A C NAVIGATIONAL SYSTEM

THESIS

Presented to the Graduate Council of the University of North Texas in Partial Fulfillment of the Requirements

For the Degree of

MASTERS OF COMPUTER SCIENCE

By

James D. Hammerquist, B.S.

Denton, Texas

May, 1989

The C Navigational System (CNS) is a proposed programming environment for the C programming language. The introduction covers the major influences of programming environments and the components of a programming environment. The system is designed to support the design, coding and maintenance phases of software development. CNS provides multiple views to both the source and documentation for a programming project. User-defined and system-defined links allow the source and documentation to be hierarchically searched. CNS also creates a history list and function interface for each function in a module. The final chapter compares CNS and several other programming environments (Microscope, R^n, Cedar, PECAN, and Marvel).
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LIST OF FIGURES</strong></td>
<td>iv</td>
</tr>
<tr>
<td><strong>Chapter</strong></td>
<td></td>
</tr>
<tr>
<td><strong>I. INTRODUCTION</strong></td>
<td>1</td>
</tr>
<tr>
<td>Purpose</td>
<td></td>
</tr>
<tr>
<td>An Overview of This Paper</td>
<td></td>
</tr>
<tr>
<td>Definitions</td>
<td></td>
</tr>
<tr>
<td>Major Influences for Programming Environments</td>
<td></td>
</tr>
<tr>
<td>The Components of Programming Environments</td>
<td></td>
</tr>
<tr>
<td>Putting It All Together</td>
<td></td>
</tr>
<tr>
<td><strong>II. THE C NAVIGATIONAL SYSTEM</strong></td>
<td>36</td>
</tr>
<tr>
<td>The Structure of CNS</td>
<td></td>
</tr>
<tr>
<td>A Walkthrough of CNS</td>
<td></td>
</tr>
<tr>
<td>How can CNS Improve Software Development</td>
<td></td>
</tr>
<tr>
<td>What Is Missing?</td>
<td></td>
</tr>
<tr>
<td><strong>III. CNS AND OTHER ENVIRONMENTS</strong></td>
<td>61</td>
</tr>
<tr>
<td>MicroScope</td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
</tr>
<tr>
<td>Cedar</td>
<td></td>
</tr>
<tr>
<td>PECAN</td>
<td></td>
</tr>
<tr>
<td>Marvel</td>
<td></td>
</tr>
<tr>
<td>Conclusion</td>
<td></td>
</tr>
<tr>
<td><strong>REFERENCES</strong></td>
<td>75</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Domain versus programming knowledge</td>
<td>5</td>
</tr>
<tr>
<td>2.</td>
<td>The Programmer's Apprentice System</td>
<td>22</td>
</tr>
<tr>
<td>3.</td>
<td>An example WEB program</td>
<td>30</td>
</tr>
<tr>
<td>4.</td>
<td>Collapsing a window</td>
<td>41</td>
</tr>
<tr>
<td>5.</td>
<td>The standard CNS window configuration</td>
<td>42</td>
</tr>
<tr>
<td>6.</td>
<td>Multiple windows within a function</td>
<td>44</td>
</tr>
<tr>
<td>7.</td>
<td>An example with various types of links</td>
<td>47</td>
</tr>
<tr>
<td>8.</td>
<td>An example of hierarchical documents</td>
<td>51</td>
</tr>
<tr>
<td>9.</td>
<td>Views of program execution in BALSA</td>
<td>70</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Purpose

The C Navigational System (CNS) is a programming environment for the C programming language. Although the system presented here is written for the C language, the same concepts can be used to support any structured programming language. The term function (from the C language) will be used to denote an executable piece of source code, although the word function can be replaced with procedure or whatever term is applicable. The system consists of a structure-oriented editor and a database system. The use of a structure-oriented editor provides the system with a syntactical structure of the source code that is being edited. The purpose of this structure is to facilitate the job of the database system in the task of searching and maintaining the names of identifiers used in a program. The structure-oriented editor helps the programmer find syntax errors and provides templates for editing. However, this particular feature of CNS will not be discussed in this thesis.

The system is intended to support the design, coding, and maintenance phases of software development. The main
purpose of the system is to provide multiple views of a software system as it is being developed and as it is maintained throughout its existence. These views are enhanced by the ability to link portions of the documentation and source code together through user-defined windows. The purpose of the windows is to enhance the readability of a program by providing logical groupings of such items as program statements, program functions, or documentation. The ultimate goal behind CNS is not only to provide as much information as possible to someone using, creating, maintaining, or browsing source code, but also to bring all this information together in a consistent and integrated environment.

An Overview of This Thesis

This paper is divided into three chapters. The first chapter presents some background information on the subject of software development environments of which programming environments are a subset. This chapter defines terms with emphasis on the various classifications of different programming environments, the major influences from other areas of software engineering, and the major components of software development environments. The next chapter presents the design of CNS, the features in CNS that should improve the software development process, and the problems that may arise from the use of CNS. The details of how
windows are used, created, and referenced are also discussed in this chapter. The final chapter compares CNS to several existing programming environments such as MicroScope, R^n, Cedar, PECAN, and Marvel.

Definitions

The purpose of this section is to provide the reader with some basic definitions of programming environments and to discuss some of the difficulties in the programming process that necessitate the use of such systems. First, it is important to know what tasks are involved in the programming process and which of these tasks are supported by programming environments. As will be discussed later, most programming environments provide support for only part of the programming life cycle. The term software development environment will be used to refer to environments that support all activities associated with software development. The next section on programming environment classifications describes the various types of programming environments along with brief examples of each type of environment.

Finally, a second classification scheme which includes an extension of the software development process to a company-wide effort will be introduced. Although the system presented in this paper is not this ambitious, one of its
primary goals is to also link the programming environment to
the maintenance and design phases.

Programming tasks

Most programmers would never agree on the best way to
develop a large software system. However they do agree that
developing such a system is an extremely complex task.
Beyond the myriad of details, the almost infinite number of
possible interactions among modules and the maze of
interfaces lies the world of managers, consultants, systems
analysts, testers, manual writers, and of course the end
user. A further complication arises when, as is generally
the case, all the people involved in the development process
do not share the same field of knowledge. The diagram in
figure 1 (modified from Charles Rich and Richard C. Waters,
21 (August 1988):42) shows the relation between the domain
knowledge and programming knowledge during various phases of
the software development process. Initially, domain
knowledge is the primary knowledge required for the
programming project. As the project progresses towards the
programming phase, the domain knowledge becomes less
important and programming knowledge becomes the dominant
factor. The testing and validation phases are unique in
that they require the use of both domain knowledge in order
to validate the source code against its requirements and
programming knowledge to fix design problems and bugs in the software.

Requirements  Design  Coding /  Testing /  Maintenance  Integration  Validation

Domain Knowledge

Programming Knowledge

Figure 1 Domain versus programming knowledge

This diagram ignores some major steps in the software development process such as the project definition, which requires only rudimentary domain knowledge, and the documentation of the system, which is an ongoing task. As indicated in the above diagram, Rich and Waters emphasize one very important issue in the path from the end user's needs to the realization of those needs.

There is no point at which someone who knows nothing about programming communicates directly with someone who knows nothing about the application domain. (Rich and Waters 1988, 40)

Thus, programming environments cannot exist for just domain experts or just programming experts, but must be designed to work for both types of people.
The maintenance phase is usually the most costly and lengthy phase of the software development cycle (Yeh 1983). Digital Equipment Corporation (DEC) estimates that 45% of the cost of a software project, over its entire life, is related to maintenance (DEC 1987). The complications arising from maintenance are not reflected in the previous diagram. Maintenance may involve a change in the requirements specification. If the original programmer is no longer available, the maintenance task will require a detailed review of the program design as well as additional coding, testing, and validation. All of these steps require both domain and programming knowledge. The maintenance task, however, is usually the least structured of all of the programming tasks. People who maintain programs are faced with the task of piecing together fragments of information. The current system may have steadily evolved from a domain-specific description to a program understandable only in the context of a specific programming language. Thus the person maintaining the program must deal with the pre-existing program as well as new domain-specific knowledge. Studies have questioned the usefulness of software tools to solve the problem of software maintenance. There is little question that tools alone cannot solve the problem (Benington 1983). One study on the management of software maintenance concluded that that development tools and organizational controls had little impact on the time
required to maintain an application system (Lehman and Swanson 1983). This same study noted that the use of certain tools resulted in a software product of enhanced quality (easier to repair). The use of software tools was found to reduce the amount of time needed for fixing program bugs and to increase the time utilized by programmers for enhancing the performance and maintainability of programs.

Programming environment classifications

One definition of programming environments (Dart and others 1987) is that they only support the coding phase of software development, the so-called programming-in-the-small tasks such as editing, compiling, linking, and debugging. A second, broader definition states that software development environments support all activities associated with software development. These activities include programming-in-the-small tasks, configuration management, and project and team management (programming-in-the-many). Dart further classifies software development environments into four categories (Dart and others 1987).

- language-centered
- structure-oriented
- toolkit environment
- method-based
In a language-centered environment, the operating system and its associated tools are designed around one particular language. Interlisp, Cedar, and Smalltalk are all examples of this type of environment. These environments support an exploratory style of programming which encourages incremental programming and rapid prototyping. In such environments the development environment is the run-time environment. A substantial effort to deliver a stand-alone application is required due to the lack of a separate run-time environment. One disadvantage of language-centered environments is that they are deficient with regard to project management support tools. Since language-centered environments tend to be designed for single workstations supporting a single programmer, a lack of support for managing groups of programmers is not unexpected. Language-centered environments also tend to become too large for one person to comprehend and maintain, possibly because the programmer's application is embedded in the development environment.

The program editor is the key component in structure-oriented environments. The editor may be a syntax-directed program editor, or it may also support the editing of more complex data structures such as history logs or module interconnections, as occurs in the module editor in R^n (Carle and others 1987). A syntax-directed editor is based on an annotated tree which represents the internal structure
of the program being edited. As the programmer modifies the program, the tree is updated. Templates representing program constructs are used to provide fill-in-the-blank type editing tasks. The editor can, and in some cases does, force the user to enter only valid programs. This rigid editing style can be very cumbersome for users, especially when modifying a program. Some editors have solved this problem by providing mechanisms for automatically converting programming constructs from one type to another, such as converting a while loop into an equivalent repeat loop. Other editors allow invalid source code fragments to exist temporarily in order to ease the user’s burden of making program modifications. A more common approach is to allow the editor to operate in two different modes. In such a system, the user can edit the text directly or operate on the program structure as a tree. This dual mode of operation was incorporated into the R^n system when the developers discovered that some users found the syntax-directed editor awkward to use.

The template-driven editing feature, although convenient for beginning users, is not the most useful part of a syntax-directed editor. The major power of such editors derives from their ability to show multiple views of the various structures of a program. Because different representations can be easily generated from the same structure, users can view programs from different levels of
abstractions. Documentation, in the form of comments, annotations, and references to other documents, is one possible view that is displayed by this type of editor. For the purpose of debugging, the editor might display the flow of control within the program. The program's structural representation (as in a parse tree) would be another view presented to someone who is editing the source code. The plain text of the program would be presented to someone interested in seeing the program source on a screen.

Perhaps the most familiar development environment is the toolkit environment. Toolkits are simply collections of programs that support primarily the coding phase of program development. Two widely used examples of toolkits are the Programmer's Workbench (PWB) in Unix and the VMSTM Vaxset for Digital Equipment Corporation (DEC 1987). These systems are extremely flexible. They can be extended by placing high-level interfaces over the operating system. However, this flexibility comes at a price to the user who must ensure that all of the tools are used in a consistent manner.

Toolkit-environments are stronger with respect to configuration management than either language-centered or structure-oriented environments. For example, DEC's code management system (CMS) monitors changes to files to avoid conflicts among groups of users while at the same time allowing team members to work concurrently on the same file.
CMS maintains a record of who is working on a particular file and a historical record of file changes. The CMS record keeping mechanism saves on file storage, since only the changes to a file are kept rather than multiple copies of essentially identical source code. The user can also recover any previous version of a file or identify changes that have been made to different versions of a program.

Files that are related can be collected into groups and classes. A group contains files that are functionally related such as all of the documents for a project. A class contains time-related files such as a particular version of a project.

The last program environment class is termed the method-based environment. These environments tend to support programming functions that are not found in other systems such as requirements analysis, design, and management. They usually support a particular software development approach either informally, requiring the user to do much of the work, or formally, with underlying theoretical models from which the program description or design can be verified automatically.

Anna is an example of a formal method-based environment (Luckham and von Henke 1985). Anna is a specification language for Ada which provides facilities for explaining a program by adding extensions to the language. It provides machine-processable information about such things as
functional requirements and module interactions. Anna provides such information through the use of formal comments. Thus Anna programs are valid Ada programs. A formal comment can provide programming concepts that would not normally be part of the program. Such concepts can be useful for program testing and validation. The purpose of this formalism is to make programs more readable and to provide error checking at compile time and, more importantly, at run time.

Another classification scheme for software development environments is suggested by Perry and Kaiser (Perry and Kaiser 1988). They group environments into four models—individual, family, city, and state. The individual model is supported by tool-induced policies and is normally referred to as programming environments. The family model is based on a policy of coordination and supports such tasks as configuration management. The city model uses enforced cooperation as its main policy. The state model is a proposed environment that provides a generic model with tools and supporting structures for company wide software development. In the state model, the environment tailors itself to meet the needs of each project. The environment manages the differences among projects in order to support the movement of information among projects. An implementation of the state model will occur sometime in the future; since the system would in essence need to learn how
a company developed software, an extremely difficult and poorly understood task.

Major Influences for Programming Environments

Initially, programming environments were merely programs that allowed programmers to use computers. As the hardware became more powerful, it became clear that some of the processing power of the computer could be used not only to run applications but also to help the programmer develop those applications. The first software development environment developed for computers was undoubtedly the Lisp language environment which allowed programmers to develop programs to aid in the software development process. One of the major advances in software development environments was the creation of the personal workstation which was built at Xerox Palo Alto Research Center (PARC) and supported Smalltalk. The last part of this section presents several other systems that have had an impact on the development of programming environments, which include automatic-programming, Computer Aided Software Engineering (CASE), and hypertext.

AI and Lisp

The Lisp language and associated systems have been major influences in the development of programming environments. Indeed, the creation of programming environments has been a major goal of AI research (Fischer
and Schneider 1984). Lisp's equal treatment of programs and data has made it possible to have programs manipulate other programs. The interactive nature of Lisp led to the interleaving of the coding, executing, testing, and debugging processes. Unfortunately this flexibility has usually been at the cost of specification, documentation, and maintenance.

The lack of concern for software engineering practices in Lisp environments is a natural outgrowth of the research setting in which many Lisp programs are developed. In such a setting a premium is placed on fluidity of development and ease of modification (Barstow 1987). In research environments the act of developing the program, not running it, constitutes the experiment (Sandewall 1984). The programs developed in research environments tend to be large and complex, undergo drastic revisions, and are likely to be thrown away before being completed.

This combination of Lisp and research environments has led to the use of a unique method of incremental program development referred to as structured growth (Teitelman and Masinter 1984). Structured growth means that the initial program is allowed to grow by increasing the ambition of the modules. This process is continued recursively as each module is rewritten. The difference between the Lisp programming style and conventional programming is best summarized by Sandewall who states:
Programs in early stages of growth can be executed and programs in early stages of refinement cannot. (Sandewall 1984, 63)

Despite these differences, some valuable lessons have been learned. First, the user must be able to view a program from many different viewpoints: as a parse tree (for editing), as a document (for reading), as a control flow graph (for debugging), and as a visual image (for viewing the source on a screen).

Second, the user must be able to develop rapid prototypes in order to test ideas. The concept of prototyping can be expanded even further to the idea of exploratory programming (Sheil 1984). In exploratory programming environments, it is recognized that some applications are design problems rather than implementation projects. Exploratory programming requires the design and program to develop simultaneously. In a typical system, the design comes first, which usually protects it from unintentional changes. The weakness with this approach is that it also protects the design from intentional changes. This forces the implementation team to push for exact specifications early on in the project. The result is a program specification that is incomplete. When the oversight is corrected (usually when the designers see a preliminary working version of the source code), the implementation team must "kludge" the existing source code.
Lastly, regardless of why environments are developed, the process of developing programming environments is highly experimental. This has led to the development of systems that exist solely for the purpose of developing other programming environments. Gandalf is such a system (Habermann and Notkin 1986). Gandalf generates families of software development environments and frees the programmer of any hand-coding. The environments generated are language-centered editing systems that support source-level debugging and either interpretation or incremental compiling. The Gandalf environment is based on a structure-oriented editor, but it provides for more support of the development process than both language-centered and structure-oriented environments. The Source Code Control System (SCSS) in Gandalf is much more sophisticated than in systems like Unix or VMS. If a comment in a module is changed, the module does not have to be re-compiled since the SCSS has an understanding of the structure of a module. Gandalf also supports project management tasks through mechanisms such as access lists and revision histories. To avoid having to re-link an entire project when one module changes, Gandalf uses invariant addresses for procedures within modules (branch tables). Outside a module, procedures are referenced by a fixed index in an entry vector.
Object-oriented programming

The roots of object-oriented programming go back to the SIMULA language, which first introduced the concept of classes as a generalization of the program block (Stefik and Bobrow 1986). Smalltalk is perhaps the most well known object-oriented programming language. Smalltalk, like Lisp, is well suited to an exploratory style of programming.

Classes in a Smalltalk system can represent almost anything such as numbers, text, databases, text editors, processes, and compilers. This uniform representation provides a framework for organizing information so that software maintenance and modification can be enhanced (Goldberg 1984). Classes allow the programmer to group related concepts so that the concepts do not have to be repeated. The use of classes to group concepts reduces the number of changes that would be necessary if the system is modified. The idea of subclasses inheriting information from the parent class further simplifies the changes to a system.

Smalltalk systems also provide users with a unified interface. The user can easily move from programming, to compiling, to testing, and to debugging. This flexibility is no accident, but rather the result of viewing an application in Smalltalk in the same way as the fundamental units from which the system is built. Alan Kay, the
inventor of Smalltalk, saw the blurring of distinctions between the operating system and applications as the basis for integrated environments (Tesler 1981).

Smalltalk also extends object-oriented concepts to programming environments. Smalltalk commands are really messages that are sent to an object. Commands may be executed either through pop-up menus or by typing the command directly into the system. Commands can be executed from any window, although a separate work-space window is used most often. This same concept serves as the basis of the Macintosh Programmer’s Workshop (MPW), which is built around a single program that serves as an editor and command shell (Meyers and Parrish 1988).

Unix

Unix has had a major impact on programming environments since it was first designed (Kernighan and Mashey 1984). Some of the major features of Unix are:

- Data written by one program can be used easily by another program since there are no file types or structure.
- Files, I/O devices, and programs are all treated the same. Thus a program does not need to worry about where its data is coming from.
Complex programs can be created by interconnecting smaller components. Tools can be interconnected by using the command language interpreter (the shell).

The shell is a major component of the Unix programming environment. It is like a programming language itself, with variables, control flow, subroutines, and interrupt handlers. Prototypes can be written quickly using the shell. Once the prototype has stabilized, it can be rewritten in a high level language. The Unix system, like Lisp, has led programmers to believe in a different philosophy about the software development cycle as expressed by Kernighan.

...almost any program must be continually modified to meet changing requirements, and no amount of initial design work is a complete substitute for actual use. In fact, too much design without experience can lead to a first-class solution to the wrong problem. A program may require a period of rapid and drastic evolution before stabilizing. (Kernighan and Mashey 1984, 189)

Other influences

Other systems that have affected software development environments are automatic programming, Computer Aided Software Engineering (CASE), and more recently Hypertext.

There are three main approaches to automatic programming (Rich and Waters 1988). The first approach uses "very high level" programming languages and is known as the
bottom-up approach. These systems include fourth-generation languages, database query systems, and prototyping languages. The second approach tries to narrow the domain knowledge to such an extent that it becomes reasonable to build a program generator that communicates directly with the end user. A very common example of such a system is a spreadsheet. A spreadsheet is a domain-specific interface for applications such as report writing, financial analysis, and forecasting. End users can use spreadsheets with no programming experience since a spreadsheet can be modified directly. A spreadsheet, however, is more than an interface. Maintaining a spreadsheet can be as complex as maintaining a program. Finally, a third approach to automatic programming suggests that one should build an assistant for the user to help in the various aspects of programming. This approach is really no different than creating an interactive programming environment.

Some systems have attempted to be more than mere assistants. The Programmer's Apprentice (Waters 1984) actively assists programmers with the task of programming. It is based on plans or flow charts in which both data and control flow are represented by explicit arcs. Using the plans, the system generates source code (in Lisp). The user can modify the source code, while a program analyzer will verify that the source code is still consistent with the plans. The programmer builds plans by using a plan editor.
to combine common program fragments from a library of plans. The user converses with the plan editor via a limited English-like syntax. The plan editor can show the user which parts of a plan are incomplete, describe parts of the plan, and display the program fragments that correspond to the plan. The plan editor, however, shows only the resultant source code and not the internal structure of the plan to the user. This is due in part to the inability of the system to show a concise representation for plans. The system does contain a drawer for printing plans, but it is limited to printing simple plans. Figure 2 shows the major elements of the Programmer's Apprentice along with the flow of information: modified from Richard Waters, "The Programmer's Apprentice: Knowledge Based Program Editing," Interactive Programming Environments (New York: McGraw-Hill, 1984), 470.

The current system is limited in that it does not understand anything about data structures, specifications, and interrelationships between library fragments. There is, however, little doubt that future systems must contain detailed and integrated knowledge of the source code.

CASE is related to software development environments, but is primarily concerned with the management of the programming tasks as opposed to programming itself (Weiser, Deutsch, and Kessler 1988). The following definition argues that CASE is more concerned with the information involved in
the development of software and less with the actual task of developing software.

Simply put, a CASE environment is a collection of software tools that "cooperate" with each other for the purposes of storing, controlling, representing, and transforming information related to various phases of the software development process. (Nejmeh 1988, 45)

Originally CASE tools were limited to programs that supported programming environments. However, CASE has recently been extended to include tools which support the requirements, design, project management, configuration management, and testing aspects of software development.

The key component in any CASE system is a database which manages large volumes of diverse information for programmers involved in the software development process. It is the

Figure 2 The Programmer's Apprentice System
fact that they can handle complex relationships among the various components in a development environment that brings CASE and software development technologies together.

Hypertext systems can also be considered programming environments. The basic concept that underlies hypertext systems is that one associates windows on the screen with objects in a database and provides links between such objects (Conklin 1987). The use of static data structures limits the use of hypertext systems in programming environments because most programs are, by nature, dynamic. The system could add links to objects after the program is completely written, but this would only help in the maintenance phase. What is probably more practical is a system that automatically links objects together as in a relational database. This would free the user of the responsibility of creating and maintaining all of the links in a program or document.

The concept behind hypertext, however, is indeed applicable to programming environments. The process of linking objects allows the user to organize text in a nonlinear manner. Although all programs, from a textual point of view, are linear, the relationships among the various routines and data structures within a module are definitely hierarchical. Links in a hypertext system form a network that can be either followed graphically or searched for specific occurrences of some object.
The Components of Programming Environments

The last part of this chapter presents the major components of programming environments. This list includes the system components, user interface, database system, documentation methods, programming language, source code, communication system, and management system.

System components

Programming environments are composed of many parts. The minimum requirements of any system are the processing components such as editors, compilers, and linkers as well as the secondary components such as the file system and the operating system. At the heart of most programming environments is the editor. The editor is the mechanism which allows the programmer to issue system commands in order to invoke the other basic system components.

The Macintosh Programmer’s Workshop (MPW) is an example of a programming environment in which the editor is the central part of the system (Meyers and Parrish 1988). The MPW editor serves as both an editor for source code and as a gateway to the operating system. A system command may be executed anytime by entering the text for the command and pressing ENTER instead of the return key. The output of the command will be directed to the editor and becomes a part of the text that is being edited.
User interface

Because the major purpose of programming environments are to support programmers in their development of programs, it seems natural to assume that some of that support should be directed toward developing an intuitive interface. One of the first, and most influential user interfaces, was the Star user interface from Xerox. It was based on a small set of principles that were consistently used throughout the entire system (Smith and others 1982).

The first principle in the Star user interface stated that the system should support the user’s conceptual model (the concepts that a person gradually acquires which explain the behavior of a system). The Star system used an analogy of the office as its user model. The system did not over extend this analogy. Just as the typewriter can be thought of as a model for a word processor, such a model does not imply that the user will re-type a page whenever he wishes to edit a page. A programming environment might begin with a similar user model and then extend it to other tasks associated with program development. The model may hide some of the more mundane tasks such as compiling and linking, while enhancing the more prominent tasks such as program editing and execution.

The second and most difficult principle that the developers of the Xerox Star wanted to achieve was
consistency. Consistency simply means that the system will react in the same way whenever the user enters the same commands. The Star interface achieved consistency by adhering to a strict set of paradigms for the operations a user performs in the editing environment. These paradigms were limited to editing text, retrieving information, and copying.

Other principles used by the Star included the use of the screen as a "visual cache" which meant that something visible happened for every command that was entered, and the use of "What You See Is What You Get" for document creation. The designers enforced the notion of consistency by using universal commands such as move, copy, delete, show properties, copy properties, again, undo, and help which performed the same functions regardless of which object was selected. Finally, a user interface should be simple to use. The goal of simplicity is to make the system uniform, consistent, and to be nonredundant. As Alan Kay states.

Simple things should be simple; complex things should be possible. (Smith 1982, 274)

Database system

The database system, though possibly considered a system component, is an integral part of programming environments because of the voluminous information that must be maintained. The relationships between data varies from
the simple relationships between program modules, procedures, variables, and documentation to the complex relationships involved in the execution of an entire programming system.

The information associated with programming environments often requires the storage of more than just data and static relationships. Some environments (such as Microscope and Marvel which are discussed in chapter three) use knowledge-bases to store both implicit and explicit facts along with rules for inferencing about the data. In such environments, the database is an active participant in the development process. Changing the source for a file may automatically trigger its compilation and even the execution and verification of related test cases. For example, the Marvel system (Kaiser, Feiler, and Popovich 1988) refers to its database as an object base since it contains objects that represent both the system and its development history. Objects in the Marvel system include modules, procedures, types, designs, user manuals, and just about everything else in the system. The most unique thing about Marvel is that it is a file-less environment. The user only sees logical entities and does not have to worry about how they are represented physically in the system.

The database in programming environments also serves to integrate the various individual tools. If tools have access to common data, then the user can spend less time
managing the interactions between the various pieces of data. In the Genera programming environment (Walker and others 1987), for example, the compiler maintains a database of argument lists which the editor, debugger, and other tools access to offer on-line help. The compiler also maintains a list of warnings and error messages which the editor uses to inform the user of a particular problem.

The classification of the data in a database can be arranged in a hierarchical or network model. Regardless of which classification scheme the designer selects, he should model the relationships between various objects in the programming system. These relationships can be structural, semantic, or system dependent.

Documentation methods

Complexity may be the bane of every programmer (Fastie 1988), but documentation is often viewed as nothing more than a necessary evil. Jon Bentley, author of "Programming Pearls" in the Communications of the ACM (Association for Computing Machinery), expressed the sad state of most documentation when he wrote:

... most programs are written to be executed, a few are written to be maintained, but almost no programs are written so someone else can read them. (Bentley and Knuth 1986, 364)

He wrote this as a preface to an article on Knuth's "literate" style of programming (WEB). WEB (Knuth 1984) is a mixture of a programming language (initially Pascal) and a
documentation language. The idea is that the documentation and the source code are interwoven (hence the name WEB) into a single "stream of consciousness." There is no rigid method of programming enforced. WEB allows a programmer to specify a complex piece of software by specifying its single parts and simple relations between those parts in whatever order is best suited for comprehension. The WEB system then takes this output and produces the documentation that describes the problem (a process Knuth calls weaving) and a machine-executable program (called tangling since the resulting source code is barely readable by humans). An outline of a sample WEB program is shown in figure 3. The documentation is not inserted in the source code, but it appears as prefaces to each section. Since sections can be defined regardless of the particular language syntax, the developer can write each section in whatever order seems appropriate.

The key feature of this example is that the documentation is tightly integrated with the source code. Most systems split the documentation between comments in the source code and separate design documents, with no explicit links between the two. As will be shown later, the approach that CNS takes is to provide explicit links between source code and documentation.
<the program> ::= { this is the main program }

program x (output);
<constants>
<variables>
begin
 <initialization>
 <calculate>
 <output it>
end.

<constants> ::= { sections are added as defined }
const NUM = 1000;
MAX = 2000;

<variables> ::= 
var x, y: integer;

<initialization> ::= 
x := 1;
y := 2;
<more initialization>

<calculate> ::= 

<more initialization> ::= 

<variables> += { sections can be added on to }
var z: real;

Figure 3 An example WEB program

Programming languages and source code

Mark Weiser of the Xerox Palo Alto Research Center once compared software to kitchen appliances (Weiser 1987). If kitchen appliances were like programs, he stated, they would be gray, featureless boxes that processed food. You would never be sure exactly what happened to the food. Working in a kitchen would become a matter of becoming familiar with
the idiosyncrasies of these boxes. You would never get the results you really wanted. Several times a year the manufacturer would send you a new box; it might do more, but since you were never sure what it did in the first place you would never know. Weiser goes on to argue that even though not perfect, the source code is the best thing for describing how a program actually works. Since a programmer cannot anticipate all of the possible uses of a program, any details omitted may later become vital to one's understanding of the program.

There is little question that source code is an essential ingredient in programming environments. A system's ability to provide the source code for a program, however, is complicated by problems such as the loss of proprietary information, the proliferation of different, possibly unsupported versions of a program, and the availability of disk space. The problem with providing source code to the user may not be a question of availability, but what will the user do with it once he gets it? In a system such as WEB, the source code and documentation are tightly integrated. The Unix operating system provides on-line documentation to system callable functions and, in some cases, even the source code. Thus, in both of these systems, the user has access to the related documentation for the source code.
The question of programming language support is usually based on many reasons such as financial status and emotional ties. A particular language may be supported only because people like it. If a company has a large amount of money or resources invested in a particular language, there is little doubt that the language will be supported by a programming environment. Alternatively, programming environments can offer support for multiple languages. Knuth has even suggested that every good computer scientist should be able to use a system that is programmed in several different languages (Knuth 1984). However, multiple language support is often achieved at the expense of an efficient implementation.

Communication and management system

Although programming environments are primarily concerned with programming-in-the-small tasks, there is still a need for programming environments to communicate with all parts of the system and manage the resources at the disposal of the programming environment. An example of how programming environments can manage resources is the use of contexts. A context is used for the purpose of collecting system components into information domains. These information domains can represent anything from a set of test routines to an entire programming project. Contexts can also be used to manage different versions of a
programming project. To support configuration management, each version of a program can be assigned to a different context. The context, in this case, could represent the entire programming project or possibly a particular version of the source code.

The various tools in a programming environment must also communicate with each other. From a rather simplistic view, whenever the individual programming tools in a system reach a certain level of integration, they can be classified as a programming environment. Although there is more to a programming environment than a set of integrated tools, a programming environment cannot exist without a high degree of integration among its individual components.

Putting It All Together

The software development process can be broken into five major phases: project definition, project design, coding and integration, testing and validation, and maintenance phases. The maintenance phase is considered the most costly and lengthy phase. By conservative estimates, the maintenance phase consumes at least half of the entire software development time (Lehman 1983, Digital Equipment Corporation 1987). Solving the maintenance problem will not be easy. The use of development tools alone will not solve the problem.
Programming environments primarily support the coding phase of software development which includes such tasks as editing, compiling, linking, and debugging. A more encompassing software development environment attempts to support all of the activities associated with software development. Programming environments can be classified into four categories - language-centered, structure-oriented, toolkit environment, and method-based. In a language-centered environment, the operating system and associated tools are designed for a particular language (such as Interlisp and Smalltalk). The editor is the central component of a structure-oriented environment. A toolkit environment consists of a collection of tools designed to work together for the purpose of supporting the coding phase of development. The method-based environments also offer support for the requirements and design phases, and the management of software development.

The Lisp community had the earliest influences on the development of effective programming environments. This was largely due to the interactive nature of Lisp. An interactive programming environment makes the job of the developer easier than using a compiled language. Other influences on programming environments include object-oriented programming systems, Unix, automatic programming, Computer Aided Software Engineering (CASE), and Hypertext.
Programming environments are composed of many smaller programs that must interact together. The system components include the various development tools such as editors, compilers, and linkers as well as the operating system. The user interface is simply the viewpoint afforded to the user of the system. The Database system (which may be a system component) integrates the individual tools in a programming environment by providing a common access to the data needed to develop a program. The method of documenting source code and the choice of a particular programming language are difficult decisions that must be made in the design of programming environments.
CHAPTER 2

THE C NAVIGATIONAL SYSTEM

The Structure of CNS

The C Navigation System (CNS) is an editor-based programming environment. The term "programming environment," as stated in the definition of CNS, refers to the design, coding, and maintenance phases of software development. CNS is editor-based because the editor is the primary interface which allows the programmer to use CNS. In CNS there is one editor for both program source and documentation, with explicit links between the source code and its associated documentation. Both program source and documentation are displayed on a display screen using a window format.

The implementation of the windows will not be presented in this paper, since the major reasons for using windows is that they provide a mechanism for grouping specific sections of documentation or source code. The windows could be implemented either graphic-based or text-based, but it will be assumed in this paper that windows will be implemented graphically. The graphic implementation of windows is more flexible than the text-based system because it allows for the specification of more character attributes (such as bold faced, italics, and underline), and it can use icons.
(pictures) to represent windows. Each window may itself contain other windows, but a window must contain only source code or documentation (for purposes explained later). Documentation may include design specifications, history lists, program interfaces, or any other related text. While developing a program, the user can switch easily between editing the source code and editing the documentation.

The use of links to navigate through source code and documentation is similar to that used in hypertext systems. CNS differs from a hypertext system in that links are made implicit by the use of the name of a window or program function to tie the objects together. If a document references the name of a window or a function calls another function, then the system provides an automatic link between the two references. The user may add other explicit links if needed. However, the links provided by the system can be followed in either direction.

The system is able to distinguish between windows that contain documentation and windows that contain source code. The reason for this feature is that the system uses the name of a function for either source code or related documentation, depending on the context in which the name is used. For example, the default window for a given name will be the source code if the user is currently viewing source code, and the default window for a given name will be the documentation window if the user is currently viewing
documentation. The user can over-ride the default window name by qualifying the name with either the source code or documentation attribute. The programmer can also qualify any name to specify related documentation. Related documentation includes both documentation created by the programmer and windows automatically created by the system.

The system will automatically create two windows for each function. The first window created by the system is the history list. The history list contains the author of the function, the date the function was created, and a list of modifications to the function's source code. Although the system creates the history list, the user must enter some of the fields such as the name of the author, while all other data are provided by the operating system. The second window provided by CNS is the function interface. The function interface contains a list of the parameters of the function, the local variables of the function, global variables referenced by the function, and a list of the other functions called by the function. If the programmer creates a documentation window with the name of the function, the system will display that particular window whenever the programmer asks to see the documentation associated with the function's source code. The user may also provide an explicit link between the source code and other related documentation.
If there is no implicit link (a documentation window with the name of the function) or explicit link (a documentation window name provided by the programmer) from a function to its associated documentation, the system will provide a link. The system will search for documentation associated with the parents of the function’s window name. This will provide a hierarchical link of the documentation for a programming project. In figure 7 for example, the functions GetToken and GetId inherit any link to the documentation for parse.c along with the link for the documentation to the project window (CNS in this case).

CNS tries to integrate the source code with documentation, which is similar to the concept as introduced in WEB (Knuth 1984). In WEB, the source code and documentation are both contained in the same file. Because source code and documentation in WEB can be mixed together, it is natural for the programmer to write the source code and documentation at the same time. CNS uses this same concept, but implements it in a different manner. Although physically distinct, the source code and documentation are connected through system and user-supplied links. The programmer may also switch between editing the source code and editing the documentation at any time. In short, CNS is a distributed version of the localized mixture of source code and documentation as represented in WEB.
While in the CNS editor, any piece of text (source code or documentation) can be enclosed within a window. Placing text within a window does not change the meaning of the text. Windows merely provide a mechanism to group portions of related information. If there is text before or after a section of source code that has been converted to a window, the text retains its previous ordering. When a window is expanded, the surrounding text moves up and down to make room for the resizing of the window. Windows can be assigned logical names similar to file-names. The window’s logical name is used to associate it with other windows.

Each window can be independently scrolled either horizontally or vertically. Windows can also be moved, copied, or deleted as entire units. Windows can be collapsed, causing the text within the window to become part of the surrounding text (as shown in figure 4). When a window is collapsed, all references to the window are changed to point to the containing window. For purposes of implementation, the modification of references can be performed either as the window is collapsed or later when the old window is referenced (in which case a list of all collapsed windows would be maintained).

When editing source code, the system will normally configure each function as a separate window (figure 5). A single command performs this operation in order to
facilitate converting existing programs to CNS format. The reverse process removes all windows from a program so that the program can be converted to either a normal text file or a group of files (each window in a separate file).

The default logical name of a window will be the name of the function in the window. This does not prevent the user from entering more than one function in a window. When a window contains more than one function, the user will provide a name for the window. The default name will be the first function in the window.

A window that contains a function may itself contain other windows. The nesting of windows provides the user
void parse() {
    int index, temp, stat;
    char name[5];

    get_file( srcf );
    cpy_file_num = 0;
    src_line_num = 0;
}

main( int argc, char *argv[] ) {
    char filename[81], *s;

    cmd_options.x = FALSE;
    if (argc-- < 1)
        error( INVALID_ARGUMENTS );
    else
        while (--argc > 0 && (**argv[0] == ' ')) {

Figure 5  The standard CNS window configuration

with the ability to view multiple portions of a function at
the same time (see figure 6). This idea is similar to other
editors that provide split screens. However, when a CNS
window is assigned, it becomes a permanent part of the
system, until the user removes the window by deleting or
collapsing it. In this manner, the developer of a
particular function may assign pieces of the source code to windows to emphasize the structure of the program. The assignment of a particular piece of a program, some algorithm for example, to a window provides a logical link between the source code and the documentation as well as other parts of the program. This process of breaking a function into smaller pieces can also be accomplished by splitting the code into separate functions. The advantage of using separate windows is that the individual windows retain the scope of the containing function. When a function is split into smaller functions, the programmer must rewrite the code to account for the change in the scope of variables. Windows can be assigned or re-assigned without any changes to the code.

CNS windows will have three different display formats. The long-format has been used in all of the previous examples. A long-format window contains the text of the source code or documentation. The text can be scrolled, and the window can be sized so as to display varying amounts of information. A window must be in the long-format in order to edit the text in the window. The long-format window may require the use of large amounts of display space, even if only a small part of the text is visible. To alleviate this problem, each window will also have a short-format version.
```c
main( int argc, char *argv[] ) {
    int i;

    get arguments
    while (--argc && (*++argv[0] == ' ')
        for (s = argv[0] + 1; *s; s++)
            switch(tolower(*s)) {

    initialization
    clearscreen();
    init_files();
    for (i = 0; i <= MAX_OPTIONS; i++)
        set_option( flags[i] );

    /* now call main simulator */
    if (ok_to_call)
        simulator( open_msg );

    termination
    save_memory( save_mem_flag );
    save_options( flags );
    close_files();
}
```

Figure 6 Multiple windows within a function

that displays a brief summary of the function or
documentation contained in the window.

The need for the programmer to enter the summary
information (automatically retrieving the data seems
unreasonable) may itself cause more problems then it solves.
Windows can also be reduced to iconic representations. The
system provides a set of icons to represent common types of
routines such as a disk for I/O routines or a monitor for screen control routines. The user may also create custom icons through a special icon editor.

The use of icons to represent windows is helpful for displaying a software library of functions. A software library contains groups of icons with each icon bearing the name of the associated function. Common routines can be grouped together using similar icons that make them clearly visible to the programmer.

CNS requires the programmer to provide a module definition for each programming project. A project encompasses all of the files (both source code and documentation) associated with the development of a particular system. One or more windows can be associated with each file in a project. The module definition lists the files that are part of a programming project. CNS allows the user to enter a brief description of each file in the list. The module definition also specifies how the individual files are to be compiled and linked. The interdependencies between files does not need to be specified (as they would in a "make" utility), since CNS can tell which files need to be recompiled whenever any file is changed. This information can be extracted easily by the syntax-directed editor.

For each source file in the module definition, CNS will create a function list. The function list specifies each
function in the source file along with a brief description of the function. A programmer can easily scan all of the functions for a project by referring to the module definition and the function lists. If more detail is required, the programmer can view both the source code and the documentation of a function.

In figure 7 the module definition window is labeled CNS Modules. This window displays the name of three source files in the CNS project (all of the filenames may not be displayed at once). In the example, the function list for parse.c is also displayed. The function list contains the name of two functions, GetToken and GetId. The source code for GetId is displayed below the function list. When the function SymSearch within GetId is selected, several windows associated with SymSearch are also displayed. The various windows contain displays of the CNS design document, the history list for SymSearch, the interface definition for SymSearch, the source code for SymSearch, and a list of related test cases for the file parse.c.

Windows in CNS are not restricted to a sequential structure as implied by figure 6. Since the name of a window or function provides a link between the identifier and the identifier’s references throughout the project, a program can be traversed hierarchically as a network. Though this representation may create some confusion among people reading the source code, it is useful for two
Figure 7: An example with various types of links

### CNS Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>screen.cns</td>
<td>Screen control</td>
</tr>
<tr>
<td>cns.cns</td>
<td>Main control code</td>
</tr>
<tr>
<td>parse.cns</td>
<td>Syntax analysis</td>
</tr>
</tbody>
</table>

### parse.c

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getToken</td>
<td>Get next token</td>
</tr>
<tr>
<td>getId</td>
<td>Return token id</td>
</tr>
</tbody>
</table>

### parse design

SymSearch was written by Tom B. using the algorithm from Knuth.

### SymSearch history

- Created: 8/12/88
- Author: Tom B.
- History: 8/22/88 made sympitable global

### SymSearch interface

```c
tok SymSearch(char *s) {
    tok t;
    global variables used sympitable
    calls binsearch
}
```

### SymSearch

```c
tok SymSearch(char *s) {
    tok t;
    t=binsearch(symtable)
    if (t == NO_SYMBOL) {
        t=binsearch(s);
    }
}
```

### parse tests

- stest0019.c
- stest0020.c
- stest0030.c

---

Figure 7: An example with various types of links
reasons. First, it allows the existing source code to be easily adapted to CNS. As programmers become familiar with the system, they can add more structured links. Second, a hierarchical structure allows a user to "jump out of" the structure at any time while viewing source code.

Figure 7 shows a possible set of windows for a program. The module window lets CNS know which files are part of this particular programming project. CNS creates a window for each function listed in the file. In this example, the user creates two additional links. The first link associates a portion of the design document with the function SymSearch and hence to any references to SymSearch. The links to the history list, function interface, and source code for SymSearch are automatically generated by CNS. An additional link created by the user connects a list of test cases that are applicable to testing function SymSearch. If the section of text in parse design devoted to SymSearch was assigned to a window named SymSearch, then an explicit link from the source code to the documentation would not have been necessary since the system would have provided an implicit link.

A Walkthrough of CNS

The first step when using CNS is to create a module definition file. The module definition file contains an entry for each file in the project under development. A
file may contain source code, documentation, or executable code. For each executable file listed in the module definition file, the user supplies a list of the source code and library files necessary to create the executable file. As previously mentioned, the CNS editor will maintain all current information concerning executable files. In addition to relieving the user of the responsibility of building a "make" file, the editor is smart enough to know when a file does not need to be recompiled just because a comment is changed.

For each source file, CNS will create a function list. The function list will contain an entry for each function in the source file. The list will be updated as new functions are added or removed. The syntax-directed editor within CNS will inform CNS of such changes. When a function is created, the user must supply a brief definition of the function. This definition is displayed in the function list and whenever the short-form of a function window is displayed.

CNS also maintains an identifier dictionary of the simple identifiers used in each source file. Simple identifiers include variable names, constant names, and type definitions. Given any identifier, the system can retrieve all references to the identifier throughout the project. This search is not the same as a link since a search for a simple identifier is restricted to specific files and
functions. A link, on the other hand, is much more powerful. The use of a function name provides a direct link to the function and any associated documentation. No searching is required to follow a link since the CNS editor maintains a list of all the links in a project. In the case of a function that is declared local to a source file, the link may have to be qualified with a filename since duplicate names can occur. In such a situation, the system will notify the programmer and provide a list of file names from which he can select. A link can also be used to obtain more detailed information such as the short-form, the history list, the documentation, and the interface for a function.

The programmer can create documentation at four levels. The different levels provide a hierarchical structure to the documentation and makes it easier for a programmer to scan the documentation for appropriate information. The levels of documentation in CNS encompass the following domains.

- Project wide
- File specific
- Function specific
- Window specific

An example of the hierarchical document structure is shown in figure 8. In this example, the window specific
Figure 8  An example of hierarchical documents

document is entitled Tokens. A link has been established between the use of the variable test and the window Tokens. If the person who views this program wants to see more information on tokens, he can follow the link from the variable test to the document on tokens. The system provides an automatic link from the function GetId to the window GetId Design. To navigate through the hierarchy of documents for tokens, the user would pass from the Tokens window to the GetId Design window, to the Parse Design window, and finally to the CNS Design window. The Screen
Design window would not be traversed in such a search. The advantage to the hierarchical arrangement is that it gives the user the ability to browse through a series of documents without having to know the specific links for each document. If a document on a specific level is missing, the document on the next higher level becomes the next part of the chain.

CNS also supports the execution phase of software development. When the user runs an executable file, the system verifies that all of the associated object files are current. If a source file or one of its related files (such as an include file) has been changed, the source file is recompiled. Since the syntax-directed editor in CNS maintains a syntax tree for each program, the syntactic information can be passed to the compiler. The compiler must also include linenumber and identifier name information in the object file. This information is used by the CNS debugger to support source level debugging.

Source level debugging lets a user single step through the source code, one statement at a time. Though not the same as an interpreter, it still provides essentially the same results. Variables, including records, may be accessed or modified by symbolic name instead of machine addresses. Breakpoints may be set at specific statements or on entry or exit from functions. A breakpoint may also be set to occur when a variable acquires a specific value or range of values. As an example, if the programmer wants to detect
when an index variable is going out of range, he can set a breakpoint to occur when the value of the index exceeds its maximum value. The value of specific variables may also be monitored by the programmer as a monitoring request. A monitoring request causes the value of a variable to be displayed in a separate monitoring window. As the variable changes, the values in the monitoring window are updated.

The programmer may also view any source code or documentation in the project. The programmer may also modify the documentation to reflect design or program changes. Modifying the source code may result in a loss of integrity between the source code and object code (a copy of the old source code could be retained for debugging purposes and discarded later). Therefore, the decision as to whether to allow the modification of source code while debugging will be left as an implementation issue.

The final area of support provided by CNS is that of testing and validating programs. Typically "test" managers execute the program using specific inputs and compare the outputs of that program to a user-defined file called a benchmark. Although this process is called the "testing" phase, the use of benchmarks is actually more like a program validation. Validation is the process of verifying that a program meets its requirements. This approach to validation, however, has one major weakness which is evolving changes. Each change to the output of a program
requires a corresponding change to all of the benchmarks. When validating multiple releases of a program, however, the use of benchmarks is an indispensable tool. Since benchmark testing can be performed entirely "outside" of CNS, its design will not be presented here.

Testing a program requires more than simple verification of the output of the program. A test should also inform the programmer about where the problem lies. Benchmark tests only provide "black-box" testing; the program tester is only concerned with input and output and is unaware of how the program actually works. CNS will also provide a programmer with the ability to test individual functions and modules which is the "white-box" method of testing. White-box testing is not without its drawbacks, especially if the programmer performs the test. Designing a test to exercise a function, or more precisely the design of the function, is a difficult task. Cleanroom software engineering (Mills 1988), a recent approach, suggests that one should assign the task of testing to a separate group of programmers or designers.

The example in figure 7 contains a list of test cases for the module parse. Each test file may test one or more functions in the parse module. A test file must specify the input to each function to be tested and the expected output. CNS will execute the test cases when instructed by the programmer and report the results. A test file may also
contain a special section which initializes any global variables either directly or indirectly through other functions. The use of global variables, however, limits the usefulness of module testing since it is often impossible to track modifications to global variables.

**How can CNS Improve Software Development**

The need for intensive documentation cannot be overstated. Herbert Benington, who worked on the Semi-Automatic Ground Environment (SAGE) project for the Navy said that documentation must be done on every level from sales brochures for management to source code listings for maintenance engineers (Benington 1983). He realized that such a vast amount of documentation would require us to develop new methods and languages to help organize the material as well as make extensive use of the computer to assist in the process. This statement was made in 1955 in reaction to a project that required half a million instructions (one-quarter were devoted to operational use). By tightly integrating the source code and program documentation, it is hoped that CNS will provide a method that will help documentation to remain current as the software continues to change. Using CNS, the programmer can develop documentation along with the software. After the prototype software is developed, it can be discarded (or at
least set aside for reference) while the documentation can remain until the final system is developed.

The other major area of software development addressed by CNS is maintenance. Schnedewind (Schneidewind 1987) listed four tasks that a maintenance tool needs to perform.

- Look for structure (procedural, control, data, I/O)
- Follow data flow (where the data originates, where it is used)
- Follow control flow (consequences of executing each path must be understood)
- Understand different versions of a program

CNS helps a programmer to define the structure of a program through the use of windows and links. It also allows the programmer to navigate through a program’s source code, data, and documentation. CNS also helps a programmer to follow the flow of data and program control through source level debugging (single stepping of statements and breakpoints for both code and data). CNS supports primarily the design, coding, and maintenance phases of software development. As such, it is not designed to directly support version control. Version control is only supported through the use of contexts. A context specifies a group of related files such as a particular release of a project. It is up to the system outside of CNS to manage the various contexts. Version control, however, is only part of the configuration management task. The other half of
configuration management requires that the system maintain the integrity of a project. The integrity of a project is maintained by performing such tasks as compiling a source file, when the source code changes, or re-linking modified object files, in order to keep executable files current. CNS performs the task of maintaining the integrity of a project automatically by using the module definitions provided by the user.

What is Missing?

Competent personnel remain the most significant factor in productivity and quality on a software development project... (Nejmeh 1988, 48)

This statement effectively summarizes what CNS cannot do for the program development process. CNS cannot make poor programmers better programmers, nor can it replace programmers or the art of programming. Like all programming environments it is just a tool. Like all software tools, CNS must overcome two major obstacles before it is accepted. The first major obstacle is the result of the often weak software background of corporate management which causes them to be unsympathetic to investments in software tools (Yeh 1983). This problem is compounded by the fact that software tools are generally paid for with funds earmarked for specific projects. This often means that tools must be justified on the basis of their narrow purposes rather than for broad-based company needs. The final, and perhaps the
most difficult obstacle to overcome, is that of programmer's resistance to change. Programming is a discipline that evolves over years of practice. After years of experience, programmers tend to develop specific strategies for writing software. New tools, even if they are better, must be adapted to fit the old established methods. If the new tools can be easily adapted, or if they are quickly accepted by the programmers, then they have a chance to make an impact.

Programmer's resistance to change probably explains why the Unix environment, with its simple yet powerful set of tools, has had such a major impact on software development. Although Unix does not have the most sophisticated tools, it is easy to use. Users can select the tools that fit their needs. If a particular tool does not fit, the programmer can modify the tool or develop a new tool that does the job. Ease of use has been a major concern in the design of CNS; however CNS, unlike Unix, is not a tool-kit environment. CNS is not composed of separate pieces that can be linked together by programmers to create a programming environment. The flexibility of CNS ensues from the ability of programmers to create networks of source code and documentation.

This ability to create a network from source code and documentation is also a potential weakness. Even a sequentially organized system in CNS will be navigated as a
network. Using the names of functions to access the network, a user will be able to navigate through a program and all of the programs documentation. One problem with using this type of navigation system is that the users can easily get lost in the system. Conklin refers to this problem as getting "lost in space" (Conklin 1987). In a large network, information can be difficult to locate. Providing the user with clues that will help him find information is not easy. A "highway map" of the network may appear to be more like a tangled web than an aid.

A data query system may help to alleviate the problem of getting "lost in space" while searching through a network of links. The Hypertext Abstract Machine (HAM) provides a set of query operations to traverse links in a hypertext-based system (Bigelow 1988). Conditional searches can be performed using various system-defined attributes. The result of using conditional searches is a reduction in the number of links that must be traversed for a query operation. The HAM system also allows the user to collect related links into a context. The use of contexts in the HAM system allows a programmer to select a subset of a project and make local modifications to the subset. After making the necessary modifications to the subset, the programmer merges the changes with the system.

The HAM query method could be expanded by allowing user definable attributes. For example, a system may contain
predefined attributes for local and global variables. By searching for all variables with the global variable attribute, the user can find all references to the global variables. The user may then define a new attribute for a specific set of global variables and use this attribute to search for only the accesses to those variables.
CHAPTER 3

CNS AND OTHER ENVIRONMENTS

The purpose of this chapter is to examine CNS in light of several other environments. The comparisons are not meant to be exhaustive, but instead concentrate on the characteristic features of the particular environment that is being discussed. The environments presented are Microscope, R^n, Cedar, PECAN (with family member BALSA), and Marvel. These environments cover a wide range of programming languages ranging from Fortran and Pascal to Lisp. The individual environments also vary as to their purposes. As will be seen in the following sections, the purpose of an environment defines much of its behavior. The purpose of Microscope, for example, is to help programmers comprehend and modify programs. As such, Microscope emphasizes the presentation of multiple views of both the static and dynamic features of a program. The same can be said of a teaching tool such as PECAN. The purpose of R^n, on the other hand, is to help programmers develop more efficient programs. Thus, R^n concentrates on providing information to the compiler so that the compiler can generate more efficient code. As the programmer moves from programming environments to software development
environments, the emphasis shifts towards the sharing of information between programmers.

The consequences of making a change to a program also becomes more critical as the number of programmers and programs increases. A major concern the programmer has when operating in an environment is, "If I change something, what are the consequences?" Answering such a question requires extensive knowledge of all of the programs in a project, along with knowledge of all the rules for deciding the impact of changes. Changing something such as a comment should have little impact on a program. However even a simple variable change can completely change the meaning of a program. Consider the following two program fragments.

```c
int index=0;
func() {
    index=10;
}
main() {
    func();
    while(index--)
        printf("+");
}
```

The program on the left prints ten plus signs and then stops. The program on the right prints nothing. The only difference between the two programs is the added declaration of the local variable `index` in the function named `func`. This simple change can be easily detected by a compiler as a variable that is initialized but never used. But imagine a program with several hundred functions and a complicated
flow of control. A more difficult aspect of this problem is deciding if a change to a program preserves the original meaning of the program. For example, the following two statements appear to perform the same operation (converting a single character to lower case).

```c
charx = tolower(charx); if (charx >= 'A' && charx <= 'Z')
    charx = charx + 0x20;
```

On most systems, the compiler will convert `charx` to a lower case character. However, if the computer uses the EBCDIC character set, the statement on the right will not work because of the non-alphabetic characters embedded in the range from 'A' to 'Z.' So, even if an environment "understands" a program, the knowledge will be worthless without an equally thorough knowledge of the system on which the programming environment resides.

**MicroScope**

Microscope is a program-analysis system that helps programmers comprehend and modify programs for Common Lisp and Common Objects (Ambras and O'Day 1988). Microscope's key feature is a knowledge-base. The knowledge-base stores implicit data representations as well as explicit facts. Rules provide inferencing capabilities on the data in the knowledge-base. Integration is achieved by allowing individual tools to share information through the database. The ability of tools to access a common knowledge-base means
that the programmer will spend less time managing the interactions between tools.

Like CNS, Microscope provides the programmer with multiple views of a program. These views can be provided at different levels of "magnification" so that programmers can browse through a module's structure and zoom in on details of interest. Other views provided by Microscope include static references, dynamically determined call order of functions, and execution histories. Documentation and source code are linked through the use of annotations which are similar to links in CNS.

Microscope users issue monitoring requests to view the dynamic analysis of a program. A monitoring request specifies which events to look for, such as the value of a specific variable, and what to do when the events occur, which may include the modification of other variables. The minimum response to a monitoring request is storage of an execution history of the program. A complete profile of a program's execution can be obtained by saving the timing information along with the execution history of a program.

Flow analysis, a proposed mechanism, is used to locate all of the statements in the path that caused a specific event to occur. Flow analysis should drastically reduce the amount of time a programmer spends in the debugging phase of software development. A typical debugging session consists of walking (or wandering) through the execution of a program.
to find the cause of a specific problem. Flow analysis allows a programmer to ask the system for the chain of events that led to the problem, assuming the problem can be traced to a specific event. Although not always necessary for debugging programs, flow analysis should provide the programmer with a more structured method of tracing the cause of a bug in a program.

\( R^n \)

\( R^n \) supports the development, testing, debugging, and maintenance of large Fortran programs. \( R^n \) is designed for expert programmers whose primary goal is the development of efficient programs. Independently compiled modules severely restrict the level of optimization possible with a Fortran compiler. To address this issue, \( R^n \) uses a mechanism called compositions. A composition is a hierarchical specification of a program's structure in terms of modules (independently compiled procedures) and other compositions. A composition editor checks for consistency in procedure calls and completeness. The composition editor is, in essence, a syntax-directed editor for a module interconnection language. The composition editor allows programmers to build incomplete programs. The environment can complete an unfinished composition by including existing code from a library or previous program. By completing an unfinished composition, the environment facilitates the programmer's
task of constructing a program which is similar to pre-existing code.

A mirror image of the composition editor is the module editor. The module editor is a syntax-directed editor that allows the programmer to edit programs. The module editor plays a major role in gathering information to support the compilation process. By contrast, the CNS editor gathers the same information but primarily for the purpose of supporting the development and maintenance phases of programming.

Programs in R^n are executed by using the execution monitor. A highly optimized program is difficult to debug since the executable code may bear little resemblance to the original source code. In order to avoid the overhead of interpreting the source code, R^n uses a hybrid approach. The execution monitor can execute compiled and interpreted code. Thus, it provides a system whereby modules to be tested can be interpreted, while stable code can be directly executed.

Cedar

Numerous programming environments have been developed at the Xerox Palo Alto Research Center (PARC). The Smalltalk system is an interpreter for an object-oriented language. The Interlisp programming environment (Teitelman and Masinter 1984) introduced many new features that are now
common in other programming environments. The Interlisp Masterscope facility in the Interlisp environment analyzes a program to determine information such as which functions are called, and how and where variables are changed or referenced. The DWIM (do what I mean) utility is invoked whenever the system detects an error. DWIM attempts to guess user's intentions by using a spelling corrector to find the closest match within a list of relevant items. The DWIM feature is based on the assumption that system facilities should make reasonable interpretations when given unrecognized input. The Programmer's Assistant in Interlisp records the user's input in a history list along with the side effects of each operation. The REDO command in Interlisp uses the history list to repeat an operation. The now widely used UNDO command cancels the effect of a previous operation.

The Cedar programming environment (Teitelman 1984), also developed at PARC, was one of the first interactive, experimental programming environments based on a strongly-typed, compiler-based language. Cedar consists of a structure-oriented editor, a document preparation facility, and various tools for creating and debugging programs. Although Cedar is a programming environment, non-programmers also use it to prepare documents. The current system supports the Cedar programming language, although the original intent was to support other languages such as Lisp
and Smalltalk. The Cedar editor (Tioga) is used for editing both documentation and programs. The Cedar editor is similar to the CNS editor, except that the Cedar editor is not based on lines or paragraphs but a tree structure. The purpose of the tree structure is to allow a programmer to represent a hierarchical structure explicitly. When the display of lower level nodes in the tree structure is suppressed, the top level nodes effectively provide a table of contents. Each node can be traversed in order to obtain a greater level of detail. In CNS, a hierarchical structure can be created through nested windows or with links.

PECAN

PECAN (Reiss 1984) was developed at Brown University as was a related environment called BALSA (van Dam 1984). The PECAN environment was designed to support student programming. PECAN provides the user with multiple, dynamic views of a program as it is being executed. These views include the program source, data type diagrams, flow graphs (a computer-generated flowchart of a program), and the symbol table. The program source is displayed using a syntax-directed editor. Built-in and user-defined data types are displayed graphically. If a data type contains a pointer that would generate a recursive display, the recursion is displayed to only one level. The symbol table view includes a representation of the scope of each variable
along with the class of each variable: variable, type, or label.

The BALSA system, which the developers plan to integrate with PECAN, uses overlapping windows to display the calling sequence of procedures with the most recently invoked procedure always being visible. The BALSA system can draw pictorial views of common data structures such as pointers, records, and lists. An example display of a C program is shown in figure 9. In the example in figure 9, the execution of the function `search` causes an overlapping window to display the source code for `search` as it is executing the function `test`. The variable `tests` is also displayed pictorially. Although the BALSA system creates a view that is easily comprehensible, the view occupies an enormous amount of the screen space. For small programs that usually exist in an educational setting, however, this situation is entirely acceptable. CNS is designed primarily for a production environment and as such it would be extremely difficult to handle a large number of variables on a typical screen in CNS.

Marvel

Marvel is a programming environment that supports error checking and responds to questions about programs (Kaiser, Feiler, and Popovich 1988). Marvel also supports project
management by recording and maintaining the status of modules and the necessary communications for developing each module. The major emphasis in Marvel is on the coordination of knowledge between individual programmers and between programmers and managers.

The Marvel environment is composed of an object-oriented database and a process model. The object base consists of objects that represent both the system and its development history. Objects can be modules, procedures, types, designs, manuals, and development steps. The process model consists of rules that link tools and objects in the object base. The Marvel object base is a persistent, fileless environment. The user sees the object base only in terms of logical entities.
The user can browse the object base by accessing its logical structure. The logical structure consists of the following hierarchy: libraries, modules, procedures, and the various programming constructs (such as types, variables, and statements). The object base may also be searched by using queries. Example queries are "retrieve all software objects with name xyz" or "retrieve all modules that contain errors." Marvel contains several predefined queries.

- What components use a particular function?
- Are certain components not used at all?
- Does anyone intend to use or modify a component?

Rules control the active participation of Marvel in the process of developing a program. Rules in Marvel consist of three parts. The first part, the precondition, states what condition must be true for the rule to take effect. The second part of the rule specifies which tool should be activated and what arguments should be passed to that tool. The last part, the post-condition, updates the database depending on the result of the execution of the rule. Rules are executed automatically by the database system in an opportunistic manner. The rules are executed in a manner similar to an expert system using both forward and backward chaining. The current system, however, is unable to undo activities whenever a chain of events fails, since an activity may have permanently modified the database.
Conclusion

The software development process can be divided into different phases or tasks. Programming environments support the various phases or tasks of the software development process. Each environment provides different levels of support for a particular phase or task.

Being able to support the flow of information within a programming environment, however, is more important than knowing what the environment supports. The flow of information in a programming environment, more than any other feature, determines how the environment can or will be used. The various repositories of information include the programmer, the manager, and the environment.

For programming-in-the-small tasks, the programmer is the primary person receiving and giving information within a programming environment. Such systems are most often based on a syntax-directed editor. The primary purpose of the environment is to provide the user with numerous views of the program being developed. The possible views include the program source, the program documentation, the symbols used in the program, and the module interconnections.

The environment itself may also use the gathered information. This is especially true in environments that support primarily the coding phase of software development and use information to optimize the code produced by the
compiler. For configuration management tasks, the environment needs to maintain records of software releases (dates and reasons for releases) and system modifications (changes to modules in a project). Configuration management systems also ensure the integrity of a project by keeping source and object files consistent (such as in the Unix make facility).

The programming-in-the-many tasks of a software project require that managers receive a substantial amount of information as to the development progress of the project. This information may include such things as programmer productivity reports concerning how much code was produced by each program, project reports on cost or schedule status, status of changes to fix bugs or to incorporate design changes, and the status of testing and validation.

The C Navigational System is primarily designed to provide the programmer with information about a program. CNS provides this information by means of windows. Windows provide different views of a program including the source code, the documentation, the program history list, and the function interfaces. CNS helps the user, wherever possible, in the task of creating and maintaining windows, even to the point of automatically creating the function interface and history list. The multiple views of a program are enhanced by the capability to link them with pointers. Linking
windows with pointers allows the programmer to easily navigate through programs and documentation.

Ideally the display of windows should not be hindered by the two-dimensional display. CNS allows a programmer two methods to partially overcome these limitations. Windows may be "stacked" as shown in figure 5 which allows a programmer to quickly browse through a group of functions much like he would flip through a stack of index cards. Windows may also be reduced to an iconic representation and displayed in a library. The icon for a function can represent the type of processing performed by the function. Thus, a programmer can quickly scan a library by looking for specific icons.

I hope that the combined features of CNS, connecting the source and documentation together with links, system provided history lists and function interfaces, flexible display of windows, iconic representations of functions, and source level debugging, will give programmers a better picture of how their programs can be developed and maintained.
REFERENCE LIST


