styleguide

Documentation Best Practices

“Say what you mean, simply and directly.” - Brian Kernighan

Contents:

Minimum Viable Documentation

A small set of fresh and accurate docs is better than a large assembly of “documentation” in various states of disrepair.

Write short and useful documents. Cut out everything unnecessary, including out-of-date, incorrect, or redundant information. Also make a habit of continually massaging and improving every doc to suit your changing needs. **Docs work best when they are alive but frequently trimmed, like a bonsai tree**.

See also these Agile Documentation best practices.

Update Docs with Code

Change your documentation in the same CL as the code change. This keeps your docs fresh, and is also a good place to explain to your reviewer what you’re doing.

A good reviewer can at least insist that docstrings, header files, README.md files, and any other docs get updated alongside the CL.

Delete Dead Documentation

Dead docs are bad. They misinform, they slow down, they incite despair in engineers and laziness in team leads. They set a precedent for leaving behind messes in a code base. If your home is clean, most guests will be clean without being asked.

Just like any big cleaning project, **it's easy to be overwhelmed**. If your docs are in bad shape:

- Take it slow, doc health is a gradual accumulation.
- First delete what you're certain is wrong, ignore what's unclear.
- Get your whole team involved. Devote time to quickly scan every doc and make a simple decision: Keep or delete?
- Default to delete or leave behind if migrating. Stragglers can always be recovered.
- Iterate.

Prefer the Good Over the Perfect

Documentation is an art. There is no perfect document, there are only proven methods and prudent guidelines. See Better is better than best.

Documentation is the Story of Your Code

Writing excellent code doesn't end when your code compiles or even if your test coverage reaches 100%. It's easy to write something a computer understands, it's much harder to write something both a human and a computer understand. Your mission as a Code Health-conscious engineer is to **write for humans first, computers second.** Documentation is an important part of this skill.

There's a spectrum of engineering documentation that ranges from terse comments to detailed prose:

1. **Meaningful names**: Good naming allows the code to convey information that would otherwise be relegated to comments or documentation. This includes nameable entities at all levels, from local variables to classes, files, and directories.

2. **Inline comments**: The primary purpose of inline comments is to provide information that the code itself cannot contain, such as why the code is there.

3. **Method and class comments**:

    - **Method API documentation**: The header / Javadoc / docstring comments that say what methods do and how to use them. This documentation is **the contract of how your code must behave**. The intended audience is future programmers who will use and modify your code.

    It is often reasonable to say that any behavior documented here should have a test verifying it. This documentation details what arguments the method takes, what it returns, any "gotchas" or restrictions, and what exceptions it can throw or errors it can return. It does not usually explain why code behaves a particular way unless that's relevant to a developer's understanding of how to use the method. "Why" explanations are for inline comments. Think in practical terms when writing method documentation: "This is a hammer. You use it to pound nails."

   - **Class / Module API documentation**: The header / Javadoc / docstring comments for a class or a whole file. This documentation gives a brief overview of what the class / file does and often gives a few short examples of how you might use the class / file.

     Examples are particularly relevant when there's several distinct ways to use the class (some advanced, some simple). Always list the simplest use case first.

4. **README.md**: A good README.md orients the new user to the directory and points to more detailed explanation and user guides:

   - What is this directory intended to hold?
   - Which files should the developer look at first? Are some files an API?
   - Who maintains this directory and where I can learn more?

   See the README.md guidelines.

5. **docs**: The contents of a good docs directory explain how to:

   - Get started using the relevant API, library, or tool.
   - Run its tests.
   - Debug its output.
   - Release the binary.

6. **Design docs, PRDs**: A good design doc or PRD discusses the proposed implementation at length for the purpose of collecting feedback on that design. However, once the code is implemented, design docs should serve as archives of these decisions, not as half-correct docs (they are often misused).

7. **Other external docs**: Some teams maintain documentation in other locations, separate from the code, such as Google Sites, Drive, or wiki. If you do maintain documentation in other locations, you should clearly point to those locations from your project directory (for example, by adding an obvious link to the location from your project's `README.md`).

Duplication is Evil

Do not write your own guide to a common Google technology or process. Link to it instead. If the guide doesn’t exist or it’s badly out of date, submit your updates to the appropriate directory or create a package-level README.md. **Take ownership and don’t be shy**: Other teams will usually welcome your contributions.