

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** using **GCL**). 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.