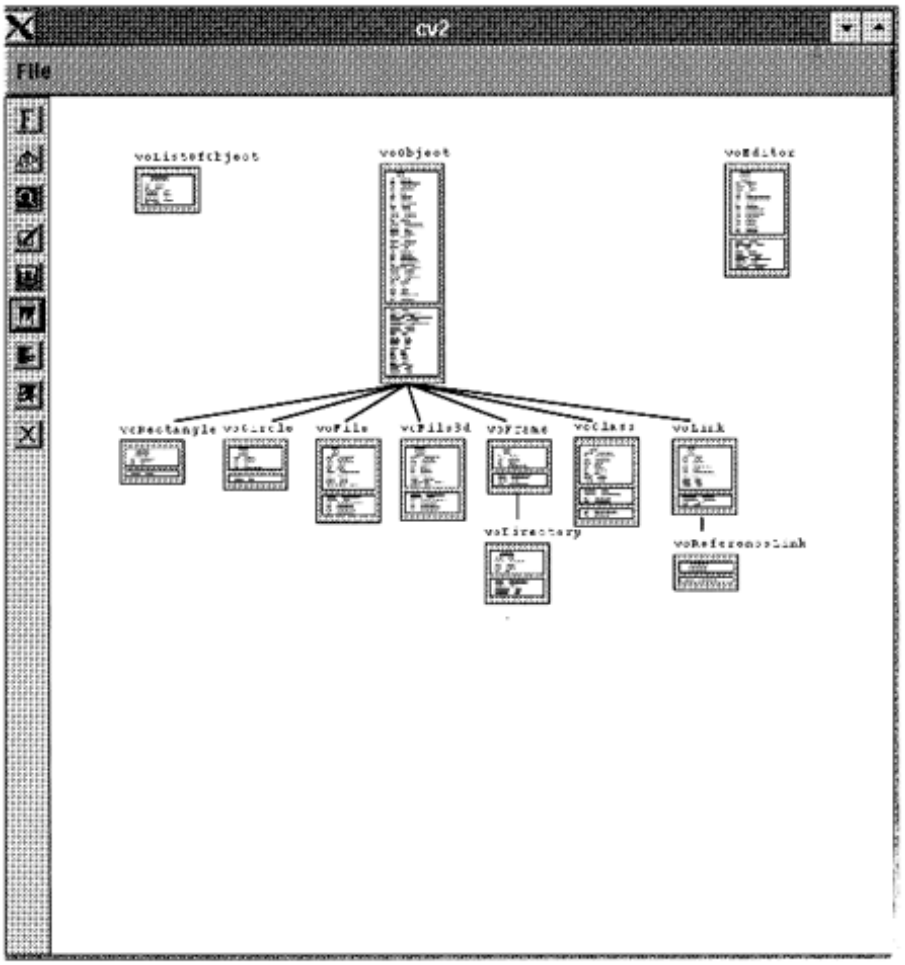


Figure 8
Demonstration of the C++ Class View

Table 1
Components in the Prototype Design

Component	Description	Lines of Code	Classes
VO	Base classes, voObject, and voEditor. Also, voFile class and other classes derived from voObject. Implements features for selecting, moving, resizing, and drawing objects.	5,000	10
TOOL	Base tool class, voTool, and classes derived from it. Includes voLens, voProbe, voMagTool, and voXRefTool.	2,500	10
VIEW	The vBaseView class is derived from voEditor. The three main views of the tool are then derived from vBaseView. The main views are vFileView, vFile3dView, and vClassView. This component also contains executable test programs for each view.	2,400	4
SDM	The software data model component contains the language-specific scanners and parsers. The base class AnnotatedFile is used by text and binary files.	4,500	15
ZWIN	Portable graphics interface. A single class interface for windowing and drawing functions is provided. Two separate implementations of the interface exist, one for MS Windows and one for the X Window System.	11,000	30
UTIL	Various miscellaneous classes for data structures, file access, etc. It also contains an interface to some common operating-system–dependent routines.	3,300	12
Total		28,700	81

would build upon our advanced development work. Building a separate tool had a number of advantages: We could develop a data visualization tool apart from the other DEC FUSE tools. We could implement it in C++ and thus use some of the design from the SoftVis tool, if not the code. The impact on current tools was minimal: only small changes to their user interfaces and an added capability for sending data to the Data Visualizer were needed. By implementing a separate tool that receives messages from other tools, we would be following the style of tool integration used in the DEC FUSE environment.

Many changes had to be made to the prototype to move this work from advanced development into a product. Functions had to be added and removed. The design was changed in a number of places. Some changes resulted from the requirement to follow the tool integration standards for theDEC FUSE product. Other changes were merely good ideas that came about once we started the work of integration.

Data Visualizer Tool

Two major features were added to integrate the Data Visualizer tool into the DEC FUSE programming environment. First, all the data that composed the view was coming from outside the tool, unlike the prototype where data for the view was generated internally by analyzing source files. Now activities performed in other tools would generate this data and send it to the Data Visualizer. Second, multiple tools would be sending data that would need to be merged within the Data Visualizer into a single view. The remainder of this section summarizes the features in the Data Visualizer tool.

The Visualization DataSet File The Visualization DataSet file is used to pass information to the Data Visualizer for display. It contains two types of data. Software component data describes the files, directories, libraries, and functions to be visualized. Event data describes the data to be associated with these components. The types of events are defined in the file by the tool creating the file, but they must adhere to one of the predefined formats. An example of an event is a memory leak detected by a memory analysis tool. In the file, the memory analysis tool defines an event type for memory leaks and then passes as many events of this type as there are leaks detected. By allowing event types to be defined in the Visualization DataSet file, the Data Visualizer can easily support any tool that creates a file in this format.

Each set of events sent to the Data Visualizer from a particular tool is logically grouped into an entity called a DataSet. For example, a single DataSet contains all the results from a single search tool inquiry. Subsequent searches yield separate DataSets.

Condensed File Views In this paper, software components are shown in both the condensed file view introduced in Figure 2 and the 3-D view depicted in Figure 4. Each of these gives the tool a concise, information-dense representation capable of displaying up to 30,000 lines of source code. Program structure is revealed by the indentation and color coding.

Event Highlighting, Filtering, and Tracking Events in the DataSet are highlighted on the screen in a number of ways. Event types are assigned a color, and that color is used to color the line of the associated event. The coloring can occur in the foreground of the line or the background. Once a user's attention has been drawn to the line, the user can obtain more information about the event at that line from the small descriptive window that appears whenever a hot cursor is moved near that line. Figure 9 shows an example produced by the Data Visualizer tool. In addition, when the event contains more information than can be displayed on a single line, for example, when a complete program call stack is logged with the event, a separate window appears with this information. This is also shown in Figure 9.

The tool's legend/filter control window shown in Figure 10 serves the dual purposes of providing a color key to the events that appear in the view and a mechanism for toggling on/off the appearance of events of a particular type. This control window also allows the user to toggle on/off the appearance of all the events in a DataSet. When multiple DataSets are present, they are placed on top of each other. Each DataSet can be thought of as a transparency that contains only the event's highlighted coloring. These transparencies are stacked on top of each other (the user can control the ordering) to show all the events together.

The Data Visualizer also provides a mechanism for keeping track of events that are seen or unseen by the user. This feature can be used when there are many events to examine and the user needs assistance in tracking what work has been finished and what remains to be done. This information can be saved between invocations of the tool so that a user can put this work aside and come back to it at a later date.

Merging DataSets As mentioned earlier, one of the important features that was added was the ability to merge the data received from multiple tools into a single displayed view. This allows the combination of the results of two or more tools that normally could not be merged or even know of each other. For example, the output from a memory analysis tool that shows where memory leaks occur and their size can be combined with the output from a search tool that locates the occurrence of a function name in the program.

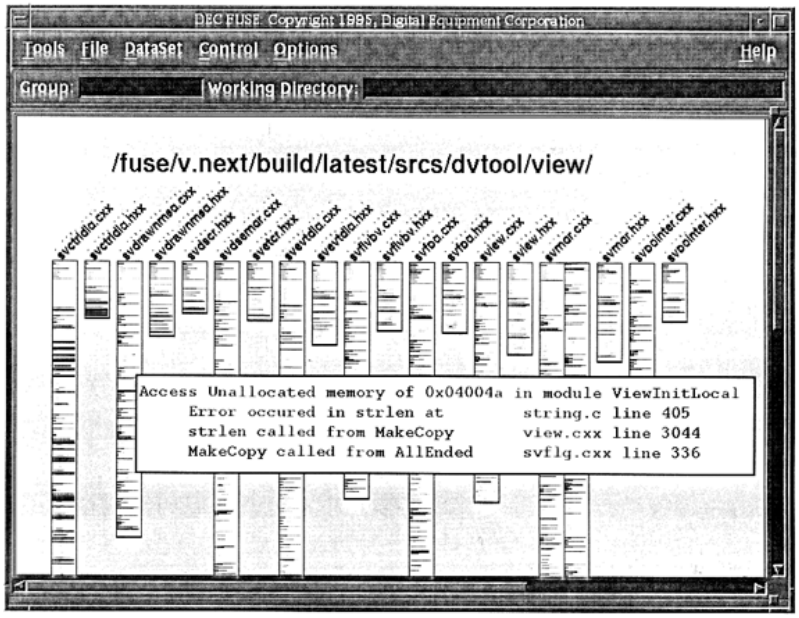


Figure 9
Highlighted Event with Call Stack

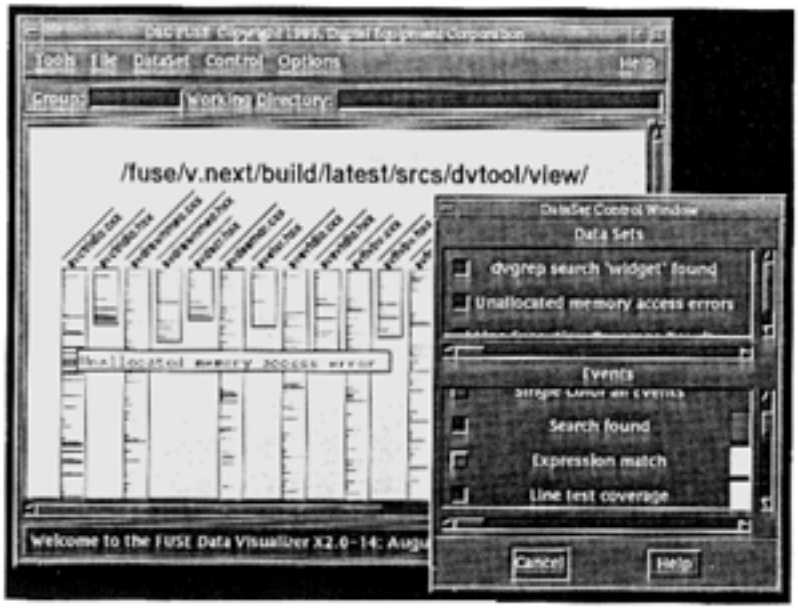


Figure 10
Event Filtering

The tool uses a number of methods for merging DataSets, and the type of merge that is performed depends on the types of events. The simple transparency model described earlier explains how events can be additively combined to display the sum of all events. In this model, when two or more events are associated with the same line in a file, they are treated as separate events that pertain to that line. For some event types, however, this is not the case. The tool supports the combination of same line events in different ways. For example, two runs of a performance analysis tool generate line execution times that can be combined by averaging the execution time values to give the user a reading on the average performance of the code. As an alternative, these same two events can be combined by creating a new event that shows the difference of the execution times to reveal improvements that may have occurred between runs.

Integration with Other DEC FUSE Tools The Data Visualizer is well integrated with the other tools in the DEC FUSE programming environment. The profiler, the heap analyzer, and the search tool all have the ability to send data to the Data Visualizer at a user's request. The Data Visualizer makes good use of the DEC FUSE editors to examine source code in detail. From within the Data Visualizer, the user can double-click at any point in any of the displayed files to have that source loaded into their preferred editor. This capability is shown in Figure 11, where the results obtained from the search tool are used to create a view in the Data Visualizer and load files into the editor.

Revised Design

As seen in Table 2, some of the prototype components were reused in the final product design. We changed the SDM component internally to handle more data, but we retained the basic design. We also retained the design of the UTIL component. Since portability between MS Windows and the X Window System was no longer a concern, we redesigned the ZWIN component into the WinDraw component. Due to this change, the size of this component decreased by 7,600 lines of code.

In addition to modifying components, we developed three new components. The FUSETool component handles the code common to all the DEC FUSE tools. It contains abstract base classes that can be used to derive new tools. The DVTool component contains the main program and the bulk of the user interface code. The View DataSet File (VDSF) component provides functions for reading and writing these files. It contains class libraries for C++ programs and C routines.

Note that this design maintains some of the plug-and-play characteristics of the earlier design. Although the tool component no longer exists, the VO (Visual Object) and the view components are present and provide extensibility for future objects and views.

Conclusions

The last section gives an overview of the software design from advanced development into final product. The section concludes with some future plans for this work.

Project History

During the process of transferring this work from advanced development into a product, many important features were added to enhance the usefulness of this technology. The final product retained the ability to visualize large amounts of data in a condensed yet comprehensible format; it also included features, like event tracking and DataSet merging, that made it a much more useful productivity tool. Figure 12 shows how the design evolved over time. The events

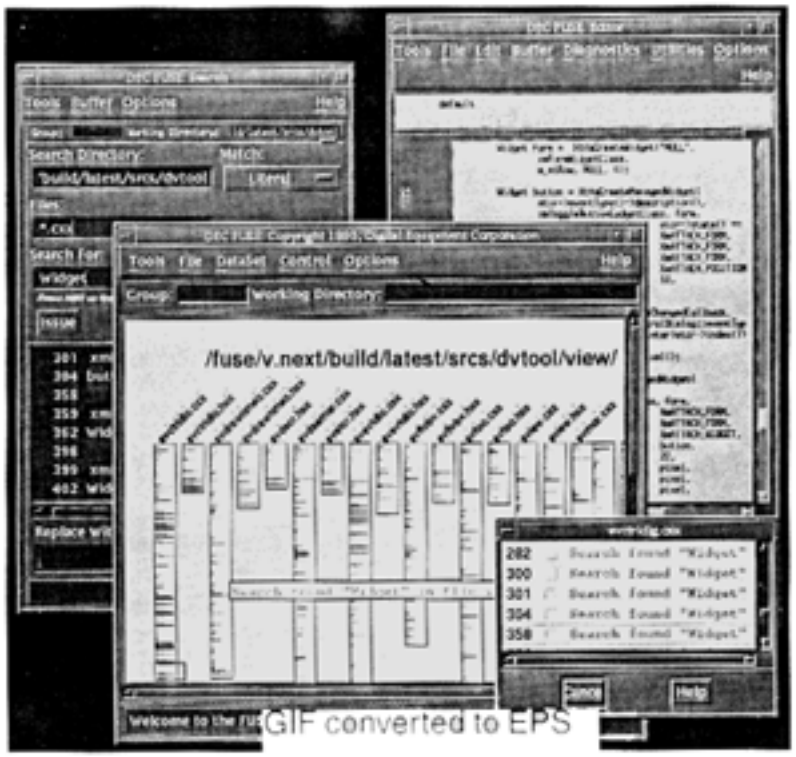


Figure 11
Integration with Other DEC FUSE Tools

Table 2
Components in the Data Visualizer

Component	Description	Lines of Code	Classes
FUSETool	Base class for building a DEC FUSE tool. Contains code common to all DEC FUSE tools.	3,000	8
DVTool	The Data Visualizer main classes. Contains the main program and most user interface classes.	2,400	10
VO	Contains the svObject base class and its derivations, the svFile, the svDirectory, and the svLibrary.	2,000	5
VIEW	Contains the svView class and its derivations, the svFileView and svFile3dView classes.	3,500	8
SDM	Software data model component. Contains the language-specific scanners and parsers. Defines the program's internal data model.	3,500	15
WinDraw	Provides C++ encapsulation of graphics drawing functions.	4,100	12
VDSF	The VisualizationDataSet Format component provides reading and writing routines for this file format.	1,000	4
UTIL	Various miscellaneous classes for data structures, file access, etc. It also contains an interface to some common operating-system–dependent routines.	2,000	8
Total		21,500	70

described in this paper occurred over the course of two years and three months. The advanced development project began in January 1993, and the final design of the Data Visualizer tool was complete in March 1995.

In Figure 12, the rectangles represent software components of the design. A software component is a collection of C++ classes that was designed to accomplish a single function; these components correspond to the design components described earlier in this paper. The oval shapes represent prototypes that were built from these components. Solid arcs connecting components with prototypes show which components were used to build that piece of software. Dotted lines between components show how components evolved over time.

Figure 12 indicates that the work involving 3-D objects and some of the early prototype components were never used. It also shows that the condensed file view component and the ZWIN component did evolve into the final product. Figure 12 further reveals that toward the end of 1994 several documents were produced, but no work was done on the design or any of the components. During this period of negotiation and redesign, the advanced development technology was being converted into a product.

Future Work

We would like to expand the capabilities of the Data Visualizer tool in several areas.

Many of the capabilities for merging DataSets are not available for selection by the user. We would like to extend the tool to have the added flexibility of allowing the user to decide how DataSets should be merged and how events should be combined. For example, the tool might show only the intersection of two DataSets, that is, display only those events that point to a file-line combination that is common in both sets.

We will also consider other ways of displaying in a condensed file format and additional types of files to visualize. The file types might be complete directories shown as a single, condensed object, or shared and nonshared libraries as a single object.

We have an ongoing effort to take the output from existing tools and visualize it in this tool.

Final Remarks

The decision to include the Data Visualizer tool in the next major release of the DEC FUSE programming environment was not an easy one to make. Many important features were being considered, but not enough resources were available to perform the work. Prioritized goals were established, and all work items were evaluated against these goals. The Data Visualizer tool was included for two important reasons. First, it supported the short-term goals of the project by adding features that current tools could use in the upcoming release. Second, it provided long-term benefits by opening up the DEC FUSE product to new capabilities in the area of software visualization. We believe that the presence of both these reasons was necessary for its inclusion in the DEC FUSE product. Had it provided support for only the short-term product goals, it would have been evaluated against the many other short-term work proposals and probably would not have been selected. Had it supported only the long-term goals, it would have been left out for lack of ties to the current tools.

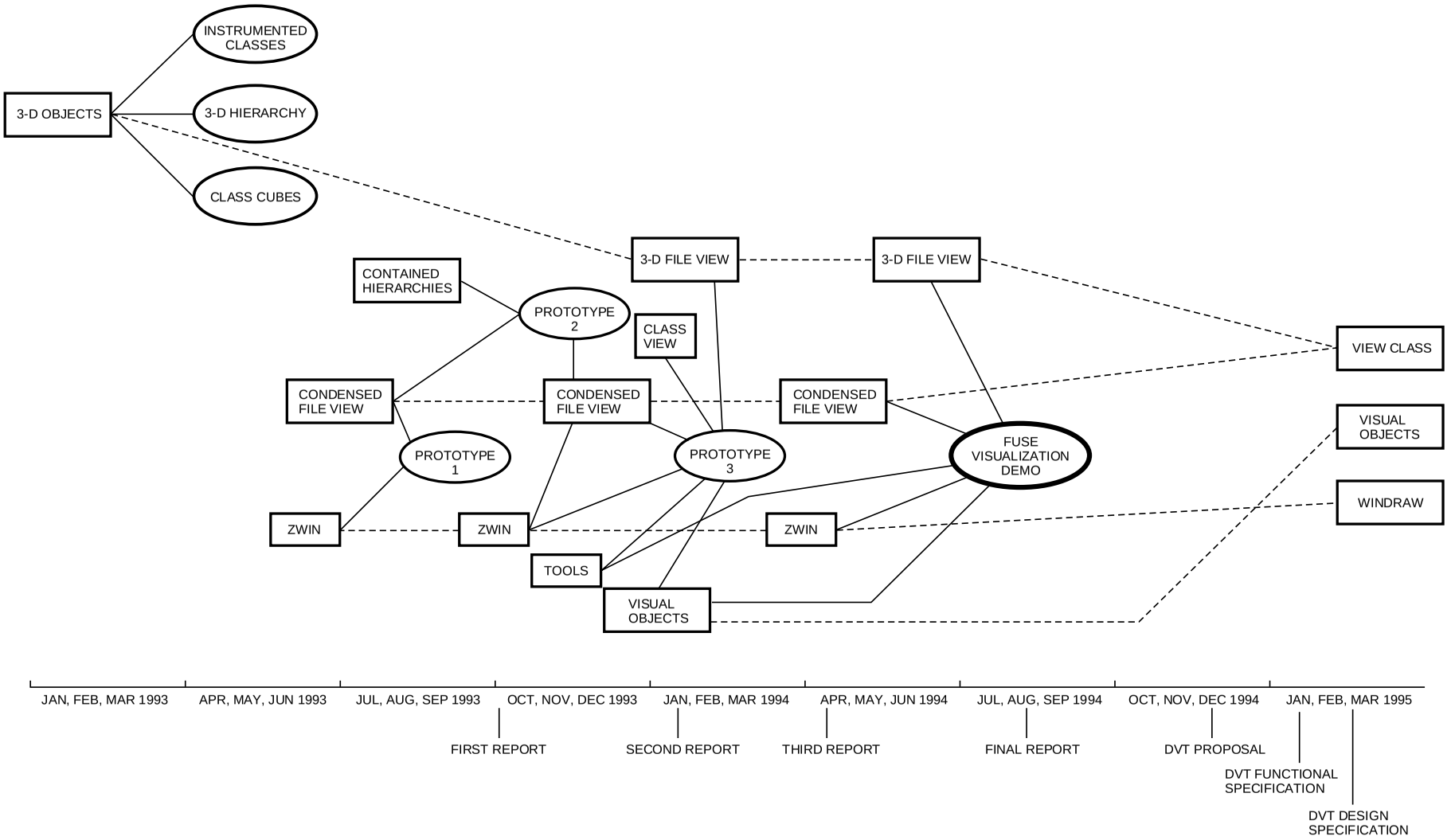


Figure 12
Project History

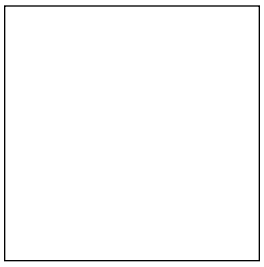
Acknowledgments

I would like to thank a number of people who supported me during this effort: John Ellenberger for his continuing guidance throughout the entire process; Mike Candella for the early work we did together in the Advanced Development Group; Glenn Lupton for his help in deciding how this prototype would fit into the DEC FUSE product; the DEC FUSE management team for supporting and encouraging this work; and finally, everyone on the DEC FUSE development team.

References

1. R. Hart and G. Lupton, "DEC FUSE: Building a Graphical Software Development Environment from UNIX Tools," *Digital Technical Journal,* vol. 7, no. 2 (1995, this issue): 5–19.
2. S. Eick, "SeeSoft—A Tool for Visualizing Line Oriented Software Statistics," *IEEE Transactions on Software Engineering,* vol. 18, no. 11 (1992): 957–968.
3. S. Eick, "Graphically Displaying Text," *Journal of Computational and Graphical Statistics,* vol. 3, no. 2 (1994): 127–142.
4. S. Eick, M. Nelson, and J. Schmidt, "Graphical Analysis of Computer Log Files," *Communications of the ACM,* vol. 27, no. 12 (1994): 50–56.
5. K. Perlin and D. Fox, "PAD—An Alternative Approach to the Computer Interface," *SIGGRAPH 93 Proceedings* (1993): 57–64.

Biography



Donald A. Zaremba
The project leader of the FUSE Data Visualization team, Don Zaremba is a principal software engineer in Digital's Unix Development Environment Group. He was responsible for designing and implementing the Data Visualizer tool. Since joining Digital in 1980, Don has contributed to the DEC Test Manager project and has worked on software development tools and fault analysis tools. He received a B.A. in mathematics from the State University of New York and an M.S. in software engineering from Wang Institute.