INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Supporting Multiple Program Comprehension Strategies During Software Maintenance

by

Susan Elliott Sim

A thesis submitted in conformity with the requirements for the degree of Master of Science

Graduate Department of Computer Science

University of Toronto

© Copyright by Susan Elliott Sim 1998

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-40750-0

Canadä

Abstract

Software maintainers are task-oriented knowledge seekers. They focus on getting the answers they need to complete a task and they use a variety of sources and strategies to do it. This thesis describes the development of a search tool `grug` intended to support program comprehension. This design was based on two user studies and previous work on program comprehension models and tools developed by other researchers. The first study looked at the habits of software maintainers with access to a software visualization tool, the Portable Bookshelf (PBS). The strategies used by subjects to complete maintenance tasks indicated PBS could be improved by adding a search tool, so that information relevant to the immediate task could be more easily located. A second study was undertaken to further characterize programmers' source code searching behaviour to determine what functionality to include in the search tool. Based on these studies and a review of other source code searching and analysis tools, `grug` was designed. This tool supports bottom-up code comprehension strategies by allowing users to search for semantic elements in source code, which they can use to build higher-level concepts. When integrated with PBS, `grug` provides a means of relating program code to the pictorial elements in the software visualization, thereby supporting top-down code comprehension strategies. The suite of tools taken together support multiple comprehension strategies and transitions between them.

Acknowledgements

My first debt of gratitude goes to those who participated in the user studies. Without these people, this research would not have been possible. Standard ethics procedures do not allow me to identify these people for their own protection, so I must thank this faceless cast anonymously.

I would like to thank my supervisors, Ric, for getting me started, and Charlie, for getting me to stop. Thanks also to my "second" reader, Marsha, for her helpful comments.

Finally, many thanks go to my husband, Jeff, for his unfailing support and encouragement.

Table of Contents

Chapter 1: Introduction 1
1.1 Motivation 1
1.2 Starting Points 2
1.3 Organization 4
Chapter 2: The Portable Bookshelf 5
2.1 Overview 5
2.2 The Software Bookshelf Concept 5
2.3 Software Landscape 7
2.4 Creating a PBS 8
2.4.1 Generating the Factbase 9
2.4.2 Recovering the Software Architecture 10
2.5 Populated Bookshelves 11
2.6 Summary 12
Chapter 3: Empirical Studies of Software Maintainers 13
3.1 Overview 13
3.2 Methods for Studying Software Maintainers 13
3.3 Code Comprehension Models 15
3.4 Maintenance Tasks 17
3.5 Work Practices 18
3.6 Summary 20
Chapter 4: Studies of PBS Users 21
4.1 Overview 21
4.2 The Software Project 21
4.3 Software Immigrants 22
4.3.1 Method 23
4.3.1.1 Data Collection 24
4.3.1.2 Data Analysis 25

4.3.2 Results....27
4.3.2.1 Mentoring....28
4.3.2.2 Difficulties Outside of the Software System....30
4.3.2.3 First Assignment....32
4.3.2.4 Predictors of Job Fit....33
4.3.3 Application of the Results to PBS....35
4.4 Project Veterans....36
4.4.1 Questions about Edges....37
4.4.2 Maintainers' Comments on Anomalies....38
4.4.3 Journalism-Style Questions....38
4.4.4 Code Migration....39
4.5 Summary....40
Chapter 5: Source Code Searching Survey....42
5.1 Overview....42
5.2 Method....43
5.2.1 Formulate the Research Questions....44
5.2.2 Create a Data Gathering Instrument....44
5.2.3 Define the Population and Sampling Method....48
5.2.4 Administering the Survey....49
5.2.5 Analyze the Data....50
5.2.6 Methodological Considerations....50
5.2.6.1 External Validity....50
5.2.6.2 Reliability....51
5.3 Results....51
5.4 Participants....52
5.5 Search Tools....53
5.6 Situations....54
5.6.1 Coding and Analysis of Anecdotes....55
5.6.2 Search Targets....55
5.6.3 Motivations for Searching....56

5.7 Searching Archetypes 58
5.7.1 Common Searches 59
5.7.2 Uncommon Searches 61
5.8 Respondents' Suggestions for Features 62
5.9 Implications for Tool Design 66
5.10 Application of the Results 66
5.11 Summary 67
Chapter 6: Supporting Queries on Source Code 69
6.1 Overview 69
6.2 `grep` 70
6.2.1 Strengths of `grep` 70
6.2.2 Limitations of `grep` 72
6.2.3 Analysis of `grep`'s Attributes 74
6.3 `cgrep` 74
6.4 `sgrep` 74
6.5 `agrep` 75
6.6 `tksee` 76
6.7 SCRUPLE 76
6.8 LSME 77
6.9 Comparison of Tools 78
6.10 Lessons Learned 79
6.11 Summary 81
Chapter 7: Design of `grug` 82
7.1 Overview 82
7.2 Platform Requirements 82
7.3 Functional Requirements 83
7.3.1 Basic `grug` Functionality 83
7.3.2 Requirements for the command-line version 85
7.3.3 Requirements for the Graphical User Interface Version 85
7.3.4 Requirements for Operating Across the World Wide Web 87

7.4 Non-Functional Requirements 88
7.5 Specification of `grug` 89
7.6 The GCL Query Language 89
7.7 Markup Schema and Macros for `grug` 96
7.8 Preliminary Implementation of `grug` and the Searchable PBS 101
7.8.1 Expanding the Factbase 102
7.8.2 Augmenting GCL 103
7.8.3 Using `grug` 103
7.8.4 The Searchable Bookshelf 105
7.9 Summary 107
Chapter 8: Conclusion 109
8.1 Observations 109
8.2 Future Work 110
8.2.1 User Testing 110
8.2.2 Organizational Studies 110
8.2.3 Source Code Searching 111
8.2.4 Tool Implementation 111
References 113
Appendix A: Dictionary of Terms 124

List of Tables

Table 4.1: Summary of Respondent Characteristics 24
Table 4.2: Summary of Question Set Usage 24
Table 5.1: Origin of Participants By Newsgroup 52
Table 5.2: Origin of Participants by Email Domain 53
Table 5.3: Tools Used 53
Table 5.4: "Other" tools used 54
Table 5.5: Summary of Common Searches 56
Table 5.6: Summary of Motivations for Searching 58
Table 6.1: Comparison of Tool Characteristics 79
Table 7.1: Markup and Macros for Function Declarations 98
Table 7.2: Markup and Macros for Function Definitions 99
Table 7.3: Markup and Macros of Function Calls 99
Table 7.4: Markup and Macros for Variable Declarations 100
Table 7.5: Markup and Macros for Variable Definitions 100
Table 7.6: Markup and Macros for Variable References 100
Table 7.7: Markup and Macros for Structural References 101
Table 7.8: Available Options in `grug` 105

List of Figures

Figure 2.1: Portable Bookshelf of the GNU C Compiler....6
Figure 2.2: Objc Subsystem of GCC....8
Figure 2.3: Syntax of Tuple Attribute Language in Backus-Naur Format....9
Figure 2.4: Example of TA Language Applied to Software....10
Figure 2.5: Factbase Schema for Software Landscape....11
Figure 4.1: Question Set One....26
Figure 4.2: Question Set Two....26
Figure 4.3: Question Set Three....26
Figure 4.4: Variables Used During Analysis....27
Figure 5.1: Introductory Page of Questionnaire....45
Figure 5.2: Text of Questionnaire....47
Figure 5.3: Message to Solicit Participants....49
Figure 5.4: Example of Scenario Anecdote....54
Figure 5.5: Coding Categories....55
Figure 5.6: Usefulness of Searching Source Code by Task....57
Figure 5.7: Example of Wish List for Features....63
Figure 6.1: Call Graph Extractor for C in SCRUPLE from [Griswo96]....76
Figure 6.2: Call Graph Extractor for C in LSME [Griswo96]....78
Figure 7.1: Example of `grug` GUI Search Dialog Box....86
Figure 7.2: Syntax of the GCL Query Language in Backus-Naur Form....90
Figure 7.3: Code Sample from gcc.c of the GNU C Compiler R2.7.2....91
Figure 7.4: Line 21 of Code Sample with Hypothetical File Positions Labeled....92
Figure 7.5: Markup Index for Line 21 of Code Sample....93
Figure 7.6: Example of Function Declaration....98
Figure 7.7: Example of Function Definition....99
Figure 7.8: Example of Function Call....99
Figure 7.9: Example of Variable Declaration....100
Figure 7.10: Example of Variable Definition....100

Figure 7.11: Example of Variable Reference....100
Figure 7.12: Factbase Schema for GCL Index in TA Format....102
Figure 7.13: The Extended GCL Syntax....104
Figure 7.14: The Searchable Bookshelf....106

Chapter 1: Introduction

1.1 Motivation

Research in program comprehension is predicated on the notion that if it were easier for maintainers of large complex software systems to understand the program source, it would be easier to make modifications to that source. A robust model of the cognitive processes and work habits of software maintainers could be used in the design of tools to support program comprehension. An effective program comprehension tool would have many benefits for maintainers, particularly for those working on large systems. Fewer errors in the comprehension process would result in faster modifications and program code with fewer errors. The combination of reduced effort and increased quality would result in lower software maintenance costs. Considering the proportion of software development costs that is typically allocated towards maintenance, the savings could be quite significant.

Source code has a structure that renders it difficult to read in a linear fashion, and this problem is compounded when the system is large because it is infeasible to read the entire corpus of code. Consequently, the maintainer must read selectively, which makes the task more difficult and susceptible to error. Currently, most programmers rely on general-purpose search tools such as text editor features and operating system utilities. There has been little work on tools specifically for searching source code [Singer97a, Paul95] and even less on searching as a component of maintenance tasks, so there is a great deal of work to be done in this area.

There are both technical hurdles and human factors in developing a program comprehension tool. The technical challenges revolve around the creation of a factbase about the software system and the means to access the factbase. Sometimes the source code itself is used as the

factbase because it is often the only complete and reliable documentation for the system. More often, the data is extracted from the source code using some combination of search utilities, parsers, and code analyzers. Whatever mechanism is used, we must ensure that it is capable of populating the factbase with the necessary information. We also need to provide software maintainers with a mechanism to access the factbase. In some cases the factbase may be viewed directly by the user; in other circumstances a query language is necessary to search it. Both of these issues are examined in this thesis.

The human issues around creating a source code searching tool are perhaps more difficult to resolve. In order to be successful, a program comprehension tool must help software maintainers to complete their work by providing relevant information in a timely fashion. We need to anticipate the information requirements of software maintainers, so these needs can be reflected in the design of the factbase schema. We must also ensure that the technical solution developed is usable by the target user group. A subtle, but important, point is that a tool is not successful if it is not adopted by the intended user group. The likelihood of adoption can be increased by making the tool fit with how software maintainers work and cooperate with their existing tools. A user study that focuses on usage patterns can assist in this process.

1.2 Starting Points

This thesis evaluates an existing program comprehension tool, the Portable Bookshelf (PBS), through user testing and proposes the addition of a search tool as an improvement. (A dictionary of terms, tools, and acronyms can be found in the Appendix.) The main contributions of this thesis are the results from the user studies and the design for a search tool `grug` (**grep u**sing **G**CL). The `grug` tool is essentially a "semantic `grep`", that supports searches for meaningful elements in source code, i.e. those elements that are "understood" by the compiler. Although these elements are sometimes called syntactic, they will be referred to as semantic for the remainder of the thesis. Searches for these elements are specified using the GCL query language, developed by other researchers at the University of Waterloo [Clarke95a, Clarke95b].

Existing work on program comprehension tools and models served as the basis for the work in this thesis. From a tools standpoint, the starting point was PBS, a reverse engineering tool targeted for the re-documentation phase of a re-engineering or migration effort. The PBS tool suite is an instantiation of the Software Bookshelf paradigm. This tool along with the utilities to construct it have been developed over the years by researchers at the University of Toronto, University of Victoria, and the Centre for Advanced Studies at IBM Canada Ltd. [Penny92, Farman97, Finnig97, Holt97]. The core of PBS is Software Landscape, an abstract visual representation of a software system. This work is discussed in greater detail in Chapter 2.

This thesis also draws on studies to define models of how programmers understand source code. Traditionally, there are two types of code comprehension models: bottom-up and top-down. Bottom-up models state that as source code is read, abstract concepts are formed by amalgamating low-level information into meaningful units [Shneid79, Pennin87]. Top-down models state that a programmer uses domain knowledge to build a set of expectations that are mapped onto elements in the source code [Brooks83, Littma86, Solowa84]. A more recent development was the integrated code comprehension model which states that programmers switch back and forth between the two strategies to complete a task [vonMay95, Letovs86]. This last model is consistent with the findings of the user studies performed as part of this thesis. When presented with low-level information, programmers wanted to relate it to high-level concepts. When presented with high-level abstractions, they wanted to relate it to low-level artifacts. A more detailed review of program comprehension research as it relates to code comprehension models and software maintenance tasks is presented in Chapter 3.

The `grug` tool and the Searchable Bookshelf were designed with these strategies in mind. Software maintainers can use `grug` to search for semantic elements in source code, thereby building higher-level, more abstract concepts about the software. The grug tool can be used to search for semantic elements such as function and variable definition and declarations. From our second user study, these were the most common targets of searches on source code

when performing maintenance tasks. By assigning meaning to a portion of the text, a collection of keywords, operators, and identifiers, becomes associated with a concept or a step in an algorithm. They can also use `grug` within the Searchable Bookshelf to relate elements in Software Landscapes to source code. The box-and-line drawings of the software architecture give a conceptual view of the software system. In order to use this information to complete a task, a software maintainer needs to relate the various pictorial elements to source code. These relationships can be established using `grug`. Together, `grug` and Software Landscape support integrated code comprehension models.

1.3 Organization

As mentioned in the previous section, background information on PBS and studies of software maintainers are given in Chapters 2 and 3, respectively. The first of two user studies is discussed in Chapter 4. This study looked at how software maintainers used a deployed PBS in their daily work. Newcomers, or "software immigrants", were examined in detail, and project veterans were also interviewed. This study indicated that a search tool should be added to PBS. The second study, which is described in Chapter 5, characterizes the habits of programmers as they relate to source code searching.

Based on the findings of the two user studies, we examined existing searching tools for their strengths and limitations. This review helped guide decisions made in the design of `grug` and the Searchable Bookshelf. Among the tools examined in Chapter 6 are search utilities from the `grep` family, tools for searching source code and source code analyzers.

In Chapter 7, the design of `grug` and the Searchable Bookshelf is presented. Included in this chapter are the requirements and specification of the tool, along with the description of a preliminary implementation. Finally, the thesis concludes with a summary and a discussion of future work in Chapter 8.

Chapter 2: The Portable Bookshelf

2.1 Overview

The PBS (Portable Bookshelf) was the starting point for the investigations in this thesis and this chapter presents the concepts underlying PBS and one of its essential components, Software Landscape. The implementation and construction of PBS are described, along with a discussion of its shortcomings.

2.2 The Software Bookshelf Concept

Software Bookshelf is a reverse engineering tool that focuses on the re-documentation phase of a re-engineering or migration effort [Finnig97]. As its name implies, the tool is based on a bookshelf metaphor where each book on the shelf corresponds to particular view or aspect of the system. Books are not limited to being written material, such as design documents and source code, but also include tools, indices, and annotations. Some tools that have been included on Software Bookshelf are Software Landscapes (discussed section 2.3) [Penny92], Rigi [Müller93], and Refine [Markos94].

PBS is a web-based implementation of the Software Bookshelf paradigm. It uses a web server to deliver information from a shared repository through HTML (hypertext markup language) pages, MIME (Mutipurpose Internet Mail Extension) types, Java applets, and CGI (common gateway interface) scripts [Tzerpo97]. By using a Java-enabled web browser as a front end, PBS presents users with a recognized interface that affords a familiar interaction style.

Figure 2.1 shows PBS accessed from the Netscape Navigator web browser. The narrow column along the left side contains the table of contents which lists the books for a subject

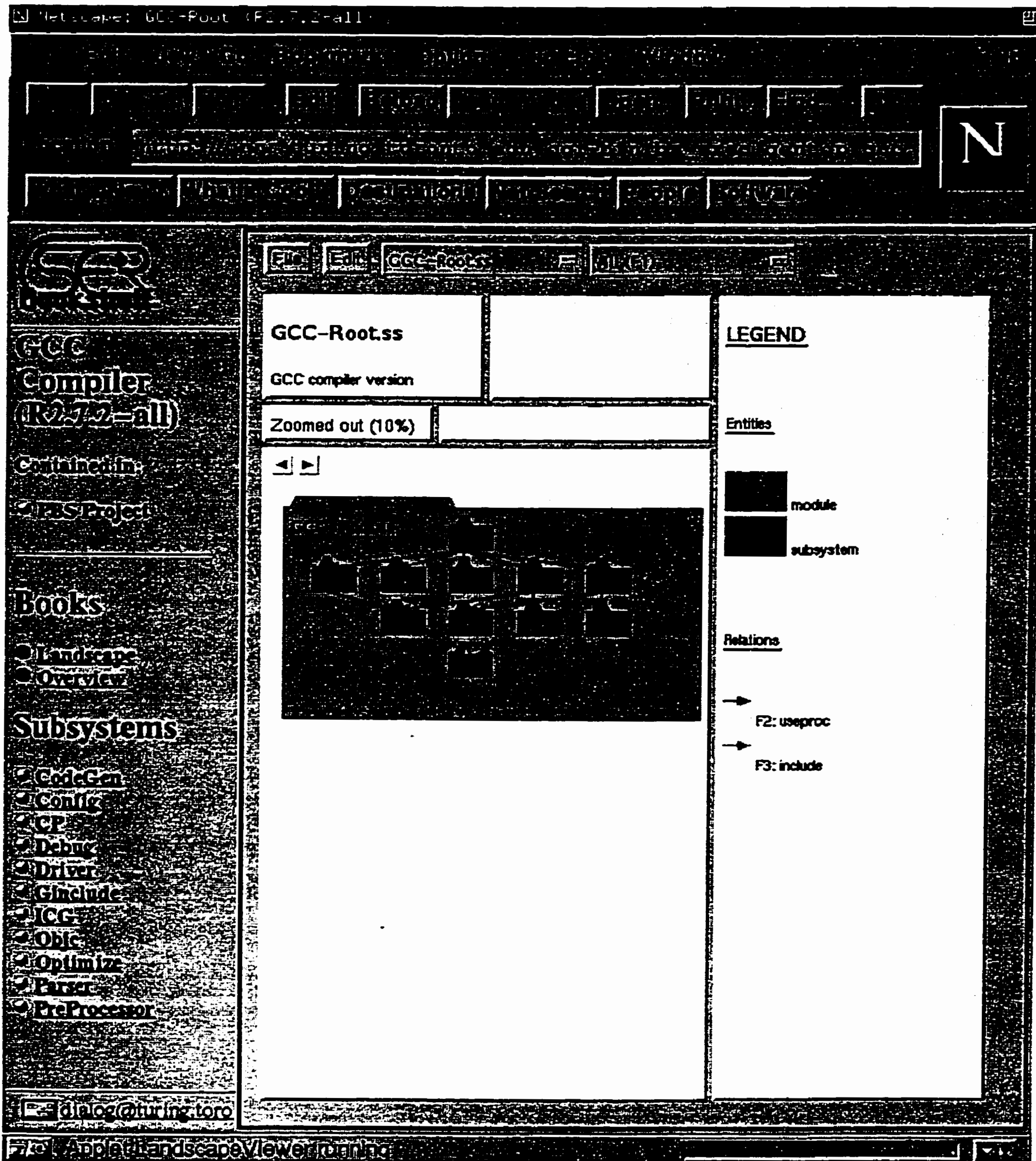

Figure 2.1: Portable Bookshelf of the GNU C Compiler

system. The Software Landscape diagram is found in the large frame on the right. A Landscape is a pictorial representation of the architecture of the software system being documented. It is generated by a series of static analysis tools that use the source file as the basic unit of analysis.

One shortcoming of PBS and Software Landscapes arises from the fact that that they were designed to be browsed. The basic mode of navigation through PBS is point and click, just as it is with other World Wide Web constructs. There is no mechanism for searching, and by extension, hypothesis testing. Another problem is that PBS does not provide easy linkage between top-down and bottom-up comprehension approaches. The Landscapes provide high-level information, while source code provides low-level. Documentation provides information at varying levels. The interface does not have the means to support smooth transitions between the levels. These shortcomings become more conspicuous in the user studies that are discussed in the Chapters 4 and 5.

2.3 Software Landscape

A Software Landscape of the Objective C subsystem of the GNU C Compiler (GCC) is in Figure 2.2. This diagram was reached by clicking on the "Objc" box in Figure 2.1. In that landscape, the Objc subsystem is the right-most box on the third row from the top. Subsystems are drawn as gray boxes with a tab on the top left to give the appearance of a folder. Modules are drawn as blue boxes with the top right corner folded down to give the appearance of a piece of paper or a document. Green edges represent variable references and red edges represent function calls between these units.

The graph is drawn using a nested box formalism [Harel88]. For example, at the top level is a landscape diagram of a folder representing the entire system. In it are other folders depicting a system decomposition. In order to see the contents of a subsystem, the user can click on its box (select using a mouse button) which brings up a lower level Landscape. Further branches of the decomposition can be browsed by pointing and clicking. Sometimes the entire set of diagrams for an application is referred to as a Landscape.

Figure 2.2 shows a subsystem further down the decomposition. The row of boxes at the top of the diagram are "clients" and the one at the bottom is a "server". Clients are other subsystems that *use* a variable or a function contained in the subsystem being viewed. By extension, servers are subsystems that contain a variable or function that is *used by* the subsystem being viewed. By using this formalism, the target subsystem can be viewed in context [Penny92].

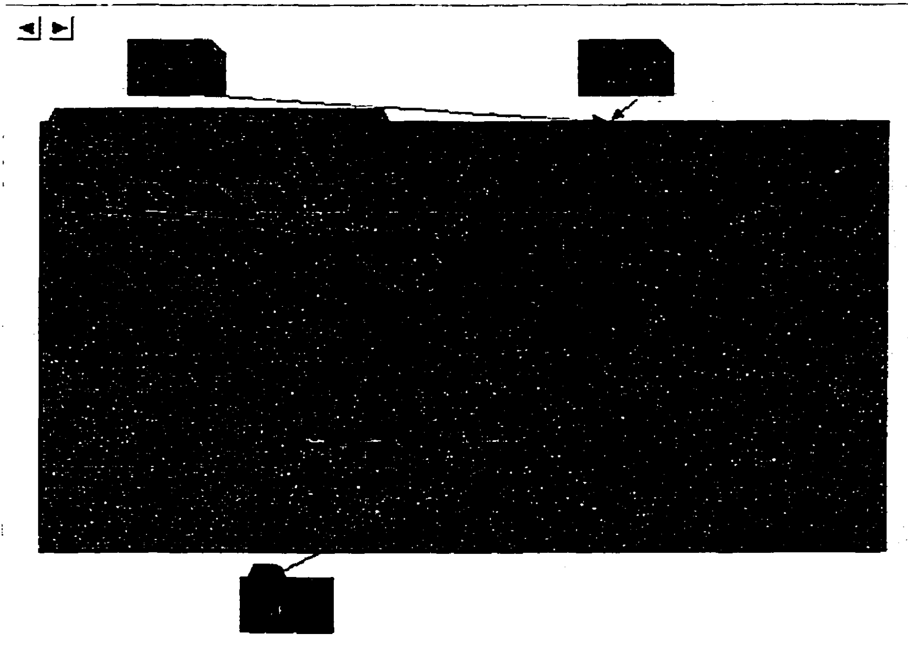

Figure 2.2: Objc Subsystem of GCC

In the next section, the steps and the tools involved in creating a PBS for a software system are described.

2.4 Creating a PBS

The first step in creating a PBS is to extract the necessary facts from the subject software system. A parser creates a factbase consisting of function calls and variable references that cross file boundaries. Function calls and variable references that are local to a file are not included in the factbase because these facts are not necessary for recovering the architecture of a software system. The factbase and the information passed between tools are stored in ASCII files using the Tuple Attribute Language (TA) [Holt97]. The decomposition of the

software is recovered by clustering files into subsystems using manual techniques and a tool called `grok`. The resulting information is drawn and manipulated using Java applets on the PBS [Farman97]. In the remainder of this section, each of these steps will be discussed in greater detail.

2.4.1 Generating the Factbase

Parsers are the only language-dependent tool required to populate a PBS. The remainder of the tools operate on factbases written in an intermediary language. A factbase for PL/IX (a variant of the PL/I programming language) source code is constructed using `plix2rsf`, while one for C source uses the combination of `cfx` and `fbgen`. The factbase is stored in TA, a language for representing coloured graphs.

```
tupleLanguage ::= { stringToken stringToken stringToken }

attributeLanguage ::= itemId "{" {attributeSetting} "}"

attributeSetting ::=
          attributeId
        | attributeId "=" attributeValue
        | attributeId "{" {attributeSetting} "}"


attributeValue ::=
          stringToken
        | "(" {attributeValue} ")"

itemId ::= stringToken
        | "(" stringToken ")"   // Relation (edge class)
        | "(" stringToken stringToken stringToken ")"

TALanguage ::= {section}

section ::=
          SCHEME TUPLE : tupleLanguage
        | SCHEME ATTRIBUTE : attributeLanguage
        | FACT TUPLE : tupleLanguage
        | FACT ATTRIBUTE : attributeLanguage
```

Figure 2.3: Syntax of Tuple Attribute Language in Backus-Naur Format

A software system can be thought of as a coloured graph. Entities such as variables, functions, files, modules, and subsystems can be represented as nodes. Relations between them such as “use”, “call”, and “contain”, can be represented as edges. Consequently, TA can be used to represent a software system. Furthermore, nodes and edges can have

```
$INSTANCE summary.c module
$INSTANCE utils.c module
$INSTANCE end_of_month function
$INSTANCE printRecord function
end_of_month {
      defloc = summary.c:56
      deflocend = summary.c:87
}
printRecord {
      defloc = utils.c:365
      deflocend = utils.c:425
}
funcdef summary.c end_of_month
funcdef utils.c printRecords
call end_of_month printRecords
```

Figure 2.4: Example of TA Language Applied to Software

attributes, such as a visual representation (colour, size of box, location of box, etc.) or location in source code.

Figure 2.4 gives an example of TA applied to software. The first four tuples represent nodes and the last three represent edges. The two stanzas in the middle assign attributes to the `end_of_month` and `printRecords` entities. In the example, the file `summary.c` defines the function `end_of_month` on lines 56-87. The function `printRecord` is defined in the file `utils.c` on lines 365-425. Finally, the `end_of_month` function calls the `printRecord` function.

2.4.2 Recovering the Software Architecture

The low-level factbase is processed to produce a high level factbase with the tuple schema given in Figure 2.5. Processing is done using the `grok` tool, essentially a binary relational "calculator" that can be used to induce relationships about factbases. It can perform operations analogous to those in SQL, such as selects, joins, and intersections, as well as transitive closure. Common sequences of operations can be written into scripts, as is the case for the operations to create a high-level factbase from the ones produced by `plix2rsf` and `cfx/fbgen`.

```
$INSTANCE [name] module
$INSTANCE [name] subsystem
useproc module module
useproc module subsystem
useproc subsystem module
useproc subsystem subsystem
usevar module module
usevar module subsystem
usevar subsystem module
usevar subsystem subsystem
implementby module module
contain subsystem module
contain subsystem subsystem
```

Figure 2.5: Factbase Schema for Software Landscape

The final step in creating the architectural level factbase is to cluster the files into subsystems. A domain expert is consulted to identify a system decomposition and to allocate files to subsystems. This information is used by `grok` to make further inferences about interactions between subsystems and modules. Attributes are added as necessary by tools that use the factbase. For example, `lslayout`, a tool for drawing software landscapes, adds attributes such as colour, size, and location. An expert is often consulted during this phase of the recovery as well. Although this approach is time consuming, the results are often better than those generated using automated approaches alone. Users tend to find the visualizations more aesthetically pleasing and consistent with their own mental models of the system [Tzerpo96].

2.5 Populated Bookshelves

A number of PBSs have been populated for "real world" software systems including GCC, Linux, and TOBEY. The first two are relatively large systems (200-300 KLOC) with publicly available source code, which makes them appropriate for testing PBS construction tools. With a world-wide user base, both pieces of software are modified frequently, often by people who do not know the source code intimately. Consequently, Software Bookshelves of these systems have a purpose beyond an academic exercise.

The TOBEY (Toronto Optimizing Back End with Yorktown) system is a compiler component maintained by a team at IBM Canada Ltd. A PBS for this software has been

populated in anticipation of a migration to C++ from PL/IX. When the PBS was constructed, there were about ten people on the development team, and the system consisted of approximately 250 000 lines of code. Source code, some documentation, and Software Landscapes had been put onto the PBS. Software immigrants and veterans of this group were studied to evaluate the effectiveness of PBS as a program comprehension and reverse engineering tool, and this work is reported in Chapter 4.

2.6 Summary

In this chapter, PBS concepts and architecture were explained. The PBS is a web-based reverse engineering tool based on the metaphor of a bookshelf. On the shelf are books corresponding to views of the system as presented by written material and tools. A web server is the delivery mechanism for this repository. The main view in PBS is a visual representation of a software system decomposition, known as Software Landscapes. In these diagrams, document and folder icons represent source files or collections of source files, and relationships between them are represented by arrows. Different levels of the system decomposition depicted by Software Landscape can be browsed by pointing and clicking.

Also discussed in this chapter is a brief overview of how a PBS is constructed for a subject system. The different tools in PBS use a language-independent factbase written in TA (Tuple Attribute) Language. There exists a suite of lightweight tools to generate and manipulate the factbases. The chapter concludes with the presentation of a list of software systems that have been re-documented using PBS.

Chapter 3: Empirical Studies of Software Maintainers

3.1 Overview

In this chapter, developments in empirical studies of software maintainers relevant to the work in this thesis are summarized. These developments include techniques used to study software maintainers, as well as the areas that have been examined such as models of code comprehension, strategies for performing maintenance tasks, and work practices. It is important to look at these aspects of software maintainers and their work to ensure the tools that researchers develop for them are relevant. As the various studies are reviewed in this chapter, the results will be related to the design of software tools for maintainers.

3.2 Methods for Studying Software Maintainers

The techniques used for studying software maintainers can be divided into two groups: those that originate in psychology and those that originate in sociology. Although these methods may have been adapted by empirical studies in software maintenance from intermediary disciplines, such as human-computer interaction, education, and business management, their roots can be identified by their philosophical underpinnings.

Those that are psychological in origin are centered around studying *individuals in controlled experiments*. Psychology experiments tend to have a very narrow focus and consequently are used primarily in studies of programming-in-the-small or "maintenance-in-the-small". These studies may require performing tasks or filling out questionnaires, and they may occur in a laboratory or an office.

Those that are sociological in origin are centered around making observations of *people working in systems*. Sociological studies tend to have a broader focus, looking at interactions between people, particularly in groups, and consequently are used in studies of

communication and work practices. These studies may involve surveys, field observations, or examination of archives.

Techniques from both psychology and sociology can be used at various stages of research, that is, they can be used both for exploratory work (theory building) and hypothesis testing (theory validation). It should be noted that the research question, rather than the method, that should dictate the approach used to analyse the data produced by a study. The two most commonly used approaches in empirical studies of software maintainers are case studies [Yin94] and exploratory studies. Normally, psychology experiments form conclusions using tests of statistical significance and sociology studies characterize populations using summary statistics and error estimates. The purpose of generating these measures is to generalize the results to a population. At this time in software engineering there are few studies that can do so appropriately because there is insufficient demographic information about the software maintainers and the systems they work on. There is almost no literature on number and distribution of software maintainers working in industry and their characteristics such as education, experience, skill level, and productivity. Similar information is also lacking about the systems they work on, such as size, age, number of releases, and language of implementation [Zvegin97]. As a result, the majority of studies performed in software engineering have theories and models as their end products.

Psychological methods have primarily been applied to code comprehension and task performance. The two most common techniques are protocol analysis and controlled experiments. The researchers best known for using protocol analysis are Von Mayrhauser and Vans [VonMay93, VonMay97]. In protocol analysis, a programmer is asked to articulate her thoughts while performing a software maintenance task. The session is recorded on audio or video, and is later analyzed along prescribed dimensions [Solowa88]. Controlled experiments have been used by many researchers to develop code comprehension models [Shneid80, Littma86], and to examine tool use [Storey96]. In these studies, a set of "conditions" are established in which each has a different "level" of independent variables. For example, in an experiment to determine whether the amount and type of light affected the

performance of programmers, there are two conditions. The experiment could have two levels for the type of light (incandescent and fluorescent) and three levels for the amount (0 watts, 40 watts, and 100 watts), for a total of six "treatments". Subjects are randomly assigned to a treatment and their performances measured. These results are compared to determine whether a particular independent variable affected any dependent variables.

Sociological methods have been applied to the study of work practices and, to a lesser degree, task performance. Perry et al. [Perry94] and Singer et al. [Singer97] have used techniques such as direct observation, questionnaires, and personal logging to characterize the work practices of software maintainers at large telecommunications companies. Eisenstadt used an informal survey to study what made some bugs more difficult than others [Eisens97]. Seaman and Basili used observations and interviews to examine the effect of different types of communication on code inspections [Seaman97].

In the remainder of this chapter, the results of the research performed using these methods are presented.

3.3 Code Comprehension Models

Work has gone into developing a reliable and valid model of source code comprehension because it is a fundamental part of so many software maintenance tasks. Before software can be modified, the maintainer needs to understand the existing system. A robust model of code comprehension is crucial to developing tools and processes that will assist maintainers with their tasks. Early research in this area tended to use undergraduates as subjects in experiments where the task was to modify small programs of 1000 lines of code or less. Some experiments were bold enough to use "large" programs of 3000 lines of code. Recent research during this decade have used industrial software maintainers as subjects and their task was often chosen by the experimenter from the subject's list of pending tasks. Although this method sacrifices experimental control, the studies more accurately reflect how software maintainers work.

Code comprehension models can be grouped into the following taxonomy: bottom-up, top-down, and integrated. Bottom-up models are arguably the oldest of the code comprehension models. They state that as source code is read abstract concepts are formed by *chunking* together low-level information [Shneid79, Pennin87]. These models were based on observations of a small number of subjects reading source code. One of the strengths of this model is it fits with some existing psychological models of how short-term and long-term memory operate.

Based on observations of themselves and other programmers, some researchers found this model unsatisfactory, particularly in situations where subjects didn't or couldn't read all of the source code, as is the case with large legacy systems. Another drawback of the bottom-up model is that it doesn't account for factors such as programmer expertise, domain knowledge, and the complexity and design of the subject system. As a result, top-down models were proposed. These models state that a programmer uses domain knowledge to build a set of expectations that are mapped onto the source code [Brooks83, Littma86, Solowa84]. The programmer looks for *beacons* or cues to indicate the functionality of a piece of source code without piecing together the algorithm one line at a time.

Others felt that top-down models also failed to adequately explain comprehension strategies used by software maintainers. These researchers proposed integrated comprehension models which state that a programmer switches between strategies as dictated by the available information [vonMay95, Letovs86]. The von Mayrhauser and Vans "Integrated Metamodel" (IM) will be discussed in detail, as it encompasses many of the ideas in top-down and bottom-up models, and is the dominant model in software maintenance research.

The IM has four components: the top-down model, the situation model, the program model, and the knowledge base. This last component is a set of rules used to construct and link the three sub-models during comprehension. The top-down model matches ideas with beacons in essentially the same manner as Brooks' model [Brooks83]. The situation model relies on domain knowledge to match operations in the code with real-world objects. The program

model captures the programmer's knowledge of how the program functions and is constructed by understanding the structure of the source, such as control-flow. Information in any of the models can be abstracted as necessary by chunking. According to the IM, programmers construct all three sub-models simultaneously and switch freely between them as information appropriate to a particular sub-model becomes available. The IM is by no means a definitive model, but it is attractive because it accounts for a large proportion of the strategies used by maintainers of large software systems.

3.4 Maintenance Tasks

Aside from code comprehension, debugging is the maintenance task that has been most studied. One possible explanation for this concentration is its prevalence as a maintenance task. Another is the lack of good debugging tools; programmers still rely on their brains and print statements as their primary means of finding bugs [Lieber97]. Beyond this, we have debuggers and profilers, much the same tools that we had twenty years ago. Some studies looked at what makes bugs difficult [Vessey89, Eisens97] and others looked at what strategies were used to find the bugs [Spohre85, Katz88]. All of these studies, except for one by Eisenstadt [Eisen97], examined how subjects behaved in controlled experiments using small programs, some as short as 10 lines.

Eisenstadt on the other hand solicited anecdotes from programmers on electronic forums such as USENET newsgroups and bulletin board systems, about particularly nasty bugs that they had encountered. The author found that the most common reasons for defects being difficult to fix were: large temporal or spatial gaps between the root cause and the symptom; and bugs that rendered debugging tools inapplicable through situations such as race conditions. An example of the former reason is a line of code that overwrites a portion of memory, but the program does not crash until much later when the memory is read. These two reasons accounted for over half the anecdotes reported.

Some of Eisenstadt's findings contradict one of Katz and Anderson's experiments, which basically reported that if a subject could find a bug, she could repair it. With sufficiently

complex software, this result not longer holds. However, Singer et al. found using several different measures that searching was the most common activity for software engineers [Singer97c]. To quote one of their subjects, "First we search to find where the problem is, then we search to find potential solutions, then we search to do impact analysis."[Singer97b] This result suggests that software maintainers could benefit a great deal from a good search tool.

Publications on other maintenance tasks, such as adding a feature to a large system and supporting an undocumented system, have tended to be personal experience reports. Lakhotia [Lakhot93] describes his experiences modifying the GNU C Compiler. He argues software maintainers rarely try to understand a source code in its entirety. More often, they only want and need to understand the minimum to get the job done.

There also exist some experience reports on working with undocumented systems. Fay and Holmes [Fay85] suggest specific strategies for dealing with the political situation and for developing a sufficient understanding of the software system to complete the assigned maintenance task. Pigoski and Looney report on their experiences on a team that had been given the responsibility of supporting a system without any prior training and insufficient documentation [Pigosk93].

3.5 Work Practices

Some of empirical studies of software maintainers look at their work habits and how their teams function. The rationale behind this type of study is that before we can help software maintainers, we must first understand how they currently work. This understanding is beneficial for process improvement, so that the negative aspects can be eliminated and the positive enhanced. In the case of tool design, a good software maintenance tool should fit with how the intended users do their jobs. Researchers in this area tend to use methods that are sociological in origin. The focus is on the software maintainer as part of a development team and a corporation, rather than as an individual toiling away on a task alone.

Perry et al. [Perry94] and van Solingen et al. [vanSol97] examined how programmers spend their time. Both studies found that developers spend about half their time working on source code. About 15% of their time is spent dealing with interruptions such as telephone calls, email and visitors. These findings suggest that a possible way to increase the efficiency of maintainers is to improve the management of information flow. For example, the work day could be organized so that certain times are reserved for working alone and others are allocated for communication. Another possibility is to determine the dependencies of a particular task and ensure that requirements are met, so a maintainer is less likely to become blocked while coding. Seaman and Basili [Seaman97] also looked at communication, but focussed on its effect on code inspections. They found that inspection teams that were less familiar with each other and were more physically and organizationally distant from each other spent longer on their inspections and found more defects in the code.

Singer and Lethbridge [Lethbr97, Singer97a] have studied work practices of software engineers for the purpose of uncovering requirements for tool design. They have used techniques such as questionnaires, interviews, and job shadowing. In a series of studies, they identified 14 categories of tasks that maintainers perform. In one study, eight programmers were each shadowed for one day as the performed their daily work. A note was made each time they changed from one task category to another. The three most common activities were searching, changing the source code, and using an editor [Singer97a].

In another study, Lethbridge and Singer looked at the positive and negative aspects of software tools that maintainers currently use. Many of the comments in both areas relate to usability issues. The subjects liked tools that were easy to use, had useful or necessary features, and had responsive performance. They disliked tools that were poorly integrated or incompatible with other tools, tools that were not powerful enough or had features missing, and tools that had too many features or were too big. The maintainers were also asked about what kinds of tools they would like to have. The two most common requests were for better exploration tools and automated testing tools [Lethbr97].

Many of Singer and Lethbridge's results indicate that there exists a niche for a good search tool. They have applied these findings the development of `tksee` (Software Exploration Environment using a `tk` interface), which will be discussed in Chapter 6.

3.6 Summary

When designing a tool for software maintainers, it is important to ensure that the tool fits with how they work. To do otherwise would reduce the already slim chances of the tool being adopted. In this chapter, some methods to uncover software maintainers' work habits and tool requirements are presented. As well, some considerations in tool design are discussed: code comprehension models, tasks performed by maintainers, and the work habits of maintainers. The research methods were adapted for two user studies that are reported in this thesis. Chapter 4 describes the experiences of a development team with access to PBS for their software system. Chapter 5 presents the results of a survey of programmers on their requirements for a source code searching tool. The developments from this chapter and the user studies are reflected in the design of `grug` and the Searchable Bookshelf in Chapter 7.

Chapter 4: Studies of PBS Users

4.1 Overview

Although a number of Software Bookshelves had been constructed, they had not been evaluated as a software comprehension tool. A Software Bookshelf had been constructed for a commercial software product, and usability studies were conducted with the team maintaining the product. The intention was to evaluate how PBS helped them with their daily work, but for myriad reasons it was difficult to find PBS users. Many of the reasons were organizational, but some were due to shortcomings in the tool. The studies of PBS users, their findings, and implications for PBS are documented in this chapter.

4.2 The Software Project

A PBS had been populated for a compiler maintained by a team at IBM Canada Ltd. in anticipation of a migration to C++ from a PL/I dialect. The software was originally created fifteen years ago and has had twelve major releases. It consisted of approximately 250 000 lines of code in 1000 files. The system was supported by a team of ten developers.

Source code, some documentation, and Software Landscapes had been put onto the bookshelf. The system's fundamental abstract data structure and a small number of subsystems were well documented, but aside from the Software Landscapes there was little on the remainder of the system.

The Software Bookshelf was designed with three groups of users in mind: newcomers to a project who needed to learn about the system, project experts who required a reference on the software, and project managers who wanted to track the maintenance effort. Ideally, detailed studies should have been performed on each of these user groups, but this was not possible due to organizational constraints.

The newcomers, or software immigrants, were relatively easy to study, since they did not yet have a lot of responsibilities and consequently had time available to spend with a researcher. As a result, a detailed, formal study was conducted with them using a method that was sociological in origin. They were expected to use PBS a great deal, but they did not for the reasons that are presented in Section 4.3.3. What started out as a study of tool use, concluded in a broader characterization of the acclimation process for new employees. Presented in this chapter is a subset of the results already published elsewhere [Sim98a].

The project veterans tended to be quite busy, and were only available for occasional meetings. The findings on this user group are based on informal meetings and conversations. The project manger, while eager to help, was probably the busiest person on the team, and as a result there are no findings for this user "group". In this case, missing a single individual, unfortunately resulted in the omission of an entire class of users. Nonetheless, valuable lessons were learned from studies of the other two user groups.

4.3 Software Immigrants

New staff members are usually experienced programmers who already have a rich set of skills and background knowledge. Despite their personal assets, they often lack basic knowledge about the specific project. For these reasons, we call these new team members "software immigrants", since their experience is analogous to those of people who arrive in a new land and need to learn its language and culture. Software immigrants are often referred to by other terms such as newcomers, newbies, recruits, new hires, rookies, and even "fresh blood". Novice is an inappropriate term since it implies a lack of experience. Extending this analogy, the process by which software immigrants adapt to a new project is called "naturalization". Others may call it acclimation, re-tooling, start-up, ramp-up or bringing someone up to speed.

Studies have been undertaken in software engineering and cognitive psychology on working with legacy systems. There are some experience reports which give practical advice for working on undocumented software systems[Fay85, Pigosk93] and anecdotes from

practitioners and consultants[Brooks95, DeMarc87]. The most significant contribution comes from Berlin [Berlin93] who studied the interaction patterns between mentors and apprentices at the conversation level and found that mentoring is a highly effective way to transmit information about the system. Mentors provide not only answers to apprentices' questions, but also explanations of design rationale. Their conversations tend to be highly interactive in nature, using techniques such as confirmation and re-statement to verify that a message has been passed correctly. While mentoring has its merits, it tends to be a time-inefficient method to train a software immigrant because it results in a net decrease in team productivity in the short term. As an antidote, Berlin suggests capturing the information that mentors convey in documentation or an intensive course for apprentices.

4.3.1 Method

A multi-case study was performed with four respondents, all immigrating into a single team. By interviewing subjects, we hoped to identify commonalties and differences in their experiences, and to infer naturalization patterns from this comparison.

In this study, our goals were to:

- describe the naturalization process,
- identify shortcomings and successes of the process, and
- characterize the strategies software immigrants used to adapt to the new job.

In order to highlight areas that would profit from modification or improvement, we must first identify strengths and weaknesses in the existing naturalization process.

The unit of analysis in this study is a single naturalization. The rationale for this choice is that each participant could be studied more than once as they naturalized to a different projects and teams. Data was collected using structured interviews, and was analyzed using qualitative data analysis methods. During analysis, variables of interest were identified using a pattern matching technique. A data matrix was populated with these variables to articulate cross-case patterns [Miles94]. In the following three subsections, we describe the data collection procedure, and the data analysis techniques that were used.

4.3.1.1 Data Collection

Interviews were conducted from February 1997 to June 1997 with four respondents. Data collection began with S1 and S2 shortly after they joined the company. As the study proceeded, S3 and S4 were identified as relatively new software immigrants, and were willing to participate in the study. Consequently, using "controlled opportunism" [Eisenh89], they were interviewed using a sub-set of the questions used with the first two respondents. At the time of interview, S4 was on an educational leave of absence. The background of each respondent is summarized in Table 4.1.

Case	Interview Frequency	Experience on Team at Time of Interview	Highest Level of Educational Attained	Previous Work Experience
S1	Every 3 weeks for 4 months	0-4 months	Masters in CS (compilers)	4 co-op work terms
S2	Every 3 weeks for 4 months	0-4 months	Masters in CS (compilers)	3 years as Windows system programmer
S3	Once	7 months	Bachelors in CE	2 years with an optimizing compiler
S4	Once	8 months (on leave)	Doctorate in CS (artificial intelligence)	Summer jobs

Table 4.1: Summary of Respondent Characteristics

Structured interviews were used with all respondents. They were asked standard questions and were allowed to elaborate as appropriate to their situation. All interviews were conducted by a single investigator and were tape-recorded. Prior to being interviewed, respondents signed consent forms. All raw data is kept confidential, and the anonymity on the respondents is maintained.

Cases	Question Set 1 Used During:	Question Set 2 Used During:	Question Set 3 Used During:
S1, S2	First interview	Interview every 3 weeks	Last interview
S3, S4	Only interview	No	Only interview

Table 4.2: Summary of Question Set Usage

Three sets of questions were used: the first set of questions inquired about the respondent's background, both educational and industrial; questions from the second set probed the respondent's growing understanding of the software system and naturalization process in progress; and the last set explored the respondent's naturalization experience in retrospect.

Question set one was used during the first interview with a respondent, and set three during the last. With S3 and S4, these occasions coincided. Question set two was used only with S1 and S2 as we followed them through their naturalization. The usage of these question sets with the respondents is summarized in Table 4.2. These question sets can be found in Figures 4.1-4.3.

At the end of the four months, we concluded our interviews with S1 and S2 because we felt that the immigrants had reached a plateau in their naturalization. This is not to say that they were completely familiar with the software system, but rather they had settled into a stable work routine and would be making a steady transition to being fully productive team members.

4.3.1.2 Data Analysis

Since a single investigator conducted all of the interviews, hypotheses were formulated throughout the study, using a method of constant comparison [Eisenh89]. After data collection concluded, notes and recordings made during the interviews were reviewed entirely. During this stage, seventeen variables of interest in five major areas of inquiry were identified using cross-case comparisons. It is important to note that the variables used were not scalar, but quantitative. A "value" assigned to a variable could be numerical, but textual descriptions and lists are also valid. The variables are listed in Figure 4.4 , and the areas are:

- respondent characteristics,
- orientation and training,
- difficulties outside of learning about the system,
- timing and type of tasks given, and
- approaches used to understand the system.

Question Set One: Subject's Background

1. What is your educational background?
2. What experience have you as a professional software developer? What kinds of projects did you work on? What tools and languages did you use?
3. Are there any educational materials that your found particularly useful such as books, manuals, guides, course, videos ?
4. What do you enjoy most about your work?
5. What do you dislike most about your work?

Figure 4.1: Question Set One

Question Set Two: Observing the Naturalization Process

1. What is your current assignment? What have you been working on over the last week?
2. How did you gather information about the problem?
3. What resources did you use? What documentation did you read? Who did you consult?
4. What new things did you learn over the last week?
5. What new tools did you use over the last week?
6. Did you use Software Bookshelf? Include information about how and why if appropriate.
7. Over the last week, what have you done to become more familiar with the software system?
8. Draw a diagram of your current understanding of the system.

Figure 4.2: Question Set Two

Question Set Three: Recalling the Naturalization Process

1. How long have you been working at this job?
2. What administrative issues did you have difficulties with? (i.e. badges, logins, machines, payroll, etc.)
3. How many different computer systems do you have to use to do your job?
4. How many different tools or applications do you have to use to do your job?
5. What technical issues did you have difficulties with? (i.e. missing background knowledge)
6. What difficulties did you encounter when learning about the system you are working on?
7. How long did it take you to become comfortable with your new environment? (i.e. office, building, cafeteria)
8. How long did it take you to figure out office numbers?
9. How long did it take to become productive?

Figure 4.3: Question Set Three

Data from the interviews were used to assign values to these variables and this information was put into a data matrix. A pattern matching technique was used, in which several pieces of information from one or more cases were related to a theoretical proposition [Eisenh89, Miles94, Yin94]. Seven propositions or "patterns" were found. Some of the propositions were grouped together because their causes or effects were tightly linked. These patterns will be presented in the next section.

- educational background
- work experience
- orientation
- training
- mentoring relationship
- IDs acquired
- computer systems used
- tools used
- time to fully functioning workstation
- system administration tasks reported
- initial task
- time until initial task assigned
- time until working independently
- shortcomings of technical background
- approach to learning system
- time to comprehend office numbering system
- other

Figure 4.4: Variables Used During Analysis

4.3.2 Results

In this subsection the findings of the study are presenting beginning with a narrative overview of the naturalization process, then continuing with analytic results. Counts of some variables will be presented, where relevant, using the following notation: (A, B, C, D) units. This tuple indicates a count of A units for S1, B units for S2, and so on. There are sufficiently few cases that it is possible to present all the data, and this notation allows us to do so compactly.

When software immigrants began work, they were each assigned a mentor. Only S3 received a three-hour formal orientation session from the human resource department; the remainder received informal orientations from their manager. Some respondents attended external formal courses, but they did not find them relevant to their work; respondents attended (0, 1, 0, 2) courses. Mentors acted as primary sources of information to software immigrants, and they passed on a wide range of information to respondents. This information tended to be practical low-level information, such as file naming conventions, system set-up, and pointers on tool usage.

The first two weeks were focused on administrative issues, that is, providing the software immigrant with the equipment, tools, and user identifications necessary to do his or her job. Half the respondents received their first task after two weeks, the other half after three. These first tasks tended to be isolated modifications to the software, or open-ended investigations with no predetermined goal. After four months, five in the case of S4, respondents were working independently of their mentors on tasks that had gradually increased in scope. Although respondents did not yet have a thorough understanding of the system, they were on their way to acquiring one. In the words of S3, "I'm fairly comfortable now. I can read the code and understand it. ...I know where to look for problems, and that's half the battle and I know who to consult when I don't."

In the remainder of this subsection, patterns in the naturalization process will be discussed. The pattern is substantiated with details from the cases, then its implications are discussed and, where possible, related to the literature.

4.3.2.1 Mentoring

- **Pattern 1**: Mentoring is an effective, though inefficient, way to teach immigrants about the software system.
- **Pattern 2**: Lack of documentation forces software immigrants to rely on mentors or consultants.

Evidence

When respondents joined the team, each was assigned a mentor who helped them with all aspects of naturalization. This assistance ranged from providing basic information about the software system, to workstation system administration, to steering them around food choices in the lab's cafeteria. Initially, mentors spent many hours a day with their charge. This time may have been lumped together into a long lecture or it may have been spread out over two or three question and answer sessions in a day. This frequency was maintained for about two weeks and then tapered off quickly. The intensity and duration of the initial contact period was less for subjects whose mentors were working on time-critical tasks. Although contact with their mentors decreased over time, it never stopped completely as maintainers often consult experts about esoteric parts of the software system. By four months, S1's interaction with his mentor consisted of a short question every two days or so. In contrast, S4 had a steady ongoing contact with her mentor because they worked closely together on the same problems.

There is a paucity of documentation for this system; what information does exist resides primarily in the minds of those developers who designed the system architecture and continue to maintain it. S3 stated, "Most people operate under the assumption that there are no documents, so you shouldn't try asking for one." This shortage means that for immigrants, their mentors become their primary source of information about the software system.

Beyond passing on knowledge, mentoring fills a social function as well. Mentors act as a means for integrating an immigrant into the social life of the software team, by providing them with introductions at the lunch table and during coffee breaks. Newcomers need to become conscious of their fellow team members and their areas of responsibility, so that they can turn to the appropriate consultant when necessary.

Implications

A major drawback of mentoring is that it is very time consuming for the senior developer, a phenomenon discussed in the introduction of this chapter. Despite the inefficiencies of mentoring, it may not be possible, or even desirable, to eliminate the system. Mentors function as more than mere repositories of data about the legacy system; the information they provide extends into the administrative and social domains as well. In light of the lack of documentation, it is important to identify who the experts are to new team members.

If changes are to be made to the naturalization process, the mentoring system should be complemented, but not replaced. The experiences of software immigrants in this study were consistent with those found by Berlin [Berlin93]. Like the apprentices in that study, these software immigrants also had interactions with their mentors that were highly interactive, in which they received timely feedback about their comprehension of the software. Efforts should be made to reduce the time commitment required by mentors, so they can still maintain their productivity, while providing adequate guidance to a software immigrant. As a result, an immigrant who has a mentor with a busy schedule, can still receive the necessary training.

4.3.2.2 Difficulties Outside of the Software System

- **Pattern 3**: Administrative and environmental issues were a major source of frustration during naturalization.

Evidence

In every case, almost the entire first two weeks were spent dealing with administrative and environmental issues. These difficulties included setting up their computers, configuring software, acquiring access to systems or tools. In many instances, there was overhead involved in performing simple tasks. Respondents had to maintain (11, 11, 5, 5) identifications, accounts or registrations to do their job.

Only S3 had a fully functioning workstation on the first day of work. Respondents had to wait (3, 6, 0, 1) weeks for fully functioning machines. S4 had a computer on the first day, but had to spend a week configuring it to be usable. S2 did not even have a workstation on his desk for the first three weeks, and then needed another three weeks to set it up to meet his needs.

Sometimes these problems are interrelated, as recalled by S1, "I tried to [set up backups for my machine], but I got stalled because I had to register my machine. So when that comes back, I'll continue…" Although his computer was basically operational after three weeks, S1 had to deal with system administration problems throughout the study.

Items ranging from user identifications to light bulbs had to be requested. Some requests could be serviced quickly but most requests required an overnight wait. Once, when S2 returned to his office with a binder, his office mate asked him, "Where can I apply to get a binder?" Ironically, binders, unlike so many other supplies, did not need to be requested.

Although respondents worked hard to comprehend a large under-documented system, at no time did they describe the task as frustrating. In contrast, frustration was a word that every respondent used with respect to at least one system administration task. This difficulty could be attributed to respondents' lack of experience performing system administration, or the feeling that machine problems were keeping them from their real jobs—programming. Regardless of the causes of this sentiment, it is a problem common to software immigrants during naturalization. Later discussions with the project manager indicated that difficulties with the lack of computing resources were experienced by all members of the team.

Implications

The problems with administrative and environmental issues, particularly the computing resource shortage, would be worth addressing for this team, since benefits would be felt not only by software immigrants but also by veterans. Some real productivity gains could be made here if developers were not distracted by administrative issues. It is not very efficient

for every team member to invest the time to learn how to perform system administration, an activity not directly related to writing code. Many of the processes could be streamlined or combined; for example, user identifications for a set of tools could be linked so that access to them does not need to be requested separately.

4.3.2.3 First Assignment

- **Pattern 4**: Initial tasks were open-ended problems or simple bug repairs, that were begun no earlier than two weeks after a software immigrant's arrival.
- **Pattern 5**: Mentors tend to pass on low-level information about the software system.

Evidence

Shortly after respondents had functioning machines, they received their first assigned task, which occurred at (3, 4, 2, 2) weeks. These initial assignments tended to be limited in scope and complexity, and did not have a fixed deadline. Three of the respondents were given open-ended problems to explore, for the purpose of improving the compiler's performance. S3 was given a bug repair that had been screened for excessive complexity by his mentor. S4's first assignment was to add a feature to a subsystem, and she recalls, "It was a small enough project and I didn't have to know anything else about the rest of the code. So it was a matter of modifying, maybe three or four files... It didn't seem very challenging, but looking back, I appreciate the fact that they gave me something so isolated. It allowed me to gain familiarity with at least those four files."

Three of the four mentors concentrated on conveying low-level information to immigrants. These lessons tended to concentrate on the subsystem that an immigrant would be working on and as a result tended to focus on knowledge that was immediately useful. Only S1's mentor began with high-level system design concepts, but even these lessons were limited to a single subsystem. By concentrating on pragmatics, software immigrants were able to start working with source code quickly.

Implications

Clearly, patterns four and five are closely related: Given the types of information conveyed by mentors, small, non-critical tasks are appropriate first assignments for software immigrants, and vice versa. Even in the absence of pressure from the team, respondents tended to push themselves to contribute. S1 observed this in himself, saying, "Sometimes it's me trying to do several things at the same time: trying to set up my machine and …be a little bit productive for the team." In such situations, the additional demands of a task with a tight deadline is unnecessary. The relationship between these two patterns can be viewed as symbiotic. Any modifications to one pattern, must be reflected in the other. Clearly, the initial task needs to provide an opportunity for software immigrants to use the lessons learned.

4.3.2.4 Predictors of Job Fit

- **Pattern 6**: Programmers who prefer to use bottom-up comprehension approaches have a smoother naturalization than those who don't.
- **Pattern 7**: There needs to be a minimal interest match between immigrants and the software system.

Evidence

At the end of the study, cases S1-3 were still working on the software team, but case S4 was on a temporary educational leave. This provides an opportunity to examine the differences between a team member who may pursue other interests, and ones who are satisfied working as software maintainers on a compiler. The three key differences were S4's inclination to take a top-down approach to comprehending the software system, and her lack of previous experience with compilers, coupled with her depth of background and interest in another field.

Immigrants were trained up from simple tasks to more complex ones. Consequently, software immigrants acquired their understanding of the software, one subsystem at a time, in

other words, in a bottom-up fashion. S1-3 took this approach when they tackled a problem by reading the source code or by profiling the subsystem. In contrast, S4 preferred to take a top-down approach, although there were no real tools or documentation that supported this line of inquiry. She said, "The system was humungous and I didn't know what comes first or anything. So the only way to do it is to dump everything [execution traces]. I didn't do that from the beginning, but I found it really frustrating because I wouldn't know what was actually being done. You need to know... or you don't know where to start."

S4's background also differed from those of the other respondents. During their Masters degrees, S1-2 both wrote theses in the area of compilers. S3 had previous experience working on a highly similar software system. S4 had completed a Doctorate in artificial intelligence. She indicated this was another reason she did not find her work compelling, "I had spent four years working on my Ph.D. and I got hired into an area that had nothing to do with my Ph.D. I just never found it fascinating. They knew that when they hired me. ...They just wanted some one they felt could pick things up quickly."

At this point, it must be stated that S4 was not an unsuccessful software maintainer. Although she is on leave, she has not given any indication that she will not return. When describing her work, she included as many low-level details of the software system as S1-3. She was able to handle tasks that were as complex as the ones given to other respondents. Furthermore, throughout the interview she emphasized that despite the interest mismatch she had congenial relations with the development team. She stated, "The actual group was amazing. I think I was very fortunate to be in that group," and " ...it was a positive experience. I don't regret working there."

Implications

Any improvement in job fit is, indirectly, an improvement on the naturalization process, since reducing a possible turnover rate decreases the time spent in this area by the team as a whole. When hiring new employees to be software maintainers on a large project, managers should look for at least a minimal interest match and a preference to work with system details

in a bottom-up fashion. This is not to say that immigrants without these characteristics are certain to fail or leave, but they will face greater frustration in their early months on the job, a time that has its own share of difficulties. A newcomer with a strong interest match is more likely to be buoyed by a high level of initial excitement about the position, a feeling that does much to mitigate many of the frustrations he or she may face. Indicators of an interest match could be experience in a related field, or it may be as simple as an expressed preference. A scheme to give employees choices in the work they undertake is proposed by DeMarco and Lister [DeMarc87].

4.3.3 Application of the Results to PBS

Taken as a group the patterns provide a coherent explanation for why software immigrants use PBS so little. They spent very little time accumulating domain level knowledge about the system, usually about two weeks. During this time, software immigrants were struggling to get a computer set up. For as long as they did not have a fully functioning computer, software immigrants could not access PBS. By the time they had a computer on their desktop, immigrants were assigned a maintenance task by their mentors and they began tackling it immediately. These initial tasks were localized and only required immigrants to be familiar with a small number of files in a single subsystem. When software immigrants could access PBS, a Software Landscape was the only information available on the subsystem they had been assigned to work on. These diagrams were too abstract to help them understand an algorithm or a specific source file. After some experiments with PBS, they found that it lacked the information they needed for their assigned tasks and put the tool aside.

Software Landscape's main strength is that it reduces the complexity of a software system that is inherent in a directory of source files, by presenting a picture of its conceptual organization. While S3 found PBS to be an invaluable resource, he was the only respondent who had a computer on his desk from the outset. He reported that PBS probably saved him about two weeks of time accumulating background knowledge. Despite these positive early experiences, S3 gradually stopped using the tool after his initial learning period. He

described PBS as a tool of "last resort" that he turned to when all other information sources were exhausted.

The evidence regarding the efficacy of PBS for software immigrants is inconclusive. Although there were some positive and negative comments, there is sufficient evidence to make a strong argument, in either direction, for it as a program comprehension tool. The only unequivocal result is that PBS did not fit well with how software immigrants are naturalized on this development team. At a time when software immigrants would have benefited from the information that PBS excels at delivering the tool was not available due to organizational constraints. In order to keep PBS relevant throughout a software immigrant's naturalization, it needs to be able to provide information that will help them with their initial tasks as well. Some mechanism needs to be added that can provide structural information at a lower level than the Landscapes do. Furthermore, this mechanism should complement the Software Landscapes, so that PBS can provide a more complete picture of a software system.

4.4 Project Veterans

The second PBS user group studied were project veterans. It was expected that this group would use PBS as a reference, that is, to look up information about relationships between modules or to validate their knowledge of the software system. It was difficult to find software maintainers from this category using PBS in their daily work, so it was necessary to use more innovative ways of evaluating PBS as a program comprehension tool for them. Techniques that were used included formal demos, email, meetings to validate Landscapes, and "coincidental" meetings. This study could be best described as informal and consequently it does not have systematic results. However, the findings are intriguing and highlight possible directions for future research.

Demos of Software Bookshelf were organized at IBM internally and at their annual conference, CASCON. On these occasions, the concepts would be explained to team members and their reactions to the Landscapes were noted. Email was exchanged with developers to ask them about how they deal with specific maintenance tasks. Two of the

researchers developing the Software Bookshelf arranged meetings with senior team members to validate the system decomposition used in Software Landscapes [Tzerpo96]. Interestingly, during one of these meetings a bug was identified using only Landscapes. Ambushes occurred when a researcher would see a software maintainer in the hallway, cafeteria, parking lot, or in her or his own office, and ask for her or his thoughts on PBS. The remainder of this section presents comments that were either common to many developers or uncommon enough to be interesting.

4.4.1 Questions about Edges

After the basic Software Bookshelf concepts, such as the Table of Contents, a Software Landscape and the meaning of boxes and lines, were explained the first question that a developer would ask was "What's this edge? Where is it in the source?" They wanted to know what line or lines of source code were responsible for putting a particular edge on the Software Landscape. Developers had more questions about edges than nodes, because the source artifact represented by a node was clear—it was either a file or a collection of files. The source code represented by a box could be accessed just by clicking on the Landscape. There is no mechanism in `lsview` to click on an edge and display the source code it represented.

The question "What's this edge?" is difficult to answer for a number of reasons. First, the low-level factbase only has information about entities such as variables, functions, and files, but not source lines. Even this information is induced to the file level to create the factbase for a Software Landscape. As a result, variables and functions aren't even represented on a Landscape, only the artifacts, such as boxes and arrows, that imply their presence are. Second, it's not clear that "what's this edge?" is really the question that maintainers want answered. Suppose that it was possible to bring up a list consisting of the lines of source code that an edge represented and the list was similar to the following:

```
asm.c:65:          if(memory[i] == NULL)
codegen.c:168:     memory[startMSP] = IorD;
machine.c:59:                  fprintf( sinkFile, "%8hd",
memory[i] );.
```

This list only leads to other questions such as: What variable type is "memory"? What functions are these lines from? Upon examination, the "What's this edge?" has deeper implications beyond adding a feature to the `lsview` Java applet.

4.4.2 Maintainers' Comments on Anomalies

When presented with a Software Landscape, developers exhibit a range of reactions. The strongest reactions were from developers who thought they saw an obvious error in the diagram. These "surprises" raised questions and comments. Sometimes these anomalies were true tool errors, for example a mistake was made during the clustering process and a file was placed into the wrong subsystem. Sometimes these anomalies were a mismatch between the developer's mental image of how the system should be drawn and the representation in the Software Landscape. Other times the anomalies were actual errors in the TOBEY implementation.

The implication of these criticisms is that there is an objective and a subjective element to evaluating Software Landscapes. Both elements can manifest themselves in both the psychological and technical domains. From a psychological standpoint, the objective aspects are apparent in the design principles that should be followed when displaying visual information. The subjective aspect arises when the visual representation is compared with a developer's mental picture of a software system. From a technical standpoint, the objective element is relevant when evaluating whether files have been placed in the appropriate subsystem. The subjective arises in the definition of subsystems in the first place. All four aspects (the cross product of objective/subjective and technical/psychological) may be sources of comments from software maintainers.

4.4.3 Journalism-Style Questions

Some of the questions that senior maintainers asked when repairing a defect or evaluating a set of Software Landscapes were surprising. A senior software maintainer who had been working on a project for a long time sometimes asked journalism questions: who, what,

when, where, why and how. When looking at source code, a possible sequence of questions and answers were:

> What is this thing?
> Who did this?
> Oh, I remember, this was when X fixed bug Y.

Or

> This looks like Z wrote this.
> When was that? How long had Z been working then?
> What was he trying to do?
> He probably did this because...

These questions are indicative of an effort to recover design rationale. If a senior maintainer can determine the circumstances surrounding a change, she or he can often infer why a change was performed a particular way. In these situations, history is being used to make up for the dearth of documentation on this software project. For example, a fact extractor could be applied to configuration management or version control tools to collect historical information.

4.4.4 Code Migration

It was expected that PBS would assist software maintainers as they migrated subsystems from the PL/IX programming language to C++. Those who were involved in the migration effort used Software Landscapes to verify some of their decisions rather than as a driver of the process. There were three reasons for this reticence. One, maintainers did not completely trust Software Landscapes, as they trusted their tried-and-true software tools such as `grep` or `find`. While this reluctance was understandable, it was nonetheless disappointing. Two, maintainers who were sufficiently senior to be entrusted with the task of designing the ported subsystem already had good mental models of the software system as a whole. As a result, they did not feel that they needed to rely on Software Landscapes for a decomposition. However, they did check the Landscapes after the design was complete to verify that there were no surprises. Three, maintainers who were responsible for doing the actual migration were working at a level too low to be helped by Software Landscapes. The task required

them to ask a lot of "What's this edge?" type questions that could not be answered using the existing PBS tools.

4.5 Summary

The results from these two user studies have a number of implications for refining PBS as a code comprehension tool. The points that are most relevant to this thesis are summarized in this section.

- Software maintainers want to relate pictorial elements to lines of source code.
- Low-level information is needed in PBS to complement architectural diagrams.

Evidence for this conclusion can be found in the prevalence of the question "What's this edge?" While abstraction can be helpful for learning new concepts, it does not help developers perform day-to-day maintenance tasks. Their jobs centre around modifying a large corpus of source code. From studying how software maintainers work, we make the following two observations:

- Software maintainers use goal-directed knowledge acquisition.
- Information is gathered by searching and asking questions.

Developers who were studied spent very little time learning for the sake of knowing. Even software immigrants spent only two weeks on open-ended study. It is more often the case that maintainers want to acquire a specific piece of knowledge for a particular maintenance task. Maintainers find this information by searching source code or by asking their peers. Asking questions is a key part of the problem solving process. Developers have been observed asking questions aloud while working alone, and providing the answers themselves.

There are many conclusions that can be drawn from these lessons, two of them are:

- PBS needs a search tool that lets users make queries about source code.
- Additional studies are necessary to develop and understanding of what searches programmers perform on source code and how.

In order to keep PBS relevant to software maintainers beyond the initial open-ended learning period, a search tool needs to be added. The tool needs to be able to answer the "What's this edge?" question, and all those that follow. The primary purpose of this tool would be to

search source code for information that programmers need to perform maintenance tasks. There are two aspects of this problem that need to be examined more closely. First, an understanding of source code searching for maintenance tasks needed to be developed. To this end, a survey of programmers was undertaken and is described in the next chapter. Second, the problem of source code as a search domain presents a technical challenge. A collection of search tools and code analysis tools is examined in Chapter 6. Based on the findings from these two lines of investigation, a design for a searching tool is presented in Chapter 7.

Chapter 5: Source Code Searching Survey

5.1 Overview

A study was undertaken to characterize the source code searching behaviour of programmers. Answers were sought to four research questions:

- What tools do programmers use to search source code?
- Which tasks require programmers to perform a search?
- What do they look for when searching source?
- What do they wish their tools could do?

The tools currently used for searching can serve as role models for the source code searching tool being developed for PBS. Results from the second two questions could be used to construct a series of archetypes to characterize searching behaviours. The last question should provide not only a list of suggested features, but also provide insight into the underlying questions that programmers are trying to answer when they search. This survey and its results are reported in this chapter. A subset of this material has been accepted for publication [Sim98b].

There were two objectives in this study. The primary objective was to understand how and why programmers searched source code. We asked about the tools they used and situations in which they searched source code. Qualitative and quantitative data from participants were used to construct a model of searching behaviours. Anecdotes of the situations were used to develop a series of archetypes of source code searching.

An archetype is a concept from literary theory. It serves to unify recurring images across literary works with a similar structure [Frye57]. In the context of source code searching, an

archetype is a theory to unify and integrate typical or recurring searches. As with literature, a set of them will be necessary to characterize the range of searching anecdotes.

The secondary objective was to determine the efficacy of using a web-based questionnaire to survey programmers. Surveying is a method often used in the social sciences to collect data in a structured or systematic manner [deVaus96]. The methods in this study were similar to those used by Eisenstadt [Eisens97].

The method is further described in Section 5.2, and the results are presented in Sections 5.3-5.6 and Section 5.8. Archetypes and uncommon search situations are presented in Section 5.7. The chapter concludes with a discussion of how the results and archetypes can be applied to tool design.

5.2 Method

In a survey, the specific data gathering technique chosen, i.e. interviews, questionnaires, or archival research, depends on the phenomenon being studied [deVaus96]. Regardless of how data is gathered, there are five steps in performing a survey:

1. Formulate the research question.
2. Create the data collection instrument.
3. Select the sample and sampling method.
4. Administer the survey.
5. Analyze the data.

This survey uses a written questionnaire to collect the data and availability sampling to obtain the participants. The survey was administered using a World Wide Web page and participants were solicited from seven USENET newsgroups. Both qualitative and quantitative methods were used to analyze the data, because the survey had both open- and closed-ended questions. These steps are described in detail in the following sections.

5.2.1 Formulate the Research Questions

In this study, we wanted to understand how and why programmers searched source code. The four questions that we wanted to answer in this study were:

- What tools do programmers use to search source code?
- Which tasks require programmers to perform a search?
- What do they look for when searching source?
- What do they wish their tools could do?

These research questions are exploratory in nature and our goal was to development a preliminary characterization of source code searching. The tools currently used for searching provide role models for future tool development, and their shortcomings suggest areas for improvement. The targets and motivations for searches indicate some of the functionality required in such a tool. The answers to the last two questions were given in anecdotes, so analysis of this data resulted in a set of archetypes to further inform tool design.

5.2.2 Create a Data Gathering Instrument

A questionnaire was selected to be the data gathering instrument because we wanted to collect information from a large number of respondents, many of whom we would not be able to contact personally. The questionnaire consisted of two web pages. The first was an introductory page with an explanation of the purpose of the survey and the rights of the participants. A link at the bottom of the page led to the actual survey. This two page format was used to encourage respondents to read this preamble before beginning the survey. The introduction had two parts, each fulfilling a distinct aim: a purpose statement motivated participants to give thoughtful responses to all the questions, and the statement of participant rights informed respondents of their rights according to standard ethics procedures [deVaus96, Foddy93].

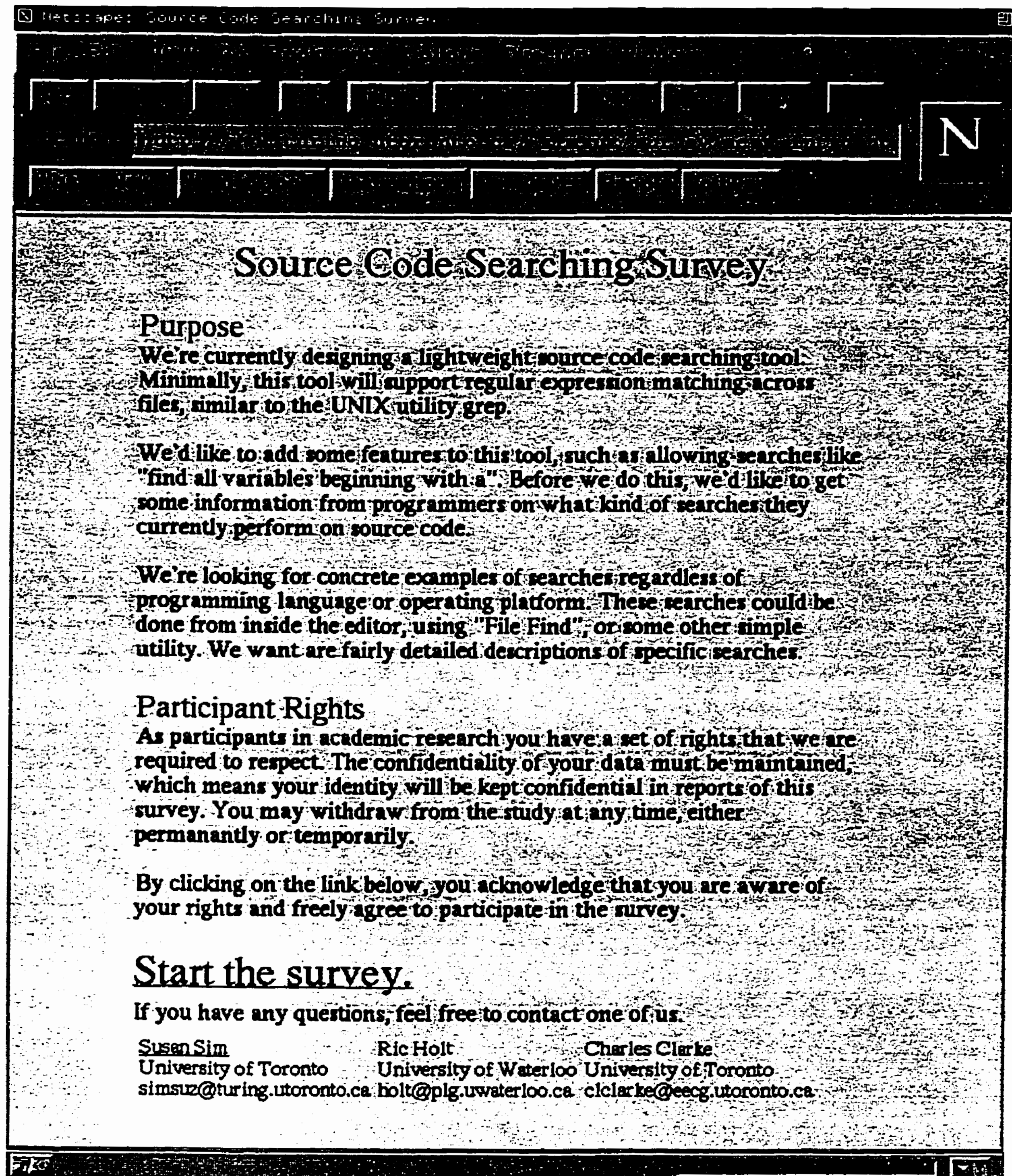

Figure 5.1: Introductory Page of Questionnaire

Question 1: Tools Used

What tools do you use to search source code? Check all that apply.

grep, fgrep, etc. [ ]

find or "File Find" [ ]

editor [ ]

e.g. vi, emacs, edit

integrated development environment [ ]

e.g. MSDS

other [ ]

Please specify: ______________

Question 2: Program Analysis Tools

Do you use an integrated software analysis and exploration tool? Two examples are SNiFF+ and CIA.

Yes [ ]

No [ ]

Question 3: Development Activities Requiring Searching

How useful is it to search source code when:

	Not at all useful				Very useful
doing low-level design?	1	2	3	4	5
writing new code?	1	2	3	4	5
testing?	1	2	3	4	5
understanding old code?	1	2	3	4	5
repairing bugs/defects?	1	2	3	4	5
adding a new feature to old software?	1	2	3	4	5
improving performance?	1	2	3	4	5
inspecting and reviewing code?	1	2	3	4	5
writing documentation?	1	2	3	4	5
maintaining documentation?	1	2	3	4	5

Question 4: Typical Usage Situations

Describe one or more situations when you needed to search source code. What did you use to find it? What were you trying to find? Why did you need to find it?

Question 5: Wish List

What types of searches would you like to be able to perform?

Question 6: Primary Responsibilities
What are your primary job responsibilities? Check all that apply.

Research [ ]
Consulting [ ]
Developing software for a customer[ ]
Maintaining software for a customer[ ]
Developing a software product [ ]
Maintaining a software product [ ]
Developing in-house software [ ]
Maintaining in-house software [ ]

Question 7: Time With Source Code Written By Others
Of your total time spent working with source code, what percentage of that time is spent working on source code written by other people?

0-20% [ ]
21-40% [ ]
41-60% [ ]
61-80% [ ]
81-100% [ ]

Question 8: Participation
Where did you hear about this survey? (Please give the name of the newsgroup or email sender.)

____

Question 9: Future Studies
Would you be willing to participate in future user studies of source code searching?

No [ ]
Yes [ ]
If yes, please provide your email address.
Email:________________

Figure 5.2: Text of Questionnaire

The questions and their wordings were tested in a pilot study of six respondents. These respondents were contacted by personal email and they were later debriefed, again by email. Our experiences from the pilot study are reflected in the final text of the survey. Data from the pilot study were not included in the analysis of the main survey. The text of the survey is found in Figure 5.1 and Figure 5.2.

5.2.3 Define the Population and Sampling Method

The population of interest for the survey was loosely defined, so a random sampling method could not be used. The population was any programmer who had worked with relatively large pieces of existing source code. Due to a lack of demographic information it was difficult to operationalize this definition. It was not possible to enumerate the population and randomly select participants. Consequently, availability sampling, also known as convenience sampling, was chosen. Normally, this method is used only in exploratory studies, such as this one.

Availability sampling is probably the least rigorous of the common sampling methods. It operates by publicly soliciting volunteers to participate in a study. The main drawback of using this technique is it does not obtain a representative sample. First, individuals with the "volunteer personality" are over-represented in the sample. Second, the sample does not represent the population of interest, in this case software maintainers.

In social research, the volunteer personality can be a serious confound because they differ systematically from the rest of the population. While it is a factor in this survey, it is less of a problem because the topic is technical rather than social. It could even be argued that its influence is positive because volunteers tend to be more intelligent, better educated, and more extraverted than the general population, resulting in a participants who can more easily describe their habits [Rosent75].

Had another sampling method been used, it still would be difficult to show the results can be generalized to the population of software maintainers. Not enough is known about the demographics of the population to determine whether a sample is representative. Since this study is exploratory in nature and its goal is to build a model of source code searching, availability sampling is adequate for the task.

Thanks to all of you who have filled out the survey. The responses
so far have been excellent. I'm posting another request for those
people who meant to do it and it slipped their mind, or for those
who just need a little more encouragement.

ses

UNIVERSITY RESEARCH SURVEY ON SOURCE CODE SEARCHING

Are you a programmer? Have you ever had to search your source code?
If you have, please visit:

http://www.turing.utoronto.ca/~simsuz/survey/scss-intro.html

We're surveying computer professionals on how they search source
code as part of the ESSME project at the University of Toronto. We're
looking for fairly basic information, such as what tools you use and what
kinds of things you look for. Our research project builds tools that are
based on what programmers actually need rather than ideas that sound good.

So next time you're waiting for a compile, or if you're having a quiet
day during the "holidays," take 5-10 minutes and fill out the
survey at:

http://www.turing.utoronto.ca/~simsuz/survey/scss-intro.html

Any anecdotes, comments, or ideas that you have will be appreciated.

Thanks in advance for your participation.

Figure 5.3: Message to Solicit Participants

5.2.4 Administering the Survey

The pages were published on a web site and participants were solicited from USENET newsgroups. A message was posted to eight newsgroups:

`comp.lang.c.moderated`, `comp.lang.c++.moderated`, `comp.lang.java.programmer`, `comp.lang.cobol`, `comp.lang.fortran`, `comp.lang.smalltalk`, `comp.lang.lisp`, and `comp.software-eng`.

The same message was reposted one week later with an additional paragraph at the beginning. There were no participants from `comp.lang.c++.moderated`, because

requests for participation were filtered out by the moderator. Figure 5.3 shows the final message that was posted. All of the data were collected within a four week period.

5.2.5 Analyze the Data

Coding is the process of assigning values to variables to represent each respondent. In the analysis of the six multiple choice questions the variables were scalar, such as counts and ratings. For the two free-form responses the variables were qualitative, meaning their "values" were text descriptions or lists. These variables were analyzed by grouping similar responses together. The anecdotes were coded using qualitative data analysis techniques in several iterations [Miles94]. Coding of situations is described in greater detail in Section 3.3.1. During this process, we used grounded analysis, that is, the categorization of search situations was driven by the data, rather than a theory of how a task is performed [Strauss90].

5.2.6 Methodological Considerations

Two issues affecting the validity of the study and the suitability of the chosen method: external validity of the results, and reliability of the respondents.

5.2.6.1 External Validity

The method selected is appropriate for the goals of the study, to build a set of archetypal source code searches. By using a structured data collection method, it is possible to can beyond looking at an interesting story in isolation. With independent confirmation by multiple sources, an anecdote becomes a thread of commonality across cases. Although some of our data is quantitative, it would be inappropriate to generalize them to a population, for the reasons stated in Section 5.2.3.

These archetypes are best applied to the design of tools intended for a user group similar to the sample. In the following sections, it becomes clear that the bulk of the participants work with procedural programming languages, either on UNIX or Windows operating systems.

There were enough Smalltalk and Lisp programmers among the respondents with unique anecdotes to show that the archetypes do not fit them well.

5.2.6.2 Reliability

Since the primary interest of this survey was the *range* of source code searching behaviour, we chose to use a survey rather than interviews or protocol analysis. Results from a relatively large number of people were needed to construct this model, and it would have been very time-consuming to use interviews or protocol analysis to collect the data. One of the advantages of interviews is that they are more dynamic, more interactive, and open-ended, so we attempted to emulate this by making parts of the survey open-ended free-form questions. Recall from Chapter 3 that protocol analysis provides information about the thought processes used while a task is performed by having subjects talk aloud during an experiment. While appropriate for constructing theories at a different level of analysis, this method would not have provided us with data consistent with our goals.

The survey relied on software maintainers' self-reports of their searching behaviour. While not as reliable as direct observation, self-reports are still a good source of data to inform research. Analysis was kept as grounded as possible, so the results presented tend to be summaries of the data rather than a complex argument constructed around the data.

5.3 Results

Sixty-nine respondents provided descriptions of 111 search scenarios and 207 suggestions for features. Overall the quality of the results were quite good; only a small number of respondents did not answer every question on the survey. Most of the responses were in point form, but their thoroughness often compensated for the lack of formality. Some anecdotes were quit long, spanning more than a page. Other, the responses were humorous, for instance, "'Show me the location of the next error I should fix' :-)."

The results of the survey are presented in four subsections. Respondents' backgrounds are described in Section 5.4 and search tools they use are reviewed in Section 5.5. Targets and motivations for both common and uncommon searches are described in Section 5.6. Finally, the various requests for tool improvements are discussed in Section 5.8. Unless otherwise specified, results are presented as counts inside brackets.

5.4 Participants

The credibility of the anecdotes depends on them originating from a variety of sources. Therefore, it is important to show in this section that a diverse group of the participants were obtained before presenting the trends in searching behaviour. The sixty-nine participants who submitted questionnaires came from a variety of newsgroups and email domains, and used a range of search tools. The participants originated from seven different newsgroups. The distribution of their origins is given in Table 5.1.

Newsgroup	Number of Respondents
comp.lang.c.moderated	28
comp.lang.lisp	12
comp.software-eng	7
comp.lang.fortran	7
comp.lang.cobol	5
comp.lang.java.programmer	4
comp.lang.smalltalk	3
unknown	3
Total	**69**

Table 5.1: Origin of Participants By Newsgroup

The last question of the survey asked if respondents were willing to participate in future studies of source code searching. Forty-five respondents were willing to participate in future studies and consequently gave their email address. An analysis of the domains of the mail addresses indicated that more than two-thirds of them were from commercial and government domains. The distribution of participants by domain name is in Table 5.2.

Domain	Number
com, gov, co.uk	26
net, org	5
edu, ac.uk	6
other	8
Total	**45**

Table 5.2: Origin of Participants by Email Domain

5.5 Search Tools

The survey included a multiple choice question on the tools that respondents used to search source code. The available choices are shown in Table 5.3. In addition, a box was provided for the name of any tool that fell into the "other" category. We found that the participants generally relied on standard tools. The `grep` category included its variants such as `fgrep`, `egrep`, and `agrep`, which perform regular expression matching over files. Although it is capable of much more, `find`, in its most basic form, is a tool that searches file names. Almost all the respondents (65) used either their editor or IDE (integrated development environment) to search source code, and yet a large number of them used other tools as well.

Tools Used	Number
editor	57
`grep`	47
`find` or "File Find"	38
IDE	26
other	38

Table 5.3: Tools Used

In the fill-in box for the "other" category, a total of nineteen different tools were mentioned. The distribution of the tools from this category is given in Table 5.4. Some participants entered more than one tool. If a tool was mentioned in an anecdotes that was not already in the list of "other" tools, then it was also added.

Included in the category of "tagging utilities", were `etags`, `ctags`, and `ftags`. The "scripts" category includes any shell scripts, Perl or awk programs, and batch files.

"Proprietary source browsers" included tools that were sold for the purpose of source browsing such as Cygnus Source Navigator, SoftBench, and tools that were bundled with third party libraries. Smalltalk and Lisp programming environments were included in "language environments." These tools were included in this category rather than IDE because they include a number of elements that are tightly integrated with the language and run-time environment. The UNIX utility `xref` builds a cross-referencing index of functions and variables. The last category included Norton Text Search, `javadoc`, the compiler, and "my brain".

Tool	Number
tagging utilities	11
scripts	7
proprietary source browsers	6
language environments	5
`xref`	4
miscellaneous	10

Table 5.4: "Other" tools used

5.6 Situations

```
I needed to understand old spaghetti code which used global
variables for everything. Say there was a variable 'foo'
which stored a critical value. I'd grep for reads and writes
to this variable, to see which functions were involved in
creating and using this value. I'd also search for it(in
emacs) in a cross-reference listing to make sure I didn't
miss some place.
```

Figure 5.4: Example of Scenario Anecdote

We received descriptions of 111 searches that ranged in length from a single line to more than a page. All but four respondents contributed anecdotes. Figure 5.4 contains a typical anecdote regarding a situation that required source code to be searched. In this subsection, the results of analyzing the anecdotes are presented. First, the search targets and the motivations for searching are discussed. In Section 5.7, the relationships between these two dimensions are examined to formulate searching archetypes.

5.6.1 Coding and Analysis of Anecdotes

Anecdotes were categorized along two orthogonal dimensions: the specific search target and the motivation for performing the search. The coding categories are presented in Figure 5.5. Search targets tended to be quite easy to categorize, whereas motivations required stricter rules for categorization. Some anecdotes had multiple search targets or multiple motivations. The example presented above, the search targets were coded as "function definition" and "all uses of a variable," and the motivation was coded as "program understanding".

Specific search target	Motivation for Search
1. function declaration	16. dead code elimination
2. function definition	17. clean up
3. function use	18. impact analysis
4. function use-all	19. bug repair
5. variable definition	20. feature add
6. variable use	21. naming conflicts
7. variable use- all	22. porting
8. class definition	23. code reuse
9. class use	24. maintenance
10. class use- all	25. program understanding
11. specific string	26. other
12. specific string- output	
13. specific string- com	
14. file	
15. other	

Figure 5.5: Coding Categories

The program understanding category was used as little as possible because it could be argued that all searches are performed for that purpose. In the example, a program understanding motivation was selected because the respondent gave no other explanation for why she was performing the search. The maintenance category was also used in a similar manner. We selected the most specific motivation for the search based on statements by the participant.

5.6.2 Search Targets

During coding of the 111 anecdotes, 154 search targets and 94 motivations were identified. The four most common search targets were function definitions (26), all uses of a function

(23), all uses of a variable (23), and variable definitions (19). Definitions are the portion of the source code that implements a function body or determines the type of a variable. Searches on functions, variables, and classes are summarized in Table 5.5. Further analysis of the searches on variables indicated that respondents were more interested in locations where a variable was written or assigned to (6) as opposed to simply read or referenced (1). Clearly, a piece of code that changes a variable affects the program more than one that only reads it.

	Function	Variable	Class	row total
declaration	10	-	-	10
definition	**26**	**19**	5	50
(single) use	11	9	1	21
all uses	**23**	**23**	5	51
column total	70	51	11	132

Table 5.5: Summary of Common Searches: Numbers shown on the table are counts of occurrences. Top four values are in bold. There were 154 total search targets.

Other common targets of searches were strings, either those output by the program or those in comments (10), and files where code was located (5). All searches for strings output by the program coincided with a defect repair. Software maintainers often take the error message in a bug report as a starting point for their investigations. They search for the line of code that is responsible for outputting that message and trace backwards from there.

5.6.3 Motivations for Searching

The motivations for source code searching were grouped into eleven categories as shown in Table 5.6. The categories and names are straightforward, with the exception of "clean-up", and "naming conflicts". These two categories will be discussed in greater detail and examples for each are provided.

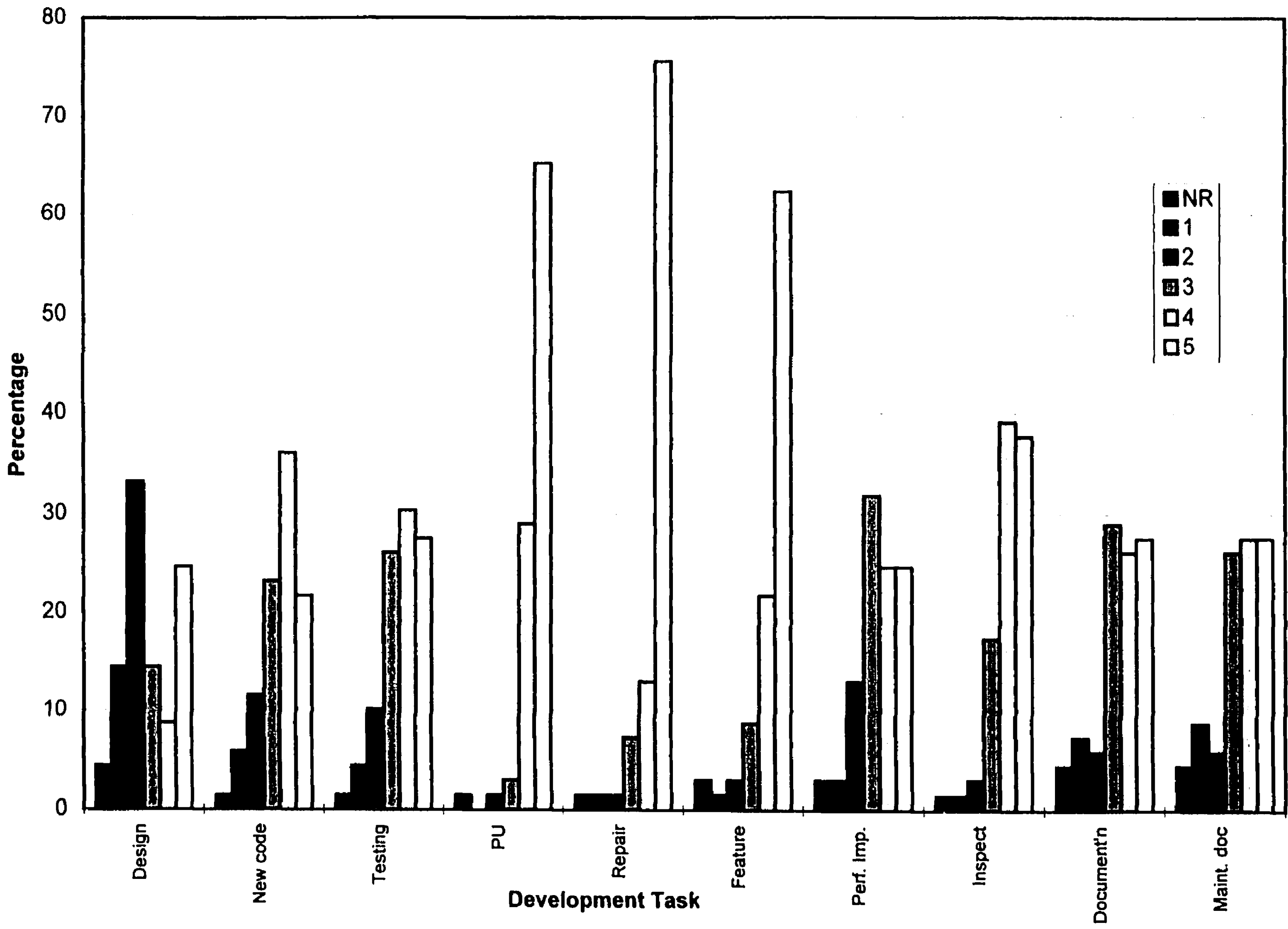

Figure 5.6: Usefulness of Search Source Code By Task. This histogram shows the distribution of ratings for each development task. A bar shows the percentage of respondents giving that rating for a task. The tasks are low-level design, writing new code, testing, program understanding, repairing a defect, adding a new feature, improving performance, code inspection, writing documentation, and maintaining documentation.

Clean-up occurs before a program is frozen for release. A programmer may hard-code some strings during development, or leave notes to herself in the code. These items are removed before the code is shipped. A naming conflict occurs if a new function, variable, or class uses an existing identifier. A developer searches code to ensure a proposed identifier is conflict-free. In such cases, the programmer picks a name and searches to ensure that no conflict exists.

Motivation	Number
defect repair	19
code reuse	14
program understanding	13
impact analysis	12
maintenance	7
feature addition	7
clean-up	5
naming conflicts	4
porting	3
dead code elimination	3
other	7
Total	94

Table 5.6: Summary of Motivations for Searching

The four most common motives for searching source code were defect repair(19), code reuse (14), program understanding (13), and impact analysis (12). The results of this analysis should be compared with those from question three of the survey. It asked, "How useful is it to search source code when…" along with a list of ten activities from the software development cycle, and asked respondents to give a rating on a scale of one (low) to five (high). It was found that the tasks in which searching was most useful (median rating 5) were repairing bugs or defects, understanding old code, and adding a new feature to old software. The distribution of the ratings are presented in Figure 5.6.

5.7 Searching Archetypes

Archetypes were generated by examining the search targets and motivations presented in the previous section for patterns. Common or frequently-occurring relationships between targets and motivations were identified as a pattern. Eleven archetypes are presented in this section,

beginning with the strongest ones. Also presented in this section are uncommon searches because they complement the archetypes by capturing the additional variability.

5.7.1 Common Searches

The pattern that emerged in the impact analysis category is the most definite.

1. During impact analysis, developers often looked for all uses of a variable or function.

Of the twelve searches with this motivation, nine were for all uses of a function or variable. Impact analysis is usually done to evaluate a change to the software. The developer wants to make sure that she has not broken anything inadvertently, therefore checks all uses of the modified component. This relationship is credible not only because the underlying explanation is plausible, but also because the numbers in this category are consequential.

In the program understanding category there were two main patterns of searching.

2. Searches motivated by program understanding sometimes sought function and variable definitions.
3. At other times, the search targets were a use of a function, variable or object.

Of the thirteen searches performed for this purpose, five were looking for definitions of functions or variables, and five were looking for function or variable or object use. In the case of definitions, the maintainer was trying to determine the effect of a particular function call or the data type of a variable. In the case of the latter, she understood the object, variable, or function, but wanted to know how it fit with the rest of the program.

The code reuse category revealed two patterns of searches.

4. To reuse code, a programmer searched for function signatures to call it correctly.
5. Alternatively, a programmer searched for functionality that was known to exist, but the name may not have been known.

Of the fourteen searches undertaken for the purpose of reusing code, seven were for function definitions and three for function declarations. When reusing code, one of two scenarios may occur: the developer knew the name of the function but needed to check the parameters in the

declaration or definition; or the developer knew that code to perform a certain procedure existed, but was unsure of its name, so she performed a search.

In the bug repair category, there were a large number of examples (19) with a variety of search targets.

6. Maintainers tackled bugs by identifying the function that was misbehaving.
7. Another approach was to track usage of a variable.
8. An output string served as the starting point for a bug-hunt.

The three most common targets were function definitions (4), all uses of a variable (3), and output strings (3). The first pattern corresponds to a situation where a programmer knew that something was going wrong and was looking for the function responsible. Consequently, she looked at a lot of function implementations or definitions. The second archetype corresponds to a scenario where a maintainer knew a variable was set incorrectly during execution. In such a case, she looked at all uses of that variable to find the error. In the case of the third pattern, the programmer has received a bug report containing an error message. The search for the faulty code began by tracing how the message came to be printed. This pattern was particularly strong because all instances of searches for output strings were motivated by bug repairs.

In the porting, feature addition, and dead code elimination categories, relationships were found, but due to the small number of anecdotes it is difficult to evaluate their significance.

9. To eliminate dead code, a maintainer needed to find all uses of the entity being removed.

In all of the dead code elimination searches(3), the targets were all uses of either a function (1) or a variable (2). In order to eliminate a variable or function, the maintainer has to make sure that it is either not used at all or used only in functions that will never be called. Therefore, she needs to be able to account for every use of that function or variable. This relationship is more credible than the others that have a small number of examples because its underlying explanation was present in the anecdotes and is highly plausible.

10. When porting code, developers often examined variables.

In all of the porting examples (3), the respondent was looking for information about variables. In two cases, it was all uses of a variable, and in the third it was the variable definition.

11. When adding features, developers sometimes examine functions.

In four of seven feature addition searches, the respondents were looking for information about functions. There were no clear patterns found among the searches in the clean-up, naming conflicts, and maintenance categories.

5.7.2 Uncommon Searches

In this section, we present some of the unique anecdotes we received in the survey. These anecdotes are noteworthy because they illustrate some the issues that software maintainers have to deal with, but are easily overlooked because they are atypical. We look at searches performed for preventative maintenance, code reuse, and testing.

Although preventative maintenance is generally agreed to be a good idea, many software shops don't have time to do it. In the study, we received two anecdotes that described searches that were performed for the purpose of doing preventative maintenance, at least on a small scale. Respondent 17 recalls an occasion when she discovered a variable had been used unsafely and she went through the source to verify other uses of that variable.

```
Upon noting an unchecked strcpy() into a global char *, [I
needed] to locate the declaration for the variable to
discover it's size and locate references to that variable to
see if bounds checking was performed explicitly.
```

Another application of searching to do preventative maintenance was described by respondent 66. She would look through the code for:

```
mundane spell correction: how many ways did i spell one
variable name by accident
```

If an identifier is used only once, then it is likely an error. An unused variable can be caught by a compiler or interpreter if warning levels are set appropriately, but these discrepancies

can be a problem in languages that do not require variables to be declared before they are used. An example of one such language is Perl. Although Perl is usually considered a scripting language, used for quick and dirty programming, it is being used for increasingly larger projects on the World Wide Web. Consequently, a tool that could ferret out identifiers that occur only once could become increasingly important.

Some common code reuse examples were discussed in the previous section, and to these respondent 23 adds the following example:

```
It has also helped in the design phase to be able to find
another program that was used for the same purpose and this
helps others to develop their applications quicker.
```

Rather than just reusing existing functions during the implementation phase, her team tries to reuse code during the design phase as well. This programmer searches for code with a particular functionality to make further development easier. It's not clear how the respondent performs these searches and what tools she uses, but the possibilities are intriguing.

The usefulness of searching during testing had a low ranking (fifth out of ten maintenance tasks), but a high rating (median of 4). An anecdote from respondent 66 illustrates this finding:

```
how many ifdef? where are they? used to figure out relevant
test cases for ported code
```

Hence, searching is probably not used during the actual testing of code itself, but it can be helpful in generating test cases.

5.8 Respondents' Suggestions for Features

We were interested in the shortcomings of existing tools and what kinds of searches developers would perform if their tools could support them. In question five on the survey, under the heading of "Wish List", we asked "What types of searches would you like to be able to perform?"

Not all respondents gave suggestions for features, but those who did often had a lot to say. Forty-three respondents gave 207 different suggestions for features. An example suggestion is shown in Figure 5.7. It contains about as many suggestions as a typical response, but is more concise. Some of the requests were for features already available in existing tools, while others were novel and interesting ideas. There were suggestions that would have been more appropriate for other software tools, such as visual debuggers, or editors. As with the previous free form question, there were the humorous suggestions, one respondent wrote, "...and I want a built-in cupholder."

```
I'd like a tool (both command-line and interactive) which
deals with types, macros, local & global variables, functions,
and where you can get all sorts of listings, given a set of
source files; module & function where it is defined, modules &
functions where it is used (read vs. written)
```

Figure 5.7: Example of Wish List for Features

We found that a set of rational suggestions is not necessarily a rational set of suggestions. While each respondent gave self-consistent suggestions, as a group the suggestions were sometimes highly contradictory in nature. The list is by no means a recipe for success. Any researcher who took the entire list of suggestions and implemented them all in a single tool might be disappointed with the results.

The suggestions for features could be placed into three groups: requirements for a software tool; requests for existing features; and suggestions for functionality. The thirteen tool requirements were the most contradictory. Below are portions of three responses to this question.

```
...play well with the existing unix environment.  ie, I need
to be able to write shell scripts around it, use it
comfortably from within emacs, other utilities, etc.

If you could provide a visually oriented tool that would
allow me to construct regular expressions without having to
remember the rather arcane syntax I would be most grateful.
```

```
Here's what I would not want.  Select from a menu to pop up
a form.  Fill in the form.  Click OK.  I won't mention
names.
```

The overall message appears to be: developers want tools that fit with the way they work. So when designing a tool, it is important to know your users and how they work.

There were fourteen requests for existing functionality. Of these requests, seven were for regular expression matching, four were for searches on multiple files or a subdirectory hierarchy, and one each for optional case sensitivity, fast search summaries, and multiple search targets. There are several possible explanations for these requests. Respondents may have included these suggestions to emphasize how useful they found them. Or this functionality may not have been available in their operating environment. Alternatively, this functionality was available, but the respondent was not aware of it.

In the remaining 170 suggestions, the most common recommendations were for building in some awareness of the programming language (88) and greater ability to control the scope of the search (28). Of the former group of suggestions, 47 were for the ability to easily find the common search targets such as the declarations, definitions, uses, and all uses of functions, variables, or classes, as displayed in Table 5.5. In the case of variables, respondents again wanted to be able to discern between uses of a variable that were assignments and references. The other recommendations in this group were being able to include, exclude or focus on elements such as identifiers, comments, and quoted strings. Some respondents wanted to be able to optionally preprocess the code before searching. Others wanted the search tool to be able to resolve references, such as pointers and macros.

The ability to control where a tool searched was also important to respondents. They wanted to be able to specify multiple files, sets of files, a search path, multiple subdirectory hierarchies, or modules. Three respondents suggested limiting the search to the current compile environment, meaning the file from which the search is originating and the files that

it links with to form an executable. Others requests were to be able to search only among instances of a particular data type, among those functions and variables currently in scope, or within an `#ifdef` macro branch in C/C++. Being able to easily control the search domain meant not just being able to include elements, but exclude them as well.

Some respondents suggested being able to control the size of the successful match (7). A single line was sometimes too small a range, and at other times it was too large. Participants wanted to be able to match a target that appeared over several lines. They also wanted to match only tokens from the language.

There were three requests for the ability to do searches that optionally ignored white space, so that a search for a function call such as "`add(`" would match "`add(`", "`add \n(`" and "`add (`", as well. Although this search can be performed using regular expressions, its specification requires more typing than most users are willing to do. There were other requests that went beyond regular expressions. There were five recommendations for exact matching of a literal string. Fuzzy searching was also requested, that is, the ability to specify a target that is close to the desired result. For example, it is possible to perform a search for a variable that is "kind of long and has a bunch of vowels at the beginning". It is useful for finding a function that is known to exist, so it can be reused. Three respondents wanted to perform searches on the search results, and two wanted to be able to perform searches based on functionality, i.e. find a function that does matrix multiplication.

There were a large number of suggestions that appeared only once, and these tended to be intriguing. One respondent wanted a tool that could identify all the functions that could have an impact on a variable. This is similar to finding all locations where the value of the variable is modified, but with the results in a call graph format. Another respondent wanted to find blocks of code greater than a given number of lines that were identical. Once these common areas were found, it may be possible to replace them with a single function.

5.9 Implications for Tool Design

As is evident in both the common search targets and the suggestions for features, respondents search for semantic elements in the source code. In this context, "semantic" means units that are meaningful in the language and can be "understood" by the compiler. Sometimes the units are labeled as syntactic, and the term semantic is reserved for their real-world meaning or effects. The most common search targets were function definitions, all uses of a function, all uses of a variable, and variable definitions. A large group of suggestions for search tool features requested greater awareness of the programming language; 88 out of 180 suggestions fell into this category.

The bottom-up aspect of the integrated code comprehension model seems to be at work here. Programmers are trying to build meaningful units from text strings and by finding the understanding the semantics associated with the identifiers. This strategy contrasts with the ones applied to Software Landscapes. When presented with an abstract pictorial representation of a software system, maintainers wanted to use a top-down approach. They wanted to relate a visual element, such as a box or arrow, to source code as directly as possible. By adding a tool designed for searching source code to PBS, software maintainers would then be able to use multiple code comprehension strategies, that is, both top-down and bottom-up, in combination to understand a software system.

Although programmers are searching for semantic elements, few of their tools support searches keyed in this manner. Instead, the mechanism that they use to perform these searches is regular expression matching on the source code. Aside from editors, `grep` is the tool most commonly used to search source. Given how `grep` is already being used, it can aptly be augmented with semantic searches. Indeed, this problem has studied by many other researchers as will become evident in the next chapter.

5.10 Application of the Results

The searching patterns that were identified in this study can also be used when designing program comprehension tools. The situations presented in Section 5.6, particularly the

subsections on Common and Uncommon Searches include examples of searches. A designer can evaluate her code comprehension tool by applying it to one of these situations and ask questions such as: Could the tool provide the information that a software maintainer needs to perform this task? What are the commands that the maintainer would have to use? How well does the user interface perform in this situation? Could it be made more efficient? By using a number of these situations, the designer can determine the flexibility of the tool. Finally, the scenarios can also be used to guide the development of experiments with software maintainers on the utility of the program comprehension tool.

The secondary purpose of this study was to determine the efficacy of our research method. We were able to obtain responses from respondents in different organizations from around the world, without the drawbacks of travelling. In many ways, web-based questionnaires are superior to their paper-based counterparts: the logistics of dealing with paper are eliminated; the researcher has greater control over the format and administration of the questionnaires; and the respondent submits the data in electronic form, which removes the need for transcription.

5.11 Summary

The goal of this study was to identify patterns of searching behaviour in order to construct a model that could be used in tool design. We found that searching was most important during defect repair, code reuse, program understanding, feature addition, and impact analysis. This finding is supported by the ratings of the usefulness of various software maintenance tasks, and the most common motivations for searching in the anecdotes. The most common search targets were function definitions, all uses of a function, all uses of a variable, and variable definitions.

The main suggestions for features in a tool was for greater awareness of the language being searched and for greater control over the search domain. In other words, participants wanted a tool that could match more than just characters in very specific locations. Far behind these

two requests were others for the ability to match more than just a single line, fuzzy matching, exact matching, and searches on functionality.

In the next chapter, a number of searching and source code analysis tools will be examined to identify the various approaches to solving this problem. In Chapter 7, a design for a search tool for PBS is presented. The results of this study will be used to guide design decisions in the development of a "semantic grep" for use with PBS and Software Landscapes.

Chapter 6: Supporting Queries on Source Code

6.1 Overview

A conclusion from the previous chapter is that software maintainers seek information to complete a particular task. To this end, they search source code for elements from which to build conceptual models of how the software operates. One tool that is frequently used for this purpose is the UNIX regular expression matching utility, `grep`. Adding semantic awareness of a programming language to `grep` would help support bottom-up code comprehension models. In this chapter, search tools and source code analysis tools are examined as possible role models for a "semantic `grep`".

We make a distinction between tools to search source code and tools to analyze source code. Analysis tools are employed to extract facts from a software system. These tools will be examined for the approach they take to building a factbase. Search tools are employed to make specific queries, and are often applied to factbases. Although analysis tools can be tuned to answer specific questions, software maintainers' knowledge acquisition strategies are more oriented towards searching. As goal-directed information seekers, they ask questions and look for answers that are necessary to complete the task at hand; they are not trying to build an encyclopedia about the software system.

This chapter begins with an examination of `grep` based on work done by Singer and Lethbridge [Singer97b]. The remainder of the chapter examines different approaches to searching or analysing source code. Some of the software tools that we consider are specifically for working with source code, while others are tools for general purpose searching or information retrieval. Only relatively lightweight tools were considered because we are building a tool that fits cleanly with the PBS paradigm of lightweight tools. As a

result , tools such as Rigi [Müller93], SNiFF+ [SNiFF+96], and CIA [Chen90] were not included in this survey.

Among the general purpose search tools that will be examined are additional members of the `grep` family, `cgrep` [Clarke96], `sgrep` [Jakko95], and `agrep` [Wu92]. Two environments for searching source code, `tksee` [Singer97a] and SCRUPLE [Paul94], are included in the survey. Finally, language-independent approaches to extracting structural information from source, LSME [Murphy96] and TAWK [Griswo96] are discussed. Similar to the analysis performed on `grep`, the strengths and limitations of these tools will be examined. These attributes will provide suggestions for the design of a semantic `grep`.

6.2 grep

The `grep` tool is a UNIX utility that performs regular expression matching over files on a line-by-line basis. The success of this tool is evident in the family of tools that it has spawned. The `egrep` variant does extended regular expression matching, `fgrep` does fast string matching, `zgrep` searches in zipped files, and there are others, `cgrep`, `sgrep`, and `agrep` that will be described in Sections 6.3-6.5. In our survey, 47 out of 69 respondents used `grep` to search source code. The prevalence of this tool was also noticed by Singer and Lethbridge [Singer97b] and they make a number of observations about `grep` as a model for a program comprehension tool. A selection of the strengths and limitations they observed along with our own observations are presented in Sections 6.2.1 and 6.2.2.

6.2.1 Strengths of grep

In this subsection, the positive aspects of `grep` are described. Attributes PG1-PG5 are taken from Singer and Lethbridge [Singer97b] and PG6-PG8 come from our own experiences and analyses.

PG1. Success, little cost of failure, and understanding of limitations = trust.

The `grep` tool excels at performing a specific task. Consequently, it is easy to specify a search and the results are returned quickly, frequently with a relevant match. When the

search fails, little time or cognitive effort was wasted. Since `grep` is a small utility, users can understand its many limitations.

PG2. Command-line interface.

With its command-line interface, `grep` is able to fit with the other UNIX utilities, and by extension, users' interface style. Furthermore, `grep` can be included easily in scripts and macros to automate repeated tasks.

PG3. Straightforward specification

The only required arguments for a `grep` search are a target pattern and a search domain, all other information is optional. The target pattern can be a simple string or a regular expression. The search domain is one or more files or directories that can be specified through file name expansion by the operating environment.

PG4. Results displayed in "parallel"

Since `grep` acts as a filter, all matches to the search patterns are displayed, as they are found. These results can be scanned quickly for the relevant match. In contrast, many editors display the results in sequence, stepping the user from match to match inside a file.

PG5. Scaffolded

Regardless of how much experience a user has had with `grep` or the software system being searched, the user will be able to obtain results from `grep`. The user does not need to learn regular expression syntax and `grep` command options in order to use the tool. This knowledge can be acquired as the user feels the need, and as a result there is a smooth transition from novice to expert.

PG6. Portable and flexible

Programmers can use `grep` with whatever software systems they are working on. Software maintainers can bring their skill with this tool to any project, in any programming

language. While `grep` is primarily a UNIX tool, implementations are available on other operating systems.

PG7. Little overhead

No indexing of the search domain is needed. Users are not required to generate an index or factbase of the source before using `grep`. Sometimes this requirement alone is enough to dissuade a software maintainer from experimenting with a tool.

PG8. Responsiveness

Users don't have to open a new window and wait for it to initialize, just to perform a "simple" search. The time spent waiting is too disruptive if a programmer is trying to sustain a complex train of thought.

6.2.2 *Limitations of `grep`*

Singer and Lethbridge also listed a number of limitations of `grep`, which indicate possible areas of improvement. A subset of their observations is presented here as NG1-NG5 and we add observations NG6 and NG7.

NG1. Interpretation of output requires effort

When `grep` returns a large number of matches, it is sometimes difficult to find the most relevant one. The string matching the search target may be difficult to find and the search results themselves may need to be searched. Each match consists of a single line, which often does not provide enough context to interpret the match.

NG2. No near searches

Matches can not be approximate and must be exact according to regular expression rules. If there is a spelling mistake in a search target, there is no facility in `grep` to deal with this.

NG3. No semantic searches

The `grep` utility treats all input as straight text. When searching source code, there is no way to limit the search domain, for instance to identifiers or comments.

NG4. No memory

Search targets or contexts are not stored and cannot be revisited for refinement or modification. Saved sequences with annotations could be useful for teaching software immigrants about a software systems. The command history of a shell can help, but is not the complete solution.

NG5. No browsing

The search results can't be browsed like hypertext, for example clicking on a matched line to display the entire file.

NG6. Fixed "hit" size

Searches using `grep` return whole lines that match. Sometimes the desired unit of return or match may be larger than a line, such as a module or function definition. At other times it may be smaller, such as a function parameter, an identifier, a specific column of a line, or a literal string.

NG7. Sensitive to whitespace

Many programming languages are insensitive to whitespace, in that the number, or type, of spaces between tokens is not significant. Searches in `grep`, however, are sensitive to whitespace. For example, the expression "add();" will not match "add ();". Although, it is possible to write an expression that is insensitive to whitespace, but to do so accurately would require more effort than most users are willing to expend. Expressions such as `add[[:space:]]*();` or `add[ \t]*();` require both a non-trivial amount of knowledge of regular expressions and time to type out the specification.

6.2.3 *Analysis of grep's Attributes*

Clearly, grep acquires many of its strengths and limitations from being a UNIX utility. It would be difficult to address some of its shortcomings without modifying this interface and compromising some of its strengths. For example, browsing through pointing and clicking is difficult to achieve within a command-line interface. Adding semantic awareness would require grep to parse its input, either when the search is invoked or beforehand to build an index. Regardless of the option chosen, at least one of speed, flexibility, and portability would be affected. Another factor to consider when adding semantic awareness to grep, is the syntax needed to query the different elements. This problem is further compounded if the user wants to search in units other than files, such as modules or subsystems. A syntax that supports queries on these search elements needs to be added.

6.3 cgrep

One of grep's limitations is that matches must appear on a single line. The cgrep (context grep) tool addresses this shortcoming by treating the input as a character stream and interpreting the newline character as ordinary text, so that it can return matches with arbitrary sizes [Clarke96]. It uses a shortest-match algorithm and allows matches to overlap, but not nest, which means the program reports "every substring of the input text that matches the regular expression and that does not itself contain a matching substring." This change results in a faster algorithm and is motivated by experience with structured text databases.

6.4 sgrep

The sgrep (structured grep) tool performs searches on text files or streams that have structural markup, such as e-mail, USENET news, source code, HTML, bibliographies, etc [Jaakko95]. Searches return *regions* that are delimited by strings or tags. Regions can be arbitrarily long, overlapping, or nested. Although sgrep is essentially a command-line tool, there is a tcl/tk graphical user interface, sgtool, available.

The syntax for specifying queries in sgrep is based on the GCL query language taken from Clarke [Clarke95a]. The command-line specifications for sgrep searches can be quite

complex, but the tool can handle macros to simplify frequent targets. However, accompanying this specification complexity is a corresponding ability to handle complex search targets. For example, the query 'show the if-statements containing the string "access" in their condition in the setOptions function of the source files *.c' is specified as:

```
sgrep '"if" not in("/*" quote "*/" or ("\\n#" .. "\\n")) \\
     ("(" .. ")") containing "access" \\
          in ("setOptions(" .. ("{" .. "}")) \\
     ..("{" .. "}" or ";")' *.c
```

With the following macro definitions,

```
          define (BLOCK, ( "{" .. "}" ))
          define (COMMENT, ( "/*" quote "*/" ))
          define (PPLINE, ("#" in start or "\\n" _. ("\\n" or
               end) ))
          define (IF_COND, ("if" not in (COMMENT or PPLINE) ..
               ( "(" .. ")")))
```

the above command can be simplified to:

```
sgrep -p m4 -f c.macros -e 'IF_COND containing "access" \\
     in ( "setOptions(" .. BLOCK ) .. (BLOCK or ";")' *.c
```

Even with these simplifications, the search specification syntax is quite complex. Users would probably have to add aliases and scripts to make the tool more usable.

6.5 agrep

The `agrep` tool [Wu92] is another `grep` variant with three modifications. It allows approximate matches by permitting a user-specified number of substitutions, insertions, or deletions. Second, instead of restricting 'hits' to single lines, `agrep` can return matching records, such as entire email messages. Finally, it allows the logical combination of patterns using AND or OR.

Approximate matching would fulfill some of the suggestions from the source code searching survey that asked for fuzzy searches. It could be used in situations when the maintainer didn't know the exact name of an identifier. As noted above by Singer and Lethbridge, while the wildcards in regular expressions allow some degree of flexibility in the matches, they do require the search to be specified accurately.

6.6 tksee

Singer et al. addressed limitations of `grep` with `tksee` (Software Exploration Environment with a `tk` interface) [Singer97a]. It is essentially a semantic `grep` inside a graphical user interface. Search targets can be regular expressions, strings, identifiers, function and variable definitions and uses, macros, etc., and can be entered in a text box, initiated by a pointer, or selected from a menu. Matches and their attributes can be browsed and searched. Search history can be browsed by clicking and search sequences or sets can be saved.

The architecture of `tksee` is similar to that of PBS. Its back end is a fast, object-oriented database containing facts extracted from the source code. The factbase is language independent and contains some clustering information. Data is passed between tools in a Tuple Attribute Language variant, TA++. Clients must connect to the server, sometimes over a network, to access the factbase.

6.7 SCRUPLE

In SCRUPLE [Paul94], there are elements of both `sgrep` and `tksee`. To find matches, SCRUPLE parses the source code when the tool is invoked, and there are versions of the tool that work with C and PL/AS. Users specify search targets using a pattern language, and the results are displayed to standard output. Paul and Prakesh initially designed SCRUPLE to be a command-line tool that extended `grep`, but later added an X-windows graphical user interface. In the GUI, when a search is completed, the user is walked through the code to each match.

```
1 $t $f_decl()
2      {*
3             @*
4             @{* #{* $f_call (#*) *} *]
5             @*
6      *}
```

Figure 6.1: Call Graph Extractor for C in SCRUPLE from [Griswo96]

In SCRUPLE's pattern language, there are various wildcard symbols for different syntactic entities. There are generic wildcards, wildcards for sets, and named wildcards. For example,

a declaration is represented by "$d", a set of arbitrary declarations is "$*d", while the declaration of entity "count" is "$d_{count}". Other syntactic entities for which there are wildcards are types, variables, functions, expressions, and statements. The query, "find all declarations of the variable x" is specified as "$t x;". An example of a more complex query is "$t $f_x <xmax> ($v*) { @* }", which means "find all functions that have references to the identifier xmax." A call graph extractor can be written using SCRUPLE in just 6 lines, as shown in Figure 6.1. While this specification language is quite powerful, it is programming language specific. In other words, a new pattern language is designed for each programming language to be searched.

6.8 LSME

A "source model" is a view of a software system, for example call graphs, file dependencies, etc. and is usually extracted by parsing the source code. In contrast, LSME (Lexical Source Model Extraction) [Murphy96] extracts source models without a language-specific parser. Conceptually, LSME is much like awk [Aho79] in that it scans through the source code doing pattern matching of constructs and regular expressions, and executes commands as matches are found. Like awk, LSME specifications are conceptually closer to a script or program, than a command-line utility.

The LSME tool achieves language independence by requiring the user to specify the match syntax. Running the specification:

```
[ <type> ]<funcName>\([{<arg>}+]\)[{<type>,argcDecl>;}+]\{
     <calledFcnName> \([{,param>}+}\)
     @write ("calls", fcnName, calledFcnName) @
```

on Kernighan and Ritchie style C source code [Kernig78], will find all function calls and writes a tuple for each one. A call graph extractor can be written for LSME in 30 lines, as shown in Figure 6.3. While LSME's lexical approach is responsible for both its main advantage and disadvantage. The tool may not accurately find all matches, but it is portable across languages. The LSME system has been used to extract source models from C, C++, CLOS, Eiffel, and TCL source code.

```
1  comment /* */
2
3  [<type>]<fn>
4   @ if kywdq(fn) | opq(fn) then fail @
5   \( [{<param>}+] \) [{ {<atype>}+ ; }+} \{
6
7     <cn>
8     @ if kywdq(cn) | opq(cn) then fail @
9     \( [ { <arg> [ , ] }+ ] \)
10   @ writeCall (fn , cn) @
11
12 procedure writeCall(fn, cf)
13   static idch
14   initial idch := (&ucase ++ &lcase ++ & digits ++ '_')
15
16   realfn := (fn ? (tab(upto(idch)), tab(0)))
17   realcf := (cf ? (tab(upto(idch)), tab(0)))
18   return write(realfn, " ", realcf)
19 end
20
21 # true if a keyword
22 procedure kywdq(nm)
23   return nm == ("if" | "while" | "Switch" | "for" | "typedef")
24 end
25
26 #true if an operator (approximate)
27 procedure opq(nm)
28   return any('?;:+-*/%!&=|<>', nm) &
29   (*nm ==1 | *nm == 2 & any('+=*/%&=|<>', nm[2])))
30 end
```

Figure 6.2: Call Graph Extractor for C in LSME [Griswo96]

There is another awk variant, TAWK, that takes an approach similar to LSME. There are some technical differences, but the key distinction is that it is language-dependent [Griswo96].

6.9 Comparison of Tools

Although the tools have conceptual differences in how they approach the problem of manipulating source code, it is possible to compare their features. Table 6.1 summarizes the features offered by the tools. The basic differences are that `grep` and its variants are simple UNIX utilities that behave as filters on an input stream, while `tksee`, SCRUPLE, and LSME were designed to analyze or search source code. This is particularly evident in the last two tools in the specialized query specification languages used.

Tool Name	grep	cgrep	sgrep	agrep	tksee	SCRUPLE	LSME
Tool Type							
Searching	X	X	X	X	X	X	
Analysis						X	X
User Interface							
Command line	X	X	X	X		X	X
GUI			X		X	X	
Approximate matching	*	*	*	X	*	X	†
Semantic searches					X	X	
Source language independent	X	X	X	X			X
Requires factbase or parsing					X	X	
Query language	regular exp.	regular exp.	GCL-based	regular exp.-based	regular exp. and navigation	pattern language tailored to source language	own pattern language with ICON

* wildcard matching only
† only at the lexical level

Table 6.1: Comparison of Tool Characteristics

6.10 Lessons Learned

We made a series of design decisions for our own source code searching tool based on the user studies and tool analyses from this and earlier chapters. These decisions were influenced by our goals as described in Chapters 4 and 5, grep's strengths and limitations, and the various query mechanisms utilized by the tools.

- **Use an existing language to specify searches**

  Except for cgrep and agrep, all the tools examined used their own query or pattern language to specify searches. (In the case of tksee, complex searches are accessed through the GUI.) There are three lessons that can be learned. 1) In order to support

semantic or structural searches, the basic grep command syntax would not be sufficient; 2) The last thing the world needs is another query language. 3) Many of the languages used in the tools examined have well documented syntax and semantics; one has an algebraic basis. Reusing a query languages would take advantage of the work already done on its formal specification.

- **Start with a command-line search tool**

  Many of grep's strengths, such as ease of use, compatibility with operating environment, etc., arise from its command-line interface. A semantic grep that does not retain this aspect of the grep paradigm is unlikely to be adopted by software maintainers.

- **Add a graphical user interface later**

  Both SCRUPLE and sgrep started out as command-line tools and later a graphical user interface was grafted on top of them. This seems to be a reasonable course of action to follow, since the GUI can be added when the tool is integrated with the Software Bookshelf.

- **Maintain language-independence**

  A tool becomes much more powerful when it isn't tied to a particular programming language, as is evident in the grep family and LSME. Familiarity with a tool or pattern language becomes more valuable because of its portability. The PBS tools achieve language-independence by using a common factbase. A search tool that is part of PBS could do the same.

Further examination of these tools led to the selection of GCL (generalized concordance lists) as the query language for the search tool. The syntax and semantics of sgrep is based on GCL. There were three main reasons for this decision: it was designed to work with structured documents, of which program source is an example, and it has an algebraic basis for the grammar [Clarke95a]. Since GCL is a general-purpose query language, the search space need not be limited to source code. Written documentation, HTML pages, and any

other structured material can be searched. Later implementation of our source code searching tool could easily be adapted to include these documents as well. With this additional capability, PBS becomes more like a traditional information repository. The GCL language is described in the next chapter, along with a design of a the search tool `grug` (**gr**ep **u**sing **G**CL) and how it fits with PBS.

6.11 Summary

In this chapter, a number of searching and source code analysis tools were examined for their approaches to solving the problem of extracting semantic information from source code. The general-purpose search tools were investigated for their interface and the syntax used to specify searches. The analysis tools were considered for the mechanisms used to extract facts from source code. Based on observations of these tools, a number of design decision were made. An existing query language, GCL was chosen to specify patterns in the search tool `grug` (**gr**ep **u**sing **G**CL). Initially, `grug` will be a command-line tool and a graphical user interface will be added later. Finally, the design of `grug` should not restrict it to a specific programming language. In the next chapter, these design decisions are be applied to the specification of `grug` and Searchable Bookshelf.

Chapter 7: Design of grug

7.1 Overview

This chapter unifies the concepts on program comprehension and source code analysis presented in previous chapters. Ideas from this material are made concrete in the form of requirements and specification for a source code searching tool, grug (**grep using** GCL). The requirements outline the goals of the grug tool and can later be used as criteria for evaluating the success of the tool. The specification describes how the requirements can be fulfilled. To recapitulate, the purpose of grug is to support multiple comprehension strategies within the Portable Bookshelf by providing search capabilities. The search tool needs to be able to link high-level abstractions such as those in the Software Landscapes with source code and to search source code for semantic elements, such as function and variable definition and uses. By supporting both top-down and bottom-up formation of program concepts, grug with Software Landscapes allow users to use multiple program comprehension strategies during software maintenance.

This chapter describes the design of grug and can be divided into two major parts. The first half of the chapter is devoted to the requirements of grug and the second half is concerned with its specification. Various platform, functional, non-functional, and data requirements are discussed. The grug tool uses the GCL query language to specify searches, so a description of this language and the markup index and necessary macros are included in the specification. At the end of the chapter, a preliminary implementation of grug and its integration with PBS are described.

7.2 Platform Requirements

The development of grug shall take place within the UNIX environment, specifically SunOS 4.1.4. All the tools discussed in Chapter 6 and the existing PBS tools operate in this

environment. As a result, there is a great deal of expertise available on developing source analysis tools and program comprehension tools under the UNIX operating system. From these factors, we derive the following two platform requirements:

P1. Users shall be able to use grug from the command-line.

P2. Users shall be able to access grug from a web browser.

The first requirement is motivated the discussion of the strengths of grep in Section 6.2 Many of grep's positive attributes are due to it being a command-line utility, and these are attributes that we wish to build into grug. The second requirement is motivated by the need to integrate the resulting tool into the existing PBS.

7.3 Functional Requirements

Some of the functional requirements of grug, as listed below, are contradictory and may not be met with a single incarnation of the tool. For example, one of the requirements is to use only standard input and standard output for input and output, while another is to be able to click on the results of a search. It may be possible to fulfill all these requirements with implementation that accepts different invocations. To this end, the functional requirements have been put into four groups: 1) those that apply to all versions of grug; 2) those that apply to a command-line version of grug; 3) those that apply to a grug with a GUI; and 4) those that apply to a grug used over the World Wide Web. There is some overlap between the four groups. For example, the version of grug with a GUI will still need to fulfill some of the requirements of the text-only command-line version, such as maintaining all existing grep functionality. The groups of functionality will be discussed in order in Sections 7.3.1-7.3.4.

7.3.1 Basic grug Functionality

The functionality that must be present in all the incarnations of grug is presented in this section. These requirements could be considered the defining characteristics of grug as a semantic grep.

F1. Users shall be able to search for semantic elements in source code.

Users shall be able to search for semantic elements such as declaration, definition, and all uses of functions and variables, using `grug`. This functionality is central to the purpose of `grug` as a tool to support program comprehension.

F2. The functionality of `grug` shall be a superset of the functionality in `grep`.

Any new functionality should be added without taking away old functionality because users have come to trust and value `grep`'s capabilities. They will be more likely to adopt `grug` if they can still do the same searches as they did with `grep` as well as the new ones.

F3. The query language used to specify searches shall be programming language-independent.

Users should not have to learn a new syntax for each programming language. However, it may be necessary to learn new options to address language-specific elements such as classes in object-oriented languages or implementors in Smalltalk. By making `grug` programming language-independent, users can port their skills from one software system to another.

F4. The query language shall be independent of factbase schemata.

A corollary of making `grug` programming language independent, is to make `grug` independent of factbase schemata, i.e. it should be able to search for any element in the factbase. Searches should be driven by available information rather than by specification syntax of the search tool. The schema of the extracted factbase varies from system to system, and `grug` should be able to work with any properly formatted index. For example, a factbase for a software system written in C++ would include information in classes, while this would not be true for a software system written in C. Users should be able to use `grug` to search both software systems equally. Although `grug` does not restrict the contents of the factbase, it will have to be stored in a fixed format or syntax so that it can be read by `grug`.

F5. Matches returned by `grug` shall not be limited to a fixed unit or size.
The tool shall return matching units in sizes appropriate to the search, from an identifier to a file. The size of the hit returned should be driven by the query, not by some formatting constraint imposed by a search tool. Often a line does not provide enough context for a match. In many cases the desired match will be several lines or an entire function body. Matches in `grug` should not be limited to a specific unit or size, such as a single line.

7.3.2 Requirements for the command-line version

The command-line version of `grug` shall have the following requirement imposed upon it in addition to the ones outlined in the previous section.

C1. The `grug` tool shall retain as many `grep` options as possible.
The functionality that must be maintained includes the ability to match regular expression, and support for the existing command flags and arguments. Users must be able to leverage their existing knowledge when using `grug`. Minor differences in the user interface will unnecessarily increase the knowledge that a user needs to become an expert.

C2. The `grug` tool shall operate in the style of a UNIX utility.
It shall accept redirection of input and output. The additional functionality will be added through new command-line flags. This requirement further defines the `grep` functionality that must be retained.

7.3.3 Requirements for the Graphical User Interface Version

The `grug` search tool operating from within a GUI shall have the following requirements in addition to the ones presented in Section 7.3.1. The requirements in this section should guide the design of the interface, rather than dictate a particular implementation.

G1. Users shall specify searches inside a dialog box.
Searches shall be specified in dialog boxes, similar to the example presented in Figure 7.1 below. The left box will accept any basic `grep` search. In the centre box, the user can

specify the semantic element that she is searching for. The right box allows the user to specify the scope of the search: files, subsystems, components. The gray triangles denote the availability of a drop-down list from which the user can select an item. These lists shall be generated from the factbase schema of the software system. It should be noted that Figure 7.1 is an example only and other designs are possible.

Search for	of type	in
B*|G*	Procedure ▼	All subsystems ▼

Figure 7.1: Example of `grug` GUI Search Dialog Box

It is important to separate the standard `grep` functionality from the new functionality for a couple of reasons. Users can still use this tool as they would existing `grep` tools. Also, this separation serves to highlight the availability of additional features.

G2. The `grug` tool shall display let users access all matches simultaneously.

G3. Results shall be displayed in a drop-down window.

The user should be able to access the multiple search results from a single window, that is, the tool should not present the user with a sequence of individual matches. The results of the search should be presented in a window attached to the search dialog box, directly below it. Solutions to searches shall be displayed with some contextual information about the match. This may include the name of the file and the procedure or the subsystem in which the match was found. The matched string shall be highlighted, so it stands out from the rest of the text. After seeing the matches, the user shall be able to modify the search without re-typing all the information.

G4. The search results shall be navigable.

The results of the search should navigable, meaning users shall be able to access and edit the actual file containing the match. This linkage can be achieved using either the pointing device or some other interface mechanism. Clicking on the matched string will open the file containing the string in an user-specified editor. Clicking on the contextual information about the match shall display an appropriate landscape. The user shall also be able to define new searches using the search results.

G5. Users shall be able to save and playback searches and results.

The tool shall keep a history log of searches. The user can choose to save a set of searches to a file. When this file is loaded later, the user can playback a sequence of searches. The save searches can also include user-defined annotations. Replaying a sequence of searches can serve a number of different purposes. When doing a bug repair, a software maintainer can save the sequence of searches performed before the repair and replay them afterwards. This allows her to do a validation of the repair analogous to regression testing. In a different scenario, an "exploration sequence" could be defined for the purpose of teaching a new team member about the system. A new team member could replay these searches to "follow the footsteps" of senior maintainer giving a tour of the software system.

7.3.4 Requirements for Operating Across the World Wide Web

Since `grug` needs to be integrated with PBS, a version of it has to be accessible across the World Wide Web. In addition to the requirements in Sections 7.3.1 and 7.3.3, this version shall meet the requirements imposed in this section.

W1. The GUI shall be replicated across platforms.

The web-client must replicate the same layout and interaction as the GUI of the local `grug`. While the GUI need not be identical, users must be able to use the local version and remote versions on different platforms interchangeably.

7.4 Non-Functional Requirements

Non-functional requirements are concerned with *how* the tool operates, rather than *what* it does. In this section, some parameters on `grug` execution are stipulated.

NF1. The `grug` tool shall have a responsive start-up

Loading the GUI version of `grug`, either locally or over WWW, shall complete relatively quickly. In the local version, the time required to start-up the initial dialog box should not deter software maintainers from using the tool. A delay longer than a second would be too disruptive and discourage adoption of the tool. The WWW version shall have a similar load time, excluding the start-up of the browser. While this timing may be difficult to control due to erratic network delays, the web page should be designed with this in mind.

NF2. Compute time for queries should be short.

Queries should return relatively quickly. Mouse clicks, in particular, should return as close to instantaneously as possible. Users make queries when they are solving a problem, so delays in responses can cause them to lose a train of thought, or become impatient. An irritation such as this is sufficient to cause some users to avoid the tool, thereby making it a failure.

NF3. The `grug` tool shall be able to handle multiple users simultaneously

More than one user shall be able to use `grug` locally or over the WWW at the same time. An upper limit on the number of users for a Searchable Bookshelf is 20, approximately the same upper limit on a typical software maintenance team, however the system must continue to operate with more than 20 users. A subset of the team should be able to shall access `grug` concurrently with no appreciable degradation in performance.

NF4. The `grug` tool shall operate on source code and a markup index.

Since the intended users of `grug` are developers trying to performs maintenance tasks, they will likely have access to source code. The markup index will contain the file positions of relevant semantic elements. A pre-computed index should be used instead of run-time

parsing because this approach allows `grug` to be language independent—one of the basic requirements from section 7.3.1.

7.5 Specification of `grug`

So far in this chapter, we have focused on the requirements or the goals of the `grug` tool. In the remainder, of the chapter we give its specification, which describes how those goals should be met. The query syntax is central to the specification, since other parts of the design depend on this syntax. As mentioned in Chapter 6, the GCL query language was chosen to specify searches. Since source code is an example of structure text and GCL was designed to be a general-purpose query language for structured texts, GCL can be adapted easily for source code searches.

Following the explanation of GCL in Section 7.6, is an description in Section 7.7 of the markup schema and macros required for `grug`. As will become apparent in Section 7.6, a markup schema is required to support all the searches required by `grug` and macros will be necessary to simplify these searches. Following these two sections is a description of a preliminary implementation of `grug` and the Searchable Bookshelf. This work provides an opportunity to validate the design and obtain feedback from potential users of the final implementation.

7.6 The GCL Query Language

GCL is a query language for schema-independent retrieval from structured text, such as email, bibliographies, HTML pages, and source code. It has a formal definition [Clarke95a], and has been implemented as part of the project on Very Large Multi-User Multi-Server Text Databases [MultiT98]. GCL requires markup of the text at appropriate character locations to indicate the boundaries of various structural elements. For example, HTML tags could serve as the markup for a web document. Alternatively, an index of file locations of structural elements could serve as implicit markup of a document. The syntax of the GCL query language in Backus-Naur Form is included in Figure 7.2. Literal strings are defined as exact matches of strings and regular expressions following the POSIX standard [IEEE92].

```
statement ::=
        macro-definition
      | query

macro-definition ::=
        identifier = query
      | identifier ( parameters )  = query

query ::=
        query containing query
      | query contained in query
      | query not containing query
      | query not contained in query
      | quantity of ( queries )
      | one of ( queries )
      | all of ( queries )
      | query ...query
      | ( query )
      | quantity words
      | identifier ( queries )
      | identifier
      | quoted-string

queries ::= query | query , queries

parameters ::= identifier | identifier , parameters

quantity ::= positive-integer

quoted-string ::= single-quoted-string
      | double-quoted-string

single-quoted-string ::= regular expression

double-quoted-string ::= literal string
```

Figure 7.2: Syntax of the GCL Query Language in Backus-Naur Form

```
1  /* Search for a file named NAME trying various prefixes including the
2  user's -B prefix and some standard ones.
3  Return absolute file name found.  If nothing is found, return NAME.*/
4
5  static char *
6  find_file (name)
7       char *name;
8  {
9    char *newname;
10
11   /* Try multilib_dir if it is defined.  */
12   if (multilib_dir != NULL)
13     {
14       char *try;
15
16       try = (char *)alloca (strlen (multilib_dir) + strlen (name) + 2);
17       strcpy (try, multilib_dir);
18       strcat (try, dir_separator_str);
19       strcat (try, name);
20
21       newname = find_a_file (&startfile_prefixes, try, R_OK);
22
23
24       /* If we don't find it in the multi library dir, then fall
25          through and look for it in the normal places.  */
26        if (newname != NULL)
27          return newname;
28     }
29
30   newname = find_a_file (&startfile_prefixes, name, R_OK);
31   return newname ? newname : name;
32 }
```

Figure 7.3: Code Sample from `gcc.c` of the GNU C Compiler R2.7.2

In this section, we give a brief description of the GCL query language, since subsequent portions of the `grug` specification depend on GCL syntax and semantics. More detailed descriptions of the the GCL syntax can be found in other publications [Clarke95a, Clarke95b]

.

The GCL query language is explained using the code sample shown in Figure 7.3. The function "`find_file`" was taken from the file `gcc.c` of version 2.7.2 of GCC, and line numbers have been added for discussion purposes. The examples in this thesis are based on C source code, because C is a well-known programming language, but GCL would work with any programming language. In the example queries to follow, the search space is limited to the source in Figure 7.3. Furthermore, line 21 of the code sample:

```
21       newname = find_a_file (&startfile_prefixes, try, R_OK);,
```

will be taken and further annotated so that additional features of the GCL query language can be illustrated.

In Figure 7.4, line 21 is shown with each character labeled starting with a hypothetical database position (11). To show more than basic matches of literal strings and regular expressions, it is necessary to add markup to this line. The specific markup generated would be determined by the analysis to be performed. The type and format of the tags used in Figure 7.5 were chosen to illustrate the GCL syntax. The markup also need not be implicit, that is stored in a separate index file. An explicit markup could be used with the tags embedded in the document, as is the case with HTML files.

·	·	·	·	·	·	n	e	w	n	a	m	e	·	=	·	f	i	n	d	_	f	i	l	e
11	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**39**	**30**	**31**	**32**	**33**	**34**	**35**
·	(	&	s	t	a	r	t	f	I	l	e	_	p	r	e	f	i	x	e	s	,	·	t	r
36	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**48**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**
y	,	·	R	_	O	K	)	;	\n															
61	**62**	**63**	**64**	**65**	**66**	**67**	**68**	**69**	**70**															

Figure 7.4: Line 21 of Code Sample with Hypothetical File Positions Labeled

In GCL, the "solution" to a query is a set of "extents" or "regions" in the text. Consequently, the markup index consists of a series of start positions and end positions of regions. In a solution, the extents may overlap but they may not nest. This constraint will become clearer as GCL queries and operators are discussed. In Figure 7.5 the pair of columns on the right show a set of start markers and the pair to columns on the left show the corresponding end markers.

Although the markup in Figure 7.5 is arbitrary, it does have a rationale. The `<line>` and `</line>` are included because the GCL matching algorithm does not use the newline character as a boundary between records and as a result the entire file is treated as a character stream. If line-based matching is desired, markup is necessary.

Syntactically, line 21 shows a variable being assigned the return value of a function call. Consequently, the syntactic elements of interest are included in the markup. The tags

`<varref> </varref>` denote a reference or access of a variable and `<var> </var>` denote the variable name. The pair `<fcncall>` and `</fcncall>` indicate a function call and `<fcn> </fcn>` indicate the function name. The remaining tags markup the argument list for the function call and the individual arguments in the list.

Tag	File Position	Tag	File Position
<line>	11	</line>	70
<varref>	17	</varref>	70
<var>	17	</var>	23
<fcncall>	27	</fcncall>	68
<fcn>	27	</fcn>	35
<arglist>	37	</arglist>	68
<arg>	38, 59, 64	</arg>	56, 61, 67

Figure 7.5: Markup Index for Line 21 of Code Sample

The simplest query is a literal string or a regular expression. The query

```
"file name found"
```

returns all three word phrases that match the string exactly. In the example, there is only one match, which is located on line 3. The query

```
'str*'
```

returns all the strings that match the regular expression. Solutions from the code sample in Figure 7.3 are `strlen` on line 16 (twice), `strcpy` on line 17, and `strcat` on lines 18 and 19. String matching will span across lines and ignore markup, unless otherwise specified. Quoted string queries can be combined using the GCL operators.

The eight GCL operators fall into three categories: ordering, combination, and containment. The ordering operator "..." is used to link textual elements. For example, the query

```
'str*' ... "multilib_dir"
```

would return:

```
strlen (multilib_dir
```

from line 16 and

```
strcpy (try, multilib_dir
```

from line 17. Markup can also be used with the ordering operator. A search of

"`<fcncall>`" ... "`</fcncall>`"

on line 21 in Figure 7.4, would return

```
find_a_file (&startfile_prefixes, try, R_OK).
```

The combination operators have the basic form "*quantity* `of ( ` *list of queries* `)`". A solution covers the solutions to a specified number of queries in the associated list. *Quantity* is normally a positive integer less than or equal to the length of the query list. A solution must begin and end with a solution to one of the queries in the list. The query

```
2 of ('str*', "name", "multilib_dir")
```

would return the following solution from lines 7-11:

```
name;
{
    char *newname;

  /* Try multilib_dir
```

three solutions from line 16:

```
try = (char *)alloca (strlen
strlen(multilib_dir ,
multilib_dir) + strlen(,
```

the following solution from lines 16-17:

```
name) + 2);
strcpy
```

and the following solutions:

`strcpy(try, multilib_dir` from line 17, and
`strcat (try, name` from line 19.

Note that the last solution demonstrates the rule that solutions can overlap but not nest. The string

```
strcat(try, dir_separator_str); strcat (try, name
```

begins with `'str*'` and ends with `"name"`, but is not a valid solution because it contains a solution, the one that was reported as the last item on the list above.

The containment operators, "`containing`", "`contained in`", "`not containing`", and "`not contained in`" are used to search for structural relationships. The query

```
(2 of ('str*', "name", "multilib_dir")) containing "try"
```

would have the solution :

```
find_a_file (&startfile_prefixes, try, R_OK).
```

Or if more context was desired,

```
("{"..."}") containing (<fcncall> ... </fcncall>) containing "try"
```

could be used and it would return the expression block on lines 12-28.

Macros can be used to simplify common queries and to make complex queries more readable. They are often saved in resource files at the system level and at the user level. The following macros can be defined for the markup shown in Figure 7.5.

```
LINE = <line> ...</line>
VAR = <var> ... </var>
VARREF = <varref> ... </varref>
FCN = <fcn> ... </fcn>
FCNCALL = <fcncall> ... </fcncall>
ARGLIST = <arglist> ... </arglist>
ARG = <arg> ... </arg>
```

The query `LINE` would return every line in the database. The subsequent macros would return all instances of a particular semantic element. Macros can be used with other operators, and can themselves be used in macros. Parameterized macros, such as the ones below, provide additional expressive power for chunking concepts together. The parameterized `VARREF` macro returns all uses of the variable "name". Similarly, the parameterized `FCNCALL` macro returns all uses of the function "name". The `GREP` macro performs regular expression matching on a line-by-line basis—in the same manner as the UNIX utility.

```
 VARREF( name ) = VARREF containing VAR containing name
FCNCALL( name ) = FCNCALL containing FCN containing name
         GREP( expr ) = LINE containing 'expr'
```

The markup and macros listed so far have for a single line. Markup can easily span across lines. The tags `<fcndef>` and `</fcndef>` can be added to the beginning and the end of the entire function declaration of `find_file`. Markup could also be added for the name of the function, its return type, its formal parameters, etc. With sufficient markup, powerful

searches can be performed. For example, with appropriate markup, the search "List all the names of all the functions that are called by find_file" could be expressed as

```
FCN contained in (FCNDEF containing (FCNNAME containing
                    "find_file")).
```

Furthermore, if we include files and their names as a unit of analysis and define a subsystem to be a set of files, we could make queries about the structure of a software system.

```
FILE = <file> … </file>
FILENAME = <filename> … </filename>
CODEGEN.SS = FILE containing (FILENAME containing one of
(  "codegen.c", "codegen.h", "parsecodegen.h"))
```

Since GCL treats input as a stream of characters, the `FILE` tags are necessary to indicate the start and end of a file. The `FILENAME` tags are used to embed the filename in the character stream. A subsystem, such as `CODEGEN.SS`, would consist of a set of files. With these additional markup and macros, a query such as, "What are the names of all the functions defined in the codegen subsystem?" could be expressed as

```
FCNNAME contained in FCNDEF contained in CODEGEN.
```

From the examples in this section, it is clear that careful selection of the markup schema and the macros defined impact significantly on the types of searches the can be performed. These parts of the `grug` specification will be discussed in the next section.

7.7 Markup Schema and Macros for grug

Markup schema for source code can be separated into two groups: those that require parsing and those that do not. Schemata that do not require parsing identify elements on the basis of lexical items, such as special symbols and lexical tokens. These approaches can be quite powerful, but are susceptible to errors such as false hits and misses, as is the case with LSME. A markup requiring parsing is necessary to fulfill all the requirements for `grug`. Before describing this schema that will be used, other approaches that do not employ parsing will be discussed to illustrate some of their shortcomings.

A lexical approach to markup could be used if only structural searches are desired, like those performed by `sgrep`. The macro definitions from the discussion on `sgrep` in the Section 6.4 have been translated into GCL macros below.

```
COMMENT = ( "/*" ... "*/" )
BLOCK = "{" ... "}"
PPLINE = ( '^#' ... '$')
IF_COND = "if" not contained in ( one of (COMMENT,
     PPLINE) )... ("(" ... ")" )
```

A C-style comment could be defined as a region of text that starts with the `/*` symbol and ends with the `*/` symbol. A `BLOCK` is defined as a region beginning with `{` and ending with `}`. A `PPLINE` (pre-processor line) is the entire line beginning with "#". An if-condition (`IF_COND`) is an `if` not within a comment or pre-processor line followed by a pair of brackets.

These macros provide only approximate matching since they operated only on lexical elements. The `COMMENT` and `BLOCK` macros would under-report true matches, whereas the `PPLINE` and `IF_COND` macros would over-report them. Although comments are not allowed to nest, they may contain additional start markers `/*`. In such cases, a match would be the text beginning with the last start marker up to the close marker rather than the entire comment. Blocks have a similar problem. Because blocks of expressions are allowed to nest, only the innermost blocks would be reported. The `PPLINE` macro could report lines beginning with # inside comments and the `IF_COND` macro could report if-conditions inside quoted strings. From these examples, we can see structural approaches do not result in highly precise matching, so parsing would be required to identify accurately semantic elements of interest.

Schemata that require parsing can have semantic elements, such as variable names and function definitions, in the markup index. Parsing is necessary to obtain the markup to meet the requirements for `grug` as set out in Section 7.3. In the remainder of this section, the markup schema and macros for `grug` will be described. It should be noted that the schema given here is one of many possibilities. Other schema are possible and can be developed according to the needs of the target application. In the schema presented here, there are a

number of omissions such as user-defined types, macros, if-conditions, etc. The schema chosen was based on the results of the study reported in Chapter 5. Programmers reported that the most common search targets were function definitions, all uses of a function, all uses of a variable, and variable definitions. The schema focuses on these elements, but could easily be extended to include others. The purpose of the macros, as illustrated in the previous section, is to simplify common searches and to make complex searches more readable.

Figure 7.6 is an example of a function declaration with the character positions labeled. Normally, character positions are numbered from the beginning of the database, but in this and subsequent examples character positions are numbered from the beginning of the line for simplicity. A function declaration is the declaration of the function type without the body, such as those found in headers and forward declarations. Table 7.1 shows the markup schema and macros for a function declaration. The columns show, in order, the element being indexed, the start tag, the character position for the start tag in Figure 7.6, the end tag, the character position for the end tag in the example, and the macro for the element. All other tables of markup and macros in this chapter use the same column organization.

i	n	t	·	a	d	d	·	(	i	n	t	·	o	p	1	,	·	i	n	t	·	o	p	2
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
)	;	\n																						
25	**26**	**27**																						

Figure 7.6: Example of Function Declaration

Semantic Element	Start Tag		End Tag		Macro
declaration	<fcndcl>	0	</fcndcl>	25	FCNDCL
name	<fcndclname>	4	</fcndclname>	6	FCNDCLNAME
return type	<fcndclret>	0	</fcndclret>	2	FCNDCLRET
parameter list	<fcndclprmlis>	8	</fcndclprmlis>	25	FCNDCLPRMLIS
parameter	<fcndclprm>	9, 18	</fcndclprm>	15, 24	FCNDCLPARM

Table 7.1: Markup and Macros for Function Declarations

An example of a function definition with the character positions labeled is given in Figure 7.7. A function definition is the section of code that implements the function or attaches a body to a signature. Table 7.2 gives the markup and macros for function definitions.

i	n	t	·	a	d	d	·	(	i	n	t	·	o	p	1	,	·	i	n	t	·	o	p	2
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
)	\n	{	\n	\t	r	e	t	u	r	n	·	o	p	1	·	+	·	o	p	2	;	\n		}
25	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**

Figure 7.7: Example of Function Definition

Semantic Element	Start Tag		End Tag		Macro
definition	<fcndef>	0	</fcndef>	49	FCNDEF
name	<fcndefname>	4	</fcndefname>	6	FCNDEFNAME
return type	<fcndefret>	0	</fcndefret>	2	FCNDEFRET
parameter list	<fcndefprmlis>	8	</fcndefprmlis>	25	FCNDEFPRMLIS
parameter	<fcndefprm>	9, 18	</fcndefprm>	15, 24	FCNDEFPARM

Table 7.2: Markup and Macros for Function Definitions

A function call occurs when a function passes execution to another section of code such as a user-defined or library function. An example of a call is in Figure 7.8 and the markup and macros are given in Table 7.3.

\t	a	d	d	·	(	l	e	f	t	,	·	r	i	g	h	t	)	;	\n
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**

Figure 7.8: Example of Function Call

Semantic Element	Start Tag		End Tag		Macro
call	<fcncall>	1	</fcncall>	17	FCNCALL
name	<fcncallname>	1	</fcncallname>	3	FCNCALLNAME
argument list	<fcncallarglis>	5	</fcncallprmlis>	17	FCNCALLARGLIS
argument	<fcncallarg>	6, 12	</fcncallarg>	9, 16	FCNCALLARG

Table 7.3: Markup and Macros of Function Calls

Figure 7.9 is a variable declaration with labeled character positions. Figure 7.10 is a variable definition with labeled character positions. The two figures appear to be identical, but they differ in that the compiler allocates memory for definitions, but not for declarations. A

declaration such as the one in Figure 7.9 would typically appear in a header file or in a source file preceded by the keyword "extern". The markup and macros for declarations and definitions are given in Table 7.4 and Table 7.5, respectively.

i	n	t	·	c	o	u	n	t	e	r	;	\n
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

Figure 7.9: Example of Variable Declaration

Semantic Element	Start Tag		End Tag		Macro
declaration	<vardcl>	0	</vardcl>	12	VARDCL
name	<vardclname>	4	</vardclname>	10	VARDCLNAME
type	<vardcltyp>	0	</vardcltyp>	2	VARDCLTYP

Table 7.4: Markup and Macros for Variable Declarations

i	n	t	·	c	o	u	n	t	e	r	;	\n
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

Figure 7.10: Example of Variable Definition

Semantic Element	Start Tag		End Tag		Macro
definition	<vardef>	0	</vardef>	25	VARDEF
name	<vardefname>	4	</vardefname>	10	VARDEFNAME
type	<vardeftyp>	0	</vardeftyp>	2	VARDEFTYP

Table 7.5: Markup and Macros for Variable Definitions

A variable reference occurs either when a variable is read or has a value assigned to it. Figure 7.11 is an example of a variable being assigned a value, and Table 7.6 gives the markup and macros for variable references.

\t	c	o	u	n	t	e	r	·	=	·	0	;	\n
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**

Figure 7.11: Example of Variable Reference

Semantic Element	Start Tag		End Tag		Macro
reference	<varref>	1	</varref>	11	VARREF
name	<varrefname>	1	</varrefname>	7	VARREFNAME

Table 7.6: Markup and Macros for Variable References

Table 7.7 shows the markup and macros for structural elements. Lines and blocks are units that are smaller than a file. Files, modules, and subsystems are parts of a software system. The line is included as a structural element because it is a construct that pervades UNIX

utilities [Pike87]. The rationale for including a block element is that it is a unit that provides more context than a line, but usually less than a function.

Structural Element	Start Tag		End Tag		Macro
line	<line>		</line>		LINE
block	<block>		</block>		BLOCK
file	<file>		</file>		FILE
module					*.MOD
subsystem					*.SS

Table 7.7: Markup and Macros for Structural References

Modules and subsystems are defined using macros that allow the grouping of files into organizational units. They are denoted by the `.MOD` and `.SS` extensions. The modules and subsystems of a software system cannot be determined using a parser alone, and so there are no tags for them. They are usually defined by users or recovered using a reverse engineering tool such as `grok`. This information is used to define the modules and subsystems as macros.

7.8 Preliminary Implementation of `grug` and the Searchable PBS

After developing the requirements and specification for a tool, the natural next step is to implement it. After some initial explorations, it was determined that an implementation of `grug` as designed would be beyond the scope of this thesis. However, the construction of a prototype would provide valuable feedback on the design and experience on building a source code searching tool. As a result, we chose to build a preliminary implementation of `grug` that made use of as many existing components as possible. We used existing PBS tools and factbases, a regular expression matching library (GNU regex 0.12), and a GCL parsing module from the Multi-Text project [MultiT98]. The work performed to construct `grug` consisted of making minor modifications to the PBS tools used to construct factbases from C source code, a component to work with the GCL module, and an interface between the GCL module and the factbase.

Some modifications were made to both the PBS tools and to the concepts and syntax of GCL, so that PBS parsers and factbases could be used. There was a mismatch between the

information contained in the factbases and the information required by GCL. The easiest way to resolve this incompatibility was to make minor changes to both components. Some location information was added to the factbase and the GCL syntax was extended so searches could be based on line numbers rather than character positions. This change had a major impact on the syntax and expressive power of the query language, as is explained in the remainder of this section.

```
funcdef    file      function
funcdcl    file      function
funcdcl    file      libraryFunction
vardef     file      variable
vardcl     file      variable
include    file      file
call       function  function
ref        function  variable

function {
     defloc
     deflocend
     dclloc
}

variable {
     vardefloc
     vardclloc
}
```

Figure 7.12: Factbase Schema for GCL Index in TA Format

7.8.1 Expanding the Factbase

In existing PBS factbases, a function would have attributes for a definition location and a declaration location:

```
add {
     defloc = utils.c:234
     dclloc = utils.h:57
}
```

In order to generate solutions to a query, GCL needs to know the beginning and the end of a region. In the case of function definitions, it needs to know not only where the definition started, but also where it ended. Some modifications were made to the utilities for generating

a factbase from C source code, `cfx` and `fbgen`, so that they produced end locations as well. Figure 7.12 is the schema for the GCL index in TA language. The `defloc`, `deflocend`, `dclloc`, `vardefloc`, and `vardclloc` attributes are set to a file and line number, as in the example above. Declarations and variable definitions are expected to span only a single line, so "end" location attributes were not added to these.

7.8.2 Augmenting GCL

The other mismatch is between the address space of locations PBS factbases and the address space of GCL. The factbase stored locations in terms of line numbers within a file, whereas GCL requires locations to be specified in terms of file positions. The address space mismatch was resolved by letting GCL match on a line-by-line basis and extending the query syntax so it could match selected items that were smaller than a line.

Figure 7.13 shows the extended GCL syntax, with the new items denoted by asterixes. The *:keyword* **(** *quoted-string* **)** syntax allows searches for function declarations, function definitions, function calls, variable declarations, variable definitions, and variable references;, with the identifier as denoted by the quoted string. This syntax permits searches that distinguish, for instance between occurrences of the string "`count`" and instances of a variable "`count`." Some examples of how this syntax is used are given in the next section.

7.8.3 Using grug

In the prototype of `grug`, users can search for literal strings, regular expressions, and six semantic elements. Queries can also be combined using the eight GCL operators. A filed called "TAfile" containing the factbase should be located in the same directory as the files being searched. The command syntax is similar to one used by `grep`, which is

```
grug [options] query <files>.
```

The four options, `-h`, `-i`, `-t`, and `-f`, are summarized in Table 7.8.

```
statement ::=
        macro-definition
      | query

macro-definition ::=
        identifier = query
      | identifier ( parameters )  = query

query ::=
        query containing query
      | query contained in query
      | query not containing query
      | query not contained in query
      | quantity of ( queries )
      | one of ( queries )
      | all of ( queries )
      | query ...query
      | ( query )
      | quantity words
      | identifier ( queries )
      | identifier
*     | :keyword ( quoted-string )
      | quoted-string

queries ::= query | query , queries

parameters ::= identifier | identifier , parameters

quantity ::= positive-integer

* keyword ::= fcndef | fcndcl | fcncall |
              vardef | vardcl | varref

quoted-string ::= single-quoted-string
      | double-quoted-string

single-quoted-string ::= regular expression

double-quoted-string ::= literal string
```

Figure 7.13: The Extended GCL Syntax

Option	Result
-h	Help information is displayed.
-i	Case-insensitive is performed.
-t <filename>	Instructs grug to use factbase found in <filename>, rather than the default.
-f <filename>	Instructs grug to read queries from <filename>, rather than the command line.

Table 7.8: Available Options in grug

A typical grep search such as "find all instances of the regular expression fprintf and sprintf in stub.c" can be invoked as:

```
grug "'[fs]printf'" stub.c.
```

Notice that GCL requires regular expressions to be placed in single quotation marks. In order for these to be passed from the command line correctly, the query needs to be placed inside double quotation marks. Similarly, strings are denoted by double quotation marks, and in order for these queries to be passed from the command line, they need to be placed inside single quotation marks. This problem can be circumvented by writing these queries to a file and using the -f option. Other examples of grug invocations are given below.

The extended syntax is used when searching for semantic elements. The search 'find the definition of the function "Display_List" in .c files' is invoked as:

```
grug ':fcndef("Display_List")' *.c.
```

A search to find where the variable "memory" is defined" is performed by issuing the command:

```
grug ':vardef("memory")' *.[ch].
```

7.8.4 The Searchable Bookshelf

This subsection shows how the Searchable Bookshelf was built with grug. In Figure 8.4, the column along the left side contains the table of contents of the books of a subject system in PBS. The landscape diagram is found in the large window in the right. In the small window beneath is an HTML form that can be used to perform grug searches. It

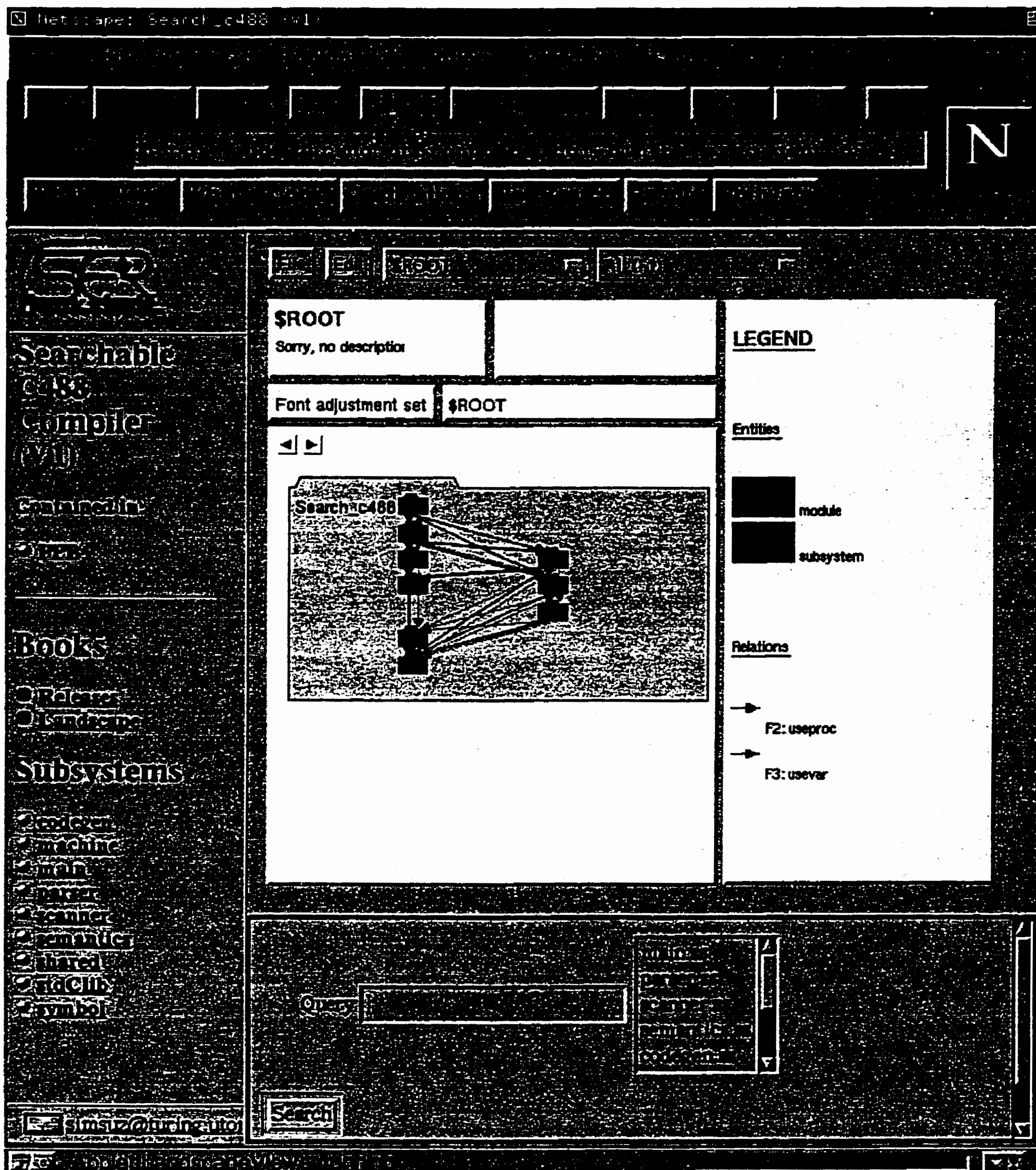

Figure 7.14: Screen Capture of The Searchable Bookshelf

consists of a text box to enter the query, a scrolling selection box from which to choose the search targets, and a set of check boxes to activate various options. A single Perl script, `bs_search`, is used to generate the form and process queries. The query is written to a file and passed to `grug` using the `-f` option. Because the query never passes through the command line, the problems with quotation marks at the shell level are circumvented, and the query can be entered literally.

The list of search targets can contain subsystems, modules, or files, depending on the landscape displayed. The landscape in Figure 7.14 is of the C488 compiler which consists of eight subsystems. These eight are reflected in the list of search targets in the query window. The `bs_search` script generates this list using the same tuple file as `lsview`, the existing Java applet that displays the Software Landscapes. Using the form, queries can be made about any or all of the subsystems.

A search is invoked by typing a query into the textbox, selecting the subsystems to be searched from the scrolling menu, and pressing the "Search" button. For example, if a user wanted to search for all the variables that were defined in the `main.ss` subsystem, she would type `:vardef('*')` in the query box, select main.ss from the scrolling menu, and click on "Search". This search would return the following:

```
main.c:12: static char mainVersion[] =
```

This result indicates that there is only one variable defined in the `main.ss` subsystem that is used by another file, and this variable is defined on line 12 of the file `main.c`. Recall from Chapter 2 that the factbase only has information on functions and variables with uses that cross file boundaries, i.e. they are called or referenced by a file other than the one in which they were defined. Functions and variables that are used only within the file in which they are defined are not included in the factbase.

7.9 Summary

The requirements and specification for `grug` were presented in this chapter. In basic form, `grug` is a command-line utility that is capable of `grep`-style searches as well as searches for

semantic elements in source code. The GCL query language is used to specify search targets which allows `grug` to be language-independent, factbase schema-independent, and to return solutions of arbitrary size. A markup schema for the GCL markup index is presented along with a set of macros to simplify search specification.

This search tool, when integrated with PBS, results in the Searchable Bookshelf, a program comprehension tool that supports integrated comprehension strategies. Working alone, `grug` supports bottom-up comprehension strategies, in which source lines are amalgamated into semantic chunks. Software Landscapes are capable of supporting top-down strategies. The two tools taken together are capable of supporting integrated strategies, in which software maintainers use multiple approaches and switch freely between them. Prototypes of `grug` and Searchable Bookshelf were described at the end of this chapter. Although they implement a subset of the functionality discussed in the design, they serve as a proof of concept and as a basis for validating the design. In future, we plan to implement versions of `grug` and the Searchable Bookshelf with the complete set of functionality described in this chapter. These tools can serve as the basis of further studies of program comprehension strategies employed during software maintenance.

Chapter 8: Conclusion

8.1 Observations

Throughout this thesis we learned many lessons about the code comprehension process during software maintenance. These lessons in turn influenced our design of `grug` and the Searchable Bookshelf. We summarize the observations made during this process here.

From the user studies, we learned that software maintainers are **task-oriented problem solvers**. They acquire knowledge to complete a specific task because it would be too difficult and time-consuming to learn about the entire system for its own sake. During problem solving, software maintainers construct mental models of the software system by using **integrated code comprehension strategies**. When looking at source code they employ a bottom-up strategy, seeking to relate lines of text to abstract concepts. When looking at Software Landscapes, they employ a top-down strategy, seeking to relate pictorial elements to code artifacts. Maintainers also switch freely between different strategies when gathering information from a single source and when synthesizing information from multiple sources.

Since software maintainers are task-oriented, their data acquisition process is guided by questions. They **ask questions** about the system and they **search to find the answers**. Based on these observations a **search tool was designed to support multiple code comprehension strategies**. The `grug` utility supports bottom-up strategies by allowing users to perform "semantic `grep`'s". Using the tool, they can search for semantic units in the source code, such as functions and variables. By integrating `grug` into PBS, the Searchable Bookshelf was created to support both top-down and bottom-up strategies. Using the Searchable Bookshelf, users can use Software Landscapes and `grug` simultaneously and switch freely between the two tools to build a mental model of the software system.

A **prototype of `grug` and the Searchable Bookshelf** was developed as a proof of concept. This prototype serves as a test of the principles and concepts laid out in the design. Although these tools have limited functionality, the experience of constructing these implementations is valuable in the development of source code searching and analysis tools. In the next section, the future work for the tools as well as user studies are described.

8.2 Future Work

As is typical of empirical research in software engineering, this thesis has identified more questions than it answered. The directions for future work discussed in this section are divided into four areas: usability testing, organizational studies, source code searching, and tool implementation. Any one of the questions raised could be the basis for a significant sequence of research.

8.2.1 User Testing

In the next iteration of the spiral model of development, the design and prototype of `grug` and the Searchable Bookshelf need to be validated. User tests need to be performed with software maintainers to determine whether `grug` meets their information needs. The results from these studies could be fed back into the development of a `grug` as a program comprehension tool.

8.2.2 Organizational Studies

The studies of software immigrants and project veterans have highlighted an area that has been largely unexamined by software engineering research. The patterns from the software immigrants study need to be validated by studying the naturalization process in other organizations to determine whether they can be generalized. This knowledge would valuable because the purpose of program comprehension tools is to assist users in understanding a software system; a tool needs to fit with how newcomers naturalize to be successful.

While software immigrants have been little studied as users of PBS, project veterans have been studied even less. The informal investigation performed in this thesis indicates that the strategies they use and the questions they ask can be quite different from those of software immigrants.

8.2.3 Source Code Searching

The study undertaken illuminated two lines of investigation, one having to do with research methodology and the other with the models created. A survey was used to collect the data, and a significant part of making a survey rigorous is the sampling technique. A very weak sampling technique, convenience sampling, was used because not enough was known about the characteristics of the population of software maintainers to create a representative sampling frame. Knowledge about the size of the software maintenance population, the amount of source code they support, the types of applications they support, and the programming languages they use could be valuable for guiding software engineering research. For example, it is not known on what platform the most problematic legacy systems reside and what programming language they are written in, yet most research into software tools use the UNIX environment and work with source code written in C.

The source code searching survey resulted in a series of archetypal searches to guide tool design. This model of searching needs to be validated and quantified, that is, it needs to be tested to determine its accuracy and the relative frequency of the searches. This could be done using either protocol analysis or another survey.

8.2.4 Tool Implementation

With respect to `grug` and the Searchable Bookshelf prototypes, the most obvious improvement would be to construct the character-based markup index for `grug`. This index would be built using a parser to generate a factbase for a software system with the schema from Chapter 7. With this index it would be possible to use the full functionality of GCL and eliminate the awkward keyword syntax. It would also facilitate further user tests to evaluate the utility of `grug` and the Searchable Bookshelf.

On a different level, the application of GCL to source code suggests the application of an information retrieval approach to software searching and analysis. It would be possible to create PBS factbases by making GCL queries using the character-based markup index. Information retrieval techniques could be applied to assist software maintenance tasks. Common operations that are candidates for moving into a utility function could be identified by making a query such as "find all regions of five or more lines that are identical". A weighting mechanism similar to those used by World Wide Web search engines could be applied to solutions returned by `grug` to make it easier to identify starting points for further investigation.

References

Aho79

A.V. Aho, B.W. Kernighan, and P.J. Weinberger. Awk – A Pattern Scanning and Processing Language. *Software Practice and Experience*, Vol. 9, No. 9, pages 267-280, 1979.

Berlin93

L.M. Berlin. Beyond Program Understanding: A Look at Programming Expertise in Industry. *Empirical Studies of Programmers, Fifth Workshop*, pages 6-25, Palo Alto, USA., 1993.

Boehm88

B. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, pages 61-72, (May, 1988).

Brooks83

R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, Vol. 18, page 543-554, 1983.

Brooks95

F.P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition.* Addison-Wesley, 1995.

Chen90

Y. Chen, M.Y. Nishimoto, and C.V. Ramoorthy. "The C Information Abstraction System." *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, pages 325-334, (March, 1990).

Clarke95a

C.L.A. Clarke, G.V. Cormack, and F.J. Burkowski. An Algebra for Structured Text Search and a Framework for its Implementation. *The Computer Journal*, Vol. 38, No. 1, pages 43-56, 1995.

Clarke95b

C.L.A. Clarke, G.V. Cormack, and F.J. Burkowski. Schema-Independent Retrieval from Heterogeneous Structured Text. *Fourth Annual Symposium on Document Analysis and Information Retrieval*, pages 279-289, Las Vegas, USA, 1995.

Clarke96

C.L.A. Clarke and G.V. Cormack. Context grep. Technical Report CS-96-41, Department of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1.

Clarke97

C.L.A.Clarke and G.V. Cormack. On the Use of Regular Expressions for Searching Text. *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 3, pages 413-426, (May, 1997).

DeMarc87

T. DeMarco and T. Lister. *Peopleware: Productive Projects and Teams.* Dorset House Publishing, 1987.

DeVaus96

D.A. deVaus. *Surveys in Social Research, Fourth Edition.* UCL Press, 1996.

Eisenh89

K. M. Eisenhardt. Building Theories from Case Study Research. *Academy of Management Review*, Vol. 14, No. 4, pages 532-550, 1989.

Eisens97

M. Eisenstadt. My Hairiest Bug War Stories. *Communications of the ACM*, Vol. 40, No. 4, pages 30-37, (April, 1997).

Farman97

G. Farmaner. *Setting Up a Portable Bookshelf.* Available at <http://www.turing.utoronto.ca/~bookshelf/docs/bookshelf.html>

Fay85

S.D. Fay and D.F. Holmes. Help! I Have to Update an Undocumented Program. *IEEE Conference on Software Maintenance*, pages 194-202, Washington DC, USA, 1985.

Finnig97

P. Finnigan, R. Holt, I Kalas, S. Kerr, K Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf. *IBM Systems Journal*, Vol. 36, No. 4, pages 564-593, (November, 1997).

Foddy93

W. Foddy *Constructing Questions for Interviews and Questionnaires: Theory and Practice in Social Research.* Cambridge University Press, 1993.

Frye57

N. Frye. *Anatomy of Criticism: Four Essays.* Princeton University Press, Princeton, 1957.

Griswo96

W.G. Griswold, D.C. Atkinson, and C. McCurdy. Fast, Flexible, Syntactic Pattern Matching and Processing. *Proceeedings of the IEEE 1996 Workshop on Program Comprehension*, Berlin, Germany, 1996.

Harel88

D. Harel. On Visual Formalisms. *Communications of the ACM*, Vol. 31, No. 5, pages 514-530, (May, 1988).

Holt97

R.C. Holt *Introduction to TA: Tuple Attribute Language.* Available at <http://www.turing.utoronto.ca/~holt/papers/ta.html>

IEEE92

Institute of Electrical and Electronics Engineers. *Standard for Information Technology--Portable Operating System Interface (POSIX)* Part 2 (Shell and Utilities) Section 2.8 Regular Expression Notation. IEEE Std 1003.2, September, 1992.

Jaakko95

J. Jaakkola, and P. Kilpeläinen. Sgrep home page. Department of Computer Science. University of Helsinki, Helsinki, Finland. Available at <http://www.cs.helsinki.fi/~jjaakkol/sgrep.html>

Katz88

I.R. Katz and J.R. Anderson. Debugging: An Analysis of bug-location strategies. *Human-Computer Interaction*, Vol. 3, No. 4, pages 351-399, (April, 1988).

Kernig78

B. Kernighan and D. Richie. *The C Programming Language.* Prentice-Hall, 1978.

Lakhot93

A. Lakhotia. Understanding Someone Else's Code: Analysis of Experiences. *Journal of Systems Software*, Vol. 23, pages 269-275, 1993.

Lethbr97

T.C. Lethbridge and J. Singer. Understanding Software Maintenance Tools: Some Empirical Research. *Proceedings of 2nd International Workshop on Empirical Studies of Software Maintenance*, pages 157-162, Bari, Italy, 1997.

Letovs86

S. Letovsky. Cognitive Processes in Program Comprehension. *Empirical Studies of Programmers, First Workshop*, pages 58-79, Washington DC, USA, 1986.

Lieber97

H. Lieberman. The Debugging Scandal and What to Do About It. *Communications of the ACM*, Vol. 40, No. 4, (April, 1997).

Littma86

D. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental Models and Software Maintenance. *Empirical Studies of Programmers, First Workshop*, pages 80-98, Washington DC, USA, 1986.

Markos94

L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. "Using An Enabling Technology to Reeingineer Legacy Systems" *Communications of the ACM*, Vol. 37, No. 5, pages 58-70, (May, 1994).

Miles94

M.B. Miles and A.M. Huberman. *Qualitative Data Analysis: An Expanded Sourcebook, Second Edition.* Sage Publications, 1994.

Müller93

H.A. Müller, M. Orgun, S. Tilley, and J. Uhl. "Reverse Engineering Approach to Subsystem Structure Identification." *Journal of Software Maintenance: Research and Practice*, Vol. 5, No. 4, pages 181-204, (December, 1993).

MultiT98

The Very Large Multi-User Multi-Server Text Bases Project Home Page. Available at <http://multitext.uwaterloo.ca>.

Murphy96

G. Murphy and D. Notkin. Lightweight Lexical Source Model Extraction. *ACM Transactions on Software Engineering and Methodology*. Vol. 5, No. 3, pages 262-292, (July, 1996).

Paul94

S. Paul and A. Prakesh. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pages 463-475, (June, 1994).

Pennin87

N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, Vol. 19, pages 295-341, 1987.

Penny92

D. Penny. *The Software Landscape: A Visual Formalism for Programming-in-the-Large*, Ph.D. Thesis, Department of Computer Science, University of Toronto, 1992.

Perry94

D.E. Perry, N.A. Staudenmayer, and LG. Votta. People, Organizations, and Process Improvement. *IEEE Software*, pages 36-45, (July, 1994).

Pike87

R. Pike. Structural Regular Expressions. Proceedings of the European UNIX User's Group Conference, 1987.

Pigos93

T.M. Pigoski and C.S. Looney. Software Maintenance Training: Transition Experiences. *International Conference on Software Maintenance*, pages 314-318, Montreal, Canada, 1993.

Rosent75

R. Rosenthal and R.L. Rosnow. *The Volunteer Subject.* Wiley & Sons, 1975.

Seaman97

C.B. Seaman and V.R. Basili. An Empirical Study of Communication in Code Inspections. *Proceedings of the 19th International Conference on Software Engineering*, pages 96-106, Boston, USA, 1997.

Shneid79

B. Shneiderman and R. Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer and Information Studies*, Vol. 8, No. 3, pages 2119-238, 1979.

Shneid80

B. Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers Inc., 1980.

Sim98a

S.E. Sim and R.C. Holt. The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize. *Proceedings of the 20th International Conference on Software Engineering*, forthcoming, Kyoto, Japan, 1998.

Sim98b

S.E. Sim, C.L.A. Clarke, and R.C. Holt. Archetypal Source Code Searches: A Survey of Software Developers and Maintainers. *International Workshop on Program Comprehension*, forthcoming, Ischia, Italy, 1998.

Singer97a

J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An Examination of Software Engineering Work Practices. In *Proceedings of CASCON '97*, pages 209-223, Toronto, Canada, 1997.

Singer97b

J. Singer and T.C. Lethbridge. What's so great about grep? Available at <http://wwwsel.iit.nrc.ca/~singer/grep/grep.html>

SNiFF+96

SNiFF+ 2.3. User's Guide and Reference. TakeFive Software. Available at <http://www.takefive.com>, (December, 1996).

Solowa84

E. Soloway and K. Erlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*. Vol. SE-10, No. 5, pages 595-609, (September, 1984).

Solowa88

E. Soloway, J. Pinto, S. Letovsky, D. Littman and R. Lampert. Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM*, Vol. 31, No. 11, pages 1259-1267, (November, 1988).

Spool92

J.M. Spool. *Product Usability Survival Techniques*. Tutorial at the ACM Conference of Human Factors in Computing Systems (CHI), Monterey, California, May, 1992.

Spohre85

J.C. Spohrer, E. Soloway, and E.A. Pope. Goal/Plan Analysis of Buggy Pascal Programs. *Human-Computer Interaction*, Vol. 1, No. 2, pages 163-207, (February, 1985).

Storey96

M.-A.D. Storey, K. Wong, and H.A. Müller. On Designing an Experiment to Evaluate a Reverse Engineering Tool *Proceedings of the Third Working Conference on Reverse Engineering*, pages 31-40, Monterey, USA, 1996.

Storey97

M.-A.D. Storey, K. Wong, and H.A. Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 12-21, Amsterdam, Holland, 1997.

Strauss90

A. Strauss and J. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*, Sage Publications, 1990.

Tzerpo96

V. Tzerpos, and R.C. Holt. A Hybrid Process for Recovering Software Architecture. *Proceedings of CASCON96*, Toronto, Canada, 1996.

Tzerpo97

V. Tzerpos, R.C. Holt and G. Farmaner. "Web-Based Presentation of Hierarchic Software Architecture", *Workshop on Software Engineering on the World Wide Web, International Conference on Software Engineering,* 1997, Also available at <http://www.turing.utoronto.ca/~vtzer>.

Vessey89

I. Vessey. Toward a Theory of Computer Program Bugs: An Empirical Test. *International Journal of Man-Machine Studies*, Vol. 30, pages 123-146, 1989.

VonMay93

A. von Mayrhauser and A.-M. Vans. From code understanding needs to reverse engineering tool capabilities. *Proceedings of the 6th International Workshop on Computer Aided Software Engineering*, pages 230-239, Piscataway, USA, 1993.

VonMay95

A. von Mayrhauser and A.-M. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, pages 44-55, (August, 1995).

VonMay97

A. von Mayrhauser and A.-M. Vans. Hypothesis-Driven Understanding Processes During Corrective Maintenance of Large Scale Software. *Proceedings of the International Conference on Software Maintenance*, pages 12-20, Bari, Italy, 1997.

vanSol97

R. van Solingen, H. Leliveld, E. Berghout, and R. van Latum. Applying Software Measurement to Organizational Issues. *Proceedings of the 8th European Software Control Metrics Conference (ESCOM)*, Berlin, Germany, 1997.

Wu92

S. Wu and U. Manber. Agrep—A Fast Approximately Pattern-Matching Tool. *USENIX Winter 1992 Technical Conference*, pages 153-162, San Francisco, U.S.A., 1992.

Yin94

R.K. Yin. *Case Study Research: Design and Methods, Second Edition.* Sage Publications, 1994.

Zvegin97

N. Zvegintzov. Session 4: Maintenance Practices Within and Across Organizations. *Second International Workshop on Empirical Studies of Software Maintenance*, Bari, Italy, 1997.

Appendix A: Dictionary of Terms

Term	Description	Source
`bs_search`	Main cgi-bin script used to access the PBS.	Holt Group
`cfx`	Extracts facts from C source code into an intermediary format that is readable by `fbgen`.	Holt Group
factbase	A database of facts (syntactic information) about a software system.	
`fbgen`	Converts data from `cfx` into a factbase.	Holt Group
GCL	--see General Concordance Lists	
General Concordance Lists	A general purpose query language for structured texts.	[Clarke95a, Clarke95b]
`grep`	A UNIX utility that matches regular expressions within a file; more generally, a family of tools with this functionality.	UNIX operating system
`grok`	Used for manipulating binary relations and can perform select, join, intersections, and transitive closure.	Holt Group
`lslayout`	A Java applet for drawing Software Landscapes inside the PBS.	Holt Group
PBS	--see Portable Bookshelf	
PL/IX	A variant of the PL/I programming language created by IBM.	
`plix2rsf`	Generates a factbase for PL/IX source code.	U. of Victoria
Portable Bookshelf	A web-based documentation repository that integrates Software Landscapes, system documentation, and other HTML files. Also known as PBS.	Holt Group [Farman97]
Refine	A reverse engineering tool design to automatically migrate code based on a formal specification.	Reasoning Systems [Markos94]
Rigi	A tool suite to explore and manipulate a software system represented as nodes and edges.	[Müller93]
Rigi Standard Form	A syntax for storing information about a software system based on triples. This format is a precursor to TA. Also known as RSF.	University of Victoria
RSF	--see Rigi Standard Form	
Software Bookshelf	A metaphor for a shared repository of information about a software system.	[Finnig97]

Term	Description	Source
Software Landscape	A visual representation of a software system that uses a nested box formalism.	[Penny92]
TA	–see Tuple Attribute Language	
Tuple Attribute Language	Syntax for describing coloured graphs, i.e. nodes, edges, and their attributes. Also known as TA.	[Holt97]