Software architecture for developers

Simon Brown

Software Architecture for Developers

Technical leadership and the balance with agility

Simon Brown

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

This version was published on 2022-05-28

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.

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:

#sa4d

For Kirstie, Matthew and Oliver

Contents

1. 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.

2. 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 model and I built Structurizr, which is a collection of tooling to help software teams visualise, document and explore their 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.

3. 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.

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 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.

4. 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 published on 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 - structure

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[1] and the interactions/relationships between these building blocks. 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. To quote Bass, Clements, and Kazman:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relations among them.

Architecture as a verb - vision

As a verb, architecture (i.e. the process of creating architecture, or "architecting") is about translating the *architectural drivers* (functional requirements, quality attributes, constraints, and principles) into a technical solution, thereby creating a technical roadmap or **vision**. Crucially, it's also about communicating that vision to a number of stakeholders both inside

[1] We don't tend to use the term "building blocks" when describing the structure of a software system. Instead we use terms such as "component", "module", "service", "microservice", "layer", etc. Unfortunately some of these terms are ambiguous. As you'll see in volume 2 of "Software Architecture for Developers", this causes a number of problems when diagramming and documenting software architecture.

and outside of the immediate software development team, so that everybody has a consistent view of what is being (or has been) built. The process of architecting is additionally about introducing *technical leadership* so that everybody involved with the construction of the software system is able to contribute in a positive and consistent way.

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 people who call themselves “architects” within the IT industry. Here, in no particular order, is a list of types of architecture 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 other organisations have a specific role that focusses on designing an overall “solution” to a problem, 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[2].

What do all of these terms have in common? Well, aside from being able to suffix 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 located at different ends of the country. One option is to find the largest reel of network cable that you can, plug it in at one office, and start heading to the other in a straight line. Assuming that you had enough cable, this could potentially work. In reality though, 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 is important. One single long piece of cable is certainly *one* approach, but it's not a very good one because of the real-world constraints. For this reason, networks are typically much more complicated, requiring a collection of smaller building blocks that collaborate together in order to satisfy the goal. From an infrastructure perspective then, we can talk about structure in terms of the common building blocks that you would 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 the most familiar with. In this context, I'm going to define an application as being a single deployable unit, written in a single technology; such as a single-page JavaScript/Angular application, an iOS or Android mobile app, a Java server-side Spring MVC 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 building

[2]My own job title for a number of years was "Technical Architect". With hindsight, this was not very descriptive or accurate, since my day-to-day focus was primarily software architecture rather than anything else.

blocks (e.g. components, layers, packages, namespaces, etc) as well as understanding the patterns, frameworks and libraries in use. In essence, it's predominantly about code, and the organisation of that 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 deployable units (e.g. applications or datastores), each of which might be built using different technologies.

As an example, you might have a software system comprising of a client-side iOS mobile app communicating via JSON/HTTPS to a Java server-side Spring MVC web application, which itself consumes data from a MySQL database. Since each of these three deployable units (the mobile app, the web app and the database) is built using a different technology, each of them will have their own internal application architecture.

However, for the software system to function as a whole, thought needs to be put into bringing all of those separate deployable units together. In other words, you also need to consider the overall structure of the software system at a high-level, and the integration of the various parts. For example, if I make a request from the mobile app, how is that request processed by the entire software system? Additionally, software systems typically 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 (e.g. programming languages, frameworks, libraries, etc), system architecture is about understanding both **software and hardware**. Admittedly, much of the hardware that we deal with nowadays is virtualised or completely hidden[3], but it's an important concern nonetheless. The reason that most definitions of system architecture include references to software *and* hardware (whether it's the target deployment platform or supporting infrastructure) is that you can't have a successful software system without it. After all, software needs to be deployed somewhere in order to run! Additionally, infrastructure typically provides constraints that must be taken into account when designing software. Examples of this range from the processing power and memory of embedded devices limiting what your software can do, through to cloud providers limiting how your applications are deployed in order to facilite better high availability and scaling.

[3]Platform as a Service (PaaS) and Function as a Service (FaaS) environments typically hide the underlying hardware completely, and instead provide higher-level abstractions on which to build and deploy software systems. Understanding the constraints of these environments is still important if you want to build software that "works".

Towards a definition of "software architecture"

Unlike application architecture and system architecture, which are relatively well understood, the term "software architecture" isn't. Rather than getting tied up in the complexities and nuances of the many definitions of software architecture that exist on the Internet, I like to keep the definition as simple as possible. For me, software architecture is simply the combination of application architecture and system architecture, again in relation to structure and vision. 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, functional programming principles, classes, interfaces, modules, inversion of control, refactoring, automated 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.
- Structural consistency and integrity.
- Consistency of approaches to solving problems and implementing features across the codebase.
- Evaluating that the foundations you're building will allow you to deliver what you set out to deliver.
- Keeping an eye on the future, and changes in the environment.

In order to think about these things, you need to step back, away from the code and your development tools. Working software is ultimately about delivering working code, so the detail *is* crucially 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 of those groups/departments, and how technology is used to support the goals of the enterprise. 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, and a different focus. It’s about breadth rather than depth, and strategy rather than code.

Central architecture groups

As a quick aside, if you’ve ever worked in a large organisation, the definition I’ve just given for enterprise architecture might be different to what you were expecting. Often, large organisations will have a “central architecture group” that might be referred to as “the enterprise architects”. Such groups typically manage lists of the approved technologies that you can (or must!) use when building software, and will often have some involvement in reviewing/guiding the output from software development teams to ensure consistency across the organisation. Although a useful role, this isn’t really what I deem to be “enterprise architecture”.

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 the words “architecture” and “design” 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, Grady 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/product 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 (options) in the potential decision space. That's the essence of design. It's about narrowing down the solution space to find an option that works given the context in which you are working.

Grady then goes on to say that:

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

This makes sense, because creating a solution, and "architecting", is essentially a design exercise. It's about narrowing down options, and making decisions. However, for some reason, there's a distinction being made about not all design being "architecture", which Grady 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 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, ports and adapters, 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

Although this sounds relatively straightforward, we, as architects, have a degree of influence over those architecturally significant decisions. Imagine you're building a simple server-side web application to deliver information to users, and that information 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. Let's also ignore non-relational (e.g. NoSQL) databases.

When building the web application, many teams will choose to use some sort of abstraction layer to communicate with the database, 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 business/domain-specific 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 only using the database for data storage (i.e. the database only contains data rather than code wrapped up functions and stored procedures), the use of the database abstraction layer allows you to, in theory, change your database via configuration, without changing any code. Since the database can be changed so easily, many teams would therefore 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 we can't necessarily make "significant decisions" disappear, we can use a number of different tactics (such as architectural layering, in the previous example) to change what those significant decisions are. There's also no explicit line between the decisions that should be deemed as significant, and those that shouldn't. Having said that, the significant decisions are usually related to key technology choices (e.g. programming languages and frameworks) and the overall structure (monolithic deployment unit vs microservices). Aspects such as "tabs vs whitespaces", or "curly braces on same line vs the next line", are definitely not architecturally significant! Everything else will fall in between somewhere between these two extremes. Part of the process of architecting a software system is about understanding what is significant and why, given the context you're working in.

Is software architecture important?

Now that we understand what software architecture is, we should wrap up this chapter by looking at the importance of software architecture. The past decade or so has seen a huge shift in the way that we build software, thanks to movements such as agile, lean, software craftsmanship, continuous delivery, DevOps, the cloud and more. Together these new approaches help us to build better software that better meets the needs of our stakeholders, 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.

As I've already mentioned, successful software projects aren't just about focussing on good code. 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?
- Do team members share the same vision for how the software will be built?
- 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. Although most software projects and products start with the best of intentions, it's easy for them to veer off track without an appropriate amount of technical leadership; both at the code level, and above it. If nobody thinks about software architecture, you often end up with codebases that are too

slow, insecure, fragile, unstable, hard to deploy, hard to maintain, hard to change, hard to extend, etc. I've personally seen (and worked on!) codebases that have exhibited the following types of problems:

- Components in a monolithic application were configured in different ways, depending upon the developer who built them. These different approaches were not discussed or documented, so deploying the resulting software required many "trial and error" loops.
- Code in a monolithic application could use one of three different data access layers, each built using a different framework, to access data from the *same* database.
- The path of a HTTP request through a server-side web application varied depending upon the developer who was implementing the feature. In essence, every developer had a different idea of what the layered architecture should be, and this was reflected in the code.
- A software system didn't perform and scale as hoped when presented with the real-world data set. In this case, one of the key technology choices wasn't able to meet the quality attributes, and this was unfortunately not evaluated before making such a significant decision.

Additionally, without technical leadership, many codebases also end up looking like the stereotypical "big ball of mud" or "spaghetti code". Sure, it has a structure, but not one that you'd want to work with! These seemingly chaotic software projects do exist in the real-world, and most of us will have one or more horror stories about the time we spent working on them. If you've never worked on such a project, you're probably in the lucky minority!

The benefits of software architecture

Thankfully, most of these problems are relatively easy to solve with the application of some good technical leadership, resulting in a team that therefore understands and thinks about software architecture. In summary, this can provide:

- 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 (inside and outside of the team) in order to ask 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”. The caveat here is that every software team should look at a number of factors in order to assess *how much* software architecture thinking, a degree of which manifests itself as up front design, 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.

Historically we’ve seen a tendency towards too much up front design, with teams trying to answer all of the questions and solve all of the problems before writing a single line of code. More recently, I’ve witnessed a trend towards the other extreme, and too little up front design. As Dave Thomas once said:

> Big design up front is dumb. Doing no design up front is even dumber.

As with many things in life, there is a sweet spot here awaiting discovery. The answer to how much is “enough” up front design and technical leadership will be explored throughout the rest of this book.

II Architects

This part of the book focusses on the software architecture role; including what it is, what sort of skills you need, and why coding, coaching and collaboration are important.

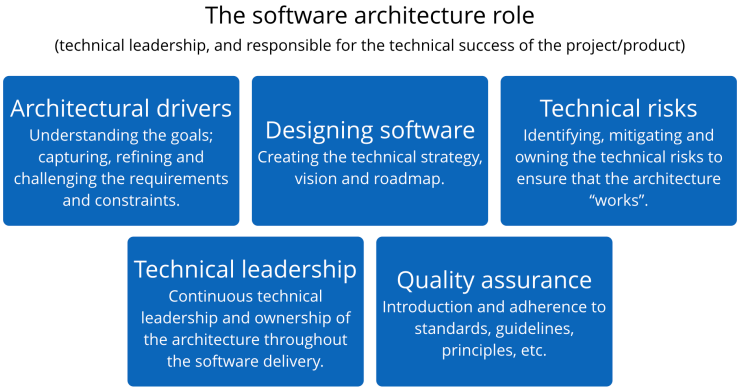
5. The software architecture role

The line between being a software developer and being a software architect is a tricky one. Some people will tell you that it doesn't exist, with architecture just being an extension of the design process undertaken by developers. Others will tell you that it's a massive gaping chasm, which can only be crossed by lofty developers who believe you must always abstract your abstractions and not get bogged down by those pesky real-world implementation details. As always, there's a pragmatic balance somewhere in between, but it does raise the interesting question of how you move from one side to the other, and therefore how you progress in your career as a software developer.

As we learnt in chapter 1, software architecture is about a number of different things; ranging from the organisation of code through to having a holistic view of the software system being built from a number of different perspectives, and making the appropriate significant design decisions to ensure success. This definition is necessarily broad, but it doesn't really describe what software architects do, and how a software developer moves into a software architecture role. It also doesn't help in identifying who will make a good software architect, and how you go about finding them if you're hiring.

Becoming a software architect isn't something that happens overnight or with a promotion. It's a **role**, not a rank. It's the result of an evolutionary process where you'll gradually gain the experience and confidence that you need to undertake the role. While the term "software developer" is relatively well-understood, "software architect" isn't. A simple way to think about the software architecture role is that it's about the "big picture" and, sometimes, this means stepping away from the code.

Here are the things I consider to make up the software architecture role, with the summary being that the role is about providing **technical leadership** and **being responsible for the technical success of the project/product**. Notice that I said "role" here; it's something that can be performed by a single person or shared amongst the team, but we'll talk about that later.



1. Architectural drivers

The first part of the role is about understanding, and managing, the architectural drivers - the functional requirements, quality attributes, constraints and principles, as we saw in the previous chapter. These driving forces have a huge influence on the resulting software architecture, so explicitly including them as a part of the software architecture role helps to ensure that they are proactively considered, and taken into account.

2. Designing software

It should come as no surprise that the process of designing software is a key part of the software architecture role. This is about understanding how you're going to solve the problems posed by the architectural drivers, creating the overall structure of the software system, and creating a vision for the delivery. Despite how agile you to strive to be, as we'll see later, you probably do need **some** time to explicitly think about how your architecture is going to solve the problems set out by the various stakeholders.

A key part of designing software is technology selection, which is typically a fun exercise, but it does have its fair set of challenges. For example, some organisations have a list of approved technologies that you are "encouraged" to choose from, while others have rules in place that don't allow open source technology with a specific licence to be used. Then you have all of the other factors such as cost, licensing, vendor relationships, technology strategy, compatibility, interoperability, support, deployment, upgrade policies, end-user environments, and so on. The sum of these factors can often make a simple decision of choosing something like a

framework into a complete nightmare. Somebody needs to take ownership of the technology selection and evaluation process, and this falls squarely within the remit of the software architecture role.

3. Technical risks

What we've looked at so far will help you focus on building a good solution, but it doesn't guarantee success. Simply throwing together the best designs and technologies doesn't necessary mean that the overall architecture will be successful. There's also the question of whether the technology choices you've made will actually work. Many teams get burnt because they believe the hype on vendor websites, or described by sales executives while spending a few hours together on the golf course. I always find it surprising how few people seem to ask whether a technology actually works the way it is supposed to, evaluating the technology where needed to prove that this is the case. Technology selection is all about managing risk; reducing risk where there is high complexity or uncertainty, and introducing risk where there are benefits to be gained. All technology decisions need to be made by taking many factors into account, and all technology decisions need to be reviewed and evaluated.

The key question that you need to ask yourself is whether your architecture "works". For me, an architecture works if it satisfies the architectural drivers, provides the necessary foundations for the rest of the code, and works as the platform for solving the underlying business problem. Software is complicated and abstract, which means that it's hard to visualise the runtime characteristics of a piece of software from a collection of diagrams, or even the code itself. Furthermore, I don't always trust myself to get it right first time.

Throughout the software development life cycle, we undertake a number of different types of testing in order to give us confidence that the system we are building will work when delivered. So why don't we do the same for our architecture? If we can test our architecture, we can prove that it works. And if we can do this as early as possible, we can reduce the overall risk of project/product failure. Like good chefs, architects should taste what they are producing. In a nutshell, the software architecture role is about proactively identifying, mitigating, and owning the high priority technical risks so that your project doesn't get cancelled, and you don't get fired.

4. Technical leadership

Whatever software architecture you create needs to be taken care of. Somebody needs to look after it, evolving it throughout the delivery in the face of changing requirements and

feedback from the team. If a software architect has created an architecture, they should own and evolve that architecture throughout the rest of the delivery too. This is about *continuous technical leadership* rather than simply being involved at the start of the life cycle and hoping for the best.

5. Quality assurance

Even with the best architecture in the world, poor delivery can cause an otherwise successful software project to fail. Quality assurance should be a part of the software architecture role, but it's more than just doing code reviews. You need a baseline to assure against, which could mean the introduction of standards and working practices such as coding standards, design principles and tools. Quality assurance also includes ensuring that the architecture is being implemented consistently across the team. Whether you call this architectural *compliance*, *conformance* or whatever is up to you, but any technical vision created by the people performing the software architecture role needs to be understood and followed by the rest of the team.

It's safe to say that most projects don't do enough quality assurance, and therefore you need to figure out what's important to make sure that it's sufficiently assured. A good starting point is to identify anything that is architecturally significant, business critical, complicated, or highly visible. You need to be pragmatic though, and realise that you can't necessarily assure everything given the typical budgetary and time constraints we are subjected to.

Software architecture is a role, not a rank

The software architecture role is essentially about introducing technical leadership into a software team, and it's worth repeating that what I'm talking about here is a **role** rather than a rank. Large organisations often use the job title of "Architect" as a reward for long service or because somebody wants a salary increase. And that's fine if the person on the receiving end of the title is capable of undertaking the role, but this isn't always the case. If you've ever subscribed to software architecture discussion groups on LinkedIn or Stack Overflow, you might have seen questions like this.

> Hi, I've just been promoted to be a software architect but I'm not sure what I should be doing. Help! Which books should I read?

Although I can't stop organisations promoting people to roles above their capability (often referred to as the Peter principle), I *can* describe what my view of the software architecture role is, and help people achieve it.

Create your own definition of the role

Most of the roles that we associate with software development teams are relatively well understood - developers, testers, ScrumMasters, Product Owners, business analysts, project managers, etc. The software architecture role? Not so much. In my experience, although many software teams *do* understand the need for the software architecture role, they often don't have a well-defined understanding (e.g. a "terms of reference") for it. Without this, you run the risk of the role not being performed in part or in whole. I regularly ask software teams whether they have a definition of the software architecture role, and the usual answer is along the lines of "no" or "yes, but we don't use it". Often people working for the *same team* will answer the question differently.

Although the need for thinking about software architecture is usually acknowledged, the responsibilities of the software architecture role often aren't clear. In my experience, this can lead to a situation where there is nobody undertaking the role, or where somebody is assigned the role but doesn't really understand how they should undertake it. If the role isn't understood, it's not going to get done.

Regardless of what you call it (e.g. Architect, Tech Lead, Principal Designer, etc), my advice is simple. If you don't have something that you can point at and say, "this is what we expect of our software architects", take some time to create something. Start by agreeing what is expected of the software architecture role on your team, and then move to standardise it across your organisation if you see benefit in doing so.