

Chapter 3: Empirical Studies of Software Maintainers

1.1 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.

1.2 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.

1.3 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.

1.4 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].

1.5 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 they 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.

1.6 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.