Practices of Software Maintenance

Janice Singer
Institute for Information Technology
National Research Council Canada
Ottawa, ON K1A 0R6
singer@iit.nrc.ca
http://wwwsel.iit.nrc.ca
vox: +1-613 991 6346
fax: +1-613 952 7151

Abstract

This paper describes the results of an interview study conducted at ten industrial sites. The interview focused on the work practices of software engineers engaged in maintaining large scale systems. Five 'truths' emerged from this study. First, software maintenance engineers are experts in the systems they are maintaining. Second, source code is the primary source of information about systems. Third, the documentation is used, but not necessarily trusted. Fourth, maintenance control systems are important repositories of information about systems. Finally, reproduction of problems and/or problem scenarios is essential to problem solutions. These truths confirm much of the conventional wisdom in the field. However, in fleshing them out, details were elaborated, and additionally new knowledge was acquired. These results are discussed with respect to tool design.

Keywords: empirical study, work practices, large scale systems

1.0 Introduction

The Software Engineering Group of the National Research Council of Canada has initiated a program on the empirical approach to the study of software engineering [EASSE]. We are concerned with the collection, analysis, and public dissemination of data concerning the real work practices of software engineers. We are focusing on the human, as opposed to the machine or business component of the human-machine-organization triad.

Using experimental methods to study software maintainers is becoming an increasingly important aspect of tool and process design. This is reflected in the publication of a number of recent papers (e.g., [1-8]) and the creation of the Workshop on the Empirical Studies of Software Maintenance [9,10].

The data reported in this paper involves an interview study. Several organizations were contacted and within each organization, two engineers were interviewed about their work practices. The focus was on problem-solving and problem-solving techniques.

The results are presented as five 'truths' about software maintenance: software maintenance engineers are experts in the systems they maintain; source code is the primary source of information about systems; the documentation is used, but not necessarily trusted; maintenance control systems are important repositories of information; and reproduction of problems and/or problem scenarios is essential to problem solutions. These truths confirm much of the conventional wisdom in the field. However, in analyzing the results of the interviews, details were elaborated, and new knowledge was acquired.

This paper proceeds by describing first the methodology of the study. Then the results are reviewed. Finally the impact of the results on tool design is discussed.

2.0 Data Gathering

2.1 Participants

Twelve corporate groups and one government group participated in this study. The data from ten corporate groups is included in this report.

The groups were contacted with the help of a member of the Software Engineering Group; i.e., they were not randomly selected. Managers from the groups were given a written explanation which outlined the purpose of the study and included a copy of the interview[1]. If the managers had any

[1] Pre-inspection of the interview questions by the managers was necessary. Managers wanted to know specifically what questions would be asked in

questions or reservations about the study, the experimenter addressed them. In actual fact, the managers never questioned the interview questions, and all approached organizations agreed to participate. The managers chose appropriate participants for the interviews and arranged for times.

Interview participants were seen in pairs. This made the situation more comfortable for the participants and encouraged them to verbalize their thoughts because they could talk to each other about aspects of the project/product. It was stressed to the managers that all participants should be involved in the actual maintenance of software (as opposed to leading a team, other administrative posts, etc.). Additionally, it was requested, if possible that the two participants be working on the same project (and with only one exception, they were). One final requirement for the pair selection was that the two members have roughly the same level of experience in software maintenance (i.e., no novice/expert pairs were seen).

Finally, it was required that the interviewees be working on a product that was at least 1-1/2 years old and currently in a maintenance phase. Maintenance was defined purely in terms of age of system and could include both major and minor bug-fixes and enhancements.

2.2 Questionnaire

The interview questionnaire had three parts[2]. The first part of the questionnaire, background information, was created to characterize the nature of software maintenance within the context of the particular organization. Questions in this part of the interview focused on things like project size and age, application domain, expertise of individual programmers, etc. This information was necessary so that the results from different organizations could be compared and contrasted with respect to the maintained product.

The second part of the questionnaire, task analysis, was intended to gain insight into the unique nature of software maintenance work. Questions in this part of the interview focused on things like problem solving, naming conventions, browsers, historical information, etc. This part of the questionnaire was not strictly followed, but rather intended as a set of topics upon which conversation about software maintenance could proceed.

The final part, tools wish list, let the software maintenance engineers express their needs for tools to help them do their jobs.

2.3 Interview Procedure

All interviews were conducted on-site at the participating organizations. The participants arranged for a quiet location, generally a conference room, in which the interview would take place. All interviews were tape-recorded. The participants were asked for their consent before the tape recorder was started. All participants consented. Some participants had access to the questionnaire a priori (this was at the discretion of the manager who set up the interview).

Participants were told the purpose of the study and asked if they had any questions. It was explained that all data collected would be held in strict confidentiality and that neither the names of the participating organizations nor the names of the interviewees would be identified in any report. It was stressed that even the managers would not be able to identify their particular organization in any reports. The interviewees were told that some questions may be sensitive and not appropriate in terms of the type of information the organization wanted to divulge. They were told that they were in a better position than the interviewer to recognize these questions and that where this case applied, they should just inform the interviewer and the question would be skipped. In fact, in the actual interviews, this situation never arose.

Part 2 of the questionnaire (task analysis) was taken as a guideline for interesting topics to pursue. If the interview drifted into another area, it was generally followed, however certain core questions were asked of all participants.

After the first few interviews, it was clear that the Part 3 questions (tools wish list) were difficult for the software engineers to answer without knowing what kinds of technologies were available. For this reason, these questions were asked only if time allowed, and generally it did not.

3.0 Results

All interviews were tape-recorded and transcribed verbatim. These verbatim transcripts were used as the basis for this analysis. Sections 4.1 and 4.2 review the findings from Part 1 of the questionnaire. Section 4.3 summarizes some interesting findings from Part 2 of the questionnaire.

3.1 Project Information Statistics

Project information statistics were gathered so that specific findings could be related to types of organizations. Conversely, if certain findings were true across several organizations, project information statistics would allow for generalization of findings.

Interestingly enough, one typical generalization factor, size of system, was impossible to collect in this study. First, several of the organizations had no idea of the extent of their software system. That is, they neither knew the number of files nor the lines of code programmed. Second, where the interviewees did know the size of the system, they provided the information in different (incommensurable) scales. Some organizations knew KLOC, some knew number of files, some knew number of megabytes. Because of these two issues, system size is not reported here.

order to make their decision about whether this was an appropriate study for their organization.

[2] The questionaire can be viewed at: wwwsel.iit.nrc.ca/~singer/main.html.

Tables 1-4 (below) provide information about the types of systems that were being maintained. Eleven systems are profiled because at one site, the two interviewees were working on different systems. At all the other sites, the interviewees were working on the same system, although oftentimes on distinctly different parts of it.

Application Type	#
COTS component	4
Proprietary hardware control	2
IBM mainframe applications	2
Accounting services	1
Data services	1

Table 1. Application types

Table 1 shows the application types. Four systems were COTS components: intended to be used as off-the-shelf components of another piece of software. Two systems were based on proprietary hardware that was designed for a specific purpose. Two systems were for IBM mainframe computers. One system provided accounting services. Finally, one system provided data services, both real-time data and analysis tools, to its subscribers.

Maintenance Platform	#
Windows	5
Mainframe	4
UNIX	3
Proprietary	2
Other	6

Table 2. Platforms

Table 2 shows the distribution of platforms. Note that these platforms are not necessarily the target platforms for the software, but are the platforms under which maintenance was completed. There are more than 10 platforms because some systems were being maintained under more than one platform. Five systems ran under Windows. This included Windows, Windows 95, and Windows NT. Four systems were for a mainframe computer. Three systems ran under UNIX. Two systems were developed for a proprietary hardware configuration. Finally, six systems were maintained under various other platforms.

Maintenance Languages	#
C	6
Proprietary	3
COBOL	1
Assembler	1

Table 3. Languages

Table 3 shows the maintenance languages. C was the predominant language. Where they used C, they were either also using, or expressed a need to soon use C++. Three systems were developed using proprietary languages (2 of these were for the proprietary platforms mentioned above). One system was developed using COBOL. Finally, one system was Assembler-based.

Overall, then, it appears as though the systems being profiled are representative of the industry. They cover a wide variety of applications and are maintained under various platforms using a variety of languages.

Project Info	Ave	Max	Min
Age of System	9.00	15	3
# of Platforms	4.10	15	1
# of Versions	2.22	5	1
Longest Time	6.75	10	2.5
Typical Time	3.86	6	3
# of Engineers	13.40	30	2

Table 4. Project statistics

Table 4 shows summary statistics for the various software systems. Overall the systems were relatively old, with an average age of 9 years. The systems ran on an average of 4.10 platforms, although some systems ran on as many as 15 different platforms. When systems did run under a great number of platforms, it was generally handled by having specific personnel deal with specific platforms. For example, one person might be the expert in UNIX and another person the expert in Windows. The software was being supported for an average of 2.22 versions. Generally speaking, when a company supported more than one version, it was dealt with by major and minor version numbers. For example, a company might support major release 1 and 2 which would be qualitatively different. But within each major release, the company would support only the most recent update. So a company might support Version 1.4 and Version 2.3, and at the same time no longer support Version 1.3 and Version 2.2.

The longest time any engineer had been on a project was 10 years (this question included all group members and not just those being interviewed), while the average time was 6.75 years. Typically though, engineers spent about 3.86 years on any one project. The group sizes varied greatly, anywhere from 30 to 2 members. On average, though, the groups had about 14 members. Overall, again, what emerges from these data is a snapshot of a 'typical' software maintenance organization. Typical in that these types of organizations vary widely. It appears, however, that a good cross-section of maintenance organizations is represented in this data set.

3.2 Interview Participant Statistics

Table 5 (below) summarizes data from Part 1 of the questionnaire about individual experience of the interviewees. These data paint a picture of software maintenance engineers as being both expert programmers and experts in the project in which they are working. The software maintenance engineers have programmed for an average of 9.60 years, with 6.68 years on the maintenance language. The engineers considered themselves to be expert in an average of 2.45 languages.

Moreover, many software maintenance engineers expressed the opinion that programming language knowledge was relatively abstract, and that their expertise would enable them to transfer their skills to additional languages with no problem.

Individual Info	Ave	Max	Min
Years Experience	9.60	20	3
Time on Language	6.68	15	2
# of Languages	2.45	5	1
Time on Project	4.38	10	1

Table 5. Individual statistics.

Not only were the software maintenance engineers expert programmers, but they were expert maintenance programmers. On average, they had spent 61% of their professional life on maintenance projects and only 39% in new development. It is not clear if different skills are needed for these two endeavors, but if so, then, on average, the interviewees were more familiar with the job of maintaining software programs than developing new ones.

What was most surprising about this data, however, was not the expertise of the individual programmers, but rather the length of time they had remained with their current project, an average of 4.38 years (additionally no one interviewed expressed any imminent desire to move on to another project or product group). This is surprising because the literature in software maintenance points to high turnover rate as being a major problem for maintenance organizations. However, it is possible that the managers chose their more stable employees to participate in the interviews.

3.3 Task Analysis Findings

A qualitative analysis of the Part 2 questions was completed. The data was diverse and hard to coagulate down to small chunks of knowledge. As such this section will present four 'truths' about software maintenance that stood out across most organizations. First, and probably most importantly, the source code is king, that is, if it's familiar. Second, the documentation is used, although it is not trusted. Third, problem reporting databases are important warehouses of information. Finally, replication of the problem and/or problem scenario is key to solving the problem.

This section is illustrated with quotes from the software engineers (in Geneva font). The text further clarifies the issues.

3.3.1 Source code is king

> Source code is basically the bible.

In response to the questions, "How do you go about solving a problem?" and "Where do you look for information?," seven of the ten organizations cited the source as being the primary source of information about the system.

> You still have to investigate the feasibility when it comes to future interactions. Source code's still there.

The source is cited as the place to look for bug fixes, but also as the primary place to look when adding enhancements to the system.

> If it's an area of the code that you work on, then you go directly to the source code. If it's an area that you're not familiar with, you might go talk to somebody who you know has worked on that piece of code, and either you transfer the problem to them, or you get the guidance from them, and then you fix the problem yourself.

There is one important codicil to the source being king: only if it's a familiar ruler will it be consulted first. All of the above interviewees would consult the source first only in instances where the source was known. If it wasn't known, all of the above interviewees would consult another person, if possible the author of the code, and if not, then a guru in the system.

> Typically the author, even if that person hasn't worked on it in a long, long time, you never know, it just might tweak.

The three non-conforming organizations had employees who would first consult another person. The reason another person was mentioned first in these instances is because the interviewees were assuming that they had not written the code in question.

The source, then, is the most important 'document' with respect to system maintenance. In many cases, it is the only accurate representation of the system. Joiner, et al. [2] found the same result. Unlike Joiner, et. al. [2], though, this study pointed out that when someone is unfamiliar with the source , they consult other people; this in the service of finding out more about the source, or finding out where to look in the source. Thus, the source code really is king, and this does not seem to be an exaggeration.

3.3.2 Untrustworthy documentation

> Do you ever look at the documentation?
>
> Sometimes when we're curious. Occasionally there's some nice stuff in there, but it's inconsistent.

Documentation is a touchy issue. It's hard to write generalizations about it because so many different things were said. Additionally, it seemed as though each company had a unique way of dealing with documentation. Some companies

had explicit protocols for documenting either as in-line comments or separate addenda to the code. Some companies only used in-line comments, but followed cultural standards regarding what types of things to write. Other companies were very casual with documentation, allowing the software engineers to decide the content, style, and necessity for it.

> No matter how much documentation you have, I won't believe it...because if the documentation is out of date, it's garbage.

The biggest problem with documentation was maintenance. Only in the most rigorous environments was documentation attached to a process for completing a change request. In this case it would be continually updated. In other organizations documentation would sometimes get updated, and sometimes not. In general, the accuracy of the documentation seemed to depend largely on its recency and the amount of change that had occurred in that section of code over time (with more change likely to indicate greater discrepancy between what was written and what was true).

> Well, part of the problem is there's too much documentation...And just as the software has interactions, what's written in the documents interacts. One page states something, the other changes it.

In those companies that had adopted a rigorous approach to documentation another problem commonly surfaced, namely that of the abundance with which it proliferated. It was often hard to find the document that you wanted. Searching through the documentation became as great a navigation nightmare as searching through the source. Here, programmers had to rely on other programmers to point them in the right and useful direction.

> The documentation is good to give you a high level understanding of how the feature is really intended to work and it usually follows its original intent.

Software engineers found more use and were more likely to trust documentation that described the design of a particular feature or the architecture of the system; i.e., the more abstract a piece of documentation was, the more likely it was believed to be correct. This statement must be qualified, though, by mentioning that in-line commentary was believed to be much more accurate than documentation that was generated as attachments to the source.

> You know, the payback on the documentation, especially in a low turnover environment isn't that great.

The value of having experienced engineers spend time updating was questioned at some companies. Although, it was felt that documentation was, in general, valuable, having experts document code was not perceived to be so.

Documentation is useful. Unfortunately, the flip side of documentation being useful is that it's time consuming to create and maintain. Even in those instances where software engineers are motivated to maintain the documentation, it is often unclear how it interacts with itself and the source code. Because of these difficulties, although documentation is useful, it is not trusted. It can give ideas, point to helpful sections of code, and provide certain guidelines, but in the end, the source code itself is the arbitrator of all disputes.

3.3.3 MCS stores knowledge

> So the bug tracking system is the place where information is kept?
>
> The core of the whole thing.

Nine of the ten organizations had some form of a maintenance control system. These were quite diverse in terms of complexity and coverage. In the simplest cases, a version control system was used with simple comments identifying changes between versions. Some companies then added a distinct problem tracking system that enabled users to assign and track changes to the system. In other companies, version control and problem tracking were integrated via configuration management software. In both of the later types of companies, the bug tracking database served as an important warehouse of information.

> [*We look in the bug-tracking database to see*] Was there a similar problem, was it ever fixed, was it fixed in a version and somehow we lost the fix along the way.

MCS's were seen as important repositories for historical information about the system. In particular, software engineers at four of the ten companies reported that they were useful for finding similar fixes. Finding similar problems and/or fixes allowed software engineers to either a) modify an already proven piece of code or b) provide assistance thinking through the logic of the fix. Similar problem solutions were particularly useful in multi-platform environments. Here a fix from one platform could be modified for a different one.

> The problem reports are effectively changing the functionality of the system and that functionality is not captured in the standard design document.

The bug-tracking database was also seen as capturing unique types of information. Additionally, because it was kept up to date (where documentation might not be), it was seen as a more accurate representation of the system's current state.

> Everyone from customer support can see what the status is.

One important feature of bug-tracking databases was that they supported communication across groups. Because bug-tracking databases were not specific to particular groups, but rather 'owned' at the corporate level, access was company wide. Therefore, they were often used by other software engineers (both within and outside the group), managers, business specialists, customer support, etc., for understanding how updates occurred, whether, when, and how they occurred, and subsequently, for making business decisions.

Bug tracking databases allow companies to store historical information about systems. This information turns out to be important, not only within a group, but also at the corporate level. Where documentation was often seen as an aside to the process of writing software, keeping track of bugs and fixes was seen as an integral part of the process. Because of this, bug tracking databases were more likely to be updated as needed resulting in were more complete and accurate textual representations of the system.

3.3.4 Reproduction = solution

> Typically reproduction of the thing is the most important. And if not, it's usually by guess and by golly.

At eight of the ten organizations, the issue of reproducing the problem came up at some point during the interview. In some cases, reproduction came up as the answer to "How do you go about solving a problem?" In other cases, reproduction came up as an aside to other questions. In all instances, though, the software engineers brought up reproduction; they were never asked or primed about it.

> Knowing that you can recreate something, you can actually sit down and trace what's happening - be it using a utility or even dumping it or whatever. It allows you to at least even read through the listings of that relevant code. And you'll have an understanding of what's actually taking place.

Software engineers felt that if the problem could be reproduced, it could be solved. This was because reproducing the problem scenario leads to a) better understanding of the problem b) better understanding of what the code is supposed to be doing and c) the place in the code where the problem is located.

> Generally, if you can't reproduce it, it's almost impossible to fix.

There was an important corollary to this principle: If the problem could not be reproduced, then it could generally not be solved.

> One of the things you generally have to do is reproduce the problem, and sometimes that's not trivial. Particularly, it may be a problem that's only (on) certain types of offices, certain types of configurations, or maybe only in a large office, or intermittent...

The biggest obstacle to successfully reproducing the problem was getting the same setup in the office that was used when the problem appeared. This could mean either hardware or software configuration.

> Sometimes the documentation is wrong. Sometimes, they're misusing the product, trying to do something it really wasn't designed for.

> Yeah, because if you can't duplicate it, you can't experience the problem. So you have to wonder, is he really experiencing this problem due to us or due to some other product... I don't want to call him a liar, but it may not be your problem.

Interestingly enough, one of the reasons reproducing the problem was so important was to ensure that it was a real problem, and not a perceived one or one caused by external sources, such as software interactions.

Thus reproduction of a problem leads to greater understanding of the problem and the conditions under which it exists. Reproduction can also lead to the place in the code that needs to be updated. Reproduction is also important because it tells software engineers whether a real problem actually exists. In fact, software engineers consider the hallmark of a 'do-able' problem to be whether it can reproduced or not.

4.0 Discussion and Conclusion

This study proceeded by interviewing software maintenance engineers in ten corporate environments. The environments themselves were diverse with respect to practically all defining variables. What the systems all had in common was that they were in a maintenance phase defined as post-development enhancements and bug fixes. The results were presented as five 'truths' about software maintenance.

Each of these 'truths' may appear to be self-evident. However this approach has allowed us to document them. This is important for two reasons. First, from here, we can further explore how each 'truth' specifically affects the productivity and efficiency of software engineers. Second, documenting these untold truths allows us to examine our assumptions about software maintenance. This provides the forum for

further discussion and, in some instances, disagreement. In the spirit of the second of these consequences, the rest of this discussion will focus on the impact of these truths on tool design.

Quantitatively, one 'truth' emerged from this interview. Not only had software engineers stayed with the organizations for a long time (an average of 4.38 years - and some as long as 10 years), but they were also experts in the systems on which they were working. This is important because with a low turnover-rate, tools should focus on allowing experts full access to and ease of navigation in the system. rather than on helping them learn it.

Qualitatively, four 'truths' about software maintenance arose through the discussions with software engineers. Each of these also has some important implications for tool design. First, the source code was trusted above all other forms of information, with the exception, perhaps of another person's knowledge. This points to two possible directions in tool design. First, and obviously, better tools for navigating and understanding the source (as generated from the source) would be useful. It's not so obvious, however, how to harness the second finding. If other people are consulted, how can we put this knowledge into a tool that will be used, without at the same time interfering with the social aspect of software maintenance.

Combined with the second and third truth, this points to an often overlooked aspect of maintaining software: cultural and social systems have a huge impact on what works and what doesn't in a particular organization. Why documentation was not updated and bug-tracking databases were has as much to do with social factors as with technical difficulties. Two different hypotheses may apply here. First, the perceived value of keeping information and its later use may have encouraged software engineers to maintain bug-tracking systems. Second, bug tracking systems could have been maintained because they were seen as a way of communicating results across the corporation. In either case (or another scenario), it is important to understand why, so that new tools will have as much a chance of success as MCSs did in these companies.

Finally, software engineers felt that problem reproduction equals problem solution. This 'truth' has important educational implications (besides the technical ones). If reproducing a problem is such a huge component of maintaining software, it makes sense for us to train students to do so.

In conclusion, these type of interview studies and other empirical approaches can shed light on the process of software maintenance, thereby enabling us to build better tools, and think about our underlying assumptions regarding their design and use.

References

[1] Dart, S., Christie, A., Brown, A. (1993). A case study in software maintenance. CMU Technical Report, CMU/SEI-93-TR-08.

[2] Joiner, J., Tsai, W., Chen, X., Subramanian, S., Sun, J, & Gandameneni, H. (1994). Data-centered program understanding. In the proceedings of the International Conference on Software Maintenance, Victoria, British Columbia.

[3] Jorgenson, M. (1994). An empirical study of software maintenance tasks. Software Maintenance: Research and Practice, 7, 27-48.

[4] Layzell, P., Macaulay, L. (1994). An investigation into software maintenance - Perception and practices. Software Maintenance: Research and Practice, 6, 105-120

[5] Lientz, B., Swanson, E., Tompkins, G. (1978). Characteristics of application software maintenance. Communications of the ACM, 21(6), 466-471.

[6] Vessey, I., & Weber, R. (1983). Some factors affecting program repair maintenance: An empirical study. Communciations of the ACM, 26(2), 128- 134.

[7] Von Mayrhauser, A., & Vans, A. (1995). Program comprehension during software maintenance and evolution. Computer, 28(8), 44-55.

[8] Proceedings of the 1st International Workshop on Empirical Studies of Software Maintenance. (1996). Monterey, CA, USA.

[9] Proceedings of the 2nd International Workshop on Empirical Studies of Software Maintenance. (1996). Bari, Italy.