Effective design docs

✏ 2024-09-15 ✂ 2024-09-16

Design docs are a controversial topic, especially among agile developers who "value working software over comprehensive documentation." Yet, all engineering organizations I worked at in the last decade, from tech giants to lean startups, employed writing design docs as an essential part of the development process.

This article is an opinionated guide to writing design docs for software projects. It explains why and when to write design docs, how to think like a researcher, how to put words on the page like Neil Gaiman, and what we can learn from the publishing industry.

Types of design docs

There are at least two types of documents that people call design docs: a *functional specification* describes what a system must do from the user's point of view, and a *design doc* describes a software architecture or an approach to solving a technical problem.

These types serve different purposes and require different workflows:

> Functional specs describe the system from the user's perspective; design documents deal with its internals.
>
> Functional specs describe the behavior of the entire system; design docs focus on solving a specific problem.
>
> Engineers write design documents for themselves, and product managers write functional specs for diverse audiences, such as engineers, external developers, and QA staff.
>
> Functional specs must evolve with the product; design docs are static.

The following table summarizes the differences.

	FUNCTIONAL SPEC	DESIGN DOC
Target audience	Diverse	Engineers
Abstraction level	Interface	Implementation
Author	Product manager	Engineers
Scope	Entire system	Specific problem
Evolution model	Evolving	Static

We can imagine a third document type: a *technical specification* describing the implementation details of the entire system and evolving with it. Unfortunately, I've never seen this idea working:

> The implementation moves too fast; a technical spec quickly becomes obsolete.
>
> The document's ownership is unclear. Nobody knows (and probably can't) all aspects of a large system. When many people own a document, nobody owns it.
>
> The system's source code is its most detailed technical specification.

This article deals with the second documentation type: design docs.

Why write design docs

To put thoughts in order

> *Writing is nature's way of letting you know how sloppy your thinking is.*
>
> Guindon, San Francisco Chronicle, 3/1/89 Guindon

No matter how clear an idea seems in your head, the first attempt to express it in writing turns it into mashed potatoes. This mysterious

property of writing is the main benefit of writing a design doc. It forces you to formulate your problem and a solution before you fiddle with code. Even if nobody reads your design doc, you save time by writing things down: If your architecture and interfaces look bad on paper, the code implementing them will look even worse.

To correct mistakes early

The cheapest mistakes are the ones you correct early. Your first design will be flawed; a design doc will enable your colleagues to point out flaws in your ideas and refine them before you invest weeks in implementing them.

To communicate context

One of my most frustrating experiences as a code reviewer was when colleagues from another side of the globe asked me to approve a sizeable controversial change that touched the interface between our components. The diffs were all over the place, and the system went in a direction that didn't feel right. The change author claimed everything should become clear once I see other (not yet written) patches in the sequence. Yeah, sure.

According to Christian Bird and other scientists who studied code reviews at Microsoft, the biggest challenges reviewers face are *large code changes* and *understanding the reason for a change*[1]. When you implement your designs in small incremental patches to address the first challenge, a reference to the design doc in the change description takes care of the second.

To onboard new team members

Sociologist Karl Maton envisions optimal learning of a new concept as riding a Semantic wave. The learner starts with a high-level, technical description of the concept, then *unpacks* details using

simpler context-dependent language, and returns to the high-level description, enlightened (Maton calls this last step *repacking*)[2].

Design docs are among the best resources for onboarding new team members. Instead of painfully deriving the purpose and structure of a system from the code base, they can get a high-level overview from a design doc, and then dive into the codebase with enough mental hooks to anchor their discoveries.

To record history

> *An engineering project shouldn't be considered complete until it is summarized and filed so that the information can be recalled or used again.*
>
> W.J. King, Unwritten Laws of Engineering, second edition

Have you ever worked on a project that felt like the five-monkey experiment? Everyone on the project does something in a peculiar way, but nobody remembers why exactly, and everyone is afraid of challenging the status quo.

Design docs are historical records of your team's decisions and their reasoning; they are a solution to Chesterton's fence problem. Thanks to these records, future designers will know whether the constraints you codified still apply to their context.

When to write a design doc

Most changes don't need a design doc. My heuristic is to start a design doc when one of the following conditions is true:

The change requires more than two weeks of engineering

work.

The problem has multiple solutions, and the optimal choice is not apparent.

The design involves non-trivial changes between software components.

Most importantly, write designs before you start implementing the system. Documenting the system post-factum takes away most of the benefits of writing a design doc. Furthermore, once the system works, you'll be too eager to move on and view documentation as a drag, so the chance of producing anything of value becomes infinitesimal.

How to write a design doc

> *Writing papers is a primary mechanism for doing research (not just for reporting it).*
>
> Simon Peyton Jones, How to Write a Good Research Paper

Most people hate writing and will do almost anything instead of putting words on the page: read Slack, help colleagues, stare at metrics dashboards, consume ungodly amounts of coffee, or even groom their Jira backlog. There is an easy fix for this problem: schedule your writing sessions. Neil Gaiman allows himself to do only two things during his writing sessions: sit in front of the document and do nothing or write. After some time, putting words in will seem more fun than just sitting there.

Unfortunately, engineers can't always follow the same routines as novelists. Fiction writers disengage from the world when they create their masterpieces. Software designs conceived in isolation

look plausible on paper but disintegrate once they meet reality. To avoid this trap, build prototypes—miniature versions of the system—to test whether your ideas hold water.

Should you start by building prototypes or drafting the doc? The research community has an answer. Design docs are not novels; they are research papers: Your problem is to build or reshape a piece of software, and your goal is to convince yourself and your peers that your plan is the best option. In his talk How to Write a Good Research Paper, Simon Peyton Jones, a former principal researcher at Microsoft Research, recommends starting with a paper and using it to drive the research. This recommendation suggests a workflow alternating writing and prototyping:

> Start with the background section. Explain the problem you're trying to solve, and enumerate your assumptions, constraints, goals, and non-goals.
>
> Lay out the design alternatives.
>
> Research the alternatives in sufficient depth and build prototypes when necessary. Go back and forth between writing the doc and doing the research.
>
> Pick the best design and describe it in detail. Explain why all other options are worse.
>
> Write the summary.

The summary should be the first section of a design doc, but you write it last. You can't summarize the research you haven't done. Even Mozart, who could envision large pieces of music in his head and later record them on paper speckless in one sitting, composed overtures for his operas only after finishing the rest of the score; he needed to know all the themes to create a perfect introduction into his musical worlds. Famously, he wrote an overture for Don Giovanni the night before the premiere.

What should go into a design doc

There is no one-size-fits-all template for design docs.⊕ Experiment to find what works best for your organization. Most design docs contain the following sections in one form or another:

Metadata. Place the author, creation time, and status near the top of the document. The reader should know whether to read it and whom to ask about it.

Summary. This section provides a bird's-eye view of the entire document: a single paragraph explaining the problem and the solution in the simplest terms.

Context/Background. Explain the current state of affairs and the problem you aim to solve. This section should make sense to a secondary audience, not only experts in the field.

Goals/Non-goals. State which aspects of the problem are inside and outside the project scope. For example, in an early version of a system, you might avoid handling performance considerations or integrations with external systems.

Proposed design. Go into details of your key idea. Include diagrams, schemas, and back-of-envelope calculations. Cover all the significant design aspects relevant to your field: security, privacy, scalability, portability, observability, accessibility, and backward compatibility.

Alternatives considered. Describe other design options you considered and why you chose your primary option. Your proposal is unlikely to be strictly better than alternatives on all dimensions; list the upsides and downsides of each approach. If possible, add a comparison table where one dimension is design options, and another is design criteria (complexity, cost, delivery time estimates, etc.).

Seeking feedback

If nobody reads or comments on your design doc, you lose most of the benefits of writing it. Unfortunately, tricking other people into reading your writing is hard. As Steven Pressfield puts it, “nobody wants to read your sh*t.” Furthermore, making engineers re-read new revisions of a document is nearly impossible.

When seeking feedback, we face a dilemma. If you request feedback too early, your colleagues will point out the most obvious flaws and probably never give your document another chance. On the other hand, if you slave on a doc for weeks until it becomes “perfect,” you will likely waste time developing bad ideas.

One of my colleagues found a solution to this dilemma during his time at Jane Street: each document writer should have a buddy. Once the doc has enough substance, the buddy helps the author polish the document before requesting feedback from a wider audience. This approach is a miniature version of the publishing industry workflow, where the author iterates on the manuscript with a dedicated editor before the text goes to print.

People find walls of text scary and procrastinate reading them. The best way to make your doc readable is to keep it short and to the point. Cut ruthlessly, make the structure apparent, and keep the language simple. Another way to trick people into reading is to make the document visually appealing. To combat monotonicity, intersperse the text with diagrams, use lists, and be generous with blank space. Refer to the Resources section for book recommendations that can help you with structural and visual components.

FAQ

Should I update a design doc as the code evolves?

Probably not. Design doc captures your thinking at a specific point in time. If the situation changes and you must revise the design, write a new document referencing the original. Scientists don't edit their published papers; they write new ones.

However, feel free to modify the design doc if you change your mind or discover new challenges during development. Editorial changes that make the doc more accessible are also welcome.

Resources

`designdocs.dev` offers examples and templates of software design docs.

The "Painless Functional Specifications" articles by Joel Spolsky cover many aspects of authoring a functional spec: Part 1: Why Bother, Part 2: What's a Spec?, Part 3: But… How?, Part 4: Tips.

Trees, maps, and theorems contains excellent advice on all steps of authoring a technical document, emphasizing structure.

Bugs in Writing gives you tactics for writing good technical English. The previous book is about the forest and trees; this one is about the leaves.

How to Write a Lot teaches you how to become a productive writer. In short, schedule your writing time and do writing-related work during these sessions.

If you can read only one book on graphical design, let it be

The Non-Designer's Design Book.

Similar articles

ONNX introduction

Box combinators

Transposing tensor files

The plan-execute pattern

Enlightenmentware

←Programming avant-garde
Transaction models are programming paradigms→