Software Architecture for Developers

Volume I

Technical leadership and the balance with agility

Simon Brown

Technical leadership and the balance with agility

Software Architecture for Developers - Volume 1

Simon Brown

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

This version was published on 2017-08-04



This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2012 - 2017 Simon Brown

Tweet This Book!

Please help Simon Brown by spreading the word about this book on Twitter!

The suggested hashtag for this book is #sa4d.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#sa4d

For Kirstie, Matthew and Oliver

Contents

About the book . . . . . **i**
Why did I write the book? i
A new approach to software development? ii

About the author . . . . . **iii**

Acknowledgements . . . . . **iv**

I Software architecture 1

1. What is "software architecture"? **2**
Architecture as a noun 3
Architecture as a verb 3
Types of architecture 3
Application architecture 5
System architecture 5
Software architecture 6
Enterprise architecture - strategy rather than code 7
Architecture vs design 7
Architectural significance 9
Is software architecture important? 9
The benefits of software architecture 10
Does every software project need software architecture? 11

About the book

This book is a practical, pragmatic and lightweight guide to software architecture, specifically aimed at developers, and focussed around the software architecture role and process.

Why did I write the book?

Like many people, I started my career as a software developer, taking instruction from my seniors and working with teams to deliver software systems. Over time, I started designing smaller pieces of software systems and eventually evolved into a position where I was performing what I now consider to be the software architecture role.

I've worked for IT consulting organisations for the majority of my career, and this means that most of the projects that I've been involved with have resulted in software systems being built either *for* or *with* our customers. In order to scale an IT consulting organisation, you need more people and more teams. And to create more teams, you need more software architects. And this leads me to why I wrote this book:

1. **Software architecture needs to be more accessible**: Despite having some fantastic mentors, I didn't find it easy to understand what was expected of me when I was moving into my first software architecture roles. Sure, there are lots of software architecture books out there, but they seem to be written from a different perspective. I found most of them very research oriented or academic in nature, yet I was a software developer looking for real-world advice. I wanted to write the type of book that I would have found useful at that stage in my career - a book about software architecture aimed at software developers.
2. **All software projects need software architecture**: I like agile approaches, I really do, but the lack of explicit regard for software architecture in many of the approaches doesn't sit well with me. Agile approaches don't say that you shouldn't do any up front design, but they often don't explicitly talk about it either. I've found that this causes people to jump to the wrong conclusion and I've seen the consequences that a lack of any up front thinking can have. I also fully appreciate that big design up front isn't the answer either. I've always felt that there's a happy medium to be found where *some* up front thinking is done, particularly when working with a team that has a mix of

experiences and backgrounds. I favour a lightweight approach to software architecture that allows me to put *some* building blocks in place as early as possible, to stack the odds of success in my favour.

3. **Lightweight software architecture practices**: I've learnt and evolved a number of practices over the years, which I've always felt have helped me to perform the software architecture role. These relate to the software design process and identifying technical risks through to communicating and documenting software architecture. I've always assumed that these practices are just common sense, but I've discovered that this isn't the case. I've taught these practices to thousands of people over the past few years and I've seen the difference they can make. A book helps me to spread these ideas further, with the hope that other people will find them useful too.

A new approach to software development?

This book *isn't* about creating a new approach to software development, but it does seek to find a happy mid-point between the excessive up front thinking typical of traditional methods and the lack of any architecture thinking that often happens in software teams who are new to agile approaches. There **is** room for up front design and evolutionary architecture to coexist.

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 web-based tool to create software architecture diagrams based upon the C4 model.

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 twenty countries around the world.

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

Acknowledgements

Although this book has my name on the front, writing a book is never a solo activity. It's really the product of a culmination of ideas that have evolved and discussions that have taken place over a number of years. For this reason, there are a number of people to thank.

First up are Kevin Seal, Robert Annett and Sam Dalton for lots of stuff; ranging from blog posts on Coding the Architecture and joint conference talks through to the software architecture user group that we used to run at Skills Matter (London) and for the many tech chats over a beer. Kevin also helped put together the first version of the training course that, I think, we initially ran at the QCon Conference in London, which then morphed into a 2-day training course that we have today. His classic "sound-bite" icon in the slide deck still gets people talking today. :-)

I've had discussions about software architecture with many great friends and colleagues over the years, both at the consulting companies where I've worked (Synamic, Concise, Evolution and Detica) and the customers that we've built software for. There are too many people to name, but you know who you are.

I'd also like to thank everybody who has attended one of my talks or workshops over the past few years, as those discussions also helped shape what you find in the book. You've all helped; from evolving ideas to simply helping me to explain them better.

Thanks also to Junilu Lacar and Pablo Guardiola for providing feedback, spotting typos, etc.

And I should finally thank my family for allowing me to do all of this, especially when a hectic travel schedule sometimes sees me jumping from one international consulting gig, workshop or conference to the next. Thank you.

I Software architecture

In this part of the book we'll look at what software architecture is about, the difference between architecture and design and why thinking about software architecture is important.

1. What is “software architecture”?

Let’s start with the basics. The word “architecture” means many different things to many different people, and there are many different definitions floating around the Internet. I’ve asked thousands of software developers what “architecture” means to them and, in no particular order, a summary of their responses is as follows.

- Modules, connections, dependencies and interfaces
- The big picture
- The things that are expensive to change
- The things that are difficult to change
- Design with the bigger picture in mind
- Interfaces rather than implementation
- Aesthetics (e.g. as an art form, clean code)
- A conceptual model
- Satisfying non-functional requirements/quality attributes
- Everything has an “architecture”
- Ability to communicate (abstractions, language, vocabulary)
- A plan
- A degree of rigidity and solidity
- A blueprint
- Systems, subsystems, interactions and interfaces
- Governance
- The outcome of strategic decisions
- Necessary constraints
- Structure (components and interactions)
- Technical direction
- Strategy and vision
- Building blocks
- The process to achieve a goal
- Standards and guidelines
- The system as a whole

- Tools and methods
- A path from requirements to the end-product
- Guiding principles
- Technical leadership
- The relationship between the elements that make up the product
- Awareness of environmental constraints and restrictions
- Foundations
- An abstract view
- The decomposition of the problem into smaller implementable elements
- The skeleton/backbone of the product

No wonder it’s hard to find a single definition! Thankfully there are two common themes here: architecture as a noun and architecture as a verb.

Architecture as a noun

As a noun, architecture can be summarised as being about **structure**. It’s about the decomposition of a product into a collection of smaller building blocks and interactions/relationships between them. This needs to take into account the whole of the product, including the foundations and the infrastructure services that deal with cross-cutting concerns such as security, configuration, error handling, etc.

Architecture as a verb

As a verb, architecture (i.e. the process of creating architecture or “architecting”) is about understanding what you need to build, creating a vision for building it and making the appropriate design decisions. All of this needs to be based upon requirements because **requirements drive architecture**. Crucially, it’s also about communicating that vision and introducing technical leadership so that everybody involved with the construction of the product understands the vision and is able to contribute in a positive way to its success.

Types of architecture

What we have so far is a very generic definition of the word “architecture” but, of course, there are many different types of architecture and architects within the IT industry alone. Here, in no particular order, is a list of those that people most commonly identify when asked.

- Infrastructure
- Security
- Technical
- Solution
- Network
- Data
- Hardware
- Enterprise
- Application
- System
- Integration
- IT
- Database
- Information
- Process
- Business
- Software

The unfortunate thing about this list is that some of the terms are easier to define than others, particularly those that refer to or depend upon each other for their definition. For example, what does “solution architecture” actually mean? For some organisations “solution architect” is simply a synonym for “software architect”, whereas others have a specific role that focusses on designing an overall “solution” to a problem, but stopping before the level at which implementation details are discussed. Similarly, “technical architecture” is vague enough to refer to software, hardware or a combination of the two. Interestingly, “software architecture” typically appears near the bottom of the list when I ask people to list the types of IT architecture they’ve come across. Perhaps this reflects the confusion that surrounds the term.

What do all of these terms have in common then? Well, aside from suffixing each of the terms with “architecture” or “architect”, all of these types of architecture have **structure** and **vision** in common.

Let’s take “infrastructure architecture” as an example. Imagine that you need to create a network between two offices at different ends of the country. One option is to find the largest reel of network cable that you can and start heading from one office to the other in a straight line. Assuming that you had enough cable, this could potentially work, but in reality there are a number of environmental constraints (real-world obstacles such as rivers,

lakes, roads, cities, etc) and service-level agreements (performance, bandwidth, security, etc) that you need to consider in order to actually deliver something that satisfies the original goal. This is where the process of architecting and having a vision to achieve the goal is important. One single long piece of cable is *an* approach, but it's not a very good one because of real-world constraints. For this reason, networks are typically much more complex and require a collection of components collaborating together in order to satisfy the goal. From an infrastructure perspective then, we can talk about structure in terms of the common components that you'd expect to see within this domain; things like routers, firewalls, packet shapers, switches, etc.

Regardless of whether you're building a software system, a network or a database; a successful solution requires you to understand the problem and create a vision that can be communicated to everybody involved with the construction of the end-product. In order to move towards a definition of "software architecture", let's look at a couple of types of architecture in the IT domain that are relatively well defined.

Application architecture

Application architecture is what we as software developers are probably most familiar with. In this context, I'm going to define an application as being something written in a single technology; such as a single-page JavaScript application, a Java web application, a .NET desktop application, etc. Application architecture is about looking inside the application to understand how it's designed and built. This includes how the application has been decomposed into components as well as understanding the patterns, frameworks and libraries in use. In essence, it's predominantly about software and the organisation of the code.

System architecture

I like to think of system architecture as one step up in scale from application architecture. If you look at most software systems, they're actually composed of multiple applications using a number of different technologies. As an example, you might have a software system comprised of a mobile app communicating via JSON/HTTPS to a Java web application running on a server, which itself consumes data from a MySQL database. Since each of these three applications is built using a different technology, each of them will have their own internal application architecture.

However, for the overall software system to function as a whole, thought needs to be put into bringing all of those separate applications together. In other words, you also have

to consider the overall structure of the end-to-end software system at a high-level. If I make a request from the mobile app, how is that request processed by the entire software system. Additionally, most software systems don’t live in isolation, so system architecture also includes the concerns around interoperability and integration with other systems within the environment.

Where application architecture tends to focus primarily on software, system architecture is about understand both **software and hardware**. This includes things like programming languages, libraries and frameworks through to servers and infrastructure. The reason that most definitions of system architecture include references to software *and* hardware is that you can’t have a successful software system without hardware, even if that hardware is virtualised somewhere out there on the cloud.

Software architecture

Unlike application and system architecture, which are relatively well understood, the term “software architecture” has many different meanings to many different people. Rather than getting tied up in the complexities and nuances of the many definitions of software architecture, I like to keep the definition as simple as possible. For me, software architecture is simply the combination of application and system architecture. In other words, it’s anything and everything related to the design of a software system; from the structure of the code and understanding how the whole software system works at a high level, through to how that software system is deployed onto infrastructure. But that’s not the whole story.

When we’re thinking about software development as software developers, most of our focus is placed on the code. Here, we’re thinking about things like object oriented principles, classes, interfaces, inversion of control, refactoring, automated unit testing, clean code and the countless other technical practices that help us build better software. If your team consists of people who are *only* thinking about this, then who is thinking about the other things such as:

- Cross-cutting concerns; including logging, exception handling, etc.
- Security; including authentication, authorisation and confidentiality of sensitive data.
- Performance, scalability, availability and other quality attributes.
- Audit and other regulatory requirements.
- Real-world constraints of the environment.
- Interoperability/integration with other software systems.
- Operational, support and maintenance requirements.

- Consistency of structure and approach to solving problems/implementing features across the codebase.
- Evaluating that the foundations you’re building will allow you to deliver what you set out to deliver.

Sometimes you need to step back, away from the code and away from your development tools. This doesn’t mean that the lower-level detail isn’t important because working software is ultimately about delivering working code. No, the detail is equally as important, but software architecture is about having a holistic view across your software system to ensure that your code is working toward your overall vision rather than against it.

Enterprise architecture - strategy rather than code

Although this book isn’t about enterprise architecture, it’s worth including a short definition so that we understand how it differs from software architecture.

Enterprise architecture generally refers to the sort of work that happens centrally and across an organisation. It looks at how to organise and utilise people, process and technology to make an organisation work effectively and efficiently. In other words, it’s about how an enterprise is broken up into groups/departments, how business processes are layered on top and how technology underpins everything. This is in very stark contrast to software architecture because it doesn’t necessarily look at technology in any detail. Instead, enterprise architecture might look at how best to use technology across the organisation without actually getting into detail about how that technology works.

While some developers and software architects do see enterprise architecture as the next logical step up the career ladder, most probably don’t. The mindset required to undertake enterprise architecture is very different to software architecture, taking a very different view of technology and its use across an organisation. Enterprise architecture requires a higher level of abstraction. It’s about breadth rather than depth and strategy rather than code.

Architecture vs design

One of the words that people use to describe architecture is “design”, and this raises the question of whether we should use these words interchangeably. Grady Booch has a well cited definition of the difference between architecture and design that really helps to answer this question. In On Design, he says that:

> As a noun, design is the named (although sometimes unnameable) structure or behavior of a system whose presence resolves or contributes to the resolution of a force or forces on that system. A design thus represents one point in a potential decision space.

If you think about any problem that you've needed to solve, there are probably a hundred and one ways in which you could have solved it. Take your current software project for example. There are probably a number of different technologies, deployment platforms and design approaches that are also viable options for achieving the same goal. In designing your software system though, your team chose just one of the many points in the potential decision space.

Grady then goes on to say that:

> All architecture is design but not all design is architecture.

This makes sense, because creating a solution is essentially a design exercise. However, for some reason, there's a distinction being made about not all design being "architecture", which he clarifies with the following statement:

> Architecture represents the significant design decisions that shape a system, where significance is measured by cost of change.

Essentially, Grady is saying that the significant decisions are "architecture" and that everything else is "design". In the real world, the distinction between architecture and design isn't as clear-cut, but this definition does provide us with a basis to think about what might be significant (i.e. "architectural") in our own software systems. For example, this could include:

- The overall shape of the software system (e.g. client-server, web-based, native mobile, distributed, microservices, asynchronous vs synchronous, etc).
- The structure of the code inside the various parts of the software system (e.g. whether the code is structured as components, layers, features, etc).
- The choice of technologies (i.e. programming language, deployment platform, etc)
- The choice of frameworks (e.g. web MVC framework, persistence/ORM framework, etc)
- The choice of design approach/patterns (e.g. the approach to performance, scalability, availability, etc)

The architectural decisions are those that you can't reverse without some degree of effort. Or, put simply, they're the things that you'd find hard to refactor in an afternoon.

Architectural significance

It's often worth taking a step back and considering what's architecturally significant with your own software system. Imagine that you're building a simple server-side web application that delivers information to users that is stored in a relational database. For the sake of this discussion, let's say there are no complex requirements related to security, performance or scalability, and that the database is simply being used for data storage.

When building the web application, many teams will choose to use some sort of database abstraction layer, like an object-relational mapping framework; such as Hibernate, JPA, Entity Framework, etc. One common reason to use a database abstraction layer is to make accessing the database easier. Another common reason to use a database abstraction layer is to decouple application code from the choice of database. The use of an abstraction layer is a classic technique for decoupling distinct parts of a software system; promoting looser coupling, higher cohesion and a better separation of concerns. If you're simply using the database for data storage, the use of the database abstraction layer allows you to, in theory, change your database without changing any code. Since the database can be changed so easily, many teams would consider the choice of database to no longer be a significant decision.

However, while the database may no longer be considered a significant decision, the choice to decouple through the introduction of an additional layer should be. If you're wondering why, have a think about how long it would take you to swap out your current database abstraction layer or web MVC framework and replace it with another. Of course, you could add another layer over the top of your chosen database abstraction layer to further isolate your business logic and provide the ability to easily swap out your database abstraction layer but, again, you've made another significant decision. You've introduced additional layering, complexity and cost.

Although you can't necessarily make "significant decisions" disappear entirely, you can use a number of different tactics such as architectural layering to change what those significant decisions are. Part of the process of architecting a software system is about understanding what is significant and why.

Is software architecture important?

Software architecture then, is it important? The agile and software craftsmanship movements are helping to push up the quality of the software systems that we build, which is excellent. Together they are helping us to write better software that better meets the needs of the

business while carefully managing time and budgetary constraints. But there’s still more we can do because even a small amount of software architecture can help prevent many of the problems that projects face.

Successful software projects aren’t just about good code though and you do need to step away from the code for a few moments to see the bigger picture. Ask yourself the following questions:

- Does your software system have a well defined structure?
- Is everybody on the team implementing features in a consistent way?
- Is there a consistent level of quality across the codebase?
- Is there a shared vision for how the software will be built across the team?
- Does everybody on the team have the necessary amount of technical guidance?
- Is there an appropriate amount of technical leadership?

It is possible to successfully deliver a software project by answering “no” to some of these questions, but it does require a very good team and a lot of luck. If nobody thinks about software architecture, the end result is something that typically looks like a big ball of mud. Sure, it has a structure but it’s not one that you’d want to work with! Other side effects could include the software system being too slow, insecure, fragile, unstable, hard to deploy, hard to maintain, hard to change, hard to extend, etc. I hope you’ve never seen or worked on software projects like this, but they do exist.

The benefits of software architecture

What benefits can thinking about software architecture provide? In summary:

- A clear vision and roadmap for the team to follow, regardless of whether that vision is owned by a single person or collectively by the whole team.
- Technical leadership and better coordination.
- A stimulus to talk to people in order to answer questions relating to significant decisions, quality attributes, constraints and other cross-cutting concerns.
- A framework for identifying and mitigating risk.
- Consistency of approach and standards, leading to a well structured codebase.
- A set of firm foundations for the product being built.
- A structure with which to communicate the solution at different levels of abstraction to different audiences.

Does every software project need software architecture?

Rather than use the typical consulting answer of “it depends”, I’m instead going to say that the answer is undoubtedly “yes”, with the caveat that every software project should look at a number of factors in order to assess how much software architecture thinking is necessary. These include the size of the project/product, the complexity of the project/product, the size of the team and the experience of the team. The answer to how much is “just enough” will be explored throughout the rest of this book.