How Software Engineers Use Documentation: The State of the Practice

Timothy C. Lethbridge, *University of Ottawa*
Janice Singer, *National Research Council*
Andrew Forward, *Deloitte Consulting*

Software engineering is a human task, and as such we must study what software engineers do and think. Understanding the normative practice of software engineering is the first step toward developing realistic solutions to better facilitate the engineering process.

We conducted three studies using several data-gathering approaches to elucidate the patterns by which software engineers (SEs) use and update documentation. Our objective is to more accurately comprehend and model documentation use, usefulness, and maintenance, thus enabling better decision making and tool design by developers and project managers. Our studies' results confirm the widely held belief that SEs typically do not update documentation as timely or completely as software process personnel and managers advocate. However, the results also reveal that out-of-date software documentation remains useful in many circumstances.

Most software documentation is not updated consistently, but out-of-date documentation might remain useful. We must find powerful yet simple documentation strategies and formats that software engineers will likely maintain.

The studies

The first study consisted of interviews at 12 corporate sites and one government site.[1] We interviewed participants in pairs to make the situation more comfortable and natural for them. The interview questionnaire (see wwwsel.iit.nrc.ca/~singer/main.html) had three parts: background information, task analysis, and tools wish list. The sidebar lists some of the interviewees' assessments of documentation.

We conducted the second study at a large telecommunications company.[2,3] It involved software engineers maintaining and enhancing a large, profitable telecommunications system written in a proprietary high-level language. This study had four components. First, using a Web questionnaire, we asked the SEs what

Documentation: The Good, the Bad, and the Ugly

During our interviews, software engineers expressed the following general attitudes about documentation.

The good

- Architecture and other abstract documentation information is often valid or at least provides historical guidance that can be useful for maintainers.
- Inline comments are often good enough to greatly assist detailed maintenance work.

The bad

- Documentation of all types is frequently out of date.
- Systems often have too much documentation.
- Documentation is often poorly written.
- Finding useful content in documentation can be so challenging that people might not try to do so.
- Much mandated documentation is so time consuming to create that its cost can outweigh its benefits.

The ugly

- A considerable fraction of documentation is untrustworthy.

they do. Second, we followed a single SE for 14 weeks as he worked. Third, we shadowed nine SEs individually for one hour as they worked. Finally, we obtained companywide tool use statistics.

The final study comprised a 50-question survey that focused on many documentation aspects.[4,5] We solicited survey participants directly from several high-tech companies and by using software engineering email lists.

Of the three studies, only the final one focused exclusively on documentation issues. The former two focused on gaining an overall understanding of the software maintenance process. Even so, they uncovered important information that relates directly to documentation use and usefulness.

Documentation maintenance and usefulness

First, and probably foremost, our studies confirm that SEs often do not maintain documentation. Figure 1 shows survey respondents' answers to the question, "In your experience, when changes are made to a software system, how long does it take for the supporting documentation to be updated to reflect the changes?" (Note that only 25 of the 48 survey respondents answered this question.) With the exception of testing and quality documentation (such as test cases and plans), SEs rarely update the documents. When SEs do update documents, they usually do so several weeks after changes are made to the code. Forty-four percent of the 45 respondents somewhat agreed and 24 percent strongly agreed with the statement, "Documentation is always outdated relative to the current state of a software system."

The interview responses mirrored the survey

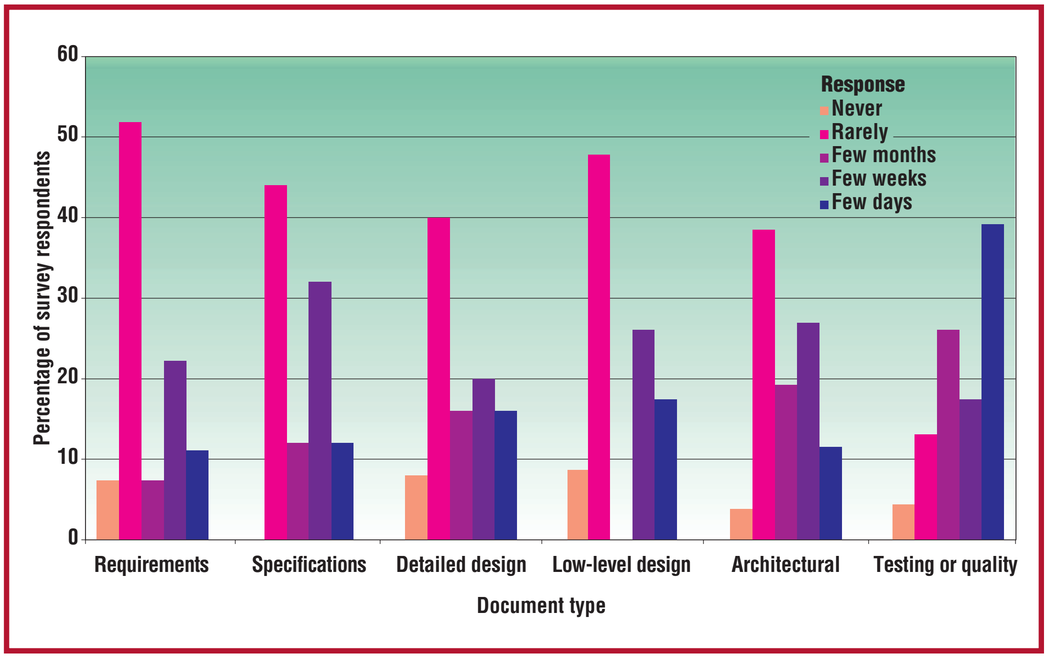


Figure 1. The time between system changes and documentation updates for different documentation types.

results. SEs were likely to update documentation continually only when it was attached to the process for completing a change request. In other contexts, SEs sometimes would and sometimes wouldn't update documentation. The documentation's accuracy seemed to depend largely on its recentness and the amount of change that had occurred in the relevant code sections (with greater source code change corresponding to greater discrepancy between the code and the documentation).

In important contrast to the lack of documentation maintenance is the usefulness of even outdated or inaccurate documentation. Fifty-three percent of the 45 survey respondents somewhat agreed and 28 percent strongly agreed that "Software documentation can be useful, even though it might not always be the most up-to-date." The survey provided data on the SEs' perceptions of different document types' accuracy and the frequency with which they consulted the documentation. Table 1 shows the correlation between the document type's perceived accuracy and the frequency with which SEs consulted the documentation.

The relationship between accuracy and consultation frequency is the highest for testing and quality documents and the second highest for low-level design documents. To a lesser degree, the accuracy of requirements, architectural design documents, and detailed design documents also correlate with consultation frequency. Specifications have almost no such correlation. The relationships seem to indicate that the closer you get to the real code, the more accurate the documentation must be for SEs to use it.

The interviews support the survey findings. SEs were more likely to use and trust documentation that describes a particular feature's design or the system architecture. The more abstract a piece of documentation, the more likely SEs were to consider it accurate and useful. One software engineer stated, "The documentation is good [for giving] you a high-level understanding of how the feature is really intended to work."

Documentation applicability

Table 2 shows the tasks for which the survey respondents rated the available software documentation effective or extremely effective.

More than one-half of the respondents found the available software documentation effective when learning a new software system, testing a system, or working with a new system. Fifty percent found it effective when other developers are unavailable or when looking for big-picture information. Only approximately one-third of the respondents found the documentation effective for maintaining a system, answering questions about the system, looking for in-depth information, or working with an established system. The results indicate that documentation satisfies particular roles for particular tasks.

Words versus actions

We must recognize that our survey results reflect how SEs perceive documentation, not necessarily how they actually use it. For example, at the telecommunications company, 6 percent of the respondents said they spent considerable time reading documentation, and 50 percent reported spending considerable time consulting

Table 1
The correlation between a document type's perceived accuracy and its consultation frequency

Document type	Correlation
Testing or quality	0.67 ($p < .005$)
Low-level design	0.58 ($p < .005$)
Requirements	0.43 ($p < .05$)
Architectural	0.41 ($p < .05$)
Detailed design	0.39 ($p < .05$)
Specifications	0.03

Table 2
Percentage of survey respondents who rated documentation effective or extremely effective for particular tasks

Task	Percent
Learning a software system	61
Testing a software system	58
Working with a new software system	54
Solving problems when other developers are unavailable to answer questions	50
Looking for big-picture information about a software system	46
Maintaining a software system	35
Answering questions about a system for management or customers	33
Looking for in-depth information about a software system	32
Working with an established software system	32

To achieve greater documentation relevance, we need to find ways to increase its power, simplicity, or preferably both.

source code. In our observational studies at the same company, however, SEs consulted the documentation only 3 percent of the time: 12 times over 357 logged events. This suggests that documentation use might be even less than reported. The higher perceived use might nonetheless suggest that SEs place considerable value on the documentation.

Discussion

Our studies raise two main issues.

Timeliness

The first is, should we force software engineers to keep documentation meticulously up-to-date? Formal-process theorists would certainly argue that we should. In fact, most published methodologies prescribe the documentation types that SEs should write and use.

But where's the real evidence that the prescribed processes work? Most of it is based on opinion or conjecture. Many software projects fail or run over budget, but evidence hints that the fault lies mostly with poor management and failure to gather requirements, not with out-of-date or incomplete documentation. As we mentioned before, our studies suggest that out-of-date documentation has value, particularly if the high-level abstractions remain valid.

Judging value: The simple-and-powerful rule

The second issue is why SEs adopt particular practices and tools. Our results indicate that software engineers create and use simple yet powerful documentation, and tend to ignore complex and time-consuming documentation.

Consider bug-tracking systems. The interviews revealed that SEs perceive them as important repositories for historical information. Documentation in bug-tracking systems stays up-to-date because SEs recognize its value—adding a simple comment as you fix a bug requires little effort, and maintenance is semiautomatic. Similarly, code-level comments stay current because they are short and "right there," resulting in relatively little maintenance work. Test cases also stay up-to-date because each one has a simple structure and obvious operational value for verifying the system. Specifications and requirements, however, are big, complex, and of varied structure. So, SEs consider updating these documents less worthwhile, especially because the high-level abstractions tend to remain useful when the details become outdated.

The necessity of designing simple, powerful design tools is evident in other software engineering areas. For example, ultrasimple yet powerful tools such as grep are still among the most widely used. They're far more popular and enduring than many CASE (computer-aided software engineering) tools that are more powerful but much more complex. SEs also widely use processes such as code inspections, which have obvious power and which they can describe in a page or two.

All this suggests that to achieve greater documentation relevance, we need to find ways to increase its power, simplicity, or preferably both. We must find ways to express the most useful information in less space and to make documentation easier to update, perhaps semiautomatically.

Some people will argue that SEs fail to update documentation because they're lazy. Many managers have responded to this assertion by trying to impose more discipline on software engineers—forcing them to update documents. We suggest that most SEs aren't lazy; they have discipline of a different sort. They consciously or subconsciously make value judgments and conclude that it's worthwhile to update only certain types of documentation.

So, rather than forcing SEs to perform cost-ineffective work, we should strive for simple yet powerful documentation formats and tools, as we just mentioned. Also, we need to better understand the various roles of software documentation and more closely match our prescribed processes to fit those roles.

An additional lesson from our research is that you can learn a lot from studying SEs in the real world—both what they do and how they think. Further studies like ours could provide rich data that can serve to help formulate research questions. Those research questions, in turn, could aid in other types of empirical studies, such as testing specific hypotheses in more constrained (artificial) settings.

References

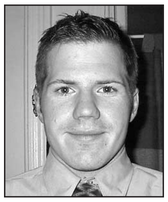
1. J. Singer, "Practices of Software Maintenance," *Proc. Int'l Conf. Software Maintenance* (ICSM 98), IEEE CS Press, 1998, pp. 139–145.
2. J. Singer et al., "An Examination of Software Engineering Work Practices," *Proc. Centers for Advanced Studies Conf.* (CASCON 97), IBM, pp. 209–223.
3. J. Singer and T.C. Lethbridge, "Studying Work Practices to Assist Tool Design," *Proc. Int'l Workshop Program Comprehension* (IWPC 98), IEEE CS Press, pp. 173–179.
4. A. Forward, *Software Documentation: Building and Maintaining Artefacts of Communication*, master's thesis, School of Information Technology and Eng., Univ. Ottawa, 2002; www.site.uottawa.ca/~tcl/gradtheses/aforward.
5. A. Forward and T.C. Lethbridge, "The Relevance of Software Documentation, Tools and Technologies: A Survey," *Proc. ACM Symp. Documentation Eng.* (DocEng 2002), ACM Press, pp. 26–33.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.

About the Authors

Timothy C. Lethbridge is an associate professor at the University of Ottawa. He investigates ways that people can more easily understand and manipulate complex information, including software. He's on the steering committee of *Computing Curriculum—Software Engineering*, sponsored by the IEEE Computer Society and the ACM. He also coauthored *Object Oriented Software Engineering: Practical Software Development Using UML and Java* (McGraw Hill, 2001). He received his PhD in Computer Science from the University of Ottawa. He's a senior member of the IEEE. Contact him at SITE, 800 King Edward Ave., Ottawa, ON K1N 6N5, Canada; tcl@site.uottawa.ca.

Janice Singer is a cognitive psychologist working in the National Research Council of Canada's Software Engineering Group. She also heads the NRC's Human-Computer Interaction program. Her interests lie in collaboration, cognition, and improving software processes and tools by understanding the cognitive and social demands of work. She received her PhD in cognitive psychology from the University of Pittsburgh. Contact her at the NRC Inst. for Information Technology, M50, 1200 Montreal Rd., Ottawa, ON K1A 0R6, Canada; janice.singer@nrc-cnrc.gc.ca.



Andrew Forward is a systems analyst with Deloitte Consulting, working in their technology practice. His academic interests are software engineering, automated testing, and documentation. He has an MS in computer science from the University of Ottawa. He's a member of the IEEE and ACM. Contact him at 106 Melrose Ave., Apt 1, Ottawa, ON K1Y 1V1, Canada; aforward@dc.com.