Writing Technical Design Docs

Engineering Insights

Talin Follow
Jan 3, 2019 · 7 min read

An important skill for any software engineer is writing technical design docs (TDDs), also referred to as engineering design docs (EDDs). Here in this article I offer some advice for writing good design docs and what mistakes to avoid.

One caveat: Different teams will have different standards and conventions for technical design. There is no industry-wide standard for the design process, nor could there be, as different development teams will have different needs depending on their situation. What I will describe is one possible answer, based on my own experience.

Design Process

Let's start with the basics: **What is a technical design doc, and how does it fit in to the design process?**

A technical design doc describes a solution to a given technical problem. It is a specification, or "design blueprint", for a software program or feature.

The primary function of a TDD is to communicate the technical details of the work to be done to members of the team. However, there is a second purpose which is just as important: the process of writing the TDD forces you to organize your thoughts and consider every aspect of the design, ensuring that you haven't left anything out.

Technical design docs are often part of a larger process which typically has the following steps:

1. **Product requirements are defined**. These will typically be represented by a Product Requirements Document (PRD). The PRD specifies what the system needs to do, from the perspective of a user or outside agent.
2. **Technical requirements are defined**. The product requirements are translated into technical requirements — *what* the system needs to accomplish, but now *how* it does it. The output of this step is a Technical Requirements Document (TRD).
3. **Technical design**. This contains a technical description of the solution to the requirements outlined in the previous steps. The TDD is the output of this step.

4. **Implementation**. This is the stage where the solution is actually built.
5. **Testing**. The system is tested against the PRD and TRD to ensure that it actually fulfills the specified requirements.

Between each of these stages there is typically a review process to ensure that no mistakes were made. If any errors, misunderstandings, or ambiguities are detected, these must be corrected before proceeding to the next step.

This process is highly variable; the set of steps listed here will change on a case-by-case basis. For example:

- For smaller features that don't involve a lot of complexity, steps 2 and 3 will often be combined into a single document.
- If the feature involves a large number of unknowns or some level of research, it may be necessary to construct a proof-of-concept implementation before finalizing the technical design.

This process also happens at different scales and levels of granularity. A PRD / TRD / TDD may concern the design of an entire system, or just a single feature. In most environments, the process is also cyclic — each design/implement cycle builds on the work of the previous one.

The dividing line between TRD and TDD can be a bit blurry at times. For example, suppose you are developing a server that communicates via a RESTful API. If the goal is to conform to an already-established and documented API, then the API specification is part of the requirements and should be referenced in the TRD. If, on the other hand, the goal is to develop a brand new API, then the API specification is part of the design and should be described in the TDD. (However, the requirements document still needs to specify what the API is trying to accomplish.)

Writing the TDD

These days, it is common practice to write technical docs in a collaborative document system, such as Google Docs or Confluence; however this is not an absolute requirement. The important thing is that there be a way for your team members to be able to make comments on the document and point out errors and omissions.

Most TDDs are between one and ten pages. Although there's no upper limit to the length of a TDD, very large documents will be both difficult to edit and hard for readers to absorb; consider breaking it up into separate documents representing individual steps or phases of the implementation.

Diagrams are helpful; there are a number of online tools that you can use to embed illustrations into the document, such as draw.io or Lucidchart. You can also use offline tools such as Inkscape to generate SVG diagrams.

The document should be thorough; ideally, it should be possible for someone other than the TDD author to implement the design as written. For example, if the design specifies an implementation of an API, each API endpoint should be documented. If there are subtle design choices, they should be called out.

Avoid Common Writing Mistakes

Probably the most common mistake that I encounter in TDDs is a lack of context. That is, the author wrote down, in as few words as they could manage, how they solved the problem; but they didn't include any information on what the problem was, why it needed to be solved, or what were the consequences of picking that particular solution.

Also, it's important to keep in mind who the likely reader is, and what level of understanding they have. If you use a term that the reader might not know, don't be afraid to add a definition for it.

It hardly needs to be stated that good grammar and spelling are helpful. Also, avoid the temptation for wordplay or "cute" spelling; while programmers as a class tend to like playing around with language, I've seen more than one case where excessive frivolity ended up costing the team wasted effort because of misunderstandings. It's all right to use occasional humor or choose colorful, memorable names for features and systems, since that helps people remember them. But don't let your desire to show off how clever you are become a distraction.

Speaking of names, choose them carefully; as Mark Twain once wrote, "Choose the right word, not it's second cousin." There's a tendency for engineers with poor vocabularies to use the same generic terms over and over again for different things, leading to overloading and confusion. For example, naming a class "DataManager" is vague and tells you nothing about what it actually does; by the same token a package or directory named "utils" could contain virtually anything. Consult a thesaurus if you need to find a better word, or better, a specialized synonym database such as WordNet.

TDD Template

When writing a TDD, it can be helpful to start with a standard template. The following is a template that I have used in a number of projects. Note that this template should be customized where needed; you are free to delete sections which don't apply, add additional sections, or rename headings as appropriate.

<Title> TDD

Author: <Your Name>

Introduction

Rationale

What are you trying to accomplish? What's wrong with things the way they are now?

Background

Describe any historical context that would be needed to understand the document, including legacy considerations.

Terminology

If the document uses any special words or terms, list them here.

Non-Goals

If there are related problems that you have decided not to address with this design, but which someone might conceivably expect you to solve, then list them here.

Proposed Design

Start with a brief, high-level description of the solution. The following sections will go into more detail.

System Architecture

If the design consists of a collaboration between multiple large-scale components, list those components here — or better, include a diagram.

Data Model

Describe how the data is stored. This could include a description of the database schema.

Interface/API Definitions

Describe how the various components talk to each other. For example, if there are REST endpoints, describe the endpoint URL and the format of the data and parameters used.

Business Logic

If the design requires any non-trivial algorithms or logic, describe them.

Migration Strategy

If the design incurs non-backwards-compatible changes to an existing system, describe the process whereby entities that depend on the system are going to migrate to the new design.

Impact

Describe the potential impacts of the design on overall performance, security, and other aspects of the system.

Risks

If there are any risks or unknowns, list them here. Also if there is additional research to be done, mention that as well.

Alternatives

If there are other potential solutions which were considered and rejected, list them here, as well as the reason why they were not chosen.

Of course, these sections are only starting points. You can add additional sections such as “Design Considerations”, “Abstract”, “References”, “Acknowledgements”, and so on as appropriate.

TDD Lifecycle

During construction of the system, the TDD serves as a reference, coordinating the activities of the team members working on the project. However, after construction is finished, the TDD will continue to exist and serve as documentation for how the system works. You may want to distinguish between “current” and “archived” TDDs.

However, there are two perils to watch out for:

First, TDDs can quickly become out of date as the system continues to evolve. An engineer using a two-year-old TDD as a reference can waste a lot of time trying to understand why the system doesn’t behave as described. Ideally, stale TDDs would be marked as obsolete or superseded; in practice this seldom happens, as teams tend to focus on current rather than past work. (Keeping documentation up to date is a challenge that every engineering team struggles with.)

Second, a TDD may not include all of the information needed to interface with the system. A TDD might only cover a set of changes to an already-existing system, in which case you would need to consult earlier documentation (if it exists) to get the whole picture. And a TDD mainly

focuses on implementation details, which may be irrelevant to someone who simply wants to invoke an API.

Thus, a TDD should not be considered an adequate substitute for actual user or API reference docs.

Finally

There are plenty of other articles on the web explaining how to write a great design doc. Don’t just read this one! Read several, and then pick a mix of ideas that is right for you.

Update

There’s a follow-on article, Writing Technical Design Documents, Revisited, that provides some additional information.