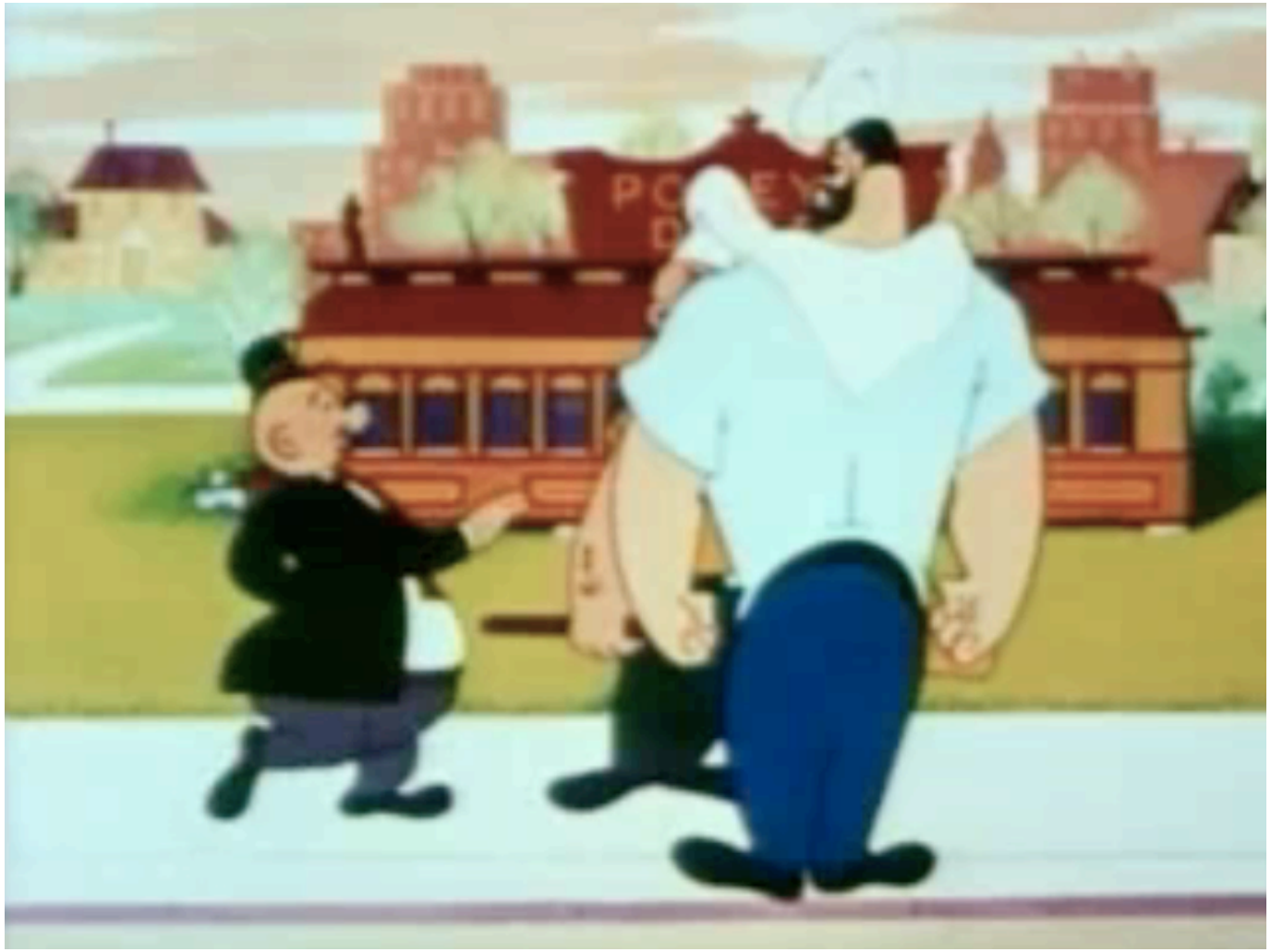
Document Your Software Project!

ian.s.dees@tek.com

Hi, I'm Ian. I'm here to talk about software documentation, with an emphasis on library documentation.

The problem with a title like this is that it's something everyone says we should do. My co-worker Tim calls these kinds of goals "motherhood and apple pie." So how can we get past a general feel-good sentiment and arrive at something practical?

“Let’s You and Him Fight”



One thing that won’t work is swooping in as a self-proclaimed expert, with no stakes in the game, and telling everyone what they should be doing. That’d be like picking a fight between two strangers.

Why We're Here

Document Misfits

The Magazine Metaphor

How to Get There

A Few Tools

Instead, I'll begin with the story of how this talk came to be, and then we'll get into some horror stories of documents gone awry. Next we'll talk about some ways to trick ourselves into doing the right thing. Finally, I'll mention a few tools that have worked for me, though please understand that you should use the tools that work for you—which may be a completely different set.

Why We're Here

PNSQC

PNSQC is an intimidating conference. Not only are you respected engineers, but you're also friends and neighbors. It took me a few years to work up the nerve to submit a talk. With this year's theme, I felt like I had something to say.

Achieving Quality in a Complex Environment

The theme of the conference this year is Achieving Quality in a Complex Environment. What associations does that trigger? Maybe something about testing? That'd be doable if I'd spent the year doing a peer-reviewed effectiveness study (I haven't). Besides, testing is already well-covered here.

The New Guy

So what has this year been about? Well, I joined a new team at work a couple of years ago, making me the new guy, or at least one of 'em. Has anyone here joined a new project or team recently? What was it that helped you get oriented?

Cultural Knowledge

For me, it was learning bits of cultural knowledge that helped the most. Not just coding standards or how to perform common tasks, but a real feel for how we do things here. Like any long-lived group of collaborators, my teammates had a rich oral storytelling tradition.

Documentation

But this oral tradition has a bias toward preservation. There's a real emphasis on doing the legwork to find out how a system works ("Ask Erik where to start,...") and then make sure that others benefit from your detective work ("... then add it to the docs.")

Culture / Team

Artifact / Project

Fragment / Subsystem

That kind of knowledge preservation wasn't just done at the team level. There were, of course, stashes of documentation for specific projects: architecture diagrams and so on. What I really want to talk about today, though, is even narrower: the scope of a single subsystem or software library.

Software Spelunking

Does anyone go software spelunking? Browsing through random directories you can find on the server and wondering, “What’s this do?” What’d you find?

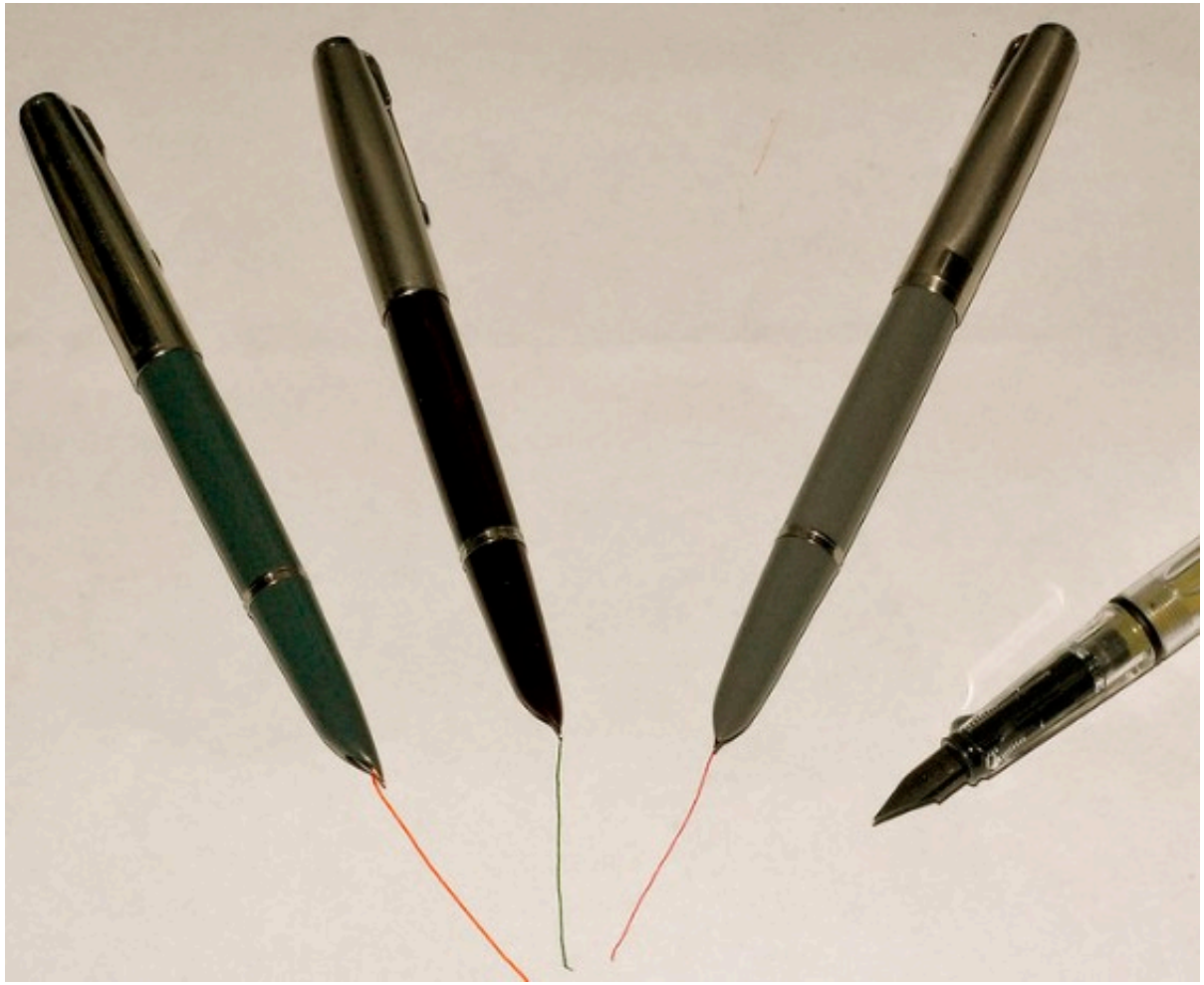
For one particular subsystem, I found a welcome mat of sorts. In a single page, the author laid out a sixty-line example that showed how to use most of the features of the library. Over the next few pages, he explained the example line by line.

Three Types

The welcome-mat document was basically a tutorial, the first of three document types that Jacob Kaplan-Moss (the Django docmeister) says that big projects need. The other types are low-level API references and topic guides (the layer in between). Today, we'll be talking mostly about tutorials and references.

Document Misfits

Invisible Ink



What kinds of things have gone wrong with documentation in your experience? One of the most common things I've seen is projects with no docs at all.

Ghost Writer

The second is relying solely on documents generated automatically from source code.

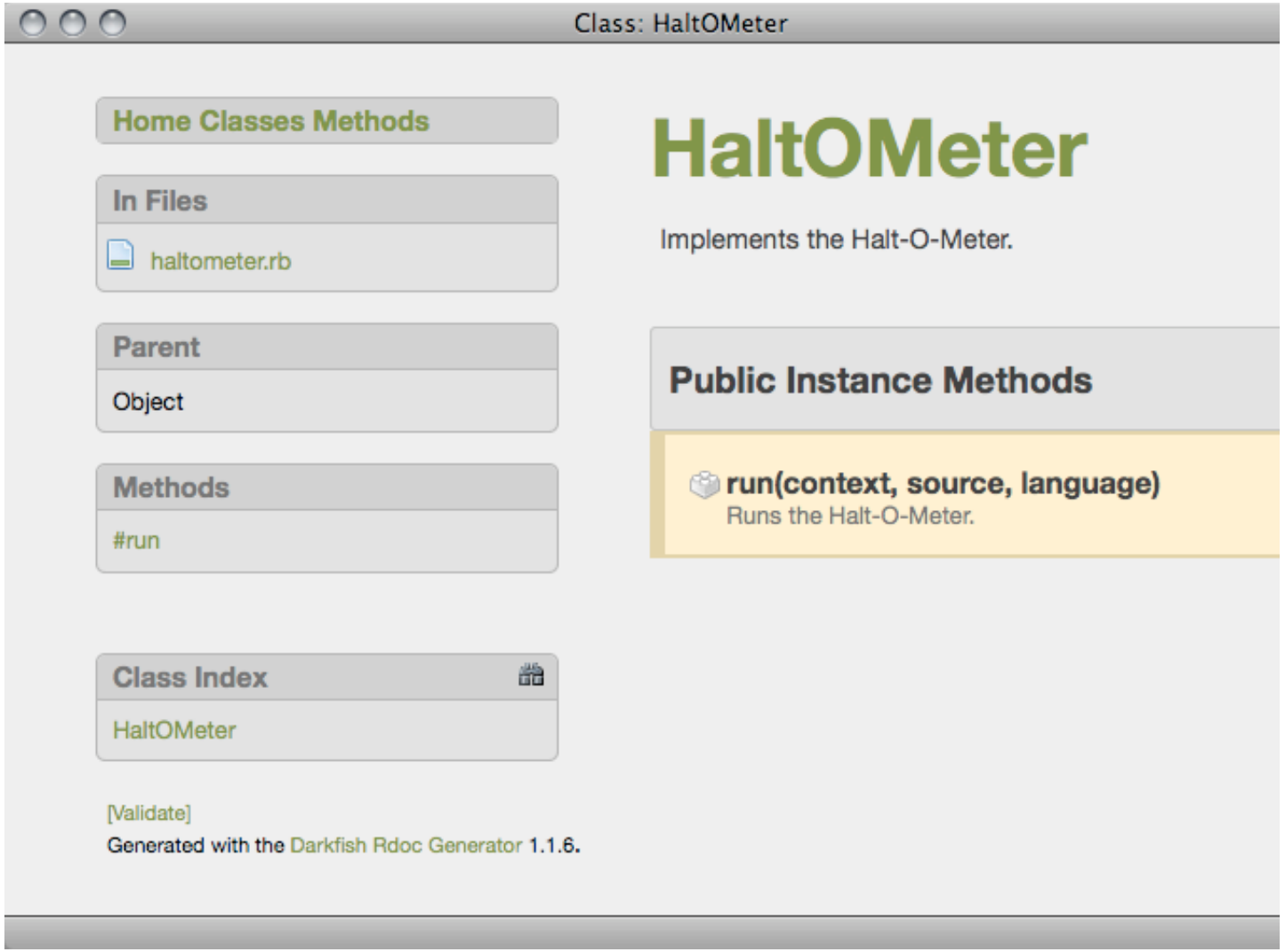
Auto-generated documentation is worse than useless: it lets maintainers fool themselves into thinking they have documentation, thus putting off actually writing good reference by hand.

Jacob Kaplan-Moss

Kaplan-Moss is particularly tough on this kind of documentation, and with good reason. Just to pick on one of my favorite languages: a lot of Ruby projects link straight from the home page to a sea of class definitions with no information on what each class is for or where to start.

```ruby
# Implements the Halt-O-Meter.
#
class HaltOMeter
  # Runs the Halt-O-Meter.
  #
  def run(context, source, langua
  end
end
```

Consider this example. The source comments don't tell us anything we couldn't have gotten from the class and method names. For instance, they don't tell us what these parameters mean.

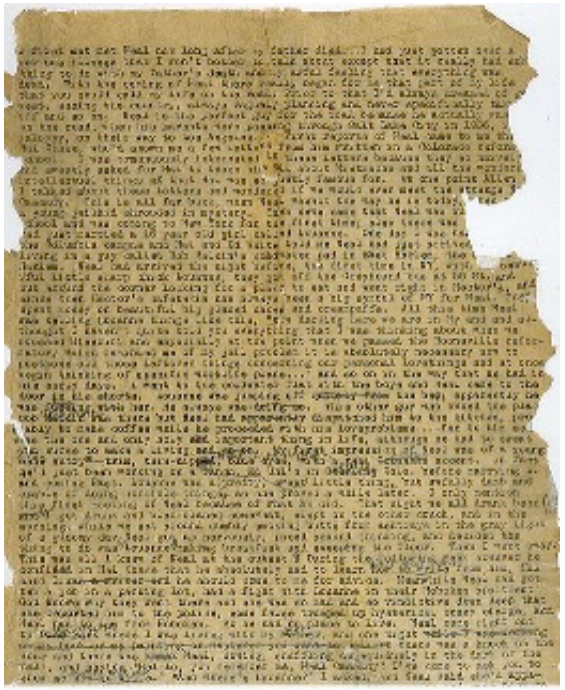


And the result is a beautiful HTML page (complete with validation link) with no content. The fix is twofold: write enough in the source comments for a reader to know how to use your class, and provide an overview telling him where to start. Both of these solutions are manual; even if you're generating docs automatically, you still need to do some writing yourself.

One Document to Rule Them All

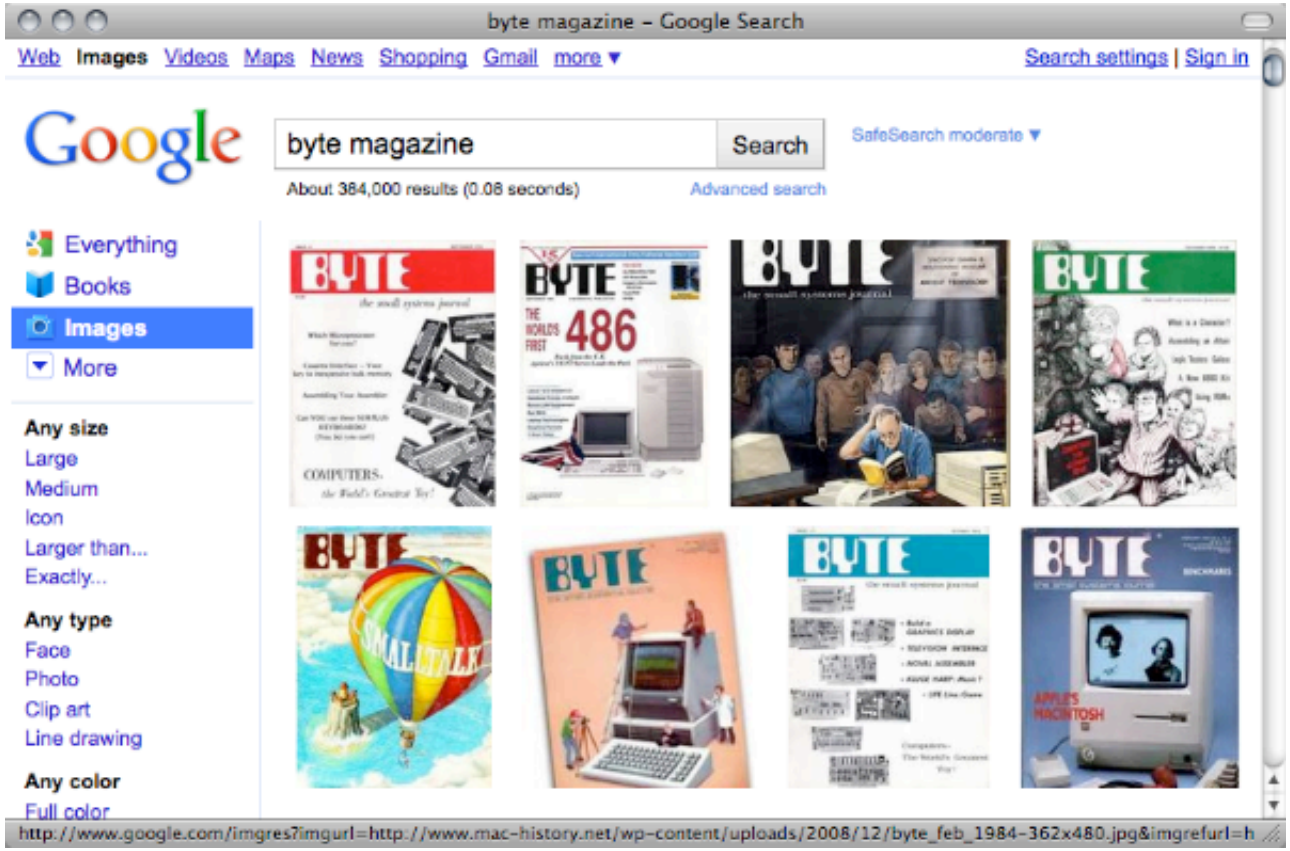
At the other end of the spectrum, we have overdocumentation. I've seen people throw a mass of inscrutable architecture diagrams and outdated requirements documents into a central binder, far away from the source code. These can be informative as a historical record, but they're seldom helpful when it's actually time to write software.

Ancient Scrolls



Documents go out of date, especially ones that are kept in a central stash that no one remembers to look in. Sometimes the text is all right, but the source examples stop running with the latest software. Later, we'll talk about ways to keep your example code tested and updated.

The Magazine Metaphor



Now that we're all depressed, let's turn to an image that has warm, fuzzy connotations for a lot of developers. What was your favorite computer magazine when you were cutting your teeth as a coder? *COMPUTE!*, *BYTE*, *DDJ*? What did you like about these?

One Sitting

I liked the fact that you could read and article over a lunch break and come away with exposure to a brand new language or technology.

Working Examples

The programs were listed right there in the magazine, encouraging you to type them in and learn by doing.

Curated Code Excerpts

The listings were carefully chosen to explain some key facet of the subject.

Short Form

Not only are these compact articles not intimidating to read, they're not intimidating to write. "Write a short articule introducing the new subsystem" is a lot less daunting than, "Document the entire system."

How to Get There

Keep It With the Code

So how might those aspects of magazine articles apply to library documentation? One thing we can do is keep a README and a few working examples right there in the source code. It's easier for newcomers to find it, and it's easier to remember to keep it up to date.

Keep It Up To Date

Speaking of keeping it up to date, you might even add compiling and running the examples as part of your build. Test-driven development meets a README!

Provide a Path

from “Hello World” to Mastery

New users of your library will ask, “What’s the Hello World equivalent for this system?” Once they’ve got that down, they’re likely to want to know where to go next. This is where you can lead them from the README to other resources like API references, wiki pages, and so on.

Use Tools That Fit Your Hand

As Tim Lister pointed out in this morning's talk, you don't need a process or system to motivate people to do something they already find fun. Granted, some codebases will never be a carnival ride, but we can at least choose tools that won't become an excuse for us no to write.

A Few Tools

Humane Markup

Markdown

Textile

reStructuredText

AsciiDoc

Speaking of tools, here are a few ways to create code-friendly documents. First, there are the so-called humane markup languages. These are like text you'd write in an e-mail message: dashes or equals signs for underlining, stars for bulleted lists, and so on.

```
Plain Text and Its Relatives
^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Don't underestimate the power of plain text. 
the command line with no additional tools, se
and tracked usefully with revision control to
the ``humane markup'' languages, such as Text
reStructuredText, and AsciiDoc. Using one of 
just writing like you'd normally do, and then
simple conventions for section titles, code e
For example, here's how a README file might b
reStructuredText:
footnote:[http://docutils.sourceforge.net/rst

----------------------------------------
include::example.rst[]
----------------------------------------
```

I wrote my paper for this conference in one of these formats, AsciiDoc. Here's an excerpt. As you can see, it's just plain text with a few conventions. Notice that I can tie in source code that lives in external files.

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
============
```

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

```
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;
  }

Now, run Halt-o-Meter::

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

Here's a bit of the PDF that came from this process. The fresh, tested code example from the external file is typeset right into the printable documentation.

Machine-Friendly Markup

LaTeX

RTF

DocBook

HTML

If you're already a whiz in one of the more typesetting-oriented text languages like TeX or RTF, you should absolutely use those. Because they're still somewhat text-based, it's easy enough to use a script to update sections that contain source code.

Word Processors

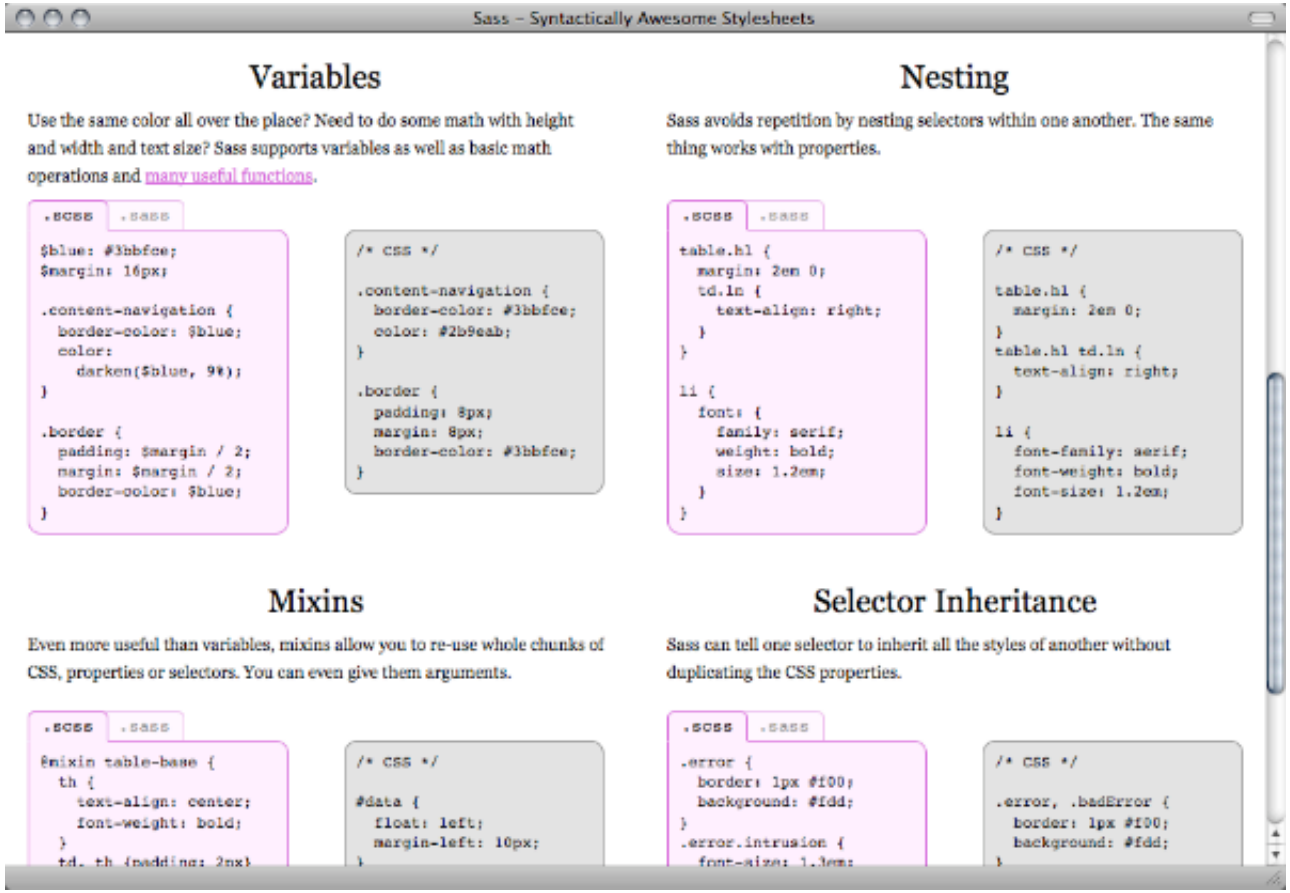
The welcome-mat document I mentioned earlier was a plain old Word file. All of the Big Three—Word, OpenOffice.org, and iWork—can save to XML-based formats, which means it's possible to update code snippets automatically. (See the snippetizer project on GitHub for an iWork example.)

Example: Sass

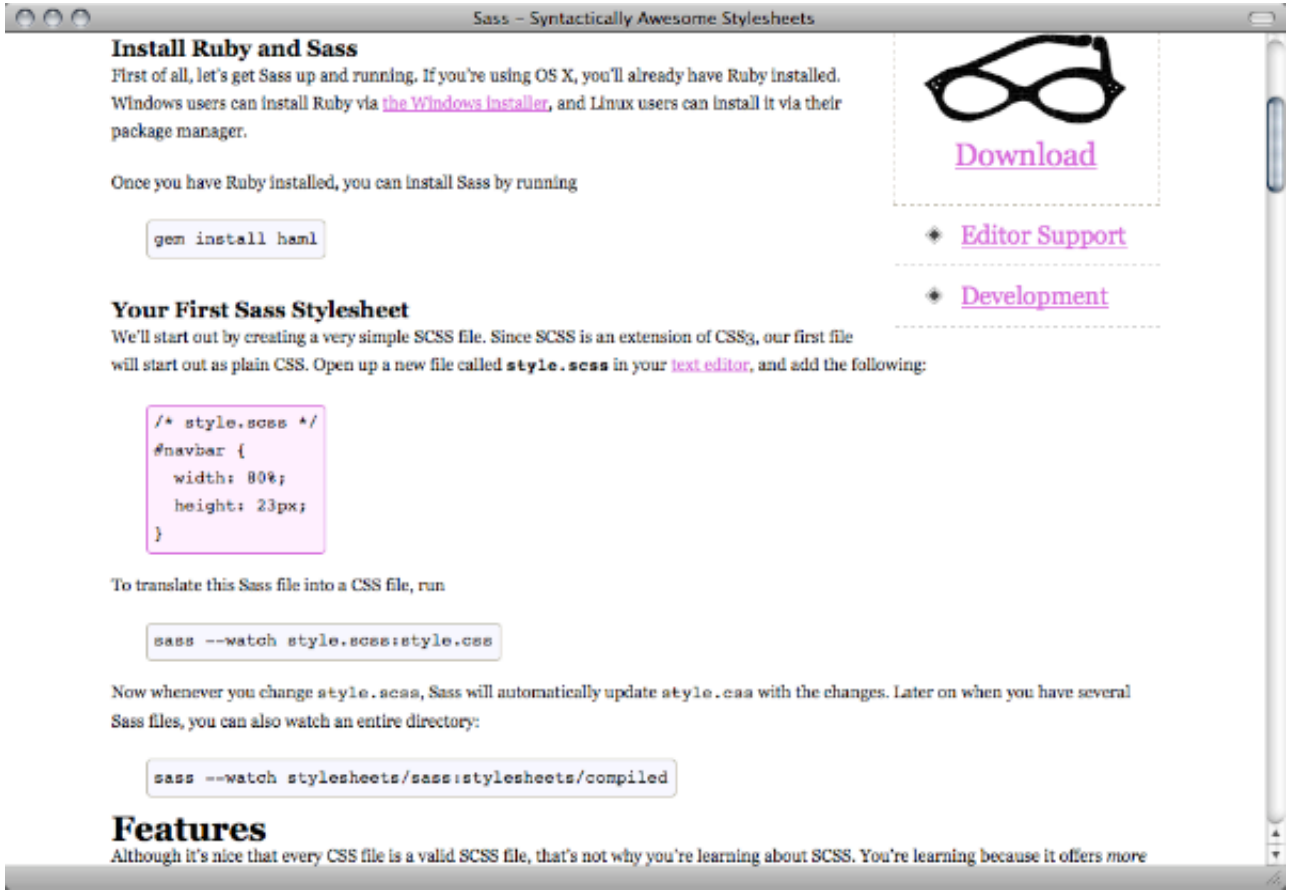
http://sass-lang.com

The Sass project for generating CSS files is the kind of documentation that's not only readable, it's also within our reach as developers to create.

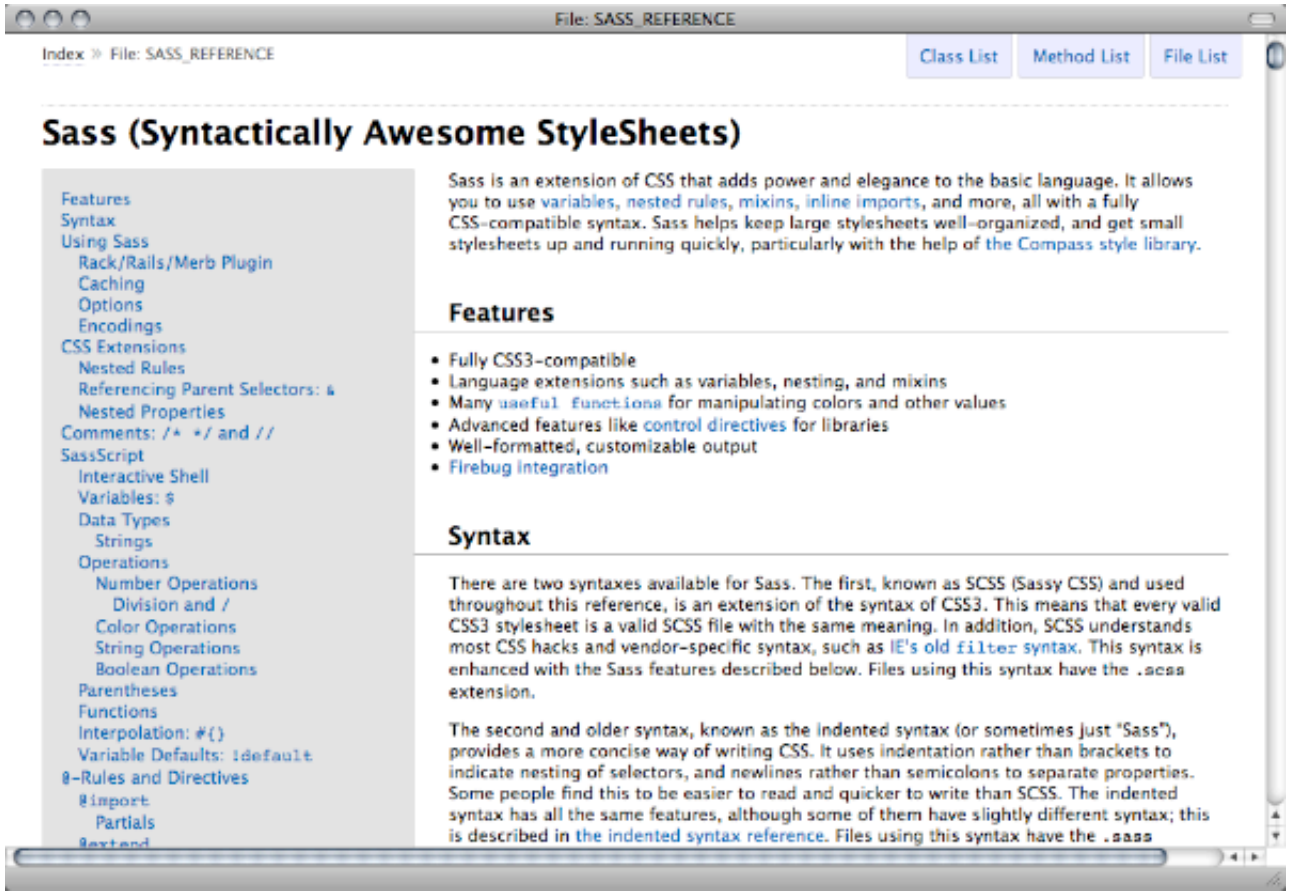
The main page begins with a three-line example for the impatient, then explains what the library is and why you'd use it.



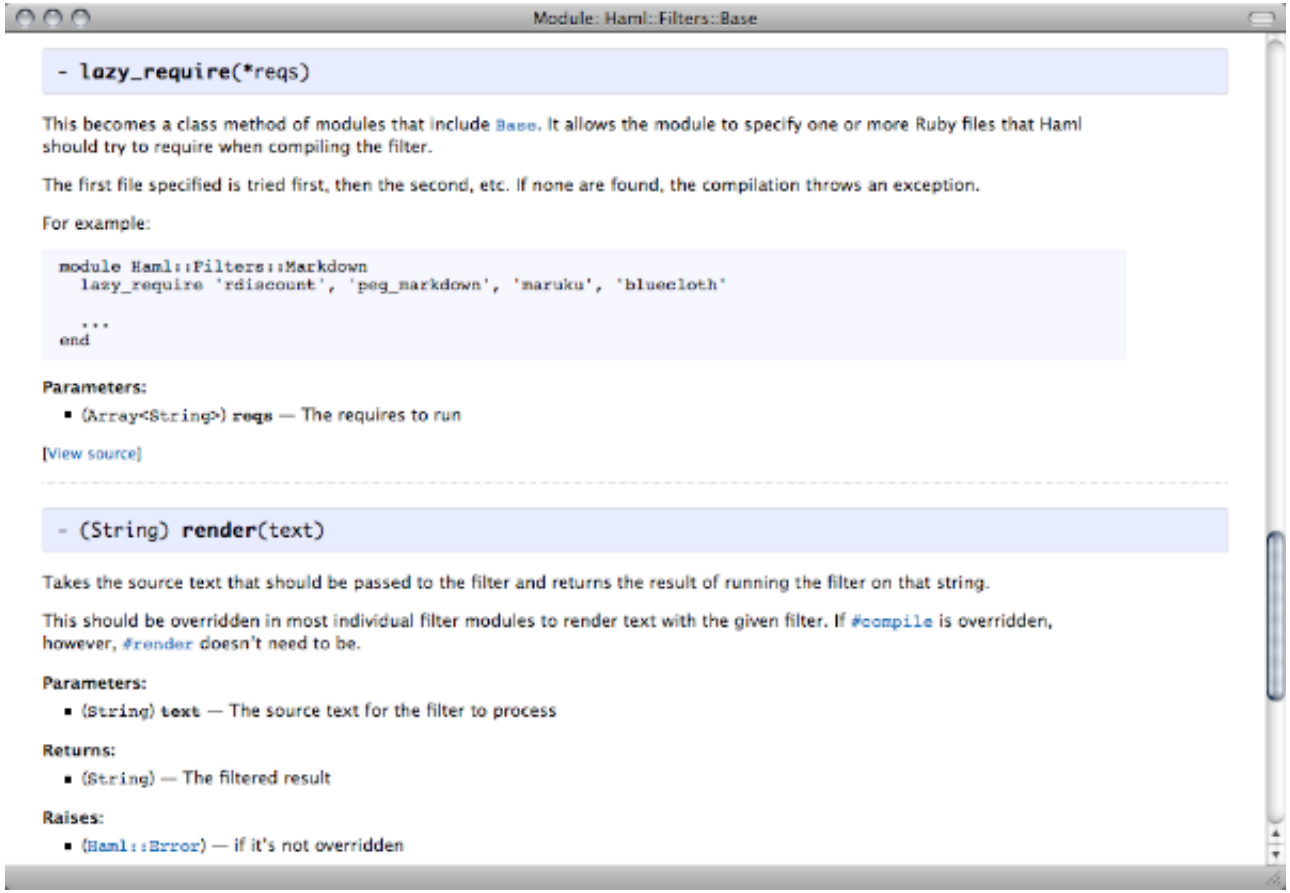
Further down the page are a bunch of real examples you're encouraged to try.



The Tutorials link takes you to a magazine-article length guide that walks you step by step through installation and your first creation.



The API documentation leads off with an equally lucid walkthrough of the major parts of the system. You have access to all the class and method documentation—but you also get advice on where to look first.



Once you've zoomed in on a particular method, this is what you see. Unlike our pathological Ruby example from earlier on, this text discusses parameters, examples, and error handling.

```
## Features

* Fully CSS3-compatible
* Language extensions such as variables, nest
* Many {Sass::Script::Functions useful functi
  manipulating colors and other values
* Advanced features like
  [control directives](#control_directives) f
* Well-formatted, customizable output
* [Firebug integration]
  (https://addons.mozilla.org/en-US/firefox/a
```

Here's a snippet of the raw text that's checked into the source tree, right alongside the code. The format is Markdown, one of the humane markup languages I mentioned earlier.

http://github.com/undees/pnsqc

This style of documentation is not only readable, it's achievable by us, the developers whose job it is to provide it.

To see the paper or the slides for this presentation, please visit the address above.

Other Voices

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

http://tom.preston-werner.com/2010/08/23/readme-driven-development.html

http://railstips.org/blog/archives/2010/10/14/stop-googling

Here's the Jacob Kaplan-Moss article mentioned earlier, along with a Tom Preston-Warner blog post on README-driven development. For a contrary (or orthogonal) point of view, see John Nunemaker's exhortation for people new to a sytem to jump straight into the code.

Photo Credits

pie: http://www.flickr.com/photos/kankan/53973269

Popeye: http://www.archive.org/details/spree_lunch

welcome: http://www.flickr.com/photos/madame_furie/2624297545

invisible: http://www.flickr.com/photos/bontempscharly/4352297633

ghost: http://www.flickr.com/photos/cocreatr/2345627792

ring: http://www.flickr.com/photos/garlandcannon/4918141830

scroll: http://www.flickr.com/photos/emdot/4876145