Chapter 10. Documentation

Written by Tom Manshreck

Edited by Riona MacNamara

Of the complaints most engineers have about writing, using, and maintaining code, a singular common frustration is the lack of quality documentation. “What are the side effects of this method?” “I got an error after step 3.” “What does this acronym mean?” “Is this document up to date?” Every software engineer has voiced complaints about the quality, quantity, or sheer lack of documentation throughout their career, and the software engineers at Google are no different.

Technical writers and project managers may help, but software engineers will always need to write most documentation themselves. Engineers, therefore, need the proper tools and incentives to do so effectively. The key to making it easier for them to write quality documentation is to introduce processes and tools that scale with the organization and that tie into their existing workflow.

Overall, the state of engineering documentation in the late 2010s is similar to the state of software testing in the late 1980s. Everyone recognizes that more effort needs to be made to improve it, but there is not yet organizational recognition of its critical benefits. That is changing, if slowly. At Google, our most successful efforts have been when documentation is *treated like code* and incorporated into the traditional engineering workflow, making it easier for engineers to write and maintain simple documents.

What Qualifies as Documentation?

When we refer to “documentation,” we’re talking about every supplemental text that an engineer needs to write to do their job: not only standalone documents, but code comments as well. (In fact, most of the documentation an engineer at Google writes comes in the form of code comments.) We’ll discuss the various types of engineering documents further in this chapter.

Why Is Documentation Needed?

Quality documentation has tremendous benefits for an engineering organization. Code and APIs become more comprehensible, reducing mistakes. Project teams are more focused when their design goals and team objectives are clearly stated. Manual processes are easier to follow when the steps are clearly outlined. Onboarding new members to a team or code base takes much less effort if the process is clearly documented.

But because documentation's benefits are all necessarily downstream, they generally don't reap immediate benefits to the author. Unlike testing, which (as we'll see) quickly provides benefits to a programmer, documentation generally requires more effort up front and doesn't provide clear benefits to an author until later. But, like investments in testing, the investment made in documentation will pay for itself over time. After all, you might write a document only once,[1] but it will be read hundreds, perhaps thousands of times afterward; its initial cost is amortized across all the future readers. Not only does documentation scale over time, but it is critical for the rest of the organization to scale as well. It helps answer questions like these:

- Why were these design decisions made?
- Why did we implement this code in this manner?
- Why did *I* implement this code in this manner, if you're looking at your own code two years later?

If documentation conveys all these benefits, why is it generally considered "poor" by engineers? One reason, as we've mentioned, is that the benefits aren't *immediate*, especially to the writer. But there are several other reasons:

- Engineers often view writing as a separate skill than that of programming. (We'll try to illustrate that this isn't quite the case, and even where it is, it isn't necessarily a separate skill from that of software engineering.)
- Some engineers don't feel like they are capable writers. But you don't need a robust command of English[2] to produce workable documentation. You just need to step outside yourself a bit and see things from the audience's perspective.
- Writing documentation is often more difficult because of limited tools support or integration into the developer workflow.

- Documentation is viewed as an extra burden—something else to maintain —rather than something that will make maintenance of their existing code easier.

Not every engineering team needs a technical writer (and even if that were the case, there aren't enough of them). This means that engineers will, by and large, write most of the documentation themselves. So, instead of forcing engineers to become technical writers, we should instead think about how to make writing documentation easier for engineers. Deciding how much effort to devote to documentation is a decision your organization will need to make at some point.

Documentation benefits several different groups. Even to the writer, documentation provides the following benefits:

- It helps formulate an API. Writing documentation is one of the surest ways to figure out if your API makes sense. Often, the writing of the documentation itself leads engineers to reevaluate design decisions that otherwise wouldn't be questioned. If you can't explain it and can't define it, you probably haven't designed it well enough.
- It provides a road map for maintenance and a historical record. Tricks in code should be avoided, in any case, but good comments help out a great deal when you're staring at code you wrote two years ago, trying to figure out what's wrong.
- It makes your code look more professional and drive traffic. Developers will naturally assume that a well-documented API is a better-designed API. That's not always the case, but they are often highly correlated. Although this benefit sounds cosmetic, it's not quite so: whether a product has good documentation is usually a pretty good indicator of how well a product will be maintained.
- It will prompt fewer questions from other users. This is probably the biggest benefit over time to someone writing the documentation. If you have to explain something to someone more than once, it usually makes sense to document that process.

As great as these benefits are to the writer of documentation, the lion's share of documentation's benefits will naturally accrue to the reader. Google's C++ Style Guide notes the maxim "optimize for the reader." This maxim applies not just to code, but to the comments around code, or the documentation set attached to an API. Much like testing, the effort you put into writing good documents will reap benefits many times over its lifetime. Documentation is critical over time, and reaps tremendous benefits for especially critical code as an organization scales.

Documentation Is Like Code

Software engineers who write in a single, primary programming language still often reach for different languages to solve specific problems. An engineer might write shell scripts or Python to run command-line tasks, or they might write most of their backend code in C++ but write some middleware code in Java, and so on. Each language is a tool in the toolbox.

Documentation should be no different: it's a tool, written in a different language (usually English) to accomplish a particular task. Writing documentation is not much different than writing code. Like a programming language, it has rules, a particular syntax, and style decisions, often to accomplish a similar purpose as that within code: enforce consistency, improve clarity, and avoid (comprehension) errors. Within technical documentation, grammar is important not because one needs rules, but to standardize the voice and avoid confusing or distracting the reader. Google requires a certain comment style for many of its languages for this reason.

Like code, documents should also have owners. Documents without owners become stale and difficult to maintain. Clear ownership also makes it easier to handle documentation through existing developer workflows: bug tracking systems, code review tooling, and so forth. Of course, documents with different owners can still conflict with one another. In those cases, it is important to designate *canonical* documentation: determine the primary source and consolidate other associated documents into that primary source (or deprecate the duplicates).

The prevalent usage of "go/ links" at Google (see Chapter 3) makes this process easier. Documents with straightforward go/ links often become the canonical source of truth. One other way to promote canonical documents is to associate them directly with the code they document by placing them directly under source control and alongside the source code itself.

Documentation is often so tightly coupled to code that it should, as much as possible, be treated *as code*. That is, your documentation should:

- Have internal policies or rules to be followed
- Be placed under source control
- Have clear ownership responsible for maintaining the docs
- Undergo reviews for changes (and change *with* the code it documents)
- Have issues tracked, as bugs are tracked in code
- Be periodically evaluated (tested, in some respect)
- If possible, be measured for aspects such as accuracy, freshness, etc. (tools have still not caught up here)

The more engineers treat documentation as “one of” the necessary tasks of software development, the less they will resent the upfront costs of writing, and the more they will reap the long-term benefits. In addition, making the task of documentation easier reduces those upfront costs.

CASE STUDY: THE GOOGLE WIKI

When Google was much smaller and leaner, it had few technical writers. The easiest way to share information was through our own internal wiki (GooWiki). At first, this seemed like a reasonable approach; all engineers shared a single documentation set and could update it as needed.

But as Google scaled, problems with a wiki-style approach became apparent. Because there were no true owners for documents, many became obsolete.[3] Because no process was put in place for adding new documents, duplicate documents and document sets began appearing. GooWiki had a flat namespace, and people were not good at applying any hierarchy to the documentation sets. At one point, there were 7 to 10 documents (depending on how you counted them) on setting up Borg, our production compute environment, only a few of which seemed to be maintained, and most were specific to certain teams with certain permissions and assumptions.

Another problem with GooWiki became apparent over time: the people who could fix the documents were not the people who used them. New users discovering bad documents either couldn't confirm that the documents were wrong or didn't have an easy way to report errors. They knew something was wrong (because the document didn't work), but they couldn't "fix" it. Conversely, the people best able to fix the documents often didn't need to consult them after they were written. The documentation became so poor as Google grew that the quality of documentation became Google's number one developer complaint on our annual developer surveys.

The way to improve the situation was to move important documentation under the same sort of source control that was being used to track code changes. Documents began to have their own owners, canonical locations within the source tree, and processes for identifying bugs and fixing them; the documentation began to dramatically improve. Additionally, the way documentation was written and maintained began to look the same as how code was written and maintained. Errors in the documents could be reported within our bug tracking software. Changes to the documents could be handled using the existing code review process. Eventually, engineers began to fix the documents themselves or send changes to technical writers (who were often the owners).

Moving documentation to source control was initially met with a lot of controversy. Many engineers were convinced that doing away with the

GooWiki, that bastion of freedom of information, would lead to poor quality because the bar for documentation (requiring a review, requiring owners for documents, etc.) would be higher. But that wasn't the case. The documents became better.

The introduction of Markdown as a common documentation formatting language also helped because it made it easier for engineers to understand how to edit documents without needing specialized expertise in HTML or CSS. Google eventually introduced its own framework for embedding documentation within code: g3doc. With that framework, documentation improved further, as documents existed side by side with the source code within the engineer's development environment. Now, engineers could update the code and its associated documentation in the same change (a practice for which we're still trying to improve adoption).

The key difference was that maintaining documentation became a similar experience to maintaining code: engineers filed bugs, made changes to documents in changelists, sent changes to reviews by experts, and so on. Leveraging of existing developer workflows, rather than creating new ones, was a key benefit.

Know Your Audience

One of the most important mistakes that engineers make when writing documentation is to write only for themselves. It's natural to do so, and writing for yourself is not without value: after all, you might need to look at this code in a few years and try to figure out what you once meant. You also might be of approximately the same skill set as someone reading your document. But if you write only for yourself, you are going to make certain assumptions, and given that your document might be read by a very wide audience (all of engineering, external developers), even a few lost readers is a large cost. As an organization grows, mistakes in documentation become more prominent, and your assumptions often do not apply.

Instead, before you begin writing, you should (formally or informally) identify the audience(s) your documents need to satisfy. A design document might need to persuade decision makers. A tutorial might need to provide very explicit instructions to someone utterly unfamiliar with your codebase. An API might need to provide complete and accurate reference information for any users of that API, be they experts or novices. Always try to identify a primary audience and write to that audience.

Good documentation need not be polished or “perfect.” One mistake engineers make when writing documentation is assuming they need to be much better writers. By that measure, few software engineers would write. Think about writing like you do about testing or any other process you need to do as an engineer. Write to your audience, in the voice and style that they expect. If you can read, you can write. Remember that your audience is standing where you once stood, but *without your new domain knowledge*. So you don’t need to be a great writer; you just need to get someone like you as familiar with the domain as you now are. (And as long as you get a stake in the ground, you can improve this document over time.)

Types of Audiences

We’ve pointed out that you should write at the skill level and domain knowledge appropriate for your audience. But who precisely is your audience? Chances are, you have multiple audiences based on one or more of the following criteria:

- Experience level (expert programmers, or junior engineers who might not even be familiar—gulp!—with the language).
- Domain knowledge (team members, or other engineers in your organization who are familiar only with API endpoints).
- Purpose (end users who might need your API to do a specific task and need to find that information quickly, or software gurus who are responsible for the guts of a particularly hairy implementation that you hope no one else needs to maintain).

In some cases, different audiences require different writing styles, but in most cases, the trick is to write in a way that applies as broadly to your different audience groups as possible. Often, you will need to explain a complex topic to both an expert and a novice. Writing for the expert with domain knowledge may allow you to cut corners, but you’ll confuse the novice; conversely, explaining everything in detail to the novice will doubtless annoy the expert.

Obviously, writing such documents is a balancing act and there’s no silver bullet, but one thing we’ve found is that it helps to keep your documents *short*. Write descriptively enough to explain complex topics to people unfamiliar with the topic, but don’t lose or annoy experts. Writing a short document often requires you to write a longer one (getting all the information down) and then doing an edit pass, removing duplicate information where you can. This might sound tedious, but keep in mind that this expense is spread across all the readers of the documentation. As Blaise Pascal once said, “If I had more time, I would have written you a shorter letter.” By keeping a document short and clear, you will ensure that it will satisfy both an expert and a novice.

Another important audience distinction is based on how a user encounters a document:

- *Seekers* are engineers who *know what they want* and want to know if what they are looking at fits the bill. A key pedagogical device for this audience is *consistency*. If you are writing reference documentation for this group—within a code file, for example—you will want to have your comments follow a similar format so that readers can quickly scan a reference and see whether they find what they are looking for.
- *Stumblers* might not know exactly what they want. They might have only a vague idea of how to implement what they are working with. The key for this audience is *clarity*. Provide overviews or introductions (at the top of a file, for example) that explain the purpose of the code they are looking at. It’s also useful to identify when a doc is *not* appropriate for an audience. A lot of documents at Google begin with a “TL;DR statement” such as “TL;DR: if you are not interested in C++ compilers at Google, you can stop reading now.”

Finally, one important audience distinction is between that of a customer (e.g., a user of an API) and that of a provider (e.g., a member of the project team). As much as possible, documents intended for one should be kept apart from documents intended for the other. Implementation details are important to a team member for maintenance purposes; end users should not need to read such information. Often, engineers denote design decisions within the reference API of a library they publish. Such reasonings belong more appropriately in specific documents (design documents) or, at best, within the implementation details of code hidden behind an interface.

Documentation Types

Engineers write various different types of documentation as part of their work: design documents, code comments, how-to documents, project pages, and more. These all count as "documentation." But it is important to know the different types, and to *not mix types*. A document should have, in general, a singular purpose, and stick to it. Just as an API should do one thing and do it well, avoid trying to do several things within one document. Instead, break out those pieces more logically.

There are several main types of documents that software engineers often need to write:

- Reference documentation, including code comments
- Design documents
- Tutorials
- Conceptual documentation
- Landing pages

It was common in the early days of Google for teams to have monolithic wiki pages with bunches of links (many broken or obsolete), some conceptual information about how the system worked, an API reference, and so on, all sprinkled together. Such documents fail because they don't serve a single purpose (and they also get so long that no one will read them; some notorious wiki pages scrolled through several dozens of screens). Instead, make sure your document has a singular purpose, and if adding something to that page doesn't make sense, you probably want to find, or even create, another document for that purpose.

Reference Documentation

Reference documentation is the most common type that engineers need to write; indeed, they often need to write some form of reference documents every day. By reference documentation, we mean anything that documents the usage of code within the codebase. Code comments are the most common form of reference documentation that an engineer must maintain. Such comments can be divided into two basic camps: API comments versus implementation comments. Remember the audience differences between these two: API comments don't need to discuss implementation details or design decisions and can’t assume a user is as versed in the API as the author. Implementation comments, on the other hand, can assume a lot more domain knowledge of the reader, though be careful in assuming too much: people leave projects, and sometimes it’s safer to be methodical about exactly why you wrote this code the way you did.

Most reference documentation, even when provided as separate documentation from the code, is generated from comments within the codebase itself. (As it should; reference documentation should be single-sourced as much as possible.) Some languages such as Java or Python have specific commenting frameworks (Javadoc, PyDoc, GoDoc) meant to make generation of this reference documentation easier. Other languages, such as C++, have no standard “reference documentation” implementation, but because C++ separates out its API surface (in header or *.h* files) from the implementation (*.cc* files), header files are often a natural place to document a C++ API.

Google takes this approach: a C++ API deserves to have its reference documentation live within the header file. Other reference documentation is embedded directly in the Java, Python, and Go source code as well. Because Google’s Code Search browser (see Chapter 17) is so robust, we’ve found little benefit to providing separate generated reference documentation. Users in Code Search not only search code easily, they can usually find the original definition of that code as the top result. Having the documentation alongside the code’s definitions also makes the documentation easier to discover and maintain.

We all know that code comments are essential to a well-documented API. But what precisely is a “good” comment? Earlier in this chapter, we identified two major audiences for reference documentation: seekers and stumblers. Seekers know what they want; stumblers don’t. The key win for seekers is a consistently commented codebase so that they can quickly scan an API and find what they are looking for. The key win for stumblers is clearly identifying the purpose of an API, often at the top of a file header. We’ll walk through some code comments in the subsections that follow. The code commenting guidelines that follow apply to C++, but similar rules are in place at Google for other languages.

File comments

Almost all code files at Google must contain a file comment. (Some header files that contain only one utility function, etc., might deviate from this standard.) File comments should begin with a header of the following form:

```
// ----------------------------------------------------
// str_cat.h
// ----------------------------------------------------
//
// This header file contains functions for efficiently
// strings: StrCat() and StrAppend(). Most of the work
// actually handled through use of a special AlphaNum t
// to be used as a parameter type that efficiently mana
// strings and avoids copies in the above operations.
…
```

Generally, a file comment should begin with an outline of what's contained in the code you are reading. It should identify the code's main use cases and intended audience (in the preceding case, developers who want to concatenate strings). Any API that cannot be succinctly described in the first paragraph or two is usually the sign of an API that is not well thought out. Consider breaking the API into separate components in those cases.

Class comments

Most modern programming languages are object oriented. Class comments are therefore important for defining the API "objects" in use in a codebase. All public classes (and structs) at Google must contain a class comment describing the class/struct, important methods of that class, and the purpose of the class. Generally, class comments should be "nouned" with documentation emphasizing their object aspect. That is, say, "The Foo class contains x, y, z, allows you to do Bar, and has the following Baz aspects," and so on.

Class comments should generally begin with a comment of the following form:

```
// ----------------------------------------------------
// AlphaNum
// ----------------------------------------------------
```

```
//
// The AlphaNum class acts as the main parameter type f
// StrAppend(), providing efficient conversion of numer
// hexadecimal values (through the Hex type) into strin
```

Function comments

All free functions, or public methods of a class, at Google must also contain a function comment describing what the function *does*. Function comments should stress the *active* nature of their use, beginning with an indicative verb describing what the function does and what is returned.

Function comments should generally begin with a comment of the following form:

```
// StrCat()
//
// Merges the given strings or numbers, using no delimi
// returning the merged result as a string.
…
```

Note that starting a function comment with a declarative verb introduces consistency across a header file. A seeker can quickly scan an API and read just the verb to get an idea of whether the function is appropriate: "Merges, Deletes, Creates," and so on.

Some documentation styles (and some documentation generators) require various forms of boilerplate on function comments, like "Returns:", "Throws:", and so forth, but at Google we haven't found them to be necessary. It is often clearer to present such information in a single prose comment that's not broken up into artificial section boundaries:

```
// Creates a new record for a customer with the given r
// and returns the record ID, or throws `DuplicateEntry
// record with that name already exists.
int AddCustomer(string name, string address);
```

Notice how the postcondition, parameters, return value, and exceptional cases are naturally documented together (in this case, in a single sentence), because they are not independent of one another. Adding explicit boilerplate sections would make the comment more verbose and repetitive, but no clearer (and arguably less clear).

Design Docs

Most teams at Google require an approved design document before starting work on any major project. A software engineer typically writes the proposed design document using a specific design doc template approved by the team. Such documents are designed to be collaborative, so they are often shared in Google Docs, which has good collaboration tools. Some teams require such design documents to be discussed and debated at specific team meetings, where the finer points of the design can be discussed or critiqued by experts. In some respects, these design discussions act as a form of code review before any code is written.

Because the development of a design document is one of the first processes an engineer undertakes before deploying a new system, it is also a convenient place to ensure that various concerns are covered. The canonical design document templates at Google require engineers to consider aspects of their design such as security implications, internationalization, storage requirements and privacy concerns, and so on. In most cases, such parts of those design documents are reviewed by experts in those domains.

A good design document should cover the goals of the design, its implementation strategy, and propose key design decisions with an emphasis on their individual trade-offs. The best design documents suggest design goals and cover alternative designs, denoting their strong and weak points.

A good design document, once approved, also acts not only as a historical record, but as a measure of whether the project successfully achieved its goals. Most teams archive their design documents in an appropriate location within their team documents so that they can review them at a later time. It’s often useful to review a design document before a product is launched to ensure that the stated goals when the design document was written remain the stated goals at launch (and if they do not, either the document or the product can be adjusted accordingly).

Tutorials

Every software engineer, when they join a new team, will want to get up to speed as quickly as possible. Having a tutorial that walks someone through the setup of a new project is invaluable; “Hello World” has established itself is one of the best ways to ensure that all team members start off on the right foot. This goes for documents as well as code. Most projects deserve a “Hello World” document that assumes nothing and gets the engineer to make something “real” happen.

Often, the best time to write a tutorial, if one does not yet exist, is when you first join a team. (It’s also the best time to find bugs in any existing tutorial you are following.) Get a notepad or other way to take notes, and write down everything you need to do along the way, assuming no domain knowledge or special setup constraints; after you’re done, you’ll likely know what mistakes you made during the process—and why—and can then edit down your steps to get a more streamlined tutorial. Importantly, write *everything* you need to do along the way; try not to assume any particular setup, permissions, or domain knowledge. If you do need to assume some other setup, state that clearly in the beginning of the tutorial as a set of prerequisites.

Most tutorials require you to perform a number of steps, in order. In those cases, number those steps explicitly. If the focus of the tutorial is on the *user* (say, for external developer documentation), then number each action that a user needs to undertake. Don’t number actions that the system may take in response to such user actions. It is critical and important to number explicitly every step when doing this. Nothing is more annoying than an error on step 4 because you forget to tell someone to properly authorize their username, for example.

Example: A bad tutorial

1. Download the package from our server at http://example.com
2. Copy the shell script to your home directory
3. Execute the shell script
4. The foobar system will communicate with the authentication system
5. Once authenticated, foobar will bootstrap a new database named “baz”
6. Test “baz” by executing a SQL command on the command line
7. Type: CREATE DATABASE my_foobar_db;

In the preceding procedure, steps 4 and 5 happen on the server end. It's unclear whether the user needs to do anything, but they don't, so those side effects can be mentioned as part of step 3. As well, it's unclear whether step 6 and step 7 are different. (They aren't.) Combine all atomic user operations into single steps so that the user knows they need to do something at each step in the process. Also, if your tutorial has user-visible input or output, denote that on separate lines (often using the convention of a `monospaced bold` font).

Example: A bad tutorial made better

1. Download the package from our server at *http://example.com*:

   ```
   $ curl -I http://example.com
   ```

2. Copy the shell script to your home directory:

   ```
   $ cp foobar.sh ~
   ```

3. Execute the shell script in your home directory:

   ```
   $ cd ~; foobar.sh
   ```

   The foobar system will first communicate with the authentication system. Once authenticated, foobar will bootstrap a new database named "baz" and open an input shell.
4. Test "baz" by executing a SQL command on the command line:

   ```
   baz:$ CREATE DATABASE my_foobar_db;
   ```

Note how each step requires specific user intervention. If, instead, the tutorial had a focus on some other aspect (e.g., a document about the "life of a server"), number those steps from the perspective of that focus (what the server does).

Conceptual Documentation

Some code requires deeper explanations or insights than can be obtained simply by reading the reference documentation. In those cases, we need conceptual documentation to provide overviews of the APIs or systems. Some examples of conceptual documentation might be a library overview for a popular API, a document describing the life cycle of data within a server, and so on. In almost all cases, a conceptual document is meant to augment, not replace, a reference documentation set. Often this leads to duplication of some information, but with a purpose: to promote clarity. In those cases, it is not necessary for a conceptual document to cover all edge cases (though a reference should cover those cases religiously). In this case, sacrificing some accuracy is acceptable for clarity. The main point of a conceptual document is to impart understanding.

"Concept" documents are the most difficult forms of documentation to write. As a result, they are often the most neglected type of document within a software engineer's toolbox. One problem engineers face when writing conceptual documentation is that it often cannot be embedded directly within the source code because there isn't a canonical location to place it. Some APIs have a relatively broad API surface area, in which case, a file comment might be an appropriate place for a "conceptual" explanation of the API. But often, an API works in conjunction with other APIs and/or modules. The only logical place to document such complex behavior is through a separate conceptual document. If comments are the unit tests of documentation, conceptual documents are the integration tests.

Even when an API is appropriately scoped, it often makes sense to provide a separate conceptual document. For example, Abseil's `StrFormat` library covers a variety of concepts that accomplished users of the API should understand. In those cases, both internally and externally, we provide a format concepts document.

A concept document needs to be useful to a broad audience: both experts and novices alike. Moreover, it needs to emphasize *clarity*, so it often needs to sacrifice completeness (something best reserved for a reference) and (sometimes) strict accuracy. That's not to say a conceptual document should intentionally be inaccurate; it just means that it should focus on common usage and leave rare usages or side effects for reference documentation.

Landing Pages

Most engineers are members of a team, and most teams have a “team page” somewhere on their company’s intranet. Often, these sites are a bit of a mess: a typical landing page might contain some interesting links, sometimes several documents titled “read this first!”, and some information both for the team and for its customers. Such documents start out useful but rapidly turn into disasters; because they become so cumbersome to maintain, they will eventually get so obsolete that they will be fixed by only the brave or the desperate.

Luckily, such documents look intimidating, but are actually straightforward to fix: ensure that a landing page clearly identifies its purpose, and then include *only* links to other pages for more information. If something on a landing page is doing more than being a traffic cop, it is *not doing its job*. If you have a separate setup document, link to that from the landing page as a separate document. If you have too many links on the landing page (your page should not scroll multiple screens), consider breaking up the pages by taxonomy, under different sections.

Most poorly configured landing pages serve two different purposes: they are the “goto” page for someone who is a user of your product or API, or they are the home page for a team. Don’t have the page serve both masters—it will become confusing. Create a separate “team page” as an internal page apart from the main landing page. What the team needs to know is often quite different than what a customer of your API needs to know.

Documentation Reviews

At Google, all code needs to be reviewed, and our code review process is well understood and accepted. In general, documentation also needs review (though this is less universally accepted). If you want to “test” whether your documentation works, you should generally have someone else review it.

A technical document benefits from three different types of reviews, each emphasizing different aspects:

- A technical review, for accuracy. This review is usually done by a subject matter expert, often another member of your team. Often, this is part of a code review itself.

- An audience review, for clarity. This is usually someone unfamiliar with the domain. This might be someone new to your team or a customer of your API.
- A writing review, for consistency. This is often a technical writer or volunteer.

Of course, some of these lines are sometimes blurred, but if your document is high profile or might end up being externally published, you probably want to ensure that it receives more types of reviews. (We've used a similar review process for this book.) Any document tends to benefit from the aforementioned reviews, even if some of those reviews are ad hoc. That said, even getting one reviewer to review your text is preferable to having no one review it.

Importantly, if documentation is tied into the engineering workflow, it will often improve over time. Most documents at Google now implicitly go through an audience review because at some point, their audience will be using them, and hopefully letting you know when they aren't working (via bugs or other forms of feedback).

CASE STUDY: THE DEVELOPER GUIDE LIBRARY

As mentioned earlier, there were problems associated with having most (almost all) engineering documentation contained within a shared wiki: little ownership of important documentation, competing documentation, obsolete information, and difficulty in filing bugs or issues with documentation. But this problem was not seen in some documents: the Google C++ style guide was owned by a select group of senior engineers (style arbiters) who managed it. The document was kept in good shape because certain people cared about it. They implicitly owned that document. The document was also canonical: there was only one C++ style guide.

As previously mentioned, documentation that sits directly within source code is one way to promote the establishment of canonical documents; if the documentation sits alongside the source code, it should usually be the most applicable (hopefully). At Google, each API usually has a separate *g3doc* directory where such documents live (written as Markdown files and readable within our Code Search browser). Having the documentation exist alongside the source code not only establishes de facto ownership, it makes the documentation seem more wholly "part" of the code.

Some documentation sets, however, cannot exist very logically within source code. A "C++ developer guide" for Googlers, for example, has no obvious place to sit within the source code. There is no master "C++" directory where people will look for such information. In this case (and others that crossed API boundaries), it became useful to create standalone documentation sets in their own depot. Many of these culled together associated existing documents into a common set, with common navigation and look-and-feel. Such documents were noted as "Developer Guides" and, like the code in the codebase, were under source control in a specific documentation depot, with this depot organized by topic rather than API. Often, technical writers managed these developer guides, because they were better at explaining topics across API boundaries.

Over time, these developer guides became canonical. Users who wrote competing or supplementary documents became amenable to adding their documents to the canonical document set after it was established, and then deprecating their competing documents. Eventually, the C++ style guide became part of a larger "C++ Developer Guide." As the documentation set became more comprehensive and more authoritative, its quality also improved. Engineers began logging bugs because they knew someone was maintaining these documents. Because the documents were locked down under source control, with proper owners, engineers also began sending changelists directly to the technical writers.

The introduction of go/ links (see Chapter 3) allowed most documents to, in effect, more easily establish themselves as canonical on any given topic. Our C++ Developer Guide became established at "go/cpp," for example. With better internal search, go/ links, and the integration of multiple documents into a common documentation set, such canonical documentation sets became more authoritative and robust over time.

Documentation Philosophy

Caveat: the following section is more of a treatise on technical writing best practices (and personal opinion) than of "how Google does it." Consider it optional for software engineers to fully grasp, though understanding these concepts will likely allow you to more easily write technical information.

WHO, WHAT, WHEN, WHERE, and WHY

Most technical documentation answers a "HOW" question. How does this work? How do I program to this API? How do I set up this server? As a result, there's a tendency for software engineers to jump straight into the "HOW" on any given document and ignore the other questions associated with it: the WHO, WHAT, WHEN, WHERE, and WHY. It's true that none of those are generally as important as the HOW—a design document is an exception because an equivalent aspect is often the WHY—but without a proper framing of technical documentation, documents end up confusing. Try to address the other questions in the first two paragraphs of any document:

- WHO was discussed previously: that's the audience. But sometimes you also need to explicitly call out and address the audience in a document. Example: "This document is for new engineers on the Secret Wizard project."
- WHAT identifies the purpose of this document: "This document is a tutorial designed to start a Frobber server in a test environment." Sometimes, merely writing the WHAT helps you frame the document appropriately. If you start adding information that isn't applicable to the

WHAT, you might want to move that information into a separate document.

- WHEN identifies when this document was created, reviewed, or updated. Documents in source code have this date noted implicitly, and some other publishing schemes automate this as well. But, if not, make sure to note the date on which the document was written (or last revised) on the document itself.
- WHERE is often implicit as well, but decide where the document should live. Usually, the preference should be under some sort of version control, ideally *with the source code it documents*. But other formats work for different purposes as well. At Google, we often use Google Docs for easy collaboration, particularly on design issues. At some point, however, any shared document becomes less of a discussion and more of a stable historical record. At that point, move it to someplace more permanent, with clear ownership, version control, and responsibility.
- WHY sets up the purpose for the document. Summarize what you expect someone to take away from the document after reading it. A good rule of thumb is to establish the WHY in the introduction to a document. When you write the summary, verify whether you've met your original expectations (and revise accordingly).

The Beginning, Middle, and End

All documents—indeed, all parts of documents—have a beginning, middle, and end. Although it sounds amazingly silly, most documents should often have, at a minimum, those three sections. A document with only one section has only one thing to say, and very few documents have only one thing to say. Don't be afraid to add sections to your document; they break up the flow into logical pieces and provide readers with a roadmap of what the document covers.

Even the simplest document usually has more than one thing to say. Our popular "C++ Tips of the Week" have traditionally been very short, focusing on one small piece of advice. However, even here, having sections helps. Traditionally, the first section denotes the problem, the middle section goes through the recommended solutions, and the conclusion summarizes the takeaways. Had the document consisted of only one section, some readers would doubtless have difficulty teasing out the important points.

Most engineers loathe redundancy, and with good reason. But in documentation, redundancy is often useful. An important point buried within a wall of text can be difficult to remember or tease out. On the other hand, placing that point at a more prominent location early can lose context provided later on. Usually, the solution is to introduce and summarize the point within an introductory paragraph, and then use the rest of the section to make your case in a more detailed fashion. In this case, redundancy helps the reader understand the importance of what is being stated.

The Parameters of Good Documentation

There are usually three aspects of good documentation: completeness, accuracy, and clarity. You rarely get all three within the same document; as you try to make a document more "complete," for example, clarity can begin to suffer. If you try to document every possible use case of an API, you might end up with an incomprehensible mess. For programming languages, being completely accurate in all cases (and documenting all possible side effects) can also affect clarity. For other documents, trying to be clear about a complicated topic can subtly affect the accuracy of the document; you might decide to ignore some rare side effects in a conceptual document, for example, because the point of the document is to familiarize someone with the usage of an API, not provide a dogmatic overview of all intended behavior.

In each case, a "good document" is defined as the document that is *doing its intended job*. As a result, you rarely want a document doing more than one job. For each document (and for each document type), decide on its focus and adjust the writing appropriately. Writing a conceptual document? You probably don't need to cover every part of the API. Writing a reference? You probably want this complete, but perhaps must sacrifice some clarity. Writing a landing page? Focus on organization and keep discussion to a minimum. All of this adds up to quality, which, admittedly, is stubbornly difficult to accurately measure.

How can you quickly improve the quality of a document? Focus on the needs of the audience. Often, less is more. For example, one mistake engineers often make is adding design decisions or implementation details to an API document. Much like you should ideally separate the interface from an implementation within a well-designed API, you should avoid discussing design decisions in an API document. Users don't need to know this information. Instead, put those decisions in a specialized document for that purpose (usually a design doc).

Deprecating Documents

Just like old code can cause problems, so can old documents. Over time, documents become stale, obsolete, or (often) abandoned. Try as much as possible to avoid abandoned documents, but when a document no longer serves any purpose, either remove it or identify it as obsolete (and, if available, indicate where to go for new information). Even for unowned documents, someone adding a note that “This no longer works!” is more helpful than saying nothing and leaving something that seems authoritative but no longer works.

At Google, we often attach “freshness dates” to documentation. Such documents note the last time a document was reviewed, and metadata in the documentation set will send email reminders when the document hasn’t been touched in, for example, three months. Such freshness dates, as shown in the following example—and tracking your documents as bugs—can help make a documentation set easier to maintain over time, which is the main concern for a document:

```
<!--*
# Document freshness: For more information, see go/fresh-
freshness: { owner: `username` reviewed: '2019-02-27' }
*-->
```

Users who own such a document have an incentive to keep that freshness date current (and if the document is under source control, that requires a code review). As a result, it’s a low-cost means to ensure that a document is looked over from time to time. At Google, we found that including the owner of a document in this freshness date within the document itself with a byline of “Last reviewed by...” led to increased adoption as well.

When Do You Need Technical Writers?

When Google was young and growing, there weren’t enough technical writers in software engineering. (That’s still the case.) Those projects deemed important tended to receive a technical writer, regardless of whether that team really needed one. The idea was that the writer could relieve the team of some of the burden of writing and maintaining documents and (theoretically) allow the important project to achieve greater velocity. This turned out to be a bad assumption.

We learned that most engineering teams can write documentation for themselves (their team) perfectly fine; it's only when they are writing documents for another audience that they tend to need help because it's difficult to write to another audience. The feedback loop within your team regarding documents is more immediate, the domain knowledge and assumptions are clearer, and the perceived needs are more obvious. Of course, a technical writer can often do a better job with grammar and organization, but supporting a single team isn't the best use of a limited and specialized resource; it doesn't scale. It introduced a perverse incentive: become an important project and your software engineers won't need to write documents. Discouraging engineers from writing documents turns out to be the opposite of what you want to do.

Because they are a limited resource, technical writers should generally focus on tasks that software engineers *don't* need to do as part of their normal duties. Usually, this involves writing documents that cross API boundaries. Project Foo might clearly know what documentation Project Foo needs, but it probably has a less clear idea what Project Bar needs. A technical writer is better able to stand in as a person unfamiliar with the domain. In fact, it's one of their critical roles: to challenge the assumptions your team makes about the utility of your project. It's one of the reasons why many, if not most, software engineering technical writers tend to focus on this specific type of API documentation.

Conclusion

Google has made good strides in addressing documentation quality over the past decade, but to be frank, documentation at Google is not yet a first-class citizen. For comparison, engineers have gradually accepted that testing is necessary for any code change, no matter how small. As well, testing tooling is robust, varied and plugged into an engineering workflow at various points. Documentation is not ingrained at nearly the same level.

To be fair, there's not necessarily the same need to address documentation as with testing. Tests can be made atomic (unit tests) and can follow prescribed form and function. Documents, for the most part, cannot. Tests can be automated, and schemes to automate documentation are often lacking. Documents are necessarily subjective; the quality of the document is measured not by the writer, but by the reader, and often quite asynchronously. That said, there is a recognition that documentation is important, and processes around document development are improving. In this author's opinion, the quality of documentation at Google is better than in most software engineering shops.

To change the quality of engineering documentation, engineers—and the entire engineering organization—need to accept that they are both the problem and the solution. Rather than throw up their hands at the state of documentation, they need to realize that producing quality documentation is part of their job and saves them time and effort in the long run. For any piece of code that you expect to live more than a few months, the extra cycles you put in documenting that code will not only help others, it will help you maintain that code as well.

TL;DRs

- Documentation is hugely important over time and scale.
- Documentation changes should leverage the existing developer workflow.
- Keep documents focused on one purpose.
- Write for your audience, not yourself.

1 OK, you will need to maintain it and revise it occasionally.

2 English is still the primary language for most programmers, and most technical documentation for programmers relies on an understanding of English.

3 When we deprecated GooWiki, we found that around 90% of the documents had no views or updates in the previous few months.