

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 be 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.