The software guidebook

Simon Brown

The software guidebook

A guide to documenting your software architecture.

Simon Brown

This book is for sale at http://leanpub.com/documenting-software-architecture

This version was published on 2022-05-28

Contents

1. About the author

I'm an independent software development consultant specialising in software architecture; specifically technical leadership, communication and lightweight, pragmatic approaches to software architecture. In addition to being the author of Software Architecture for Developers, I'm the creator of the C4 software architecture model and I built Structurizr, which is a collection of tooling to help you visualise, document and explore your software architecture.

I regularly speak at software development conferences, meetups and organisations around the world; delivering keynotes, presentations and workshops about software architecture. In 2013, I won the IEEE Software sponsored SATURN 2013 "Architecture in Practice" Presentation Award for my presentation about the conflict between agile and architecture. I've spoken at events and/or have clients in over thirty countries around the world.

You can find my website at simonbrown.je and I can be found on Twitter at @simonbrown.

2. Software documentation as a guidebook

"Working software over comprehensive documentation" is what the Manifesto for Agile Software Development says and it's incredible to see how many software teams have interpreted those five words as "don't write *any* documentation". The underlying principle here is that real working software is much more valuable to end-users than a stack of comprehensive documentation, but many teams use this line in the agile manifesto as an excuse to not write any documentation at all.

Unfortunately not having a source of supplementary information about a complex software system can slow a team down as they struggle to navigate the codebase.

The code doesn't tell the whole story

We all know that writing good code is important and refactoring forces us to think about making methods smaller, more reusable and self-documenting. Some people say that comments are bad and that self-commenting code is what we should strive for. However you do it, everybody *should* strive for good code that's easy to read, understand and maintain. But the code doesn't tell the whole story.

Let's imagine you've started work on a new software project that's already underway. The major building blocks are in place and some of the functionality has already been delivered. You start up your development machine, download the code from the source code control system and load it up into your development environment. What do you do next and how do you start being productive?

If nobody has the time to walk you through the codebase, you can start to make your own assumptions based upon the limited knowledge you have about the project, the business domain, your expectations of how the team builds software and your knowledge of the technologies in use.

For example, you might be able to determine something about the overall architecture of the software system through how the codebase has been broken up into sub-projects, directories, packages, namespaces, etc. Perhaps there are some naming conventions in use. A further deep-dive through the code will help to prove your initial assumptions right or wrong, but it's also likely to leave you with a whole host of questions. Perhaps you understand what the system *does* at a high level, but you don't understand things like:

- How the software system fits into the existing system landscape.
- Why the technologies in use were chosen.
- The overall structure of the software system.
- Where the various components are deployed at runtime and how they communicate.
- How the web-tier "knows" where to find the middle-tier.
- What approach to logging/configuration/error handling/etc has been adopted and whether it is consistent across the codebase.
- Whether any common patterns and principles are in use across the codebase.
- How and where to add new functionality.
- How security has been implemented across the stack.
- How scalability is achieved.
- How the interfaces with other systems work.
- etc

I've been asked to review and work on systems where there has been no documentation. You can certainly gauge the answers to most of these questions from the code but it can be hard work. Reading the code will get you so far but you'll probably need to ask questions to the rest of the team at some point. And if you don't ask the right questions, you won't get the right answers - you don't know what you don't know.

Our duty to deliver documentation

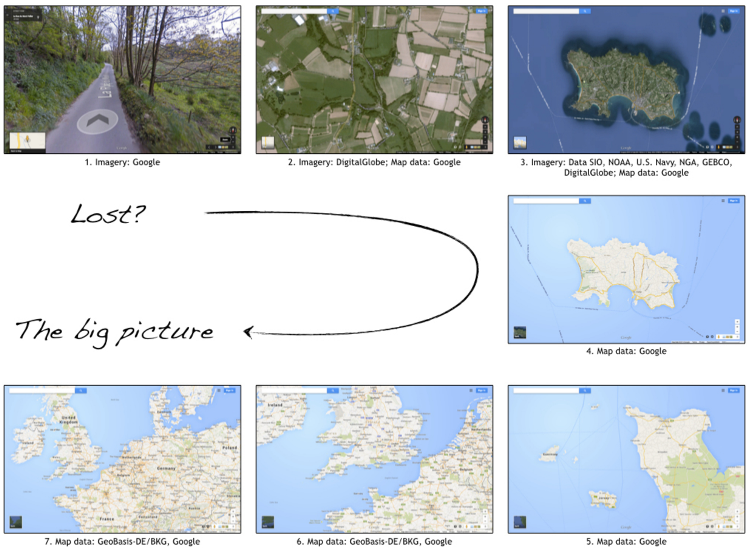
I'm also a firm believer that many software teams have a duty to deliver some supplementary documentation along with the codebase, especially those that are building the software under an outsourcing and/or offshoring contract. I've seen IT consulting organisations deliver highly complex software systems to their customers without a single piece of supporting documentation, often because the team doesn't *have* any documentation. If the original software developers leave the consulting organisation, will the new team be able to understand what the software is all about, how it's been built and how to enhance it in a way that is sympathetic to the original architecture? And what about the poor customer? Is it right that they should *only* be delivered a working codebase?

Lightweight, supplementary documentation

The problem is that when software teams think about documentation, they usually think of huge Microsoft Word documents based upon a software architecture document template from the 1990's that includes sections where they need to draw UML class diagrams for every use case that their software supports. Few people enjoy reading this type of document, let alone writing it! A different approach is needed. I like to think about supplementary documentation as an ever-changing travel guidebook rather than a comprehensive static piece of history. But what goes into a such a guidebook?

1. Maps

Let's imagine that I teleported you away from where you are now and dropped you in a quiet, leafy country lane somewhere in the world (picture 1). Where are you and how do you figure out the answer to this question? You could shout for help, but this will only work if there are other people in the vicinity. Or you could simply start walking until you recognised something or encountered some civilisation, who you could then ask for help. As geeks though, we would probably fire up the maps application on our smartphone and use the GPS to pinpoint our location (picture 2).


1. Imagery: Google
2. Imagery: DigitalGlobe; Map data: Google
3. Imagery: Data SIO, NOAA, U.S. Navy, NGA, GEBCO, DigitalGlobe; Map data: Google
Lost?
The big picture
4. Map data: Google
7. Map data: GeoBasis-DE/BKG, Google
6. Map data: GeoBasis-DE/BKG, Google
5. Map data: Google

From the detail to the big picture

The problem with picture 2 is that although it may show our location, we're a little too "zoomed in" to potentially make sense of it. If we zoom out further, eventually we'll get to see that I teleported you to a country lane in Jersey (picture 3).

The next issue is that the satellite imagery is showing a lot of detail, which makes it hard to see where we are relative to some of the significant features of the island, such as the major roads and places. To counter this, we can remove the satellite imagery (picture 4). Although not as detailed, this abstraction allows us to see some of the major structural elements of the island along with some of the place names, which were perviously getting obscured by the detail. With our simplified view of the island, we can zoom out further until we get to a big picture showing exactly where Jersey is in Europe (pictures 5, 6 and 7). All of these images show the same location from different levels of abstraction, each of which can help you to answer different questions.

If I were to open up the codebase of a complex software system and highlight a random line of code, exploring is fun but it would take a while for you to understand where you were and how the code fitted into the software system as a whole. Most integrated development environments have a way to navigate the code by namespace, package or folder but often the physical structure of the codebase is different to the logical structure. For example, you may have many classes that make up a single component, and many of those components may make up a single deployable unit.

Diagrams can act as maps to help people navigate a complex codebase and this is one of the most important parts of supplementary software documentation. Ideally there should be a small number of simple diagrams, each showing a different part of the software system or level of abstraction. Visualisation of software architecture was the focus of the first part of this book.

2. Sights

If you ever visit Jersey, and you should because it's beautiful, you'll probably want a map. There are visitor maps available at the ports and these present a simplified view of what Jersey looks like. Essentially the visitor maps are detailed sketches of the island and, rather than showing every single building, they show an abstract view. Although Jersey is small, once unfolded, these maps can look daunting if you've not visited before, so what you ideally need is a list of the major points of interest and sights to see. This is one of the main reasons that people take a travel guidebook on holiday with them. Regardless of whether it's physical or virtual (e.g. an e-book on your smartphone), the guidebook will undoubtedly list out the top sights that you should make a visit to.

A codebase is no different. Although we *could* spend a long time diagramming and describing every single piece of code, there's really little value in doing that. What we really need is something that lists out the points of interest so that we can focus our energy on understanding the major elements of the software without getting bogged down in all of the detail. Many web applications, for example, are actually fairly boring and rather than understanding how each of the 200+ pages work, I'd rather see the points of interest. These may include things like the patterns that are used to implement web pages and data access strategies along with how security and scalability are handled.

3. History and culture

If you do ever visit Jersey, and you really should because it *is* beautiful, you may see some things that look out of kilter with their surroundings. For example, we have a lovely granite stone castle on the south coast of the island called Elizabeth Castle that was built in the 16th century. As you walk around admiring the architecture, eventually you'll reach the top where it looks like somebody has dumped a large concrete cylinder, which is not in keeping with the intricate granite stonework generally seen elsewhere around the castle.

A joining of two distinct architectural styles

As you explore further, you'll see signs explaining that the castle was refortified during the German occupation in the second world war. Here, the history helps explain why the castle is the way that it is.

Again, a codebase is no different and some knowledge of the history, culture and rationale can go a long way in helping you understand why a software system has been designed in the way it was. This is particularly useful for people who are new to an existing team.

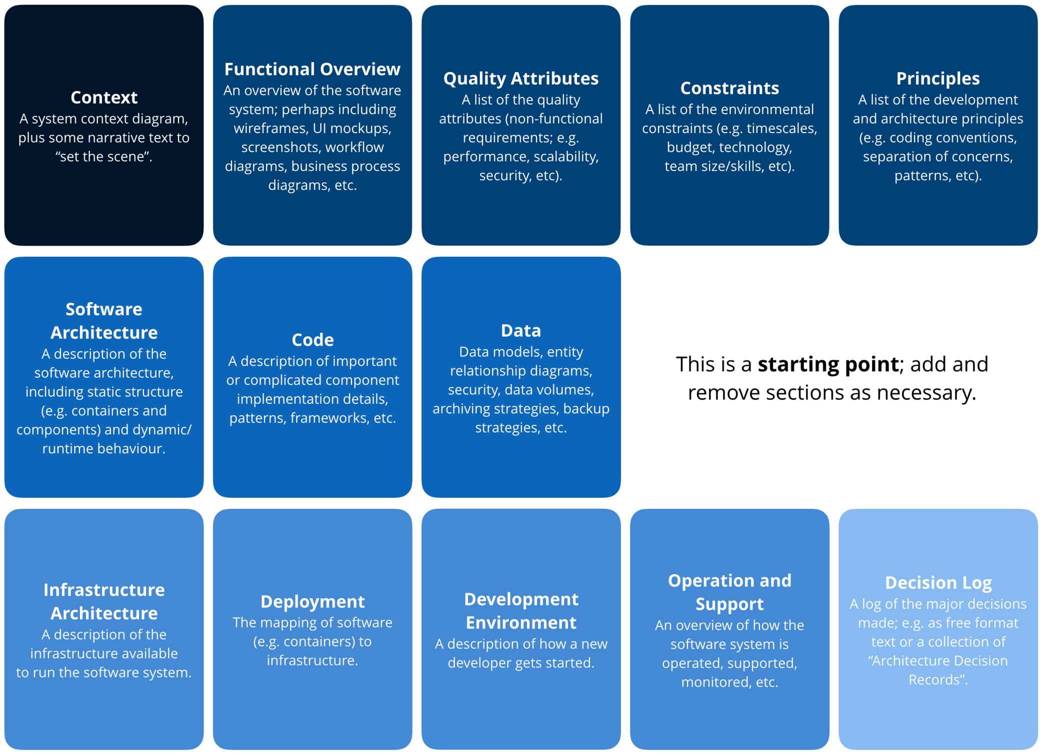
4. Practical information

The final thing that travel guidebooks tend to include is practical information. You know, all the useful bits and pieces about currency, electricity supplies, immigration, local laws, local customs, how to get around, etc.

If we think about a software system, the practical information might include where the source code can be found, how to build it, how to deploy it, the principles that the team follow, etc. It's all of the stuff that can help the software developers, support staff, etc do their job effectively.

Describe what you can't get from the code

Exploring is great fun but ultimately it takes time, which we often don't have. Since the code doesn't tell the whole story, *some* supplementary documentation can be very useful, especially if you're handing over the software to somebody else or people are leaving and joining the team on a regular basis. My advice is to think about this supplementary documentation as a guidebook, which should give people enough information to get started and help them accelerate the exploration process.



Context
A system context diagram, plus some narrative text to "set the scene".

Functional Overview
An overview of the software system; perhaps including wireframes, UI mockups, screenshots, workflow diagrams, business process diagrams, etc.

Quality Attributes
A list of the quality attributes (non-functional requirements; e.g. performance, scalability, security, etc).

Constraints
A list of the environmental constraints (e.g. timescales, budget, technology, team size/skills, etc).

Principles
A list of the development and architecture principles (e.g. coding conventions, separation of concerns, patterns, etc).

Software Architecture
A description of the software architecture, including static structure (e.g. containers and components) and dynamic/runtime behaviour.

Code
A description of important or complicated component implementation details, patterns, frameworks, etc.

Data
Data models, entity relationship diagrams, security, data volumes, archiving strategies, backup strategies, etc.

This is a **starting point**; add and remove sections as necessary.

Infrastructure Architecture
A description of the infrastructure available to run the software system.

Deployment
The mapping of software (e.g. containers) to infrastructure.

Development Environment
A description of how a new developer gets started.

Operation and Support
An overview of how the software system is operated, supported, monitored, etc.

Decision Log
A log of the major decisions made; e.g. as free format text or a collection of "Architecture Decision Records".

Do resist the temptation to go into too much technical detail though because the technical people that will understand that level of detail will know how to find it in the codebase anyway. As with everything, there's a happy mid-point somewhere. The following headings describe what you might want to include in a software guidebook:

1. Context
2. Functional Overview
3. Quality Attributes
4. Constraints
5. Principles
6. Software Architecture
7. Code
8. Data
9. Infrastructure Architecture
10. Deployment
11. Operation and Support
12. Development Environment
13. Decision Log

There are, of course, a number of different documentation templates available, and this is my starting point for my own documentation. I would also recommend taking a look at arc42, which captures the same sort of information in a slightly different format, and the "Building Block View" complements the C4 model nicely.

Product vs project documentation

As a final note, the style of documentation that I'm referring to here is related to the *product* being built rather than the *project* that is creating/changing the product. A number of organisations I've worked with have software systems approaching twenty years old and, although they have varying amounts of *project-level* documentation, there's often nothing that tells the story of how the product works and how it's evolved. Often these organisations have a single product (software system) and every major change is managed as a separate project. This results in a huge amount of change over the course of twenty years and a considerable amount of project documentation to digest in order to understand the current state of the software. New joiners in such environments are often expected to simply read the code and fill in the blanks by tracking down documentation produced by various project teams, which is time-consuming to say the least!

I recommend that software teams create a single software guidebook for every software system that they build. This doesn't mean that teams shouldn't create project-level documentation, but there should be a single place where somebody can find information about how the product works and how it's evolved over time. Once a single software guidebook is in place, every project/change-stream/timebox to change that system is exactly that - a small delta. A single software guidebook per product makes it much easier to understand the current state and provides a great starting point for future exploration.

Keeping documentation up to date

We'll talk about tooling later but, from a process perspective anyway, my approach to documentation is "little and often". If you have a "definition of done" for your work items, simply add another line to the bottom that says something like, "Documentation and diagrams updated". From my own experience, doing a number of small documentation updates is much more palatable than sitting down for a few weeks with the sole purpose of writing documentation!

Documentation length

How long should documentation be? I get asked this question a lot. And rather than talk about numbers of pages, what I'm really looking for is something that I can read in a couple of hours, over a coffee or two, to get a really good *starting point for exploring the code.*