A Source Code Navigator

Matthew Exon

This report is submitted as partial fulfilment of the requirements for the Honours Programme of the School of Computer Science and Software Engineering, The University of Western Australia, 1996

Abstract

Graphical user interfaces are now the standard mechanism for communication between users and programs. However, the area of computer programming has been noticeably slow in making the transition to graphical means of presenting information. In this thesis I will investigate some of the possible reasons why graphical user interfaces have been slow to catch on in this area, and present my own C++ source code navigator. I designed the navigator to provide several views on a single program, including a hierarchical organisation of the functions in the program, a display of scope trees, class inheritance graphs, as well as the original source code. The navigator was designed to maximise the programmer's freedom to impose whatever structure they feel is appropriate on their program. This is done through the ability to construct their own organisational hierarchy, the ability to arrange components of the display in arbitrary patterns, and the use of colour coding. The result is an aid to communication of program ideas, as well as aiding the programming process.

Keywords: Source Code Navigator
CR Categories:

Contents

CHAPTER 1

Introduction

1.1 Motivation

Traditionally, computer programs have rarely been represented as anything other than text. Since programming is primarily a process of writing text, text-based systems have seemed to be the most appropriate. However, over the last decade and a half, graphical user interfaces have become prevalent in most kinds of software, and this has led to many experiments in representing programs in a graphical way. These have concentrated on various facets of programming, from the basic semantic structure of the program [16], through the data structures it uses [3], to the overall design plan [6].

Despite the large number of experimental programming tools that have been created, commercial programming environments appear to have been primarily concerned with adding graphical interfaces to the standard text-based commands, rather than fundamentally restructuring the interface. It is obviously hard for dramatic changes to be successful. Programmers would need to learn to use the new interface, and possibly change the way they think about programs. Programmers have a natural conservatism, which is partly due to the desire to ensure that their software is compatible with older systems. Furthermore, software companies already have large collections of tools that are known to work well and be popular, and having to rewrite them all would be a massive undertaking. Nevertheless, graphical user interfaces have been dominant for over a decade now, and in that time other dramatic changes to the programming process, such as the introduction of object-oriented programming and *Java*, have been enthusiastically adopted. It is possible that if a software package with the right combination of features were released, then it may also be taken up with enthusiasm.

When taking on this project, I was primarily concerned with solving some of the problems involved with transferring a traditionally text-based activity to a graphical user interface. However, I was also interested to see if there was any fundamental reason why progress was slow in this area—I wanted to see if programming is naturally a textual activity, and graphical user interfaces were, to some extent, inappropriate.

1.2 Aim

It was the purpose of my honours project to write a source code navigator that will aid in the task of comprehending a large, unfamiliar project. I hoped to use graphical representations of program structure to convey a more intuitive sense of the structure and purpose of the subsystems of the project, and to show the inter-relationships between them. The particular representations that were used included call dependency graphs, scope trees, class inheritance graphs, as well as displays of the original code itself.

When a programmer attempts to comprehend unfamiliar code, the code itself is actually much less useful than the extra information provided by the original programmer—comments, explanatory text files, naming conventions of the files involved, formatting of the code, and so on. Banner comments are used to separate large sections of the program, while blank lines are used to separate sections within a function. The code is the most important way to express to a computer what a program should do, but a programmer will gain an understanding much quicker if other methods are used. They provide a "leg up" to the programmer, so that they can then concentrate on the details of how the goals are achieved, rather than trying to understand what the original goals were.

I believe that the introduction of graphical user interfaces can provide an important additional mechanism for the communication of this kind of explanatory information. Therefore, my program was designed in the following way: the original programmer makes use of my program to create a picture of the program, which may give an indication of their intentions. Then, they save this information in a file, additional to the usual program code. When the code is distributed, subsequent programmers will be able to restore the representation of the code as the original programmer saw it (see Figure 1.1). Hopefully, the new programmer will gain a clearer insight into the original programmer's intentions from the way

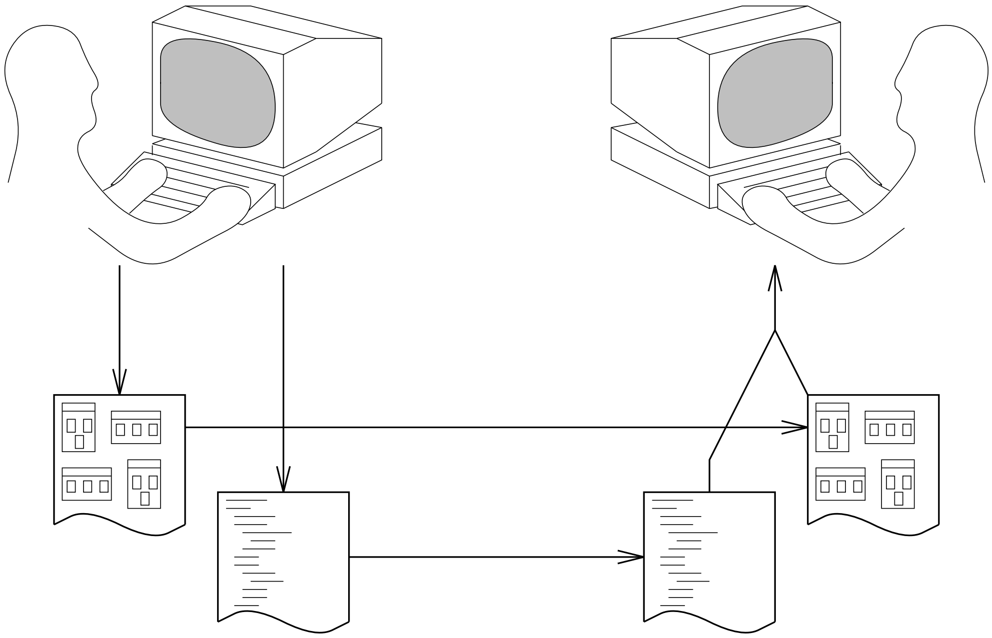


Figure 1.1: Communication between programmers via the navigator

the program is presented. The navigator will also be able to read in existing programs and present them in as structured a manner as possible through simple mechanisms such as grouping together functions that are in the same file.

I chose to make my program specific to the C++ language. This is because C++ is a language that already asks programmers to impose a more logical structure on their programs. C++ classes are logical constructs that group together functions that operate on the same data structures. This provides a useful starting point for extracting and displaying some of the structure of a program. The inheritance of members between different classes is a useful mechanism for defining the relationships between data structures, but it can be rather confusing when presented as simple code files, so the presentation of a class inheritance graph would be a useful role for the navigator to play.

I chose to write the program for the Unix environment, mainly because this is the environment I am most familiar with, but also because it seems to have been the slowest to convert to graphical user interfaces generally. If my program is to have an impact, I would like it to be in this area that needs it most. The particular system I worked on used the Linux operating system, but the navigator should be relatively easy to port to other platforms.

Since this is a heavily graphical program, it was essential to use a standard graphics interface—the X window system is the most frequently used, and the one I adopted. I used the freely distributable xview system to provide a high level interface to X, because it provides high level features such as file choosers while still allowing close control of the way the graphics are displayed.

1.3 Method

The program requires two parts—a parser to extract information about the contents of the program, and a graphical user interface to display the data.

1.3.1 The Parser

Rather than write my own parser for this project, which would obviously be a huge undertaking, I decided that it would be far more sensible to make use of an existing parsing system. My program requires only a small subset of the information that a full parser extracts from a program, and so the criticism could perhaps be made that making use of a full parsing system is a gross inefficiency. However, using an existing parsing system, and minimising the changes that need to be made to it, makes my program more flexible. If the language is changed or spawns new dialects in the future, or the parser requires bug fixes, it should be relatively simple to use my program with the new version. Also, the use of an established parser means that there are unlikely to be bugs present, and the fewer changes made to it, the less chance that new bugs will be introduced.

The system I chose was the GNU C compiler [14]. The gcc has established itself as the standard C compiler for Unix systems, and so its behaviour will be familiar to the vast majority of programs, and it should work with the vast majority of programs. The variant of *gcc* for the C++ language is called *g++* . The disadvantage of *g++* is that it is such a large and complicated system, and hence extracting just the components that are required for my program to work is that much more difficult. Rather than adopt that strategy, I decided simply to modify *g++* to send data to my program via a Unix pipe, while maintaining its role of providing full compilation of the program all the way to an executable program. This would mean that the user would not need two versions of *g++*—they could simply replace the existing version of *g++* with the modified version. The navigator relies on the program using the standard Unix *make* utility, by having a makefile which controls how the program is compiled. When the navigator needs to read in a program, it can make small modifications to the makefile, so that any invocations of *g++* are given options that signal to it that it should output data, and specify a pathname for the pipe that should be used for the output. It then invokes the makefile, which compiles the program to the final executable while simultaneously sending data to the navigator.

1.3.2 Presenting the Information

There were four mechanisms that I decided would be most useful to convey program information:

- Function group windows. A Function group is a new construct independent of the actual program code. It represents a group of functions and/or other function groups, rather like directories in standard file systems, but with files being replaced by functions. Each function group can be displayed as a window, which contains labelled icons representing the contents of that group. This provides a simple interface to the basic building blocks of the program, and allows the user to arrange these in a simple and flexible manner. The ability to colour-code icons in any manner desired also provides a mechanism to impose order on the representation of the program.

- Code windows. These provide access to the underlying code of the program, and display in text form the code for the individual functions used in the program. Programs are inevitably going to be represented as text at some level, and so it is important that the programmer can relate this text to the more abstract representations used elsewhere in the navigator.

- The variable window. This presents a tree structure representing the files and the scopes within them that make up the program. Clicking on a node in the tree brings up a list of the variables declared within that scope. This provides an alternative hierarchical representation of the program which uses a completely different definition of program structure.

- The class window. This presents a graph structure representing the classes in the program and the inheritance relationships between them. Clicking on a node brings up a list of the members of the relevant class. This makes use of the existing structure imposed by C++.

Selecting an item from any window will, where appropriate, highlight the item in other windows. Thus, these windows provide an integrated collection of views on a single program, each suited to different tasks. The views allow the user to quickly home in on particular areas, switch quickly between different areas of the program, and simultaneously view parts of the program that may be widely separated in the actual code. The individual components of each view can be related to each other easily, so that the user is not confused by the different representations available.

CHAPTER 2

Literature Review

2.1 Models of human comprehension of programs

Before I constructed the navigator, I needed to understand some of the psychological aspects of program comprehension. My task is to improve the efficiency of a programmer who is trying to understand the behaviour and construction of a large, unfamiliar program.

2.1.1 Debugging another programmer's code

Waddington and Henry [23] considered the situation in which one programmer writes a program and the other has to debug it. They proposed a model of the debugging process which revolved around three ideas: the predicted program behaviour, the actual program behaviour, and the intended program behaviour. They suggested that before a programmer can start to make predictions about the way the program will operate, some insight is required into the intentions of the original programmer.

To verify this model, they set up an experiment in which experienced programmers were given a small program (concurrent quicksort) to comprehend. In the first phase of the experiment, a version of the program that contained two bugs was presented to the programmers, and they were asked to debug it. During this phase, the programmers constructed their mental model of the program. In the second phase, the programmers were asked questions about the actual behaviour of the program, and about the original intentions of the programmer. Waddington and Henry discovered that there was no significant difference in the time taken to answer these two types of questions. They concluded that the programmers had obtained as much information about intentions as actuality, and that they had used their conception of the intentions of the programmer as part of their debugging strategy. The important conclusion for my purposes is that the navigator needs to display some indication of the programmer's thought processes, as well as the specific implementation that it led to, in order to improve programmer efficiency. The code is *not* the best way to express this kind of information, and so my navigator seeks to create a graphical means to convey it instead.

2.1.2 Remembering details about the program

In two separate studies [18, 19], Rohr investigated how people represent knowledge about the use of an application in their minds, and how this affects their performance in using the application. He ran an experiment in which subjects were asked to group a number of editing commands together according to their apparent relationship to one another. For one group, the commands were presented as icons, and for another the commands were presented as command words. The results showed that those using the icons tended to group commands according to the entity on which they operated (e.g. file operations), whereas those using command words tended to make groups consisting of collections of commands that frequently appear in sequence. Rohr then had the subjects use the command groups to perform editing operations.

Rohr found that the first group achieved a significant performance improvement over the second group. He attributed this to the icons enforcing a spatial representation of the concepts needed to understand the workings of the program. His model of the operation of the mind is one in which there is a small area of working memory, and a large area of long-term memory. When trying to understand a complex system, a user cannot keep the entire structure of the system in working memory at once. Therefore, some mechanism is needed to extract from long-term memory only the areas of the system that are required for the immediate task.

Rohr identified two mechanisms for this purpose. The first is to develop cues for sequential recall—that is, remembering a list of concepts recalled one after the other. The list can easily be recalled, but starting anywhere except at the beginning is difficult. The second mechanism is to integrate all the concepts

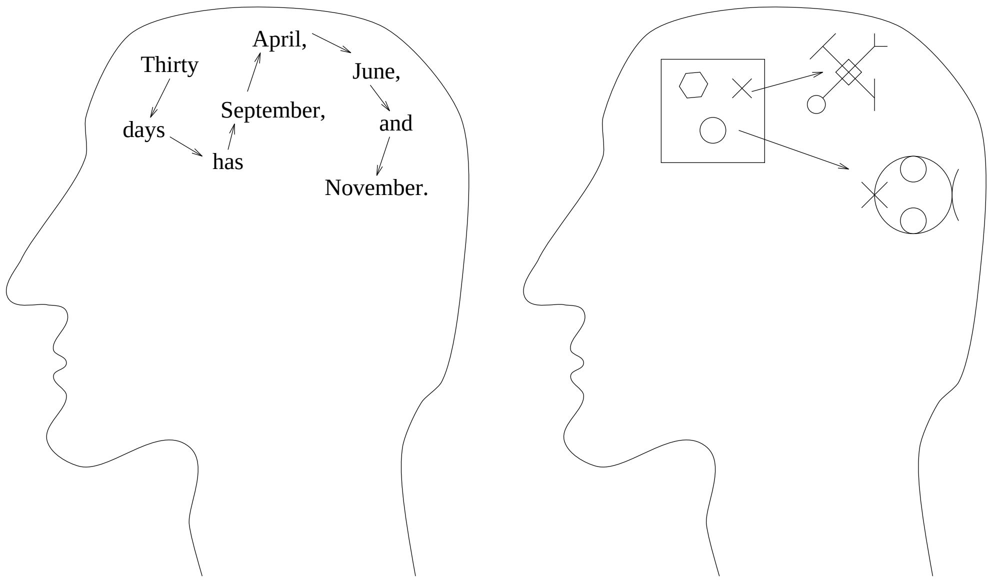


Figure 2.1: Two methods for storing information

under consideration into a picture, whose overall structure can be retrieved at will, and which then allows focus to be transferred to a specific element. With this method, it is much easier to recall just the parts of the image that are of most interest (see Figure 2.1). The first mechanism would tend to lend itself more to natural language, and hence to command words, whereas the second would be more naturally suited to icons, since they are already represented pictorially. The first group of subjects had used the pictorial representation, and were able to recall details of the software system more efficiently than the second group.

The result of this is that the choice of command words or icons to represent information can determine what kind of mental model the user constructs in order to comprehend an unknown system. Furthermore, since the former leads to sequential recall of concepts, which tends to be less efficient, the use of icons to represent ideas can lead to improved efficiency. My program makes extensive use of iconic representations to exploit this mechanism.

2.2 Using graphics to help teach programming

2.2.1 Graphics in Top-Down Design

Roberts et al. [16] have used the idea of graphical representation of program structure to help in their introductory computing courses. Through their teaching they discovered that by teaching beginners to express their design of a program in a formal, pictorial manner, they could improve the quality of the code their students produced. Students tended to keep the overall design goals in mind to a greater extent, choose more appropriate control structures such as `for` and `while` loops, and declare variables with a more appropriate scope. However, the pictorial design was always done on paper, which meant that implementation still had to be carried out in the traditional textual manner on the actual computer. This resulted in much of the benefit of the pictorial representations being lost. Therefore, they decided to implement the pictorial design process on the computer, and have the computer compile the program directly from the pictorial representation.

The pictorial design process follows three main steps. Firstly, the student designs the hierarchy of procedure calls that will decompose the program into trivial subtasks. The hierarchy of functions is represented as a tree diagram. Precisely when and under what circumstances the procedures will be called is ignored for the moment. Only when all the procedures that are going to be used are defined can the student move onto the next stage. In this stage, the student decides what control structures will be used to control program flow. The different loops and conditional statements each have a different symbol, and these are added into the tree diagram (see Figure 2.2). Finally, the student allocates variables to the control structures and implements the lowest level procedures in a normal textual way. The intention is that if students are forced to express their ideas in this formal, graphical way, then they will gain a better perception of what the program they've designed will actually do, and perhaps allow them to better understand different approaches.

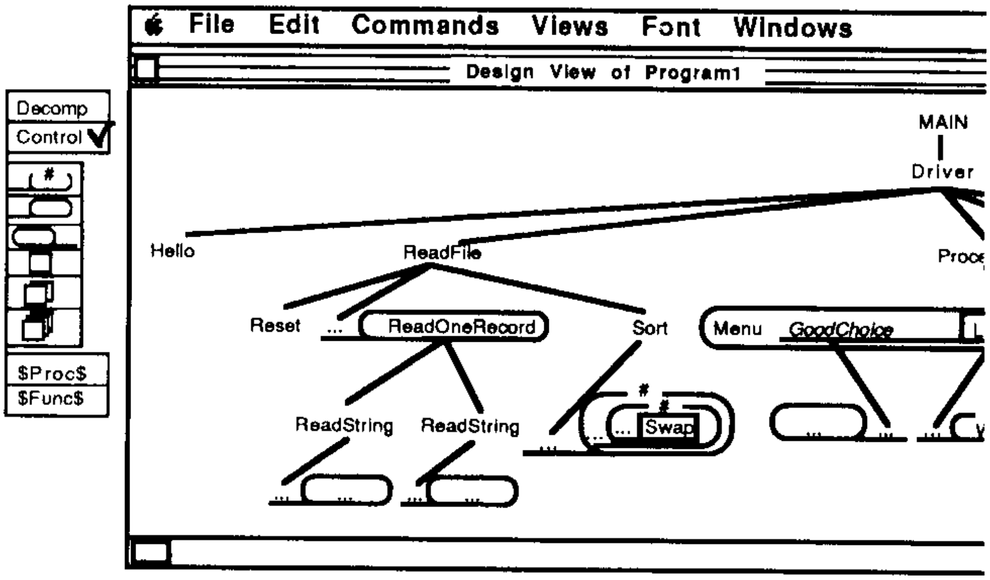


Figure 2.2: The tree representation

2.2.2 Expression of a Program as a Parse Tree

Stelovsky, Ackermann and Conti [21] have also produced a visual programming environment to help students. They suggested that many of the difficulties that students faced were due to students spending too much time concentrating on correct syntax and having to format their text to reflect the program structure. Furthermore, the text-based systems tended to force students to express their ideas in a linear fashion, whereas a program is more naturally expressed as a tree structure. To solve these difficulties, they created a visual programming environment called *Macpeth*.

Macpeth gives both a textual and a pictorial view of a program. The pictorial view is of a parse tree structure. An assignment, for example, is a node with two children: one for the variable being calculated, and one for the expression that will be assigned to it. The expression, of course, is likely to have further children (see Figure 2.3). *Macpeth* allows editing by selecting a node that hasn't been defined, and selecting a construct (`while...do`, `if...then`, etc.) from a menu. The central idea is to encourage top-down refinement: the student travels down the tree, adding more and more detail. The variables that are available at

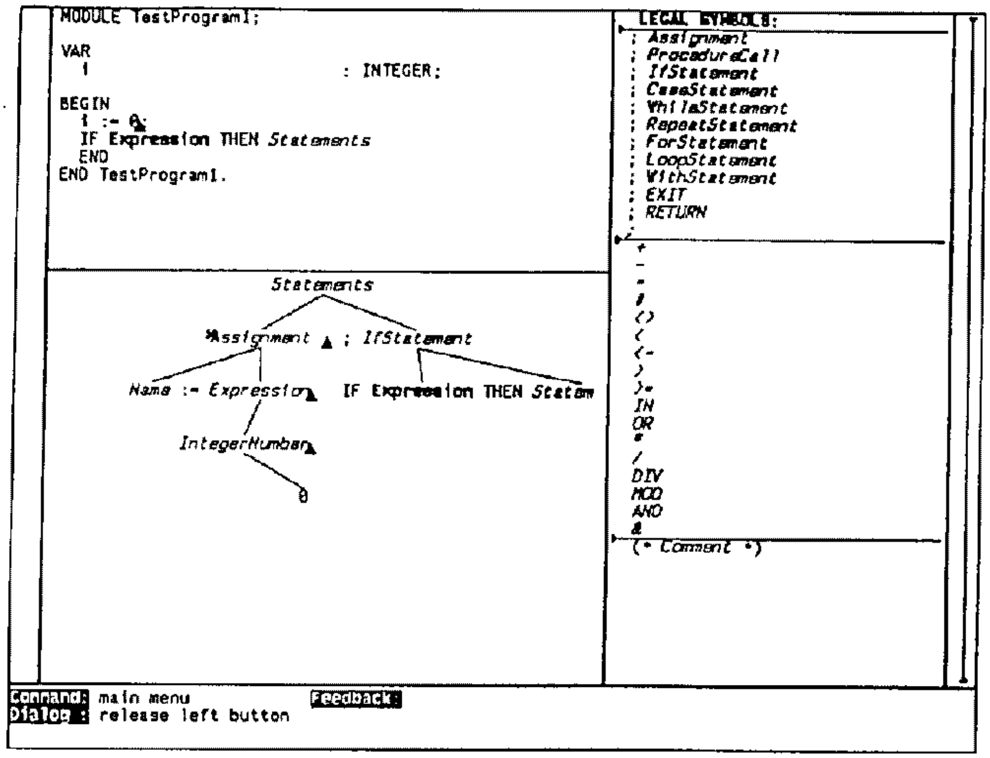


Figure 2.3: A display from *Macpeth*

each point in the program are listed in a menu—thus the student is able to see the scope of each of the variables. The declaration of any variable can be called up at will, thus saving much of the shuffling around that normally occurs during debugging.

When the program becomes too large to fit on the screen, another pictorial aid is offered. On the left hand side, the scroll bar is widened to contain parallel vertical lines which represent the nesting of the program. From the number of lines in the scrollbar, the user can gain an impression of how deep in the program they are working.

2.3 Environments for expert users

Writing visual programming environments for beginners is one thing, but attempting to create one suitable for experts is much more ambitious. Here I describe several experimental systems which had similar goals to mine but used very different methods.

2.3.1 Programming with visual objects and relations

One such attempt was described by Rogers [17]. Rogers describes his motivation as wanting to create an intuitive, powerful, visual programming language. He describes four groups that previous programming languages fell into: those tailored to a specific application, those that provide unrealistically simple models of computing, those that merely add visual interfaces to existing languages, and those that rely exclusively on programming by example. He believed that all of these had major deficiencies, and therefore suggested the need for a completely new programming language, written from scratch. The new language would conform to the expectations of computer programmers who want a powerful, general purpose language; but it would be designed around visual concepts.

Rogers felt that programming centred around the modification of data structures. Therefore, his main visual representation shows the data structures themselves, rather than the operations on them. Two constructs are displayed. Firstly, there are objects (the program is strongly influenced by object-oriented methodology), which represent specific data types that respond to particular "messages" by performing a "method" (subroutine). Secondly, there are relations between these objects that represent such concepts as `A` is a `Neighbour` of `B` , etc. These constructs provide a visual representation of the data structures involved (see Figure 2.4). In order to specify the subroutines that are used, a separate "Method editor" is used, which uses a visual syntax; but this tends to revert to the usual line-by-line format of traditional programming languages.

2.3.2 Designing and prototyping software in a visual environment

When designing a complicated software system, programmers often spend a considerable amount of time working out ideas in diagrammatic form before moving

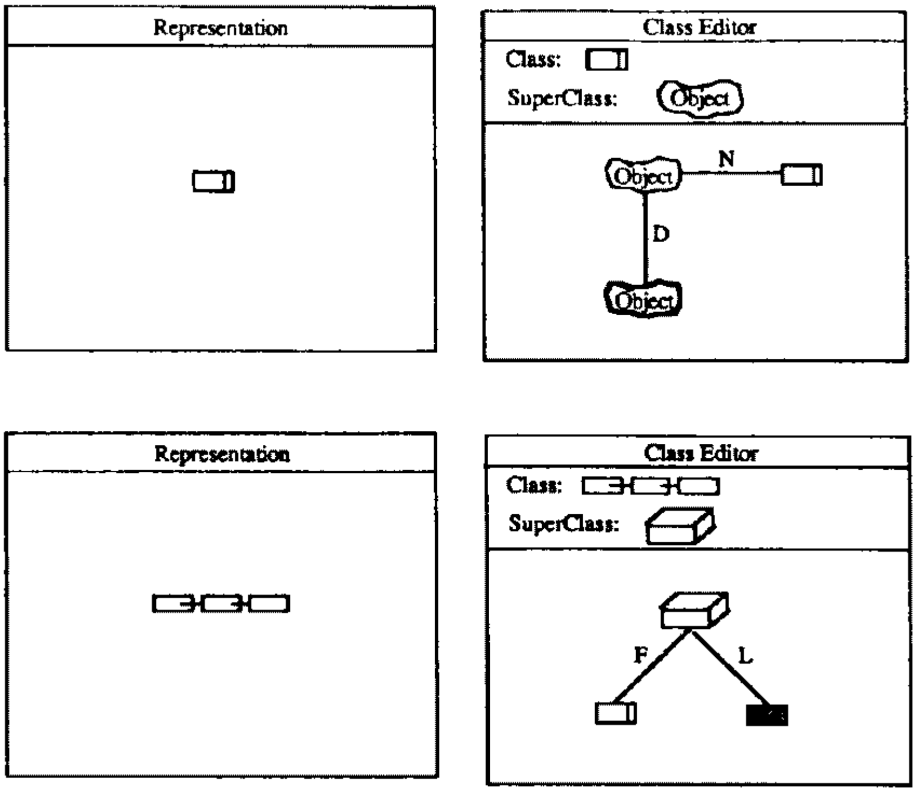


Figure 2.4: A Graphical representation of objects and relations

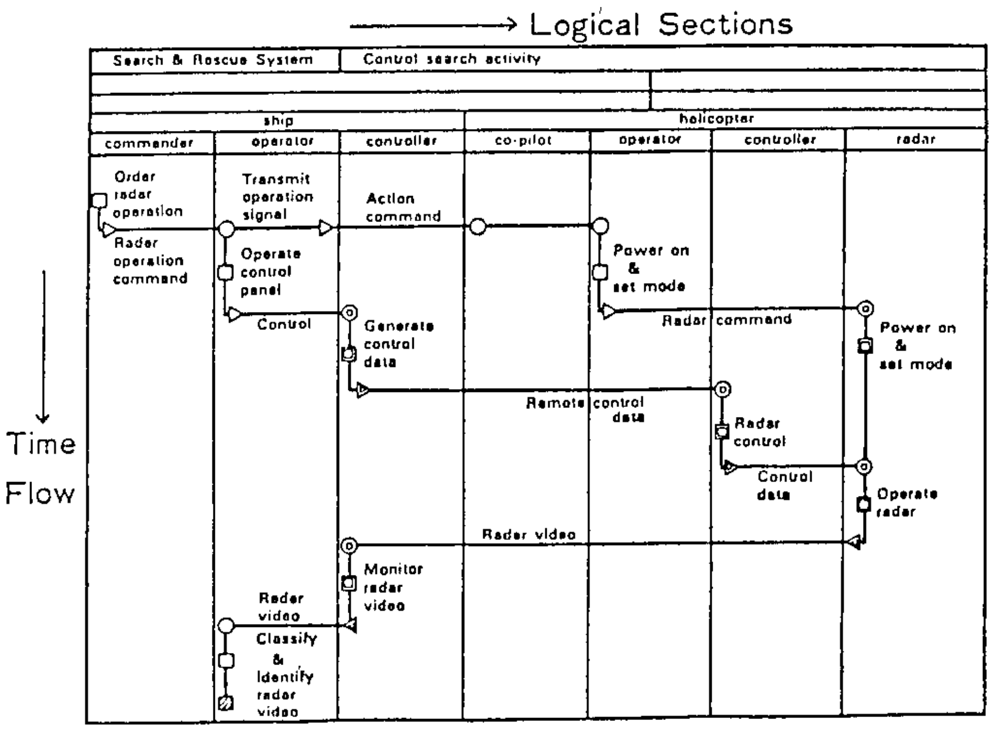


Figure 2.5: An SSD diagram

to the computer. Ikemoto et al. [6] suggested formalising this design process on a computer to ease the construction of a program specification and to improve the verification of a system design. They constructed a number of software tools to fulfill this purpose.

The first of these is an editor to construct an "SSD Diagram". The diagram is a two dimensional graph: the horizontal axis is divided into regions that represent different subsystems, while the vertical axis (roughly) represents the flow of time (see Figure 2.5). Functional boxes are arranged at different places on the graph, with arrows to represent the flow of control between the functions of the various subsystems. Thus the overall structure of the system is presented visually.

The next stage is to write the specifications for the subsystems. It is necessary here to be able to specify several possible alternatives for the design of each subsystem, and for each subsystem to be further subdivided into subtasks. Thus a tree representation is used. Each node represents a possible design alternative,

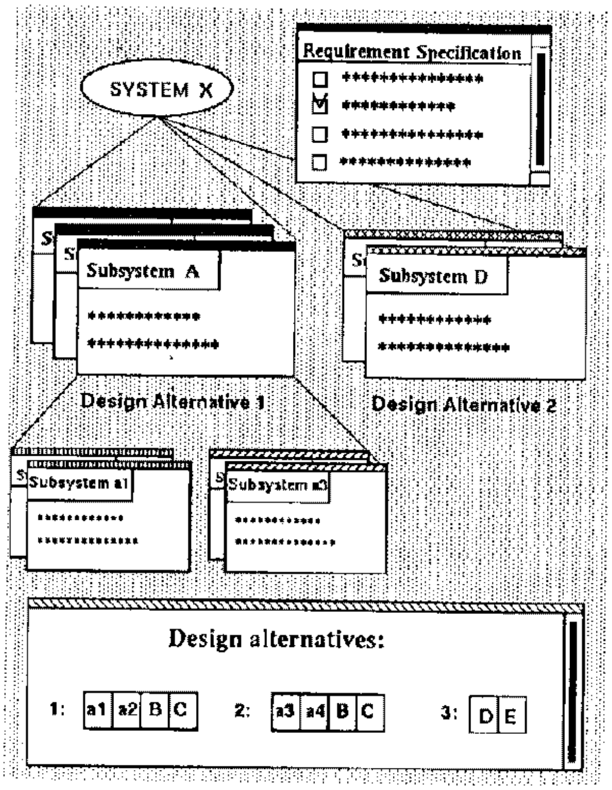


Figure 2.6: Top-down refinement

and is depicted as a stack of cards (each card representing a subsystem required to implement the design alternative). Each card contains the name of the subsystem, and a list of specifications for the functionality of the subsystem (see Figure 2.6). These specifications are automatically used to create a new stack of blank cards one step down the hierarchy. Thus the process of top-down refinement can be carried out in a visual way.

The purpose of these software tools is to help a group of system designers present their ideas in a logical and meaningful way—there is no way to automatically translate the ideas into an actual program. However, this system does demonstrate the usefulness of using graphical representations to formally define what the system is meant to do. It is hard to see how the full intricacy of a large system can be expressed without recourse to some pictorial representation, either automated as in this case or simply using pencil and paper.

2.3.3 Gaining an Overall View

Baker and Eick [1] created a tool to aid in the management of extremely large, multi-million line programs. The only practical way to display programs as large as this is with the use of statistics. Their program, *SeeSys*, works at the file, directory and subsystem level. It displays information about a program by subdividing the screen into rectangles, representing the subsystems of the program. The size of each rectangle represents the size of the subsystem, measured in non-commentary source lines (NCSL). Each rectangle is then similarly subdivided into directories. The user can focus on a particular region and have it expanded to show detail, for example a directory could be expanded to show the sizes of the files within it.

The system provides five different statistical views on the program. These are:

- non-commentary source lines,
- software complexity metrics,
- number and scope of modifications,
- number of programmers making modifications,
- number and type of bugs.

The use of these statistics allows a project manager to easily locate parts of the code that are causing problems, without actually examining a line of source code. The different statistics used are the information that is presented, not the actual code.

The most useful aspect of the system is that it allows the user to quickly browse through a very large program focusing in on details that are of interest. This is because of the hierarchical structure of the information presented, which allows the user to go from any position in the code to any other in only a few operations. However, for this technique to be successful, the hierarchy needs to be organised in a logical way, so that the user has a clear idea of which box to click on next to get to the next area of interest. The authors make note of the usefulness of spatial relationships between objects in providing this logical structure. Human beings naturally make inferences from the way objects are positioned in relation to each other, and this speeds up their processing of information.

2.4 3D Visualisation

With personal computers reaching new heights of speed and capacity, 3D real time applications have received a huge boost in interest. One attempt to represent program information in a 3D system was made by Reiss [15], in a program called *Plum*. It provides mechanisms for displaying many different aspects of a program such as call-dependency graphs, performance statistics, and the original source code. His system uses various mechanisms to neatly display complicated structures such as trees and graphs in 3D (see Figure 2.7).

One interesting aspect of the program is the use of text. Most of the objects displayed by *Plum* are labelled with text, but because of the 3D rendering these labels are frequently impossible to read unless the view is zoomed in quite close. This is less of a problem than might be anticipated, because it is the picture of the program that stays in the user's mind. The text is important for gaining an initial understanding of what an object represents, but afterwards the spatial relationships themselves provide all the cues the user needs. The program is severely handicapped by the overhead of rendering the 3D data. The advantage, however, is that it creates an extremely strong pictorial impression in the mind, and depends on text to an even lesser extent than the other programs I have discussed.

2.5 Conclusions

Some of the articles described above, and many of the others I have read, were written in the early to mid 80's, so there has been plenty of time for the ideas they have expressed to be implemented in commercial programming packages. However, the penetration of these packages into the marketplace has been minimal at best. Clearly there is some factor inhibiting the use of such environments by mainstream computer programmers.

The environments that have been produced seem to express programs in an

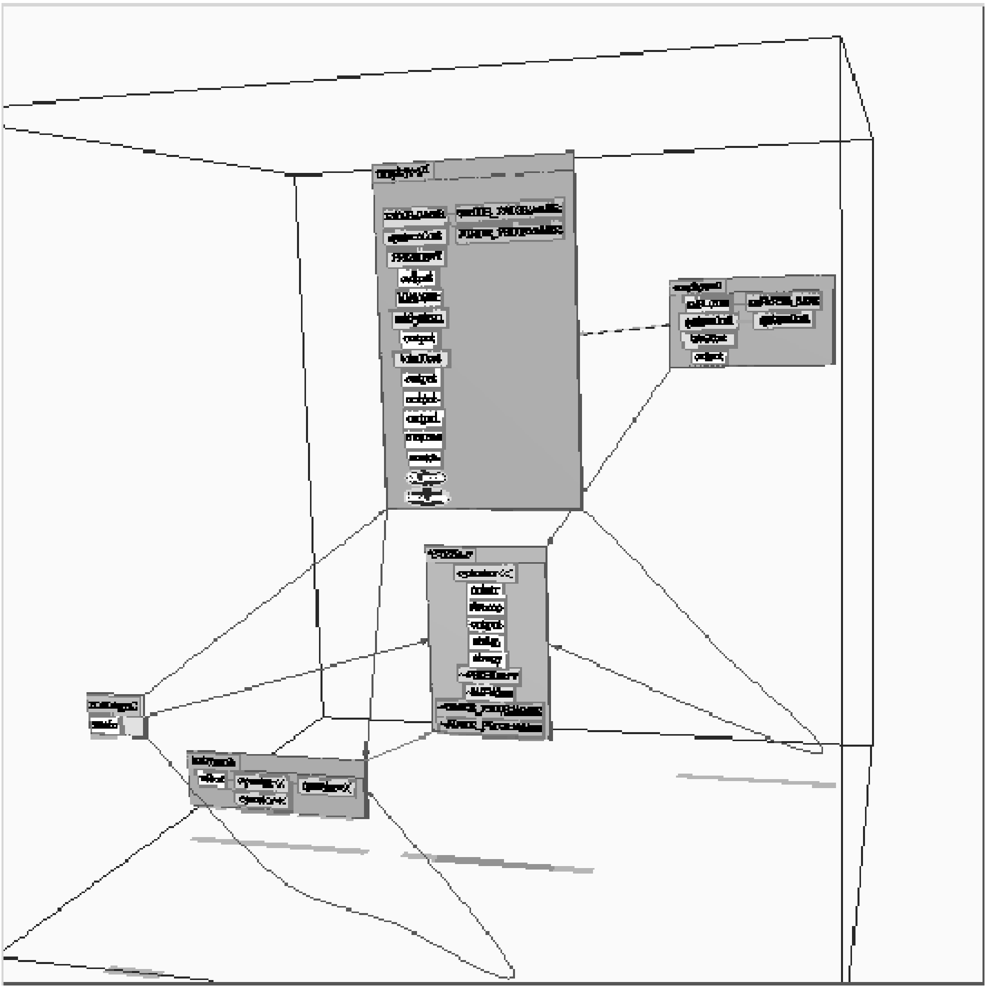


Figure 2.7: A 3D call graph

extremely inefficient way. In the paper written by Stelovsky et al. [21], one screen shot shows a program simultaneously represented as text and as a parse tree (see Figure 2.3). The parse tree representing just three lines of code takes up a sizeable proportion of the screen. A similar problem occurs in Rogers' paper [17], where a relatively simple program takes up a whole screen (see Figure 2.2). On the other hand, programs such as *SeeSys* [1] and *PLUM* [15] manage to display overall views of a program, but at the expense of ignoring the actual names of the functions of the program.

Furthermore, these environments display programs in a way that I, at least, find highly non-intuitive. It is important, when considering "intuitiveness", to distinguish your own prejudices from the genuine suitability or otherwise of a particular representation. Certainly, after several years of programming, I would be hard pressed to change my mode of thinking to a purely graphical representation. However, I believe that the standard, language-based methods of expression have a natural part to play in the expression of computer programs. Language is the means we usually use to give commands to each other, so it is only natural that we should use language to give commands to a computer. This model may only be valid for small-scale ideas, but when you *are* talking about small scale ideas, it is quite appropriate.

It is only when you start to talk about larger scale ideas that graphical means of expression start to become useful. This is where I think Rohr's model becomes quite useful. He suggested that people need a physical picture of a large system because that is the only way they can efficiently store all the details of the system in their mind. If you are constantly flicking back and forth through code, such a picture can take a long time to build up. This is why, in my project, I plan to use a combination of text for conveying the internal structure of a function, and graphics to represent the relationships between functions. I hope that this will provide an intuitive way of conveying information for a modern programmer.

CHAPTER 3

Design Goals

3.1 Graphics

3.1.1 Graphs

At various places throughout the program, it is necessary to display data as a graph. The display of graphs in an aesthetically pleasing manner is well known to be a complex and difficult problem, and many people have carried out work to achieve automation in this field [2, 22]. However, as Misue et al have pointed out [10], humans see many other relationships between the nodes in a graph apart from pure connectedness. When a person is left to arrange a graph manually, for their own purposes, they will usually arrange nodes directly underneath each other, in rectangles, and so on, according to gestalt rules [4], as in Figure 3.1. This will be an important factor in my graphical representation of information.

It is interesting to consider circuit design in relation to software design. Although they are very similar in that they involve breaking down complicated activities into trivial steps, and involve a static representation of a dynamic system, circuits have always been designed graphically, whereas software has always been designed textually. If software design is to become a more graphical process, then presumably circuit design is a useful model to follow. Petre and Green [13] have studied the way experienced electronic engineers use CAD software to design circuits. Their most important result was that experienced engineers made use of "secondary notation", a set of informal guidelines as to where objects must be placed in relation to each other to give extra information. Expert engineers read this notation almost unconsciously, often to the point where a novice's diagram, though technically legitimate, will cause great confusion to an expert.

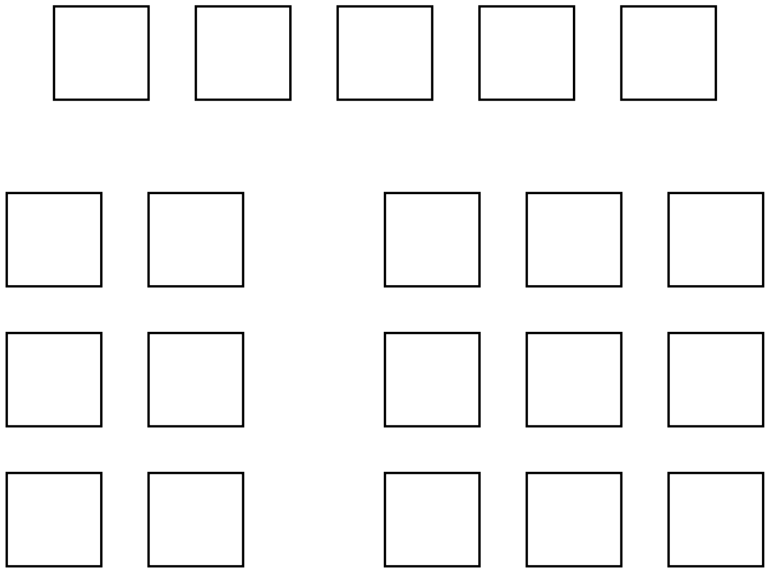


Figure 3.1: The spatial arrangement of objects can give clues as to their relatedness

These factors led Misue et al to consider a layout adjustment approach to graph drawing, as opposed to the more traditional layout creation. In their system, a graph was initially created, which the user could then modify in various ways, such as expanding or iconifying a subgraph, or moving nodes around. The graph would then be adjusted to avoid overlaps, while attempting to maintain the overall spatial relationship between the nodes.

My program is designed more as a medium for communication, rather than as an automatic generator of information. I would like to provide an original programmer with the ability to convey information to subsequent programmers with some secondary notation. The actual notation that could be used has not been investigated yet, because systems that allow such spatial notation, to the extent that they exist at all, are virtually unknown in general use. Over the years programmers have invented their own conventions for textual secondary notation [8]—I would expect a similar process to occur for graphical representations. In the meantime, the best strategy for a programming environment is to allow as much flexibility and simplicity as possible. The environment should allow components to be rearranged completely arbitrarily, while maintaining a small set of automatic visual relationships intact.

3.1.2 Colour

The use of colour in graphical representations of data has long been recognised as a useful tool in the graphical display of information [11, 20]. The advantage of colour information over other distinguishing features is that it is so abstract. Although colours such as red and green are traditionally associated with concepts such as stop and go, in general colours can be assigned to concepts completely arbitrarily, provided that consistency is maintained. In keeping with the principle stated above that the environment should allow the maximum freedom to programmers to convey information however they feel is best, the colour of objects should also be arbitrarily assignable by the user. Thus, the user has an additional facility to convey the relationships between the objects.

3.2 C++ specific issues

C++, with its additional constructs of classes and inheritance, has been greeted with great enthusiasm by much of the programming world. The rush to implement new environments based around these concepts has led to various attempts to represent classes graphically. To take one example, Microsoft's Visual C++ [9] has an online help facility which introduces new class constructs with a diagram showing their full inheritance graph, which demonstrates the key classes to which the user should refer for more information. The key problem with displaying classes is that, with complicated inheritance chains, large numbers of member variables and functions may be available as artifacts from ancestors that are quite irrelevant to the class under consideration. If a list of these members is displayed, it should be possible to limit the list only to those at the bottom of the hierarchy—but equally, it should be simple to get access to lists of the members of ancestor classes.

A large number of the enhancements to C that C++ provides can be ignored for the purposes of this project. The detailed semantics of statements are handled by *g++*, whereas I am primarily concerned with simple declarations of functions and variables. C++ features such as streams, templates, overloading, etc. require no special handling by my program, because *g++* "spoonfeeds" it a simple set of constructs that should handle any situation.

3.3 Flexibility and portability

The traditional Unix programming environment is just that—traditional. Its use is subject to numerous conventions and guidelines which specify how makefiles, directories, comments, documentation etc. are to be organised. However, all rules are made to be broken, and it seems that every program has some feature that is an exception to some rule. My program is designed to work with an arbitrary program from an arbitrary source, and thus it will have to deal with these quirks. Realistically, the goal of producing a program that will be able to deal with any situation is clearly impossible, and even trying to capture the simpler conventions formally is a huge task. Therefore, the goal I have set myself is to produce a program which can deal with a wide variety of programs, and have a better than even chance of working with a given program. This is a vague and perhaps over-simplified goal, but the task of producing a "foolproof" system is for a large software corporation, rather than a research project.

3.4 The "Complete Programming Environment"

To maximise the usefulness of any graphical representation of a program, it is necessary for it to be tightly and consistently integrated with a complete programming environment. If you can access the debugger directly from the same menu as you access the editor, you make things much easier for the end user. If you can highlight a line in the editor, and have the debugger set a breakpoint on that line, or if you can highlight a function in the call dependency graph and have the editor go directly to that function in the source code, then you can anticipate dramatic improvements in the efficiency of programmers.

An integrated system with debuggers, editors, profilers, etc would be far too difficult to achieve for a simple honours project, and I have not tried to create this. Rather, I have tried to create a program that implements features that could be incorporated into a larger programming environment. The windows are each separate entities that could easily be moved to another environment and provide an additional view onto the program that the program is working on. The features that I have implemented should be easily expandable into a more practically useful system.

CHAPTER 4

Implementation

4.1 Overall structure

When the navigator is first run, it creates a menu with several buttons on it, presenting the main options available to the user (see Figure 4.1). The "Load Hierarchy" button allows the user to load a program into the navigator. When clicked, it brings up a standard file chooser widget which asks the user for the makefile of a program. When the user selects the makefile, the navigator modifies the file so that any occurrence of "`g++`" is replaced by "`g++ -gMatsStuff`*filename*", where *filename* is a default pipe in the `/tmp` directory which will be used to send data from the compiler to the navigator. A comment is also added to the start of the file to ensure that a makefile is never modified more than once. The navigator then forks a process that invokes the standard `make` command on the modified makefile, which will invoke the modified version of g++ which parses the new argument and will use the pipe to send data to the navigator.

The data that is output by *g++* is in an ASCII text format designed to be understandable to a human with some effort. Lines are printed that represent:

- the start of a new main file,
- the start of a new included file,
- the end of the current included file,
- the start of a new class definition (which includes details about inheritance),
- the start of a function definition,

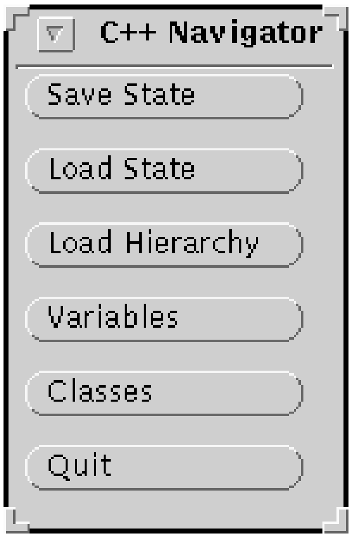


Figure 4.1: The navigator's main window

- the end of a function definition,
- the start of a new scope,
- the end of the current scope,
- a new declaration,
- a new type definition, and
- an occurrence of a particular identifier in the code.

A sample section of output is shown in Figure 4.2. Each object such as a declaration or a type is given a unique identifier which is used in cross-references.

The navigator parses this simplified representation of the program and organises it into its own data structures. The navigator then automatically displays a function window that represents the global group for the program, as a starting point for further exploration by the user. The global group remains during the rest of the user's exploration.

```
d36 108F E 105E 63 N"WindowList"
t16 1091 ""
d29 1090 1091 105E 66 Y"TreeWin"
t16 1093 ""
d29 1092 1093 105E 67 Y"TreeWin"
t16 1095 ""
d29 1094 1095 105E 69 Y"AddBaseNode"
t16 1097 ""
d29 1096 1097 105E 70 Y"AddListItem"
t16 1099 ""
d29 1098 1099 105E 71 Y"SetPositions"
d29 109A 1099 105E 72 Y"Repaint"
t16 109C ""
d29 109B 109C 105E 73 Y"Events"
t16 109E ""
d29 109D 109E 105E 74 Y"FindSelectedIcon"
t16 10A0 ""
d29 109F 10A0 105E 75 Y"GetData"
-76
]
+8
o32 1A9 10
o32 105D 11
o32 283 11
o32 1A6 14
o32 1A9 14
o32 1A6 15
o32 1A9 16
d36 10A1 1B2 1A7 10 N"GlobalScope"
c1A7
d36 10A2 106A 1A7 11 N"Window"
```

Figure 4.2: Sample output from the modified *g++*

4.2 The function window

The function window displays functions and the call dependencies between them. Because a program may have many functions, it is necessary to be able to display only a subset of the entire program at once. The strategy I adopted was to allow functions to be arranged into a hierarchy of "groups". These groups are an entirely new and independent construct in the program, and are arbitrarily created by the original programmer. This means that the programmer can use them to represent any relationship desired, such as high interdependence, use of common data structures, or functional similarity. Any group can contain any number of functions and any number of other groups.

A group is displayed as a window containing icons, representing the contents of that group. Icons that represent groups may be expanded into a new window. Within each window, icons can be rearranged in any pattern that suits the user. This is the most important improvement my program offers over previous systems. It offers the opportunity to organise the program in the following ways:

1. Functions can be grouped into separate logical entities, and these groups may be given any name. With traditional systems, files serve this function, but I believe my system provides a superior extension to this. Filenames are supposed to be used on the command line, and so their names have to be easy to type. Spaces and capital letters, which make names much easier to read, are usually avoided in filenames. My groups can have any string as their name, and these names never need to be typed, and so groups can be given clearer titles. My groups are also an important extension because they provide a single hierarchical mechanism. Files can provide one level of hierarchy, and this can be extended with the use of directories, but this requires different mechanisms for searching through and examining individual components in the system, which is more complicated for the user.

2. Functions can be given some spatial relationship to each other. If icons are presented as being close together, this is a sign that they are meant to be thought of together (see Figure 3.1). Human beings tend to look at things in horizontal and vertical sequences, so a horizontal or vertical sequence can be used to indicate relationships, or to indicate processes that appear

in some particular order. Any such mechanism that may be useful can be employed because the navigator leaves the positioning of icons entirely up to the user.

3. Functions and groups can be colour coded. Programs are complicated entities, and more than one coding mechanism can be useful in the same window. If one type of relationship, such as use of a particular data structure, is coded with spatial relationships, then another distinction, such as that between computational and input/output operations, can be encoded with colour-coding, without disrupting the existing layout. The more such mechanisms that are available to the user, the more information that can be conveyed.

Some examples of function groups are shown in Figure 4.3. They are taken from the representation of the navigator inside itself. The "Lines" function group shows the functions that read in lines of output from *g++*. The icon at the top is for the function `GetLine`. This is the generic function that controls all the others, and it is placed at the top so that it is the first one seen by the user. The other functions are divided into two areas. The first area contains functions that construct the hierarchical tree structure by pushing into new nodes and popping out of them. The pushing and popping functions are aligned in vertical columns, and they are arranged in rows according to what logical entity they operate on. The result is a two-dimensional grid that would be impossible to show in a normal text file. The second group shows a column of functions that read in the leaves of the tree—declarations, types and so on. The result is a clear indication of what it is that these functions do, and even before the user sees the code they know the names of many of the functions they can expect to see.

Note that this is only way to arrange the icons in a logical way. For example, I could have arranged them according to the data structures they operate on. However, the structure I actually used is more reflective of the way I think about the program, and the way I designed it. This should help future programmers to gain a clearer insight into my intentions, and help them maintain the program more efficiently.

When a program is first read in by the navigator, it has to provide some organisation itself, otherwise the user would be confronted by a bewildering array of functions, all in the same group. As an initial setup, each file is represented

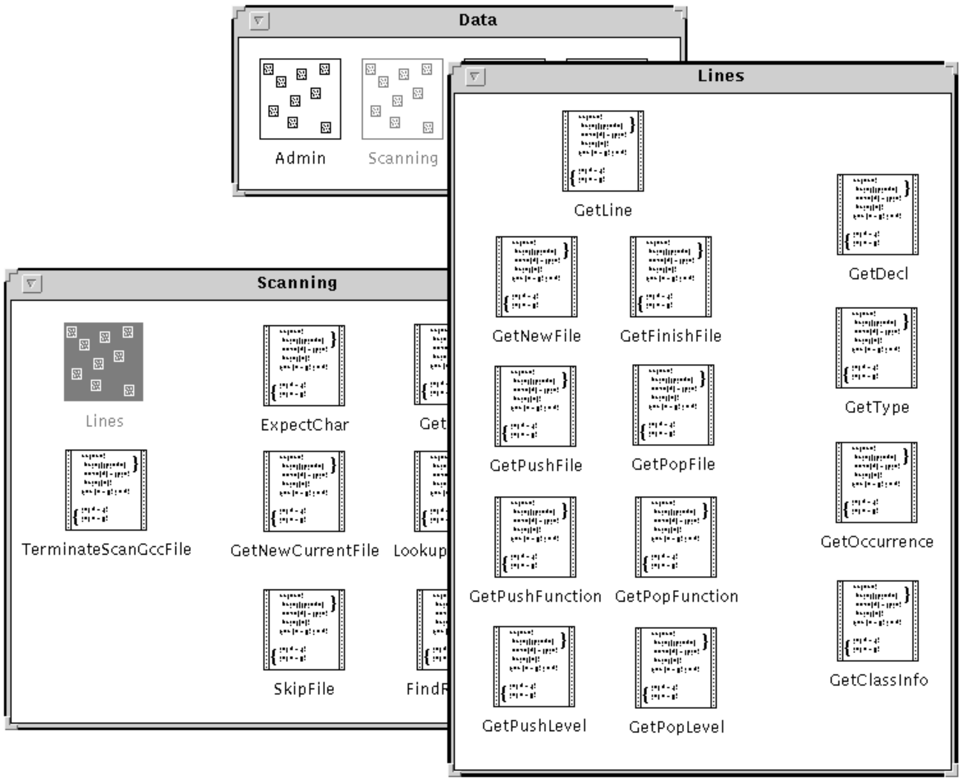


Figure 4.3: Three function groups

by its own group. Within a file's group, included files are represented by sub-groups. A distinction is made between system include files and user include files, because users are rarely interested in the system include files. They are placed in a separate group on their own. However, it may still be useful for the programmer to find out in which file a particular variable or function is declared, so it is important that the user can access the system include files if need be.

Within each group, only the functions that are actually defined are displayed. External declarations are resolved to their non-external definitions. This provides a significant break with more traditional systems in which each file is syntactically independent from the others, in that a function can be declared and used even if it does not actually exist. In the navigator, the files are presented as being part of a whole, and I believe that this corresponds much more closely with a programmer's expectations about the way a program behaves.

Each node, and the windows themselves, have associated menus that can be raised by clicking the right mouse button over them. These provide access to the operations that can be carried out on them, such as allowing groups to be iconified and expanded, and their names to be changed. The colour of any node can be changed to any of seven colours.

The navigator also provides the ability to show the dependencies between functions. From the menu for a function, the user can choose a colour and have the dependent groups and function coloured in that colour (see Figure 4.4). The navigator colours all functions that call the function, and recursively colours all groups in which they are contained. The user only sees the results of this operation in the windows that are displayed on the screen, so the operation is "fitted" to the user's view. However, this doesn't mean that the user misses some of the results of the operation, because of the way groups are coloured. If the function itself is not currently displayed, there is at least the certainty that one of the groups it is contained in will be highlight. The user can then search down from that point, simply expanding each highlighted icon.

One of the options in the menu for a function node is to view the code of the function. This creates a new window which contains only the text of the function, extracted from the original source file (see Figure 4.5). The function is disentangled from the rest of the code, and by viewing the source to multiple functions, the user can simply and efficiently see code for as many functions as desired side by side, without needing to read in an entire file, and without having

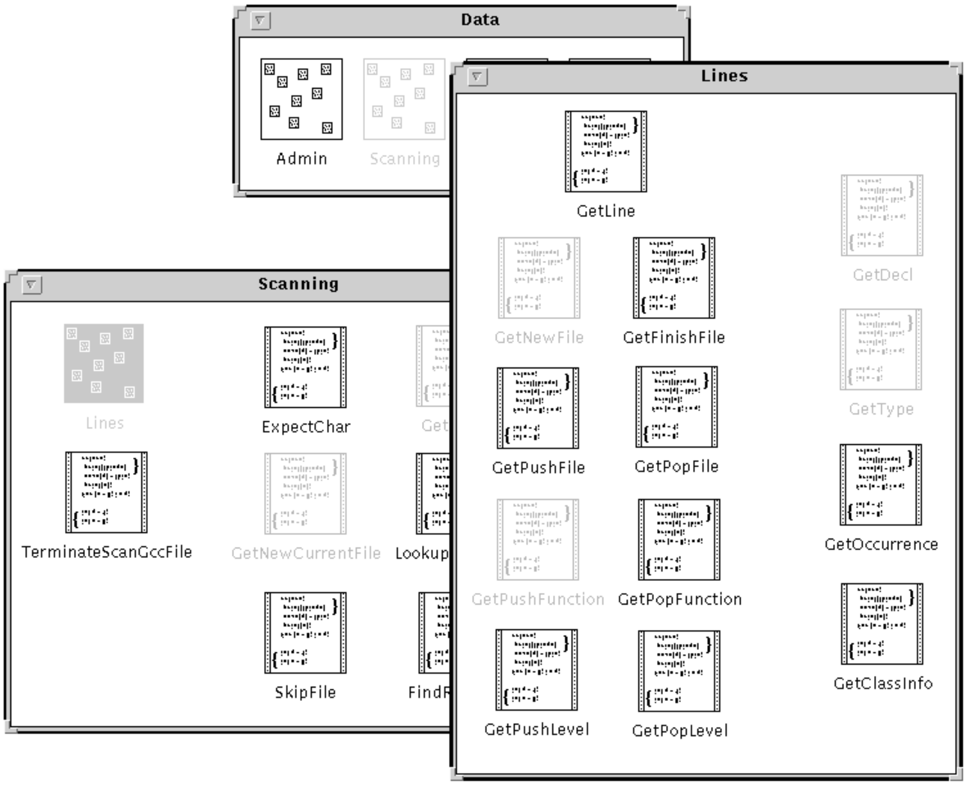


Figure 4.4: The same function groups, showing dependency on the function `GetString`

to search through a file for the section required.

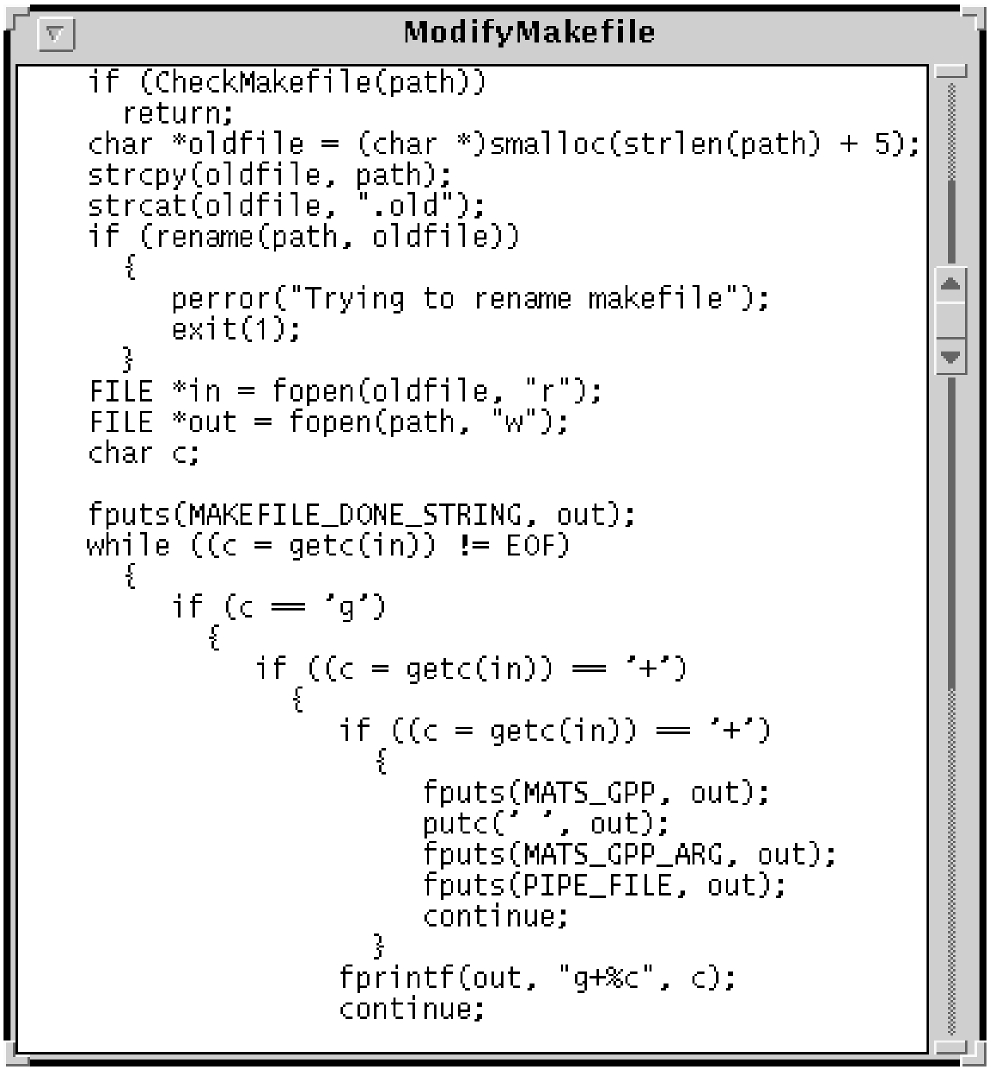
4.3 The variable window

The variable window is designed to display the variables that are declared within the program. Since any large program will have far more variables than can be sensibly viewed at one time, a means had to be devised to limit the list of variables in some logical way. The means I decided to use was to list the variables by scope. That is, the user can choose any scope and view a list of the variables that are declared within that scope.

To display the scopes that are available, the program displays a graphical scope tree (see Figure 4.6). This tree, with a node for every scope, is displayed with the outermost scopes at the top; thus, a new user sees the global scope first. A large program will have very many scopes, and so the user can choose which scopes should be displayed. Clicking on any node with the left mouse button expands the node, so that its children are displayed. The navigator stores the state of each node, so if any of those children were expanded in previous operations, then their children will be displayed when the parent is expanded. Scroll bars are provided because the diagram is likely to become very large. To help cut down on the size of the tree, subtrees are pruned if they don't contain any declarations—otherwise, every `for` loop, `while` loop, etc. would have its own node.

The navigator attempts to label nodes with some meaningful identifier: each file has its own node labelled with the filename, and function scopes are labelled with the name of the function. Other scopes are given the generic label "Local Scope".

The user can click on any node in the tree and a list of the variables declared in that scope will be added to the right hand side of the window. The user also has the option to make the list include all the variables visible within that scope (i.e., include variables from all the ancestors of the scope). If the user clicks on a name in the list, the program will highlight the node that represents the scope in which the variable was declared. This allows a user who has come across a previously unknown variable in the code to work out where it was declared. The fact that the scope tree is displayed graphically enables the user to see easily




```
    if (CheckMakefile(path))
      return;
    char *oldfile = (char *)smalloc(strlen(path) + 5);
    strcpy(oldfile, path);
    strcat(oldfile, ".old");
    if (rename(path, oldfile))
      {
        perror("Trying to rename makefile");
        exit(1);
      }
    FILE *in = fopen(oldfile, "r");
    FILE *out = fopen(path, "w");
    char c;

    fputs(MAKEFILE_DONE_STRING, out);
    while ((c = getc(in)) != EOF)
      {
        if (c == 'g')
          {
            if ((c = getc(in)) == '+')
              {
                if ((c = getc(in)) == '+')
                  {
                    fputs(MATS_GPP, out);
                    putc(' ', out);
                    fputs(MATS_GPP_ARG, out);
                    fputs(PIPE_FILE, out);
                    continue;
                  }
                fprintf(out, "g+%c", c);
                continue;
```

Figure 4.5: A code window

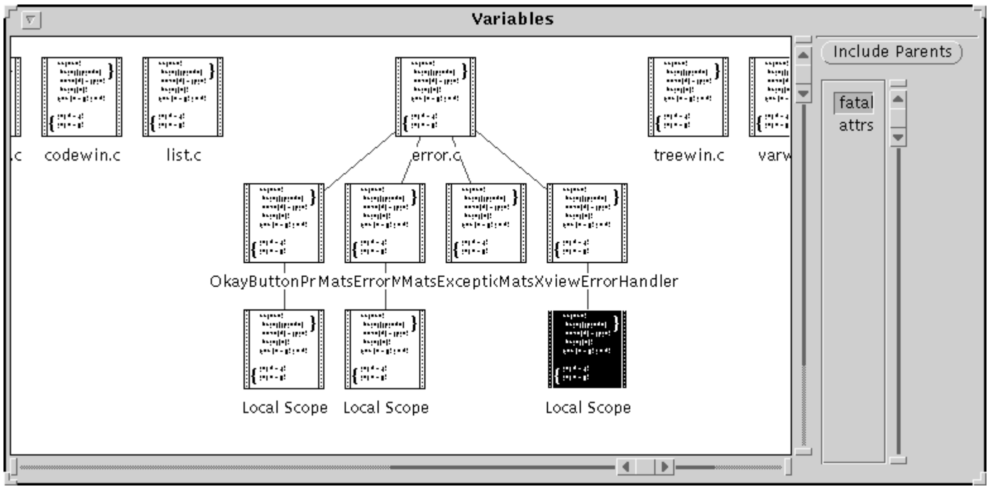


Figure 4.6: The variable window

where else the variable is likely to be used in the program, since it is visible to all nodes in the subtree with its declaration scope as root. If a user is considering making changes to the code that affect the variable under consideration, then this knowledge about what other areas of the code could be affected is crucial, and the graphical display enables decisions to be made very quickly.

4.4 The class window

The class window is very similar to the variable window, and uses much of the same code. It displays a graphical representation of the classes in the program, and the inheritance relationships between them. These relationships form a directed acyclic graph. I decided that the most useful way to display this graph would be with the base classes, that don't inherit members from any other class, at the top, and every class to be shown with its parents above it in the picture (see Figure 4.7). This means that the user first sees the most basic data structures (which are presumably the easiest to understand) and then sees more complicated data structures built from the previous ones. The picture that is created in this way is a static picture that the user cannot change, apart from scrolling over it, because there is no simple way to reduce areas of the graph to a single icon.

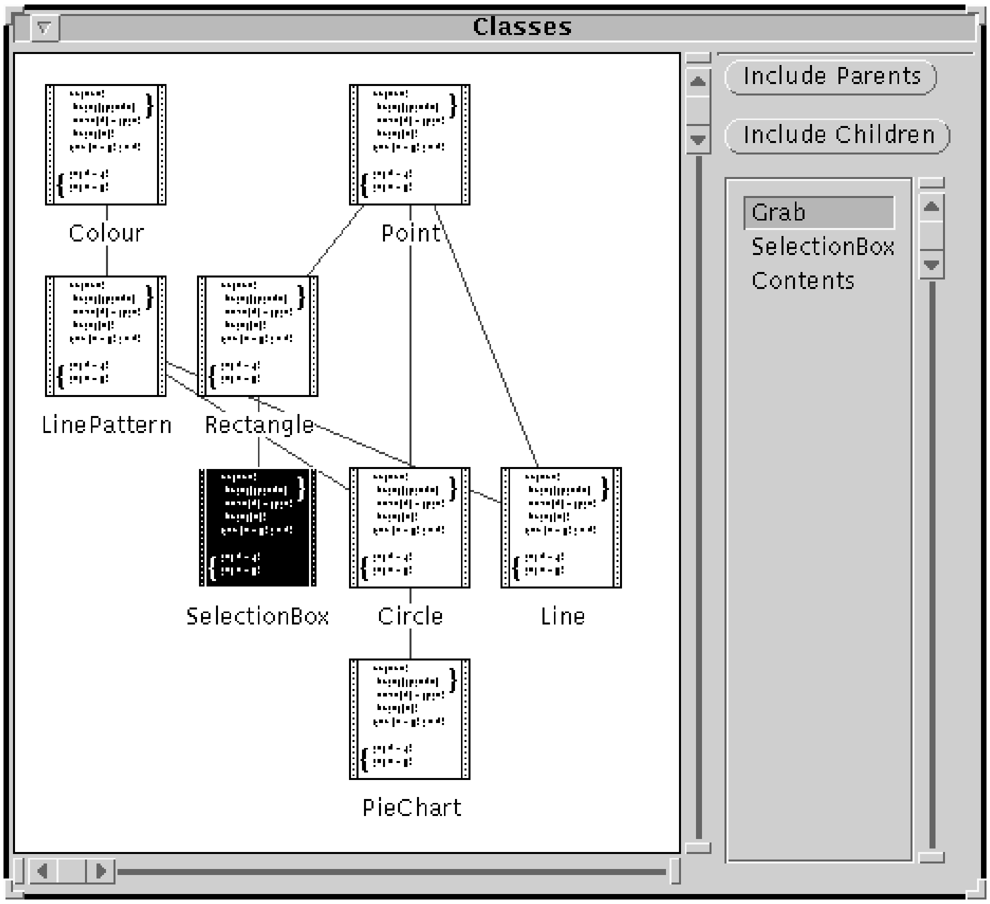


Figure 4.7: The class window

As in the variable window, the user can click on an icon to call up a list on the right hand side—in this case, a list of the members of the selected class. The user can also choose to include in the list all of the members from the parent classes, so the user can choose either to see all of the members of a class or just the members that are different from its parents.

CHAPTER 5

Results

5.1 Usability

5.1.1 The Function Window

The function window does provide a very convenient way to view different parts of a program side by side, by simply calling up two windows of different groups. One situation in which this is very useful is when there is a particular line of code which modifies a particular data structure with a function, and you are interested in seeing which other functions can be used to modify other aspects of the data structure. Because other relevant functions will most probably be placed in the same group as the original function, it is easy to see quickly what options other programmers have presented you with.

The colour-coding of functions according to call dependency is rather less clear and less intuitive than a system that displays a physical arc between nodes. This is much more difficult to implement, because of the difficulty of drawing arcs between windows, and must be balanced against the benefit of using separate windows for different groups, which is extremely useful for reducing groups to a single icon and then restoring them when you want to focus on a particular region. I has originally considered a system in which subgroups, when expanded, were displayed as subwindows of the parent - that is, clipped by the parent window so that the subwindow appears to be inside the parent window just like the original icon. This turned out to be impossible to implement with xview. The advantage of the system I eventually implemented is that it provides extra freedom in arranging windows around the screen, so that, for example, function windows can be placed next to associated code windows.

One of the best aspects of my program is, I feel, the code window, which enables the user to quickly bring up a particular function, and *only* that function. Although the spatial proximity of functions within a file is a useful guide to finding other relevant information, it is very limiting, and it is annoying having to scroll through hundreds of lines of code looking for a particular function, or having to type a search string. I have noticed when programming that when a function is displayed on the screen, and there is a few lines from another function displayed at the top, those irrelevant lines can be quite distracting and confusing. The code window of the navigator solves this problem, and displays code much more neatly. Furthermore, the hierarchy that the function window presents enables the user to reach the code of any function with just a few mouse clicks, avoiding the necessity to scroll through pages of irrelevant data, all of which must be processed by the programmer and assessed for relevance.

5.1.2 The Class Window

The class window does its job very well, and is capable of handling very complicated inheritance structures. However, I found that for most applications, inheritance is used fairly rarely, and in my display, the graph tended to be very wide and not very deep. Also, because it utilises one canvas for the entire program, programs with very many classes required such a large canvas that it took a long time to scroll. A class window is a useful thing to have, but it needs a mechanism to limit the number of structures it displays at once.

5.1.3 The Variable Window

The most useful feature of the variable window was being able to find out precisely where a given variable was declared, instantly. I experienced several situations when I was modifying *g++* when this would have been useful. When I was scrolling through one function that was several hundred lines long, I noticed a variable whose value would be useful to the navigator, but it took me some time to find out if it was a global variable or not, to see whether or not it could be used by my functions. If a variable window had been available, I could simply have selected the variable in the code and asked to be shown which scope it was declared in.

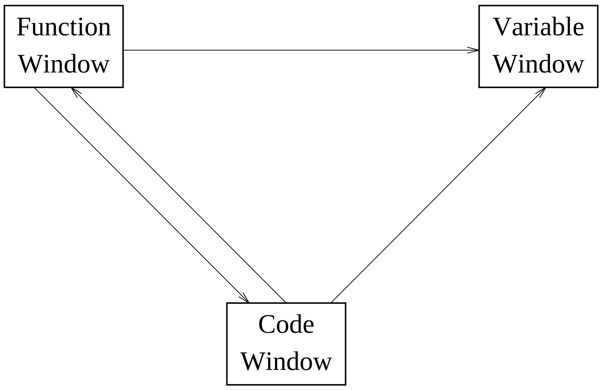


Figure 5.1: Possible paths from one window to another

5.1.4 Integration between views

The various views on the program structure are very useful, but they would be confusing if it were not for the fact the ability to relate them to each other. The user can select from the function window and call up the associated code window, or find the declaration in the variable window, or the user can select from the code window and find the appropriate item in the function window or variable window (see Figure 5.1). These allow the user to relate individual components in each view to the corresponding components in other views, thus providing an integrated whole rather than several disjointed views.

5.1.5 Integration with *g++*

The use of *g++* to simultaneously parse and compile a file does mean that the program takes quite a while to read in a new program. If a dedicated parser was used, this could probably be improved dramatically. However, this operation only has to be performed once, and the user will presumably spend quite some time using the data that is thus obtained, so the wait of a few minutes to compile a program probably won't be a great handicap. If you compare the time taken to compile a program with and without my options being used, you can see that the overhead of writing data to the pipe is not very significant, and so the efficiency of my program is comparable to a normal compile operation.

5.2 Flexibility

The traditional Unix philosophy is to create many different tools that can be combined flexibly to produce very complicated operations particularly well suited to the specific circumstances under which they are used. This is the philosophy behind the use of utilities such as gcc, make, gdb, etc. as the standard Unix programming environment. Unfortunately, it leads to the problem that when my program tries to read in a new program, it cannot possibly take into account all of the possible configurations that may be necessary. Similarly, C++ is designed to be as flexible as possible, and this makes it difficult to take into account every possible way it may be used. For example, a program that makes extensive use of preprocessor macros may not be accurately represented in my program, because variables that are used in macros will be recorded by the compiler as having occurred wherever the macro occurs. However, the code window only works with the original code, and so the user will have no way of knowing what variables are "hidden" by the macro.

So although I have tried to make my program as flexible as I could, it was impossible to take into account all the possible scenarios that a user would be likely to encounter. In order for the program to become a truly useful addition to the Unix environment, one of the following would have to occur:

- The navigator becomes integrated with other tools into a single unified system with common data structures, so that each tool would have access to a "canonical" version of the program.
- In future, programs are written in a much more standard, structured format. The second option, however, would impair the flexibility of the Unix programming environment. It is extremely unlikely that programmers would accept the less flexible system, and it contradicts the main purpose of the navigator.

5.3 Portability

My program is heavily dependent on X and xview, so porting the program to other operating systems, such as the Macintosh System or Microsoft Windows would be very difficult, although porting to other Unix systems should be relatively straightforward.

Converting its use to other languages should be quite easy, if a suitable parsing system can be found. The navigator does not utilise any syntactic information at all, and apart from the class window only displays information that is relevant to almost all programming languages. The data format for data being sent through the pipe could easily apply to any other language. The only requirement would be to find the places in the parser code to insert extra lines to send data. However, this operation did take up the whole of the first half of my project, so it can hardly be considered trivial.

CHAPTER 6

Conclusions

6.1 Implications for future systems

The graphical display of program information now has to be considered a top priority for the further development of programming environments. It seems to be an area where a relatively small amount of development can result in significant gains in programmer efficiency.

An important design goal that seems to be ignored by many experimental systems is that of flexibility. The great advantage of the traditional Unix environment is that it can easily be tailored to unusual requirements. If new systems are going to compete with current programs then they must be equally flexible. My own system makes some progress in that area by allowing the user to impose their own graphical structure on a program when they reposition, group and colour-code icons. This comes at the cost of adding an extra logical construct to the existing programming language, but not a complicated one, and one that compares favourably with file and directory structures, which it could conceivable replace.

6.2 Future Work

This navigator was written as an experimental implementation of various ideas I had for the graphical display of program information. Although a stand-alone program navigator is a useful tool, the full benefit of these ideas can only be realised once they are integrated with a larger programming environment. Reading other people's code is important, but totally useless unless you can then modify or extend it. As a result of this research, I believe that the graphical display of program information can greatly reduce the amount of time programmers need to spend trying to comprehend code, so that they have more time to concentrate on writing it. I would expect that in the future, programming environments will continue to follow the path to graphical displays.

It would be instructive to perform a study of to what extent programmers, both novice and experienced, can use the system to speed up the process of understanding a program, in a similar manner to the studies mentioned in the literature review [18, 23].

An important addition to my program is the handling of macros. As I have described previously, extensive use of macros in a program can degrade the quality of the navigator's representation of a program. The problem with macros is that they add a whole new level of complexity—just because the compiler says that a particular variable occurs on a particular line, doesn't mean that it's actually there in the source code. Checking for this involves keeping a table of macro definitions during the parsing of the code. Since *g++* uses two separate runs for the preprocessor and the actual parsing, it would require extensive modifications to *g++* to allow this to happen.

It would be very useful to replace my code window with a full editor, that would allow the user to edit the code as well as just view it. It would be important to maintain the integration of the system by allowing selections from within the editor to be used by the rest of the program for searches, etc. If this were added, then it would be possible to write a program from the ground up entirely within the navigator, and thus detach the programming process from files and directories entirely. It would be very interesting to see if professional programmers find this preferable to the existing systems.

APPENDIX A

Original Honours Project Proposal

Title: A Graphical Aid to Visualisation of Program Structure

Author: Matthew Exon

Supervisor: Dr Chris McDonald

Aim

It is the aim of this project to implement a programming aid that will allow a complicated C++ program to be displayed in a way that eases the construction of a mental model of the structure of the program. It will be designed to be used in two ways: to organise the program into a logical and meaningful visual structure, and to navigate through source code in a way that presents all useful information while hiding useless information. It will not allow the code to be edited, although this could be a future extension.

Background

Considering that programming languages are specifically designed to express complicated concepts unambiguously and logically, it is surprising how difficult and frustrating it is trying to extract meaning from a large collation of source code. Source code might be quite readable at a small scale, but programmers need to understand the overall structure of a program before they can appreciate the design of an individual function.

Normally, the structure of programs can only be indicated in a very superficial style, by breaking the code into separate files or sections, and by indentation and coordination by alignment, as investigated by Lutze [8]. It is impossible to indicate the full complexity of the inter-relationships between functions using this method. Programmers are therefore left to create their own model of these relationships before they can understand the design of a function well enough to modify it.

The use of a graphical user interface provides a possible alternative to the standard ways of expressing program structure. Graphical user interfaces introduce several factors which could simplify the process of creating a mental model of a large program. The first of these is their ability to arrange objects in two dimensions. The placement of objects in a row, for example, could denote order of execution—two rows could denote two possible sequences. On the other hand, placement of objects in a tree structure can be used to indicate a hierarchical set of function calls. This flexibility is not possible with a text based system. The second factor is their ability to use graphical objects such as lines and boxes to show relationships between objects. Text based systems can only show relationships in a linear way, whereas lines and boxes allow much more sophisticated constructions. The third factor is the ability to "point and click" on an object to perform an action on it. If the intention is to gain more information on the object, then a considerable saving is likely over the method of hunting through source code for the additional information. Finally, as investigated by Rohr [18], graphical means of presenting information force users to "think in images", which turns out to be a much more efficient way of storing and retrieving information than thinking in text.

The most important area where these factors become significant is when a programmer has to work on a program written by someone else. The programmer will have to spend a considerable amount of time simply flicking back and forth through code. Waddington and Henry [23] suggested that such a programmer needs to recreate the intentions of the original author. This suggests that it would be helpful if the *design* of the program was presented, as well as simply the final result.

There have been several attempts to create such visual environments for programming, designed for different purposes: education [16, 21], visualising data structures [12], and for software engineering [6]. However, these tend not to give the user freedom to impose their own spatial representation on the objects displayed. It is in this area that I hope that my project will extend the usefulness of programming aids.

Method

The programming aid will be designed to work with the C++ language. This language was chosen because it is one of the most frequently used languages in practice, and therefore the usefulness of the programming aid will be maximised. Furthermore, the use of classes and encapsulation in C++ means that, to a certain extent, grouping of functions will already have been defined by the programmer. However, some degree of language independence is desirable, and in the future support for other languages may be added. Design of the programming aid will keep these factors in mind.

The first phase of the project will be to create a parser which will examine source code and extract information about functions and variables. The proposed strategy is to modify an existing parsing system; the GNU C Compiler ("gcc"). The modified program will run through the code and construct a number of tables:

- A table to associate a variable name with all the appearances of that variable in the text, taking particular note of where the variable is declared, and a record of the scope of that variable.
- A table to record the dependency relationships between functions, and between functions and variables.
- Another table to record the hierarchy of classes and their member functions.

The parser should not need any information apart from the source code itself, although it may store information for future sessions.

The second phase will be to implement the front end to these tables using the X-window environment. The user will be able to call up a window which shows the functions (or some subset of the functions) as icons, linked together by lines, representing calls to those functions, producing a call dependency graph.

The user will then be able to group functions into "packages": this will allow the user to simplify the graph to any level desired. Member functions of classes will automatically be grouped in this way. It might be possible for the package to suggest groupings automatically, by searching for sets of functions with high interdependency—but it is anticipated that unless the interdependency is particularly strong this method is unlikely to produce useful results. Experimentation will be required.

Double-clicking on a function will call up its source code in a separate window. It may be possible to use an existing editing program for this, so that the function could be edited, but this is not crucial. However, it will be possible to select a variable, function, or macro and call up its declaration, its source code, or its value respectively. For a particular function, the user will also be able to call up a list of all the functions and variables whose scope includes the function.

The user will be able to call up a list of all the functions and variables used by the program, and by selecting one have the dependency graph display which functions use it. The dependency graph could also display the scope of the function or variable.

The programming aid will also offer the option to reorganise the text of a project in a logical way based on the organisation that the user has created. Thus, functions that are grouped together would appear in the same file, blank lines and comment lines would separate groups within a file, etc. The trade-off here would be between making the software configurable enough to meet the user's needs, but simple enough to be used easily.

The software will be developed for an IBM-compatible PC running the Linux operating system and the X-Window system.

A proposed timetable for the project is the following:

- Background reading and investigation of existing products
  4 weeks

- Detailed design
  3 weeks

- Implementation of the parser

  - Interpretation of preprocessor commands
    2 weeks
  - Extraction of functions and variables and creation of occurrence table
    3 weeks
  - Extraction of classes and creation of class table
    2 weeks

  7 weeks

- Implementation of front end
  - Display of function dependency graph
    2 weeks
  - Operations on function dependency graph
    * Arranging and grouping icons
    * Increasing and decreasing detail of graph

    2 weeks
  - Display of function code window
    2 weeks
  - Operations on code window
    * Substituting macros
    * Selecting names
    * Obtaining information on function calls
    * Obtaining information on variables

    2 weeks
  - Display of class inheritance tables
    * Display list of member functions and variables of a class

    2 weeks

  12 weeks

- Write dissertation
  6 weeks

Bibliography

[1] Marla J. Baker and Stephen G. Eick. Space-filling software visualization. *Journal of Visual Languages and Computing*, 6:119–133, 1995.

[2] Kylie Ding. Drawing planar graphs. Honours thesis, Department of Computer Science, University of Western Australia, 1991.

[3] Isabel Ferreira and Chadi Garman. A system for data structure visualisation. Honours project, The University of Western Australia, Department of Computer Science, 1995.

[4] James Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice*, chapter 9.5.1, pages 418–422. Addison-Wesley Systems Programming Series. Addison-Wesley, second edition, 1990.

[5] P. Gorny and M. J. Tauber, editors. *Visualization in Programming: 5th Interdisciplinary Workshop in Informatics and Psychology, Schärding, Austria, May 1986: Selected Contributions*, volume 282 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1986.

[6] Hiroyuki Ikemoto, Yoichi Kusui, Junichiro Tsuda, and Tadashi Furukawa. A visual environment for system design and prototyping. In *1988 IEEE Workshop on Visual Languages* [7], pages 160–167.

[7] Insititute of Electrical and Electronic Engineers. *1988 IEEE Workshop on Visual Languages*, 1730 Massachusetts Avenue, N.W. Washington D.C., 1988. IEEE Computer Society Press.

[8] R. Lutze. The gestalt analysis of programs. In Gorny and Tauber [5], pages 24–36.

[9] Microsoft. Visual C++. Software for Microsoft Windows, 1994.

[10] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6:183–210, 1995.

[11] Brad A. Myers. *Creating User Interfaces by Demonstration*, volume 22 of *Perspectives in Computing*. Academic Press, inc, 1988.

[12] Brad A. Myers, Ravinder Chandhok, and Atul Sareen. Automatic data visualization for novice Pascal programmers. In *1988 IEEE Workshop on Visual Languages* [7], pages 192–198.

[13] M. Petre and T. R. G. Green. Learning to read graphics: Some evidence that 'seeing' an information display is an acquired skill. *Journal of Visual Languages and Computing*, 4:55–70, 1993.

[14] The GNU Project. The GNU C compiler. Software for the Linux Operating System, 1995.

[15] Steven P. Reiss. An engine for the 3d visualization of program information. *Journal of Visual Languages and Computing*, 6:299–323, 1995.

[16] Jim Roberts, John Pane, Mark Stehlik, and Jacobo Carrasquel. The design view: A design oriented, high-level visual programming environment. In *1988 IEEE Workshop on Visual Languages* [7], pages 213–220.

[17] Greg Rogers. Visual programming with objects and relations. In *1988 IEEE Workshop on Visual Languages* [7], pages 29–44.

[18] Gabriel Rohr. How people comprehend unknown system structures: Conceptual primitives in systems' surface representations. In Gorny and Tauber [5], pages 89–105.

[19] Gabriel Rohr. Using visual concepts. In Chang et al., editor, *Visual Languages*, New York, USA, 1986. Plenum Press.

[20] Shubin, Falck, and Johansen. Exploring color in interface design. *Interactions: New Visions of Human-Computer Interaction*, III.4:37–48, July 1996.

[21] J. Stelovsky, D. Ackermann, and P. Conti. Visualizing of program structures: Support concepts and implementation. In Gorny and Tauber [5], pages 37–51.

[22] Kozo Sugiyama and Kazuo Misue. Graph drawing by the magnetic spring model. *Journal of Visual Languages and Computing*, 6:217–231, 1995.

[23] Ray Waddington and Roger Henry. Expert programmers re-establish intentions when debugging another programmer's program. In D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, editors, *Human-Computer Interaction: Interact '90*, pages 965–976, Sara Burgerhartstraat 25, P.O. Box 211, 1000 AE Amsterdam, The Netherlands, 1990. Elvesier Science Publishers.