28TH ANNUAL
PACIFIC NW
SOFTWARE
QUALITY
CONFERENCE

OCTOBER 18TH – 19TH, 2010

ACHIEVING QUALITY IN A COMPLEX ENVIRONMENT

Conference Paper Excerpt from the

CONFERENCE PROCEEDINGS

Document Your Software Project

Ian Dees
ian.s.dees@tektronix.com

Abstract

Software projects are growing in complexity. We're expected to get more done in less time than we did last year. It's vital to come up to speed quickly on any given body of code, whether it's a new third-party library or a neglected legacy subsystem.

Many projects lean on API documents that have been automatically extracted from source code comments. Other teams churn out elaborate diagrams that explain every detail of the architecture. While these can both be helpful as a reference, they do little to answer the basic questions about the code: What's this for? What file should I look in first? What's this project's equivalent of a “Hello world!” program?

In this paper, we'll use the metaphor of a magazine article to think about the ways you can help anyone come up to speed with a code base, whether it's a new hire today or you, five years from now. Getting the emphasis and level of detail right are crucial. Everything else, especially the specific choice of tools, is secondary. Even so, we'll spend some time looking at a couple of open-source software packages that may be of help.

Biography

Ian saw his first Timex Sinclair 1000 over 20 years ago, and was instantly hooked. Since then, he’s debugged embedded assembly code using an oscilloscope, written desktop apps in C++, and joyfully employed scripting languages to make testing less painful. Ian currently writes GUI code for measurement instruments as a Software Engineer at Tektronix.

Ian is the author of Scripted GUI Testing With Ruby, and co-author of Using JRuby.

1. Introduction

"Welcome to the team. Here are a couple of bugs you can fix to learn your way around the source code. You'll probably want to start by looking in this subsystem."

Have you heard this before? Perhaps it was followed by a wild goose chase through the code, as you strove to read the mind of the coder who came before you.

Or perhaps it turned out differently. You may have opened the project folder to find a welcome mat of sorts. Someone may have carefully laid out a `README` file giving a quick overview of the directory structure, names of functions to look at first, and even code samples hinting at how to use or extend the code. Maybe that someone was you, several years ago.

In this article, we're going to discuss how programming documentation goes wrong, and what you can do about it. It doesn't take a world-class technical writer to document a software project (and it certainly didn't take one to produce this paper!). It just takes a few times being baffled by an unfamiliar code base, plus a dose of empathy for your fellow developers. The number one question in their minds when they first pick up your library is, "Anything I need to know before I use this?" Let's look at a few of the ways you can provide an answer.

2. What Do You Mean, "Documentation?"

Any kind of technical documentation can benefit from care and attention to detail. In-code commentary, user manuals, architectural walkthroughs, and requirements documents should all be written in clear language and kept up to date. On any given project, each of these types is going to come with its own set of constraints on length, level of formality, and assumptions about its audience.

This article focuses on one kind of writing: software library documentation. Your audience will be your fellow programmers, very likely including yourself. The scope is something quite a bit narrower than an entire system—something along the lines of one library or subsystem.

Even the small world of a single software library can contain a vibrant ecosystem of documentation: READMEs, code comments, API references, and more. Jacob Kaplan-Moss, one of the creators of the Django web framework, classifies project documents into three broad types:[1]

1. *Tutorials* give the new programmer enough information to get started.
2. *Topic guides* pursue one aspect of the project in great detail.
3. *References* provide exhaustive coverage.

It's worth taking a look at your software project to see how you're meeting all three of these needs for your readers. This paper will be dealing mainly with the first category: getting-started documentation for new users of your library. You want to lead your reader to an early success, then challenge him with a couple more realistic examples, then point him toward the more in-depth resources that live elsewhere.

Topic guides are crucial as your project's user base grows. For that reason, they're outside the scope of what we're talking about today: how to jump into a new project (as a reader or writer) without getting bogged down. References are tangentially related to the discussion, so we'll touch on them in a couple of places.

[1] http://jacobian.org/writing/great-documentation/what-to-write

2.1. Thought Experiment

Think back to the last time you visited an unfamiliar subdirectory of your source tree for the first time. Perhaps you were assigned to help another teammate develop a feature, or simply found an interesting name in the project directory and wondered, “What’s in here?”

How easy was it to learn the purpose of that particular subsystem? Was there a `README`, or was it cultural knowledge among the team what that directory was for? Was there any documentation at all in the directory, or were you expected to look in some central location for everything?

If you did find documentation, how helpful was it? Was it boring or engaging? Did it give a quick overview of the more important files in the directory? Was there a “Hello world” code example you could try out? If so, was the example up to date?

Keep your answers to these questions in mind as you read the next couple of sections.

3. Document Misfits

People write documents for good reasons. (And not just “because management said to.”) You can probably think of several right off the bat: pride in one’s work, sympathy for one’s co-workers, or even simple self-preservation.

What goes wrong, then? Assuming authors' hearts are in the right place, we can imagine that a document that fails us may just be the wrong kind of document for the task. Let’s look at a few ways in which a software library might not be getting the textual attention it deserves.

3.1. Invisible Ink

A subsystem might have no documentation at all. This scenario is distressingly common, and to my embarrassment I’ve been party to it as well. I’m going to take advantage of the bully pulpit afforded to paper authors, and speculate on some of the causes.

We skip documentation because we run out of time. We skip it because we’re self-conscious about writing narrative text. We start a tiny new project and think, “This thing’s way too small to bother documenting; just look at the source code!” We get frustrated with flaky writing tools and bureaucratic workflows.

As we’ll soon see, there are a few simple things you can do to address nearly all of these causes of missing documentation.

3.2. Ghost Writer

A close cousin of the invisible document is the ghost-written document; that is, the automated help pages generated from the declarations (and comments, if there are any) in your source code. It’s certainly fine to use a tool to help you generate finished documentation. But tools can’t create something out of nothing.

Kaplan-Moss views auto-generated documentation as “almost worthless. At best it’s a slightly improved version of simply browsing through the source….” This is a justifiable reaction to visiting a project’s website in search of documentation, and seeing nothing but a raw list of class and method names. Hand-written text, he argues, is much more useful to the reader.

The contrary point of view is that tools like Javadoc (or its equivalent for your language) can encourage you to do a thorough job commenting all the parts of your code: function behavior, correct use, gotchas, and so on. That’s exactly the kind of information a reader is looking for when he moves on from the tutorial in search of a reference guide.

In the spirit of unifying the two camps, consider the fact that most of these tools can import external hand-written text files into their generated documents. When you've spent hours writing a detailed overview of your project's API—and hours more explaining how to use every public method of every class—does that even count as automatically generated documentation any more?

A successful project maintainer views writing documentation as an intensely (pardon the pun) manual effort, no matter whether or not the end result happens to have been generated with help from a source tool. For an example of a document maintained separately from the source code, pick any given section of the Python standard library.[2] For an integrated document that pulls in some information from the source tree, see the reference for the Haml markup language.[3]

We'll return to the subject of source comments later on in the article.

3.3. One Document to Rule Them All

Some shops demand that one file or directory be the gateway to all understanding of the project. Hardware layouts, software architecture, UML diagrams, elaborate class hierarchies, and everything else gets dumped into this one location.

It's easy to see why this approach is so common. If every fact about the system lives in one place, then one can turn to the master document for any answer—in theory. The problem is that these would-be museums quickly become mausoleums. Those elaborate diagrams that got produced not out of any real need, but because someone thought the team should have them? Dumped in the burial ground and forgotten.

The problem is not just that we might have trouble finding a specific piece of information. After all, tools like `grep` and desktop search can help us get through cluttered directories. The problem is that there are pieces of information that we don't even know exist. Too often, I've struggled with some API because the advice on how to use it was buried in a folder I'd never seen.

The exception to this anti-pattern seems to be wikis. In theory, these should be just as susceptible to information imprisonment as any other centralized documentation storage. But in practice, I've seen successful project wikis that made it easy to retrieve project knowledge—not just answers to the questions I had, but also answers to the questions I didn't know to ask. Perhaps the ease of hyperlinked, ad-hoc structure is what makes the difference.

3.4. Ancient Scrolls

Documents need to be kept up to date. It feels silly even having to say this. But I've definitely encountered libraries whose example code no longer ran unaltered, or whose descriptions didn't match the API any more. I'll bet you have, too.

People don't come into the office and decide to put off documentation updates. Maintaining the docs just doesn't become part of our habits, for the same reason that creating them doesn't become part of our habits. Every little barrier in the way of writing looks like an impenetrable wall when we think about doing it over and over again in maintenance mode.

It follows that any little thing we can do to stop ourselves from dreading writing and just *write* will have a big payoff. I'd like to discuss several such things you can do in service of this goal. To tie these various activities together, I'd like to propose a metaphor for software documentation that carries warm, familiar connotations for most of us.

[2] http://docs.python.org
[3] http://haml-lang.com/docs/yardoc/file.HAML_REFERENCE.html

4. The Magazine Metaphor

When you were cutting your teeth as a coder, did you ever subscribe to *Dr. Dobb's Journal*? How about *BYTE*, or *COMPUTE!*? Remember the thrill of diving into some brand new technique, and being presented with real, live code samples that you could actually type in and run right on the spot?

It may be a bit unrealistic to expect users of our libraries to feel quite that level of excitement about software that's just part of our day jobs. Even so, a well-written project tutorial has a few things in common with a short programming article in a magazine. Let's look into a few of these features.

4.1. One Sitting

The engineer trying out your library doesn't have all day to read an entire binder's worth of writing. There's a place for exhaustive coverage of a subsystem: the reference material. Instead, give your reader a few pages that can be printed out and read over a cup of coffee.

4.2. Working Examples

One of the most heavily leaned-on subsystems in our office has a well-loved usage guide. Even though the document doesn't score 100% on all the counts we've discussed (e.g., being easy to find and up to date), it's always worth the effort of tracking it down and reading it—because it does such a thorough job explaining how to write code for this subsystem.

In one sixty-line example that fits on a single page, the author is able to demonstrate all the library's major features. In the adjoining text, he explains what each single line of sample code does. After refreshing our memory of how that one example works, we usually have enough information to jump right in and code. For those rare cases where we need a little more, he's provided a thorough API reference in the back.

The best thing about this document is that it's like a journey through the mind of its author. Understand him, and you understand the code.

4.3. Curated Code Excerpts

The most enjoyable articles show more than just one "Hello world" example. They compare and contrast what code might look like before and after applying the technique they're describing. They peel back the API and show some facet of the underlying implementation.

I sometimes struggle with how much code to show in an introductory document. Too little, and the reader won't know where to turn after getting the first example working. Too much, and the whole thing degenerates into a parade of excerpts that don't have enough context to be meaningful. This is where the "magazine article" theme comes in most helpful to me. Articles show off a small series of curated code examples that capture the essence of a library.

4.4. A Real Page-Turner

With all this talk of keeping things short, you're probably wondering how to fit in everything the reader might need to know about the software. It's important to provide good coverage, but an introduction is not the place for that. The introduction absolutely *is* the place, however, for telling your reader where to get more information: a series of in-depth guides, full API documentation, a network of wiki pages, and so on.

5. How To Get There From Here

There's no need to go overboard with the magazine metaphor. You don't need fancy formatting, callouts, author bios, personality quizzes, or intrusive ads. Simply use the idea insofar as it helps you get the job done, and discard it as soon as it loses its usefulness.

Assuming you do find a few parts of the idea useful, how can you incorporate them into your workflow?

5.1. Keep It With the Code

The further you have to reach from your source tree to maintain your documents, the less often you're going to maintain them. It's fine to have a central place for the big stuff: giant architecture diagrams (on second thought, maybe you could just do without those), requirements documents, histories of design decisions, and so on. But your readers need *something* right next to the source code to get started—and so do you.

You may be thinking, "But we don't want to clutter up the source tree! There's no conceptual room for a giant `README`." Precisely. Keeping this material right next to the source should provide an upper bound on its size and complexity.

5.2. Use Tools That Fit Your Hand

We're all under schedule pressure. How do we make sure that the task of creating and maintaining the documentation isn't the first thing thrown out the window as the deadline draws near? Choosing a lightweight format is one way we can trick ourselves into doing the right thing. If we think of our project guide as a magazine article, it seems less difficult and more approachable.

Choosing a writing tool is another part of this psychological game. Use the tool that's not going to become an excuse for you to skip out on your writing duties. If you find yourself thinking, "Now I have to wait all day for the word processor to launch," you may prefer a plain-text format and the comfort of your favorite programming editor. If the prospect of memorizing obscure bold / italic codes annoys you, a dedicated writing tool may be the best fit.

5.2.1. Plain Text and Its Relatives

Don't underestimate the power of plain text. It can be read from the command line with no additional tools, sent around as e-mail, and tracked usefully with revision control tools. This includes the "humane markup" languages, such as Textile, Markdown, reStructuredText, and AsciiDoc. Using one of these formats means just writing like you'd normally do, and then adopting a few simple conventions for section titles, code excerpts, and so on. For example, here's how a README file might begin in reStructuredText: [4]

```
Halt-o-Meter
============
Welcome to the Halt-o-Meter! This software reads the source code of
your program and tells you whether or not it will run to
completion (and they said it was impossible!).

Save the following code in ``example.c``::

  int main() {
    for (;;);
    return 0;
  }
```

[4] http://docutils.sourceforge.net/rst.html

```
Now, run Halt-o-Meter::

  C:\> haltometer example.c
  Checking... example.c does NOT halt.
```

The other text formats look similar to this. I chose reStructuredText for this example because it's the one used by Sphinx, the tool behind the Python library documentation. [5] Sphinx is designed for writing books and articles, so it's very close to the sweet spot for introducing readers to a software library. If you happen to be writing about a Python project, Sphinx can link to your generated source documentation.

5.2.2. Machine-friendly Markup

Formats such as DocBook, LaTeX, HTML, and RTF are also text-based, but typically involve inserting instructions for the typesetting among the words of your document. To one degree or another, they're more difficult to write by hand than plain text (LaTeX is the easiest in this regard). This added complexity pays off, though, when it's time to ask a machine to do something to your text—such as print it, translate it into another format, or automatically update snippets of your source code.

Incidentally, the AsciiDoc format described earlier generates DocBook XML behind the scenes.[6] So you can get the best of both worlds: the ease of plain text, and the feature set of the entire DocBook toolchain.

5.2.3. Word Processors

The tools we've talked about so far have been pretty programmer-oriented. A few of my teammates prefer a change of scene when they're working on documentation. They'd rather see formatting changes while they're interacting with a document, rather than having to wait for a conversion to PDF.

So they fire up a word processor and start writing. They use styles and templates to avoid getting mired down in micro-decisions about the document's look. And they produce readable, useful documentation.

One thing to watch out for if you go this route is that you're going to have to work a little harder to keep your code examples up to date.

5.3. Keep the Code Up to Date

Having stale code examples that break for your first-time users is a recipe for frustration. If your library implements a fairly stable API, especially one that's defined in a standard somewhere, you may not have to worry about this much—just go ahead and paste your source examples into your document.

But if your project's programming interface changes frequently, you'll need to re-run your examples and keep the versions inside your document up to date. This task is easier with the text-based formats; most have commands that say in effect, "insert the source code from this external file here." For the ones that lack this feature, you can use a templating engine such as the Ruby-based eRubis or the language-independent m4. [7] [8]

Automatic updates are a little trickier when you're using a word processor. If you're only responsible for documenting a couple of projects, you might just add a recurring task in your calendar to try the snippets manually and make any needed fixes. Any more than that, though, and the manual approach will get old quickly. You can use your word processor's linking feature and store your snippets in external files (which are easier to test automatically).

[5] http://sphinx.pocoo.org
[6] http://www.methods.co.nz/asciidoc
[7] http://www.kuwata-lab.com/erubis
[8] http://www.gnu.org/software/m4/m4.html

You can also just update the document by writing a program. Most word processors have either a scripting API or an XML-based file format that's easy to generate automatically. [9]

5.4. Provide a Path from "Hello world" to Mastery

An engineer who stumbles across your project directory has a number of questions, which almost certainly include the following:

- What is this thing?
- How do I get it and all the dependencies installed?
- What's the equivalent of "Hello world" for this system?

Kaplan-Moss reminds us to "Be easy…. But not too easy." You want your reader to get an early success, then challenge him with a couple more realistic examples, then point him toward the more in-depth resources that live elsewhere.

6. Real Life

All this talk of misfits and magazines is all well and good, but how do these ideas apply in the real world? Do real projects use these techniques? What specific tools come in handy? And how are we software developers supposed to find the discipline to write?

6.1. An Achievable Example

Jumping into real-world examples carries a certain amount of risk. Reactions to any featured project will range from, "How can you praise the documentation for *this*?" to "You should have covered Project X instead!" Choose too many examples, and the whole discussion devolves into a laundry list.

With that in mind, let's shoot for a realistic example. We don't need to see the most beautiful, exhaustive documentation in the world. We need to see the kind of thing that you or I could reasonably produce on a tight time budget.

The tutorial for the Sass stylesheet language fits these criteria.[10]. Sass, by the way, makes it easier for web developers to implement Cascading Style Sheets (CSS) designs. (I have no connection to Sass, other than as a user.)

Sass's tutorial flows like a magazine article from an old coding journal. It starts with a simple "Hello world" equivalent, then demonstrates a few features in the order that readers are likely to wonder about first. It mentions more advanced options in passing, but wisely stays on track. You can work your way through the whole tutorial over a lunch break.

Many of the hints in the tutorial link to deeper coverage on the library.[11] This next level isn't quite the exhaustive feature reference from Kaplan-Moss's hierarchy of documentation. But it does provide hand-written discussion, options, and examples for all the major features.

The reference also contains a list of classes and methods, but doesn't leave you at their mercy. The narrative material contains several links to the classes you're likely to need to know about first when you're slinging code. The method descriptions may have been converted to HTML with the help of a tool, but they were written by a human being.

[9] For a related example of updating a presentation programmatically, see http://github.com/undees/snippetize.
[10] http://sass-lang.com/tutorial
[11] http://sass-lang.com/docs/yardoc/file.SASS_REFERENCE.html

How did the Sass team get this documentation done? They put it right into the source tree! The text is less likely to fall out of date than if it lived in some separate location. It's easy for project newcomers to view and contribute to the documentation, because it's in Markdown format and can therefore be read and written without any special tools.

6.2. Speaking of Code…

Earlier, we touched on keeping documentation with the code. In some cases, this means keeping parts of it *in* the code. It's possible to generate API documentation from source files. But it's also easy to over-rely on tools, and end up with a document that's not much more help than just reading the source code.

What level of detail is appropriate for documenting a class or a function? No single answer will suit all projects. Here are a couple of general themes that have been true on most of the project teams I've been on.

First, we have to start somewhere. Don't let the fact that some legacy class has 30 methods stop you from documenting the first one. Just think of the answers to a few simple questions about the file you're editing. Which function will project newcomers need to call first? Which one do experienced developers frequently get wrong?[12]

Second, shy away from those giant comment templates that have required sections with names like `NAME`, `DESCRIPTION`, `PURPOSE`, `ARGUMENTS`, and `REVISION HISTORY`. Your tools should pick up the function's name and parameters from its argument list.[13] Instead, just write free-form text. That'll keep you writing, and it'll save your readers from having to slog through parameter descriptions like "second_number: the second number to be added."

If we return to the example of the Haml project, we see that their source code for public APIs contains these kinds of comments. Consider their `Filters` module.[14] The file begins with a three-paragraph description, complete with a simple example. By opting for something more than just a single sentence, they've deftly avoided the trap of writing things like, "The `WidgetManager` class manages widgets." They've also steered clear of the one-size-fits-all approach. Some functions are simple enough just to need a two-liner. Others get the full treatment, with usage examples and discussions of valid parameter values.

6.3. Story Time

You've no doubt noticed that I've made it most of the way through this article without advocating any specific tools. That's by design. Not only is every developer different, but every project has its own needs.

So rather than recommending one set of tools for all situations, I'd like to tell the story of one situation, and how the choice of tools fell out of those circumstances.

A few months ago, our software lead and I were designing a new feature. We were in uncharted territory: we barely even had words for the things we were talking about. And we were supposed to be writing the requirements document for this feature.

We grabbed a conference room, sketched on whiteboards, and typed madly into our laptops. At this point, there was no conscious decision to use a particular format or processing tool. We were just using the tools that leapt into our hands first: the same programmer's text editors with which we write our code.

As the document took shape, a writing style emerged. We were subconsciously using the same kinds of text conventions we'd used for years for e-mail: dashes as crude underlines for section titles, asterisks for bulleted lists, and so on.

[12] This is also a good way to identify parts of your API that need to be redesigned. If people frequently confuse the fifth and sixth integer parameters to your function, perhaps the function should take fewer arguments.
[13] Dynamically typed languages may need a few extra hints.
[14] http://github.com/nex3/haml/blob/master/lib/haml/filters.rb

Once we were done with capturing all those thoughts, it was the work of a few minutes to transform the ad-hoc text format into one of the "humane markup" languages described earler. In this case, I chose the AsciiDoc format for its ease of making PDFs.

Because AsciiDoc generates DocBook output, any DocBook toolchain can be used to make the PDF. I happen to like Remko Tronçon's DocBook Kit, a toolbox combining a nice set of styles with a Makefile that downloads all the processing tools for you.[15] It's the work of a moment to grab a fresh copy of this kit, extract it into your project directory, adapt the sample Makefile, and type `make pdf`.

When we first showed the document to our teammates, we wondered if we'd catch flak for the fact that we weren't (yet) using the official company documentation template. We don't mind using those sorts of things, by the way—but there's a time and a place for everything.

To our delight, people appreciated the content of the writing. They said that having this kind of "theory of operation" helped them understand the feature better than a fill-in-the-blanks template would have. Of course, when we reach the state of the project where people will expect the templates, we'll preserve the narrative text.

6.4. The Discipline to Write

Stories about other other projects are all well and good, but they can only go so far. At some point, you'll be picking up your own keyboard and writing. And that's something that all the exhortations, analogies, tools, techniques, and testimonials can't help you with.

How do we as developers cultivate the discipline to write? Advice along these lines usually amounts to tautology. "The only way to write is to write," goes the line of thinking. This is sort of like saying, "The only way to diet is to eat less." Well, sure, but there are ways dieters and writers can help themselves.

For me, it comes down to removing excuses not to write. If I feel intimidated by the prospect of writing a huge document, impatient with word processors, and lost looking for templates, that gives me three chances to put off writing and go do something else.

That's why the main thrust of this argument has been: think of documentation in a way that makes it seem less daunting, choose tools that don't give you any excuse to stop using them, and keep your documents close by. If you do that, I'll be happy—not least because I'll find a well-explained project if I should ever have the good fortune to work on a project with you.

7. Closing Thoughts

If you're the maintainer of a small- to mid-sized software project, the programmers using your software deserve great documentation to help get them started. One day, one of those programmers may be you! For your sake and theirs, treat yourself to good documentation.

It's easy for writing to get lost in competition with other, more urgent, activities: testing, coding, debugging, planning, and so on. You've got to get rid of any excuse not to write. That means choosing whichever tool is going to stay out of your way. It means setting realistic expectations of what you're trying to accomplish.

If it helps, by all means choose a mental picture to rally behind. You may find inspiration in the old computing magazine articles that once inspired you to try out someone else's code.

[15] http://github.com/remko/docbook-kit