Extracted from:

Design It!

From Programmer to Software Architect

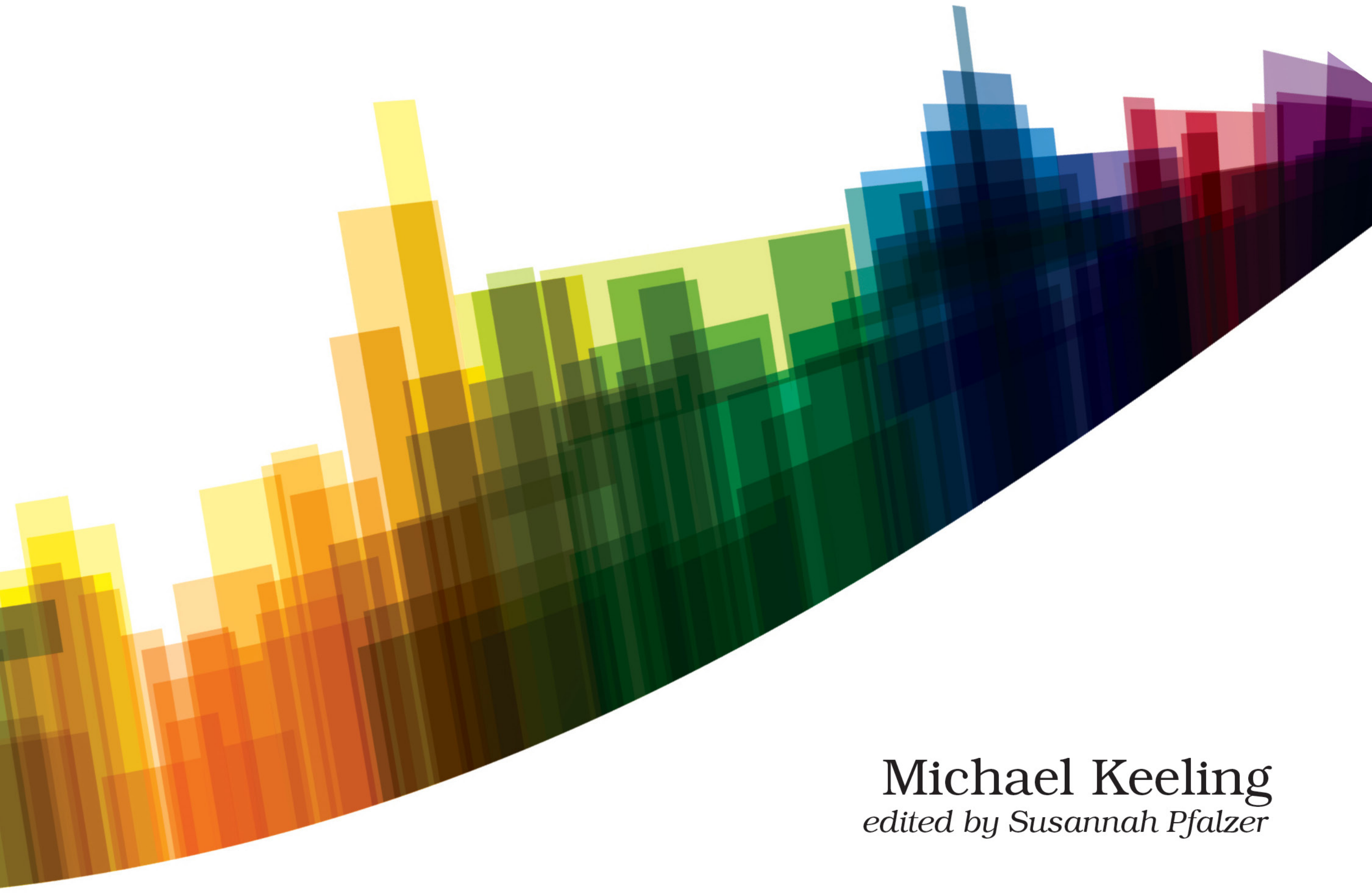
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Design It!

From Programmer to Software Architect



Michael Keeling

edited by Susannah Pfalzer

Design It!

From Programmer to Software Architect

Michael Keeling

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

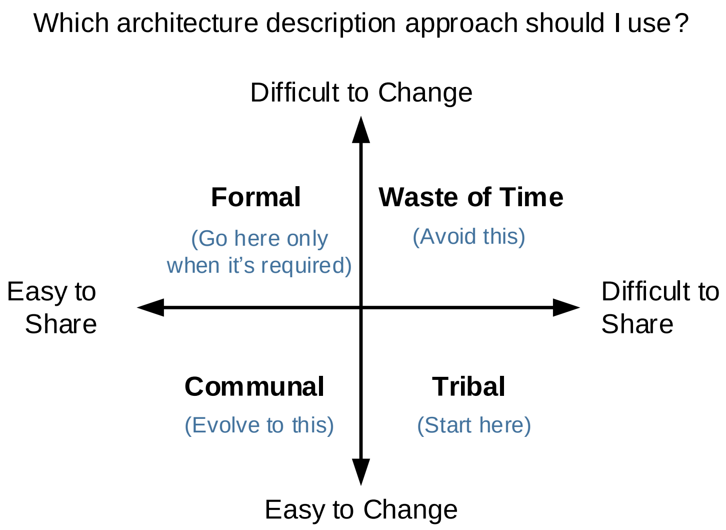
Publisher: Andy Hunt
VP of Operations: Janet Furlow
Development Editor: Susannah Davidson Pfalzer
Indexing: Potomac Indexing, LLC
Copy Editor: Liz Welch
Layout: Gilson Graphics

Printed in the United States of America.
ISBN-13: 978-1-68050-209-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2017

Match the Description Method to the Situation

There is no one-size-fits-all approach to creating software architecture descriptions. Less mature systems might change their architecture several times in a single conversation. A co-located team working on a smaller system can get by with whiteboards and storytelling. A system built for a regulated industry may be legally required to document design decisions in a specific way.

There are two questions we need to answer to decide how to document our architecture. How likely are our design decisions to change? And how far must we share our design decisions? Depending on how you answer these questions, you'll arrive at one of four types of description methods: tribal, communal, formal, or wasteful.



Create an Oral History with Tribal Methods

Tribal description methods rely heavily on oral tradition and cultural artifacts. Storytelling, metaphors, and informal sketching are all examples of tribal architecture descriptions. Always start here. Tribal descriptions are easy to change, which is perfect for the rapid design churn young architectures face.

While tribal methods are easy to create and change, they are also difficult to share. A system metaphor, described in Activity 29, *System Metaphor*, on page ?, might work well when your whole team is within earshot of your conversation, but oral histories are only alive when someone can tell the story.[1] Constantly telling stories can be exhausting, even on small teams of only half a dozen people.

Reach Further with Communal Methods

Not all aspects of an architecture evolve at the same rate or require the same kinds of description. As a rule of thumb, if you find yourself telling the same story to more than a few people, then it's time to evolve the style of architecture description to increase reach. Enter communal methods.

Communal architecture description methods are shared by the community, not just individuals of the tribe. Architecture haiku (Activity 21, *Architecture Haiku*, on page ?), architecturally evident coding styles (described on page ?), and architecture decision records (Activity 20, *Architecture Decision Records*, on page ?) are all examples of communal description methods. Communal descriptions are still easy to change, but they are also easy to share compared to tribal methods.

Most teams will evolve from tribal descriptions to communal descriptions naturally as the architecture matures and the rate of change decreases. Communal description methods are good enough for many teams. In some situations, we want something more permanent. We can get this permanence with formal architecture descriptions.

Invest in Formal Descriptions Only When Required

Formal description methods include traditional architecture description documents and formal models. These tend to be larger documents and require more effort to create. Formal models (the sort that defines the architecture with a mathematical model) require a higher degree of accuracy and precision.

High-risk systems or architecture decisions requiring a high degree of coordination are good candidates for formal description methods. Depending on your industry, you may be required to create a formal document. Even then, it pays to start with tribal methods and progress to communal methods before building formal descriptions.

Instead of starting with a traditional document, start with whiteboard sketches and a system metaphor. Record ADRs one at a time as you make decisions. Once you've made some decisions, bring everything together as an architecture haiku and continue refactoring code so that it reflects the models

1. Michael Keeling. *Creating an Architecture Oral History: Minimalist Techniques for Describing Systems*. SATURN 2012. http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=20330

in your design. After the architecture starts to coalesce, if desired (or when a stakeholder asks for it), create a traditional document.

Create a Traditional Software Architecture Description

The *traditional software architecture description*, or SAD, is a classic design document that every architect should know how to write. While these documents are time-consuming to create, they are worth their weight in gold. This does not mean you should make the document as large as possible! What I mean is that a traditional SAD is well worth the effort.

Most stakeholders—developers included—have probably never seen the architecture as a whole. The SAD is the one artifact that brings everything together to tell the whole story.

Traditional architecture documents might run 10 or 20 pages depending on the template used. I've written only a few SADs that have run 70 pages or more. Keep in mind these page counts include all the pomp and circumstance that goes along with a formal document. Even not counting the fluff, a traditional SAD is an investment.

Start a traditional architecture description by building or finding a template. You'll find many templates available online.[2] Your organization may even have a template. I strongly recommend the Software Engineering Institute's *Views and Beyond* [3] and the ISO/IEC/IEEE 42010 standard templates.[4]

All traditional architecture descriptions include the same basic parts:

Introductions and preamble information This includes the title page, update notes, signature page, table of contents, a list of figures, licensing and legal boilerplate, and other information required of a formal document. The table of contents and figure lists help readers navigate the document more easily. The rest is meant to convey the importance of the information held within the document. Some stakeholders find these preambles impressive. Remember, you may be required by your organization to include some of this information.

Overview and introduction to the SAD Briefly describes the purpose of the document as well as the methodology used to organized and create it. Your SAD could be the first time some stakeholders have read an archi-

2. http://www.iso-architecture.org/42010/applications.html
3. http://www.sei.cmu.edu/architecture/tools/document/viewsandbeyond.cfm
4. http://www.iso-architecture.org/42010/templates/

tecture description. Take this opportunity to educate them just-in-time so they can appreciate the architecture designed for them.

Summary of stakeholders, business goals, and architecturally significant requirements Since all decisions in our architecture flow from stakeholders' concerns, list them before describing the design. I like to summarize key constraints and quality attributes here as well. If you've created an ASR Workbook (introduced on page ?), then add a reference to it. Strive to keep the architecture description DRY (Don't Repeat Yourself), just like your code.

Context view Provides an overview of where the software system fits in the world. See Activity 22, *Context Diagram*, on page ? for details.

Relevant views As we discussed in *Show the Architecture from Different Views*, on page ?, architecture is too big and complex to show in one diagram. We need to create multiple views of the architecture to explain how it satisfies quality attributes and other requirements. A list of views is not very consumable, so to help our readers we'll organize views around a related set of stakeholder concerns. Each *viewpoint* shows views needed to reason about something a stakeholder cares about, such as a set of related quality attribute scenarios. You'll learn more about using viewpoints in *Organize Views around Stakeholders' Concerns*, on page ?.

Risks, open questions, future work Include a section for known risks and open questions. The purpose of these sections is to shine a light on the land mines you already know about so downstream designers can hopefully avoid them.

Appendices At a minimum include a term glossary and list of acronyms with expansions. I recommend you include a quality attribute taxonomy as an appendix as well. Some formal documents will also include change procedures and change request templates.

Creating a SAD can be exhausting. Work as a team to complete the document. Designate one person as the *Master of the SAD*. The Master of the SAD creates the template and decides who will write each section of the document. The Master of the SAD is also responsible for making sure the document is complete and has a consistent style.

Avoid Wasting Time

For the sake of completeness, there is a quadrant on our grid made up of difficult-to-change, difficult-to-share description methods. If you find yourself here, it's time to try something different. Let's look at two examples of waste-of-time description methods.

One example of a possible waste of time is the *slideware architecture description*. Presentations are a powerful tool for architects. The problem with slides is that they rarely stand on their own and can be difficult to change. Someone must present them to make sense. Someone spent hours getting all the transitions and layered boxes and connecting arrows to look just right. After so much work, people hesitate to change their beautifully laid-out diagrams, even when the world has changed. Compared to tribal methods, slides are etched in stone.

The best way to avoid wasting time is by creating great architecture descriptions. All great descriptions, no matter what method you use, have four traits:

1. They are custom built with the audience in mind.
2. They show multiple views of the architecture.
3. They clearly define the elements and their responsibilities.
4. They explain the rationale for design decisions.

In the next sections, you'll learn what these traits mean and how to put them into practice to create fantastic architecture descriptions.

George says:
Tell a Story at Many Levels

By George Fairbanks, software engineer at Google and author of Just Enough Software Architecture: A Risk-Driven Approach [Fai10]

We've all dropped into a project and struggled to understand the code. Perhaps you are browsing the repository of an open source project and find just one folder with hundreds of source files. It takes a lot of effort for you to infer how things work and you probably make mistakes.

It doesn't have to be that way. You can organize your code as a *story at many levels* so that it makes sense and tells a story as you zoom in or out. Consider what you ate yesterday evening. You recognize it as dinner, and zooming in you also recognize courses, then dishes, then ingredients.

The levels help you think clearly. If you're wondering how long dinner will last, thinking about the number of courses is helpful and thinking about allergies is best done by ingredients. But the reverse doesn't help at all.

People have been thinking about dinner for a lot longer than software, so it's already baked into our language. You will have to invent your story and levels, though you can lean on developers who have come before you. Architecture patterns give you general names to use like *connectors* and *layers*, and specific ones like *reduce stage* and *broker*.

Your story at many levels won't come for free, so you will spend time gardening, making minor refactorings as you go so that the story stays clear. Today you might have just three connectors, so they all go in the same folder, but some well-timed gardening should catch it before it grows to dozens. As with most things, a stitch in time saves nine, plus it makes you look like you knew what you were doing the whole time.

It's easy to focus your attention on the source code only, but the system's runtime deserves similar attention, as does how it is allocated to hardware or containers. If you want to think about it clearly, structure it as a story at many levels.