

Software Documentation: How Much is Enough?

Lionel C. Briand
Software Quality Engineering Laboratory
Carleton University
Department of Systems and Computer Engineering
1125 Colonel By Drive
Ottawa, ON, K1S 5B6, Canada

URL: <http://www.sce.carleton.ca/Squall/>

e-mail: briand@sce.carleton.ca

1 Introduction

It is a well-known fact that software documentation is, in practice, poor and incomplete. Though specification, design, and test documents—among other things—are required by standards and capability maturity models (e.g., SEI CMM), such documentation does not exist in a *complete* and *consistent* form in most organizations. When documents are produced, they tend to follow no defined standard and lack information that is crucial to make them understandable and usable by developers and maintainers.

Why is this the case? Recent work [4] suggests that documentation is often perceived as too expensive, difficult to maintain under the typical time pressures that are pervasive across the software industry. Interestingly enough, most people think some form of documentation is necessary but there is usually little agreement on what is needed. Even more surprising, in everyday practice people actually use incomplete, obsolete documentation and find it useful. Obviously, in practice, a tradeoff has to be found between the level of detail and scope of documentation, the frequency with which it is updated, and the cost of its development and maintenance. Intranet technologies have made the dissemination and use of documentation much easier within software development organizations.

Then a fundamental practical question, which motivated this keynote address, is to better understand what type of documentation is required, what is needed to support its completeness and consistency, and what is the level of precision required for each type of document. These questions cannot be investigated at that level of generality though. Answers are likely to be very context-dependent if they are to be precise. We focus our attention here on object-oriented development and the Unified Modeling Language (UML) [2].

2 Documentation in Object-Oriented Development

Typical object-oriented development methodologies [2, 3, 5] require that people perform and document (1) Analysis (i.e., a specification with the specific feature that the system structure starts to emerge as well as its functionality), (2) high-level or architectural design (i.e., with the main purpose of defining and describing subsystems) and (3) low level design (i.e., with the main focus of optimizing and completing the system design). Those activities are typically taking place in the context of incremental development and are therefore subject to numerous iterations.

Analysis documents typically contain a use case model describing high-level end-user functionality in a textual but structured manner. From this, an Analysis class diagram is derived, mostly containing application domain objects (i.e., objects corresponding to application domain concepts). Then interaction diagrams (e.g., sequence diagrams) are defined for each use case, thus describing the possible exchanges of messages between objects for various possible scenarios. This is in turn useful to complete the Analysis class diagram. For classes with state-dependent behavior, statecharts are usually defined so as to clearly model such complex behavior and avoid likely mistakes early on. Statecharts also help to identify missing use cases if transitions are not covered by existing use cases.

The high-level design usually decomposes the system into subsystems with clearly defined interfaces (i.e., public operations and possibly contracts) thus introducing another level of information hiding. Low-level design makes use of design patterns and modifies the design so as to complete it and optimize it. This is admittedly an over-simplification but this summary provides the overview we need here to develop our argument.

The main motivations for all existing object-oriented methodologies are as old as software engineering itself:

abstraction and separation of concerns as a means to cope with complexity. What is more specific to object-orientation is the smooth transition between phases, especially between analysis and design.

In this context, many questions arise regarding how exactly such Analysis and Design documentation is to be written and used:

- Regarding requirements elicitation: how should use cases be described, using which templates? What should be the writing style and guidelines to follow?
- Regarding the class diagram, what class taxonomy should be used in order to help assign class responsibilities (e.g., control, boundary, entity [2])? What level of detail should be required? For example, should parameter and attribute types (e.g., UML types) be provided at that stage?
- Important questions also relate to the use of contracts. Should Analysis contracts [6], defining pre-conditions, post-conditions, and class invariants be required (e.g., using the Object Constraint Language (OCL) [7]) so as to make the analysis more precise? Should they be left for the design stage only? Fusion [3] recommends the former whereas Bruegge and Dutoit [2] recommend the latter.

3 Extreme Programming and Documentation

Extreme Programming (XP) has a very different take on software documentation. The reason why this is discussed here stands from the fact that, with respect to documentation, it provides a drastically different view from what has been assumed in the last two decades or so and it is becoming increasingly popular.

XP relies exclusively on ‘oral communication, tests, and source code to communicate system structure and intent’ [1]. In other words, if faithfully applied, there is no Analysis and Design documents. The main assumptions which justify such an extreme approach are:

- XP is designed to work with small teams of two to ten programmers
- Staff turnover is supposed to be small as there is ‘less chance that a programmer gets frustrated’.
- The customer is an integral part of the team (thus providing constant feedback on requirements).
- A comprehensive suite of tests is maintained thus communicating, in a certain way, the intent of the software.
- With new technologies, the cost of change does not increase exponentially (as common wisdom and existing data suggest) and rises slowly over

time. This has a huge impact as an attitude of constant change of the requirements and refinement of the design then becomes possible. There is no need for upfront, thorough, tentatively complete Analysis and Design models and documents.

Of course, there are a number of possible contradictions in the premises and principles of XP that are worth discussing and investigating.

- How do you derive your test suite? How do you guarantee a systematic testing strategy without Analysis or Design documentation? We all know the limits of white-box testing (i.e., cannot detect missing functionality) and the drawbacks of relying on word-of-mouth for specifications.
- The specification of the project is continuously refined as the system is developed. How do you refine what is not documented? Can test suite be really a substitute with that respect?
- XP is supposed to work with programmers of ‘ordinary skills’. Can you rely on such programmers to have a total control and mastering of complex system specifications and designs?
- XP makes design refactoring a part of ‘everybody’s daily business’. Are test suites and oral communication enough to guarantee that all team members have intellectual control over the design?

So how does all this fit with standard Object-Oriented analysis and design approaches? Are the XP ideas compatible with them? Is all this just one more software engineering fad, a reaction to a decade long strong emphasis on the software process and the procedural aspects of software development?

4 So where do we stand?

The main reason why the questions above are so difficult to answer is that they cannot be investigated analytically and require an investigation *in vivo*, with actual analysts, designers and programmers. It requires the empirical investigation of human processes in realistic contexts and settings.

In a context where hard, empirical evidence does not exist, it is natural that expert opinions (sometimes self-proclaimed) then prevail thus leading to outcomes of, over the long term, questionable value. Many articles regarding agile development (e.g., XP) resemble more rambling stories than well-conducted and reported case studies [8]. Their conclusions are often not clearly supported by evidence and conveniently match the author’s viewpoints or commercial interests.

This is why software documentation, being an important practical subject, deserves research programs of its own. I do not mean necessarily working on formal notations but rather

- better understanding, at each stage of development and in well-defined contexts, the needs of software engineers as far as documentation is concerned.
- experimenting with alternative technologies addressing those needs, e.g., the use of OCL contracts during object-oriented analysis.

5 A Research Program

Any research program focusing on software documentation needs to define the following components:

- A context, e.g., OO development with the UML.
- A scope: requirements, Analysis, high-level design, etc.
- A set of hypotheses to be investigated. Examples are: (1) Analysis contracts can help reduce the number of errors introduced in the analysis model and can better convey the intent of system operations to maintainers, (2) Sequence diagrams clearly model the relationship between the system structure (class diagram) and end-user functionalities (use cases). It therefore provides the maintainers with a rationale for the design of the system.
- An operational plan for a series of experiments allowing, step by step, the convergence towards credible answers. It is unlikely that one experiment will do the job and studies with different designs are usually necessary to prevent threats to validity.

Hypotheses may involve a number of dependent variables that may be affected by documentation. Typical variables are related to the introduction rate of defects, the capacity

of people to precisely and clearly understand aspects of the system, and effort saved on subsequent phases of development. With respect to maintenance, the difficulty is that the benefit of good documentation can only be observed over time, as new releases of the software system are produced. Such studies (referred to as longitudinal) are usually complex to carry out and sustain over a long period of time.

6 Acknowledgements

This work was partly supported by an NSERC operational grant. The author is director of the SQUALL laboratory at Carleton University. The SQUALL lab focuses on software quality engineering (assurance, testing, verification, and measurement) and further information can be found on the following web site: <http://www.sce.carleton.ca/Squall/>.

7 References

- [1] K. Beck, "Extreme Programming Explained", Addison Wesley, 2001
- [2] Bruegge and Dutoit, "Object-Oriented Software Engineering", Prentice-Hall, 2000.
- [3] Coleman et al., "Object-oriented Development – The Fusion Method", Prentice Hall, 1993
- [4] A. Forward, "Software Documentation – Building and Maintaining Artefacts of Communication", Masters thesis, University of Ottawa, 2002
- [5] Robillard, Kruchten, D'astous, "Software Engineering Process", 2002
- [6] Mitchell, McKim, "Design by Contract, by Example", Addison Wesley, 2002
- [7] Warmer, Kleppe, "The Object Constraint Language", Addison Wesley, 1998
- [8] Yin, "Case Study Research – Design and Methods", 2nd edition, SAGE publications, 1994