



(Paper) Documentation Considered Harmful

Bruce Taylor

Introduction

It's been part of programmers' training since the earliest days: "Documentation is part of the product!" But our current ways of documenting requirements, architecture, design, and code are vestiges of earlier eras, when word processors and virtual paper were the only choice for communicating ideas. But consider the disadvantages of paper-based documentation:

- The written word is not always the best way to convey complicated information, especially for engineers who aren't comfortable writing large documents.
- It can be complex and costly to produce all the paper documentation required for a large system.
- Virtually no one reads with attention any document over fifteen pages in length.
- It is difficult to keep virtual paper documents up to date with the systems they describe, and document maintenance can be a large component of system costs.
- The investment in virtual paper documentation can be a barrier to refactoring components, even when they need it desperately.

And yet, the need to describe the purpose and structure of the software to people than the developers remains, so how can we get the benefits of documentation without the drawbacks of paper?

Functional Specifications

Software projects typically start with producing a Requirements Specification, which describes to the engineers the problem to be solved, and a Functional Specification, which describes back to the product managers the proposed solution. Both of these *documents* can be very large, very broad in scope, and very detailed; and they are almost never complete, consistent or correct.

But suppose that we replaced the whole process of requirements definition with a more incremental, inclusive approach. If a user team of product managers, QA staff, and customers sat down with engineers, the combined team could solve both specification problems at the same time. But instead of writing a massive Requirements specification, consider writing individual requirements as sentences or short paragraphs on story cards. And instead of writing elaborate descriptions of system behavior, think about using pictures: UML use case diagrams. Not only are these notations easier to write, they are easier to modify because the requirements are small and modular.

The Functional Specification can be expressed with feature walkthroughs, which can be described either on feature story cards, or through paper prototypes, or through mockup systems with partial UI and skeleton data and functionality. All three of these formats have the advantage of being incremental, easy to modify, and inherently unambiguous.

Architecture Documentation

Large systems often have massive architectural documents with complete descriptions of components, interfaces, and sequencing relationships. And all of this information is valuable, but only for initial architectural design and review. After the architecture is established, the only document of any value is a simple architectural overview to help a new architect understand the general structure of the system. Detailed package specifications should be generated from the code itself, rather than being laboriously written and updated by hand.

Design Documentation

Similarly, design documentation is very useful for reviewing the design of a component and ensuring that it fits into the architecture, but is not very useful to a programmer trying to learn how a component is structured: most programmers would rather dive into the component and learn from the code how it works.

The exception to this assertion that design documentation is not useful after the design review is the documentation of component interfaces. But this information should not be written in a virtual paper document, it should be kept as close to the code as possible, in a JavaDoc comments (Java) or in the .h (C/C++) header file so that it is easier for programmers to keep it up to date.

Code Documentation

One of the first lessons that novice programmers learn is the admonition, “You have to comment your code!” And yet, most experienced programmers find inline documentation to be irrelevant, annoying, or often wrong. A better investment of a programmer’s time is to concentrate on transparent coding: choosing

meaningful names, using well-known design patterns, keeping methods short and coherent, and providing an occasional hint to help the reader over a complex algorithm.

Conclusion: What’s Left?

Documentation *is* important, but it should be used in ways that help the software developers, rather than getting in their way:

- Choose a good notation for the documentation: use English as a last resort.
- Throw the documentation away after it has served its purpose.
- Keep the documentation as close to the code as possible.

About Bruce Taylor



Bruce Taylor is the principal of CoachingProgrammers.com, an executive coaching firm located near Boston, Massachusetts. Bruce helps software organizations of all sizes to create low-stress, supportive, adaptable working environments, so that the engineers, leaders, and managers can work as effectively as possible. He provides executive coaching for senior managers who are creating superior organizations, management coaching for technical leaders who are adapting to new agile practices, and individual coaching for engineers who are upgrading their skills. Bruce has a Masters in Computer Science from Duke University, a Masters in Community Psychology, and a Certificate in Job Stress and Healthy Workplace Design, both from the University of Massachusetts.