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 2022-01-09

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.

5. Architectural drivers

Now that we understand what software architecture is about, it's worth looking at what you need to think about when architecting a software system. Regardless of the process that you follow (traditional and plan-driven vs lightweight and adaptive), there's a set of common things that really drive, influence and shape the resulting software architecture.

1. Functional requirements

In order to design software, you need to know something about the goals that it needs to satisfy. To some extent, requirements drive architecture. If you ever find yourself at a whiteboard, designing software from a technical perspective, without really understanding who is going to be using your software and what they might want to do, stop. If this sounds obvious, it's because it is!

Having said that, I *have* seen teams designing software (and even building it) without a high-level understanding of the features that the software should provide to the end-users. Some have excused this behaviour as being "agile", but I call it being foolish. Even a rough, short list of features or user stories (e.g. a Scrum product backlog) and an understanding of the important user types/roles/personas is essential. This applies to product and startup compainies too, who may start with a minimum viable product and then experiment with features by iterating on them quickly. In such cases, you still need at least a high-level view of what the requirements are, even if some of the details are yet to be discovered.

2. Quality Attributes (non-functional requirements)

Non-functional requirements are often thought of as the "-ilities", and are primarily about quality of service and cross-cutting concerns. Alternative, arguably better yet less commonly used names for non-functional requirements include "system characteristics" or "quality attributes". They are mostly technical in nature and can have a huge influence over the resulting architecture, particularly if you're building systems with extremely high performance or high security in mind, or you have desires to operate at "Google scale".

The technical solutions to supporting many quality attributes are usually cross-cutting, and therefore need to be baked into the foundations of the system you're building. Retrofitting high performance, scalability, security, availability, etc into an existing codebase is usually incredibly difficult and time-consuming, because the designs to support these qualities is often reflected in the overall shape of the software system (the high-level structures) itself.

A non-exhaustive list of the common quality attributes is as follows.

Performance

Performance is about how fast something is, usually in terms of response time or latency.

- **Response time**: the time it takes between a request being sent and a response being received, such as a user clicking a hyperlink on a web page or a button on a desktop application.
- **Latency**: the time it takes for a message or event to move through your system, from point A to point B.

Even if you're not building "high performance" software systems, performance is applicable to pretty much every software system that you'll ever build, regardless of whether they are web applications, desktop applications, service-oriented architectures, messaging systems, etc. If you've ever been told that your software is "too slow" by your users, you'll appreciate why some notion of performance is important.

Scalability

Scalability is basically about the ability for your software to deal with more users, requests, data, messages, etc. Scalability is inherently about concurrency, and therefore dealing with more of something in the same period of time (e.g. requests per second).

Availability

Availability is about the degree to which your software is operational and, for example, available to service requests. You'll usually see availability measured or referred to in terms of "nines", such as 99.99% ("four nines") or 99.999% ("five nines"). These numbers refer to the uptime in terms of a percentage. The other way to think about this is the amount of downtime that can be tolerated. An uptime of 99.9% ("three nines") provides you with a downtime window of just over 1 minute per day for scheduled maintenance, upgrades and unexpected failure.

Security

Security covers everything from authentication and authorisation, through to the confidentiality of data in transit and storage. As with performance, there's a high probability that security is important to you at some level. Security should be considered for even the most basic of web applications that are deployed onto the Internet. The Open Web Application Security Project (OWASP) is a great starting point for learning about security.

Privacy

Related to security is privacy, and it's worth being aware of new regulations such as GDPR if you're handling data from users who reside in the European Union.

Disaster Recovery

What would happen if you lost a hard disk, a server or a data centre that your software was running on? This is what disaster recovery is all about. If your software system is mission critical, you'll often hear people talking about business continuity processes too, which state what should happen in the event of a disaster in order to retain continued business operation.

Accessibility

Accessibility usually refers to things like the W3C accessibility standards, which talk about how your software is accessible to people with disabilities such as visual impairments.

Monitoring

Some organisations have specific requirements related to how software systems should be monitored to ensure that they are running and able to service requests. This could include integrating your software with platform specific monitoring capabilities (e.g. JMX on the Java platform, or third-party systems such as New Relic) or sending alerts to a centralised monitoring dashboard (e.g. via SNMP) in the event of a failure.

Management

Monitoring typically provides a read-only view of a software system, and sometimes there will be runtime management requirements too. For example, it might be necessary to expose functionality that will allow operational staff to modify the runtime topology of a system, modify configuration elements, refresh read-only caches, toggle features on/off, etc.

Audit

There's often a need to keep a log of events (i.e. an audit log) that led to a change in data or behaviour of a software system, particularly where money is involved. Typically such logs need to capture information related to who made the change, when the change was made, and why the change was made. Often there is a need retain the change itself too (i.e. before and after values). Choosing an architectural style such as Event Sourcing can help with audit requirements, although naturally there are trade-offs and implications for things such as privacy.

Flexibility

Flexibility is a somewhat overused and vague term referring to the "flexibility" of your software to perform more than a single task, or to do that single task in a number of different ways. A good example of a flexibility requirement would be the ability for non-technical people to modify the business rules used within the software.

Extensibility

Extensibility is also overused and vague, but it relates to the ability to extend the software to do something it doesn't do now, perhaps using plugins and APIs. Some off-the-shelf products (e.g. Microsoft Dynamics CRM) allow non-technical end-users to extend the data stored and change how other users interact with that data.

Maintainability

Maintainability is often cited as a requirement and, as software developers, we usually strive to build "maintainable" software, but what does this actually mean? It's worth thinking about who will be maintaining the codebase in the future, and what information they will need to maintain the codebase. Maintainability is hard to quantify, but often a good set of architecture and development principles help us write maintainable code.

Legal, Regulatory and Compliance

Some industries are strictly governed by local laws or regulatory bodies, and this can lead to additional requirements related to things like data retention or audit logs. As an example, most finance organisations (investment banks, retail banks, trust companies, etc) must adhere to a number of regulations (e.g. anti-money laundering) in order to retain their ability to operate in the market. The already mentioned GDPR is another regulation that covers EU citizens who are using your software, and there are also some complicated reguations around taxation when supplying "digital services" to consumers in the EU. Regulation is everywhere, and can have some interesting effects on the resulting software architecture.

Internationalisation (i18n)

Many software systems, particularly those deployed on the Internet to a global audience, are no longer delivered in a single language. Internationalisation refers to the ability to have user-facing elements of the software delivered in multiple languages. This is seemingly simple until you try to retrofit it to an existing piece of software, and realise that some languages are written right-to-left instead of left-to-right.

Localisation (L10n)

Related to internationalisation is localisation, which is about presenting information such as numbers, currencies, dates, etc in the conventions that make sense to the culture of the end-user. Sometimes internationalisation and localisation are bundled up together under the heading of "globalisation".

Which are important to you?

There are many quality attributes that could apply to our software systems, but they don't all have equal weighting. Some are more applicable than others, depending on the environment that you work in, and the type of software systems that you build. A web-based system in the finance industry will likely have a different set of quality attributes to an internal system used within the telco industry. My advice is to learn about the quality attributes common within *your* domain and focus on those first when you start building a new system, or modify an existing system.

Working with non-functional requirements

When you're gathering requirements, people will happily give you a wish-list of what they want a software system to do, and there are well established ways of capturing this information as user stories, use cases, traditional requirements specifications, acceptance criteria and so on. This isn't typically the case with quality attributes though.

Capture

I’ve spent most of my 20+ year career in software development working in a consulting environment where we’ve been asked to build software for our customers. In that time, I can probably count on one hand the number of times a customer has explicitly given us information about the non-functional requirements. I’ve certainly received a large number of requirements specifications or functional wish-lists, but rarely do these include any information about performance, scalability, security, etc. In this case, you need to be proactive and capture them yourself.

And herein lies the challenge. If you ask a business sponsor what level of system availability they want, you’ll probably get an answer similar to “100%”, “24 by 7 by 365” or “yes please, we want all of it”.

Refine

Once you’ve started asking those tricky questions related to non-functional requirements, or you’ve been fortunate enough to receive *some* information about them, you’ll probably need to refine them. On the few occasions that I’ve received a functional requirements specification that did include some information about non-functional requirements, they’ve usually been unhelpfully vague. As an example, I once received a 125 page document from a potential customer that detailed the requirements of the software system. The majority of the pages covered the functional requirements in quite some detail, and the last half page was reserved for the non-functional requirements. It said things like:

- **Performance**: The system must be fast.
- **Security**: The system must be secure.
- **Availability**: The system should be running 100% of the time.

Although this isn’t very useful, at least we have a starting point for some discussions. Rather than asking how much availability is needed and getting the inevitable “24 by 7” answer, you can vary the questions depending on who you are talking to. For example:

- “How much system downtime can we tolerate?”
- “What happens if the core of the system fails during our normal working hours of 9am until 6pm?”
- “What happens if the core of the system fails outside of normal working hours?”

What you're trying to do is explore the requirements and get to the point where you understand what the driving forces are. Why does the system *need* to be available? When we talk about "high security", what is it that we're protecting? The goal here is to get to a specific set of non-functional requirements, ideally that we can explicitly quantify. For example:

- How many concurrent users should the system support on average? What about peak times?
- What response time is deemed as acceptable? Is this the same across all parts of the system or just specific features?
- How exactly do we need to secure the system? Do we really need to encrypt the data or is restricted access sufficient?

If you can associate some quantity to the non-functional requirements (e.g. number of users, data volumes, maximum response times, etc), you can can write some acceptance criteria and objectively test them.

Challenge

With this in mind, we all know what response we'll get if we ask people whether they need something like a specific feature. They'll undoubtedly say, "yes"! This is why prioritising functional requirements, user stories, etc is hard. Regardless of the prioritisation scale that you use (MoSCoW, High/Medium/Low, etc), everything typically ends up as a "must have" on the first attempt at prioritisation. You could create a "super-must have" category, but we know that everything will just migrate there.

Sometimes a different approach is needed, and presenting the cost implications can help focus the mind. For example:

- **Architect**: "You need a system with 100% uptime. Building that requires lots of redundancy to remove single points of failure, and we would need two of everything plus a lot of engineering work for all of the automatic failover. It will cost in the region of $1,000,000. Alternatively we can build you something simpler, with the caveat that some parts of the software system would need to be monitored and restarted manually in the event of a failure. This could cost in the region of $100,000. Which one do you need now?"
- **Sponsor**: "Oh, if that's the case, I need the cheaper solution."

Anything is possible, but everything has a trade-off. Explaining those trade-offs can help find the best solution for the given context.

3. Constraints

Everything that we create as software developers lives in the real world, and the real world has constraints. Like quality attributes, constraints can drive, shape and influence the architecture of a software system. Constraints are typically imposed upon you, either by the organisation that you work for or the environment that you work within. Constraints come in many different shapes and sizes.

Time and budget constraints

Time and budget are probably the constraints that most software developers are familiar with, often because there's not enough of either.

Technology constraints

There are a number of technology related constraints that we often come up against when building software, particularly in large organisations:

- **Approved technology lists**: Many large organisations have a list of the technologies they permit software systems to be built with. The purpose of this list is to restrict the number of different technologies that the organisation has to support, operate, maintain, and buy licenses for. Often there is a lengthy exceptions process that you need to formally apply for if you want to use anything "off list". I've still seen teams use Groovy or Scala on Java projects through the sneaky inclusion of an additional JAR file though!
- **Existing systems and interoperability**: Most organisations have existing systems that you need to integrate your software with, and you're often very limited in the number of ways that you can achieve this. At other times, it's those other systems that need to integrate with whatever you're building. If this is the case, you may find that there are organisation-wide constraints dictating the protocols and technologies you can use for the integration points. A number of the investment banks I've worked with have had their own internal XML schemas for the exchange of trading information between software systems. "Concise" and "easy to use" weren't adjectives that we used to describe them!
- **Target deployment platform**: The target deployment platform is usually one of the major factors that influences the technology decisions you make when building a greenfield software system. This includes embedded devices, the availability of

Microsoft Windows or Linux servers, and even this magical thing that we refer to as “the cloud”. Yes, the cloud has constraints. As an example, every Platform as a Service (PaaS) offering is different, and most have restrictions on the technologies you can use, and what your software can do with things like local disk access. If you don’t understand these constraints, there’s a huge danger that you’ll be left with some anxious rework when it comes to deployment time.

- **Technology maturity**: Some organisations are happy to take risks with bleeding edge technology, embracing the risks that such advancements bring. Other organisations are much more conservative in nature.
- **Open source**: Likewise, some organisations still don’t like using open source unless it has a name such as IBM or Microsoft associated with it. I once worked on a project for a high-street bank who refused to use open source, yet they were happy to use a web server from a very well-known technology brand. The web server was actually the open source Apache web server in disguise. Such organisations simply like having somebody to shout at when things stop working. Confusion around open source licenses also prevents some organisations from fully adopting open source too. You may have witnessed this if you’re ever tried to explain the difference between GPL and LGPL.
- **Vendor “relationships”**: As with many things in life, it’s not what you know, it’s *who* you know. Many partnerships are still forged on the golf course by vendors who wine and dine Chief Technology Officers. If you’ve ever worked for a large organisation and wondered why your team was forced to use something that was obviously substandard, this might be the reason!
- **Past failures**: Somewhere around the year 2000, I walked into a bank with a proposal to build them a solution using Java RMI - a technology to allow remote method calls across Java virtual machines. This was met with great resistance because the bank had “tried it before, and it doesn’t work”. That was the end of that design, and no amount of discussion would change their mind. Java RMI was banned in this environment due to a past failure. We ended up building a framework that would send serialized Java objects over HTTP to a collection of Java Servlets instead, which was essentially a workaround to reinvent the same wheel.
- **Internal intellectual property**: When you need to find a library or framework to solve some problem you’re facing, there’s a high probability there’s an open source or commercial product out there that suits your needs. This isn’t good enough for some people though, and it’s not uncommon to find organisations with their own internal logging libraries, persistence frameworks or messaging infrastructures that you *must* use, despite whether they actually work properly. I recently heard of one organisation that built their own CORBA implementation.

People constraints

More often than not, the people around you will constrain the technologies and approaches that are viable to use when developing software. For example:

- How large is your development team?
- What skills do they have?
- How quickly can you scale your development team if needed?
- Are you able to procure training, consulting and specialists if needed?
- If you're handing over your software after delivery, will the maintenance team have the same skills as your development team?

There *will* be an overhead if you ask a Java team to build a Microsoft .NET solution, so you do need to take people into account whenever you're architecting a software system.

Organisational constraints

There are sometimes other constraints that you'll need to be aware of, including:

- Is the software system part of a tactical or strategic implementation? The answer to this question can either add or remove constraints.
- Organisational politics can sometimes prevent you from implementing the solution that you really want to.

Are all constraints bad?

Constraints usually seem "bad" at the time that they're being imposed, but they're often imposed for a good reason. For example, large organisations don't want to support and maintain every technology in existence, so they try to restrict what ends up in production. On the one hand this can reduce creativity but, on the other, it takes away a large number of potential options that would have been open to you otherwise. Software architecture is about introducing constraints too. How many logging or persistence libraries do you really want in a single codebase?

Constraints can be prioritised

As a final note, it's worth bearing in mind that constraints can be prioritised. Just like functional requirements, some constraints are more important than others, and you can often use this to your advantage. The financial risk system that I use as a case study in my training is based upon a real project that I worked on for a consulting company in London. One of the investment banks approached us with their need for a financial risk system and the basic premise behind this requirement was that, for regulatory reasons, the bank needed to have a risk system in place so that they could enter a new market segment.

After a few pre-sales meetings and workshops, we had a relatively good idea of their requirements, along with the constraints that we needed to work within. One of the major constraints was an approved list of technologies that included a heavyweight Java EE stack[1]. The other was a strict timescale constraint.

When we prepared our proposal to win the project work, we basically said something along the lines of, "yes, we're confident that we can deliver this system to meet the deadline, but we'll be using some technologies that aren't on your approved technology list, to accelerate delivery". Our proposal was accepted. In this situation, the timescale constraint was seen as much more important than using only the technologies on the approved technology list and, in effect, we prioritised one constraint over the other. Constraints are usually obstacles that you need to work around, but sometimes you can trade them off against one another.

4. Principles

While constraints are imposed upon you, principles are the things that you want to adopt in order to introduce standard approaches, and therefore consistency, into the way that you build software. There are a number of common principles, some related to development, and others related to architecture.

Development principles

The principles that many software developers instantly think of relate to the way in which software should be developed. For example:

- **Coding standards and conventions**: "We will adopt our in-house coding conventions for [Java|C#|etc], which can be found on our corporate wiki."

[1]This was back in 2006, where early versions of J2EE, and the associated application servers, was a common sight in enterprise software development.

- **Automated unit testing**: "Our goal is to achieve 80% code coverage for automated unit tests across the core library, regardless of whether that code is developed using a test-first or test-last approach."
- **Static analysis tools**: "All production and test code must pass the rules defined in [Checkstyle|FxCop|etc] before being committed to source code control."
- **etc**

Architecture principles

There are also some principles that relate to how the software should be structured. For example:

- **Layering strategy**: A layered architecture usually results in a software system that has a high degree of flexibility, because each layer is isolated from those around it. For example, you may decompose your software system into a UI layer, a business layer and a data access layer. Making the business layer completely independent of the data access layer means that you can (typically) switch out the data access implementation without affecting the business or UI layers. You can do this because the data access layer presents an abstraction to the business layer rather than the business layer directly dealing with the data storage mechanism itself. If you want to structure your software this way, you should ensure that everybody on the development team understands the principle. "No data access logic in the UI components or domain objects" is a concrete example of this principle in action.
- **Placement of business logic**: Sometimes you want to ensure that business logic always resides in a single place for reasons related to performance or maintainability. In the case of Internet-connected mobile apps, you might want to ensure that as much processing as possible happens on the server. Or if you're integrating with a legacy back-end system that already contains a large amount of business logic, you might want to ensure that nobody on the team attempts to duplicate it.
- **High cohesion, low coupling**, SOLID, **etc**: There are many principles related to the separation of concerns, focussing on building small highly cohesive building blocks that don't require too many dependencies in order to do their job.
- **Stateless components**: If you're building software that needs to be very scalable, then designing components to be as stateless as possible is one way to ensure that you can horizontally scale-out your system, by replicating components to share the load. If this is your scalability strategy, everybody needs to understand that they must build components using the same pattern. This will help to avoid any nasty surprises and scalability bottlenecks in the future.

- **Stored procedures**: Stored procedures in relational databases are like Marmite - you either love them or you hate them. There are advantages and disadvantages to using or not using stored procedures, but I do prefer it when teams just pick one approach for data access and stick to it. There are exceptions to every principle though.
- **Domain model - rich vs anaemic**: Some teams like having a very rich domain model in their code, building systems that are very object-oriented in nature. Others prefer a more anaemic domain model, where objects are simply data structures that are used by coarse-grained components and services. Again, consistency of approach goes a long way.
- **Use of the HTTP session**: If you're building a website, you may or may not want to use the HTTP session for storing temporary information between requests. This can often depend on a number of things including what your scaling strategy is, where session-backed objects are actually stored, what happens in the event of a server failure, whether you're using sticky sessions, the overhead of session replication, etc. Again, everybody on the development team should understand the desired approach and stick to it.
- **Always consistent vs eventually consistent**: Many teams have discovered that they often need to make trade-offs in order to meet complex non-functional requirements. For example, some teams trade-off data consistency for increased performance and/or scalability. Provided that we *do* see all Facebook status updates, does it really matter if we all don't see them *immediately*? Your context will dictate whether immediate or delayed consistency is appropriate, but a consistent approach is important.
- **etc**

Beware of "best practices"

If you regularly build large enterprise software systems, you might consider many of the principles that I've just listed to be "best practices". But beware. Even the most well-intentioned principles can sometimes have unintended negative side-effects. That complex layering strategy you want to adopt to ensure a complete separation of concerns can suck up a large percentage of your time if you're only building a quick, tactical solution that will be thrown away in six months. Principles are usually introduced for a good reason, but that doesn't make them good all of the time.

The size and complexity of the software you're building, plus the constraints of your environment, will help you decide which principles to adopt. Context, as always, is key. Having an explicit list of principles can help to ensure that everybody on the team, both now and in the future, is working in the same way, but you do need to make sure these principles are helping rather then hindering. Listening to the feedback from the team members will help you to decide whether your principles are working or not.

Understand their influence

Understanding the functional requirements, quality attributes, constraints and principles at a high-level is essential whenever you start working on a new software system or extend one that exists already. Why? Put simply, this is the basic level of knowledge that you need in order to start making design choices.

First of all, understanding these things can help in reducing the number of options that are open to you, particularly if you find that the architectural drivers include complicated quality attributes (e.g. low latency) or major constraints (e.g. a limited target deployment platform). In the words of T.S.Eliot:

> When forced to work within a strict framework the imagination is taxed to its utmost - and will produce its richest ideas. Given total freedom the work is likely to sprawl.

Secondly, and perhaps most importantly, it's about making "informed" design decisions given your particular set of goals and context. If you started designing a solution to the financial risk system without understanding the requirements related to performance (e.g. calculation complexity), scalability (e.g. data volumes), security and audit, you could potentially design a solution that doesn't meet the goals. Assuming the data volumes are huge would result in an over-designed solution, which would take a longer time to build. On the other hand, assuming the data volumes are trivial would result in an under-designed solution, which would require major rework when it was finally discovered that the solution is not fit for purpose. Both of these extremes would incur additional time and budget, albeit for different reasons.

Software architecture is about the significant design decisions, where significance is measured by cost of change. A high-level understanding of the functional requirements, quality attributes, constraints and principles is a starting point for those significant decisions that will ultimately shape the resulting software architecture. Understanding them early will help to avoid costly rework in the future. Failing to do this may lead to some nasty surprises.

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.

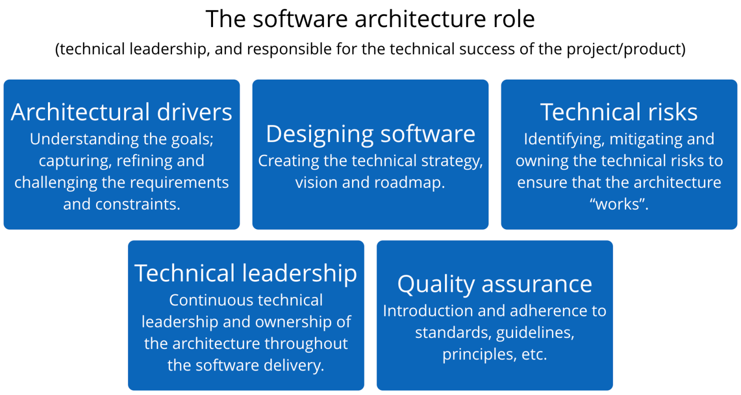
6. The software architecture role

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.


The software architecture role
(technical leadership, and responsible for the technical success of the project/product)
Architectural drivers
Understanding the goals; capturing, refining and challenging the requirements and constraints.
Designing software
Creating the technical strategy, vision and roadmap.
Technical risks
Identifying, mitigating and owning the technical risks to ensure that the architecture "works".
Technical leadership
Continuous technical leadership and ownership of the architecture throughout the software delivery.
Quality assurance
Introduction and adherence to standards, guidelines, principles, etc.

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.

7. Technical leadership

One of the key parts of the software architecture role is technical leadership, and it's applicable to every software team; from a 1-person team building a startup in their garage, through to a globally distributed team with hundreds of developers. This chapter will look at why we need technical leadership, and how to introduce it into your teams.

Controlling chaos

As I summarised in the first chapter, consciously thinking about software architecture provides a number of benefits, and goes a long way solving a number of common problems that you typically associate with chaotic teams, and their approaches to software development. To tame this chaos though, we need to introduce some level of control. But there's a problem here. Whenever I mention the word "control", some people will instantly jump to the conclusion that I'm talking about restricting what team members can do, and micro-managing their everyday tasks. There might be situations where this *is* appropriate but, in general, that's not what I mean by "control".

Provide guidance, strive for consistency

Something you'll see good teams do when working on a large single codebase is to introduce guidance and principles for the purposes of coherence and consistency. If you've ever seen software systems where a common problem or cross-cutting concern has been implemented in a number of different ways, then you'll appreciate why this is important. A couple of examples spring to mind; I've seen a software system with multiple object-relational mapping (ORM) frameworks in a single codebase, which caused no end of confusion when a new developer needed to add a feature that sourced data from the database. Another example is where components across the codebase were configured in a number of different ways, ranging from the use of XML files on disk through to tables in a database. Because of this inconsistency, the initial deployment of this particular system took *days*, with operations staff eventually just resorting to "trial and error" in order to get the system running. The ongoing maintenance of both systems was also challenging, to say the least.

At a level closer to the code, this guidance can also be about ensuring a clear and consistent structure for your codebase; appropriately organising the code into packages, namespaces, folders, modules, components, layers, etc.

Coherence and consistency can only be realised by introducing a degree of control, restraint, boundaries, guidelines and principles. Without these things, it's very possible that team members will wander off on their own tangents, or do things that are don't necessarily follow the architectural vision and intent. This is how teams with good intentions end up in a chaotic situation. If you're going to go through the exercise of agreeing upon some architectural principles that you want to follow as a team, you should also ensure that the team collectively follows them.

How much control do you need?

The real question to be answered here relates to the amount of control that needs to be introduced. At one end of the scale you have a dictatorial, command and control approach where nobody can make a decision for themselves. This isn't an approach that is generally recommended but, as we'll see shortly, there are times where it might be necessary. At the other end of the scale, you have a situation where anybody is free to make whatever choices they like. As an example of what this means in the real world, it's the difference between having full control over all of the technologies used on a software project (from the choice of programming language right down to the choice of logging library) through to being happy for *anybody* on the team make those decisions.

I've seen both approaches to software projects, and I've been asked to take over and "rescue" a few chaotic projects where everybody on the team was basically left to their own devices. The resulting codebase was, unsurprisingly, a mess. Introducing control to get everybody back on track, and contributing to a single vision once again, is really hard work on this type of project but it needs to be done if the team is to have any chance of delivering a coherent piece of software that satisfies the architectural drivers.

One of the difficulties of this, from my experience working with software teams across the globe, is that different countries and cultures place different values on control. Some (e.g. the UK) value control and the restraint that it brings, whereas others (e.g. Scandinavia) value empowerment and motivation because everybody is seen as being equal.

I like to think of control as being a control *lever* rather than something binary that is either on or off. At one extreme you have the full-throttle dictatorial approach, and at the other you have something much more lightweight. You also have a range of control in between the extremes, allowing you to introduce as much control as is necessary. So how much control do you introduce? It's a consulting style answer admittedly, but without knowing your context, it depends on a number of things:

- How much experience does everybody on the team have?
- Has the team worked together before?
- How large is the team?
- How large is the project or product being built?
- Is the business domain complicated?
- Are there complicated quality attributes or constraints that need to be taken into account?
- What sort of discussions are happening on a daily basis within the team, or is everybody working in isolation with headphones on?
- Does the team already seem chaotic?
- Does the codebase already look a mess?

My advice would be to start with *some* control, and listen to the feedback in order to fine-tune it as you progress. If the team are asking lots of "why?" and "how?" questions, then perhaps more information and guidance is needed. If it feels like the other team members are fighting against you all of the time, perhaps you've pushed that lever too far. There's no universally correct answer, but *some* control is a good thing and it's therefore worth spending a few minutes looking at how much is right for your own team. The key is to understand the team you have, and then make sure you apply the appropriate quantity and style of technical leadership.

Collaborative technical leadership is not easy

On a related note, many modern software teams strive to be agile and self-organising, but it's not immediately obvious how the software architecture role fits into this way of working. Something I touched upon the previous chapter is that the software architecture role doesn't necessarily need to be undertaken by a single person. This is often a good place to start, but the role can be a collaborative effort that is shared between a number of people.

As an example, let's imagine that you've been tasked with building a web application with a small team that includes people with specialisms in web technology, server-side programming and databases. From a resourcing point of view this is excellent because collectively you have experience across the entire stack. You shouldn't have any problems then, right?

The effectiveness of the overall team comes down to a number of factors, one of them being people's willingness to leave their egos at the door and focus on delivering the best solution given the context. Sometimes though, individual specialisms can work against a team; simply through a lack of experience in working as a team, or because ego gets in the way of the common goal. If there's a requirement to provide a way for a user to view and manipulate data in our web application, it's easy to see how each of the technology specialists might approach the problem in a very different way:

- **Web developer**: "Just give me the data as JSON and we can do anything we want with it on the web-tier, using JavaScript to dynamically manipulate the dataset in the browser."
- **Server-side developer**: "We should reuse and extend some of the existing business logic in the server-side service layer. This increases reuse, is more secure than sending all of the data to the web-tier, and we can write automated unit tests around it all."
- **Database developer**: "You're both wrong. It's far more efficient for me to write a stored procedure that will provide you with *exactly* the data that you need."

Our own knowledge, experience and preferences tend to influence how we design software, particularly if it's being done as a solo activity. In the absence of communication, there's a danger that we start to make assumptions about where functionality will be implemented, and how features will work, based upon our own mental model of how the sofware should be designed.

Getting such assumptions out into the open as early as possible can really help you avoid some nasty surprises before it's too late. One of the key reasons I prefer using a whiteboard to design software is because it encourages a more collaborative approach, when compared to somebody sitting on their own in front of their favourite modelling tool on a laptop. Like pair programming, collaborating is an effective way to approach the software design process, particularly if it's done in a lightweight way. Collaboration increases quality plus it allows us to discuss and challenge some of the common assumptions that we make based our own knowledge, experience and preferences. It also paves the way for collective ownership of the code, which again helps to break down the silos that often form within software development teams. Everybody on the team will have different ideas and those different ideas need to meet.

A word of caution, though. Be wary of any advice stating that collaborative technical leadership is easy. It's not. I regularly run software architecture katas where groups of 2-5 people are asked to design a software system, and I've witnessed some of these groups being unable to reach consensus on decisions relating to design and technology choices. In extreme cases, groups have split because of ego and personality conflicts. Remember this the next time somebody suggests that "everybody should be the architect". Speaking of which…

Everybody is an architect

In "Extreme Programming Annealed", Glenn Vanderburg discusses the level at which the Extreme Programming practices work, where he highlights the link between architecture and collective ownership. When we talk about collective ownership, we're usually referring to collectively owning the code, so that anybody on the team is empowered to make changes. In order for this to work, there's an implication that everybody on the team has at least some basic understanding of the "big picture". Think about your current project/product - could you jump into *any* part of the codebase and understand what was going on?

Imagine if you did have a team of experienced software developers that were all able to switch between the low-level detail of the code and the high-level view of the software architecture. A team of genuinely hands-on architects. That team would be amazing! All of the elements you usually associate with software architecture (satisfying the non-functional requirements, quality assurance, etc) would all get dealt with, and nothing would slip through the gaps. From a *technical perspective*, this is a self-organising team, because everybody is taking responsibility for the activities associated with the software architecture role. In other words, we have true collective technical leadership.

Except when they're not

My big problem with the self-organising team idea is that we talk about it a lot in industry, yet I rarely see it in practice. I have definitely seen some self-organising teams that work really, really well. But I've seen more teams that claim to be self-organised, but they are clearly not. This could be a side-effect of me predominantly working in a consulting environment because *my* team always changes from project to project, plus I don't tend to spend more than a few months with any particular customer. Or, as I suspect, true self-organising teams are very few and far between. Striving to be self-organising is admirable but, for many software teams, this is like running before you can walk.

In "Elastic Leadership - Growing self-organizing teams", Roy Osherove describes his concept of "Elastic Leadership" where the leadership style needs to vary in relation to the maturity of the team. Roy categorises the maturity of teams using a simple model, with each level requiring a different style of leadership:

1. **Survival model (chaos)**: requires a more direct, command and control leadership style.

2. **Learning**: requires a coaching leadership style.
3. **Self-organising**: requires facilitation to ensure the balance remains intact.

Roy uses the ScrumMaster role as an example. Teams in the initial stages of their maturity will benefit from a single person undertaking the ScrumMaster role to help drive them in the right direction. Self-organising teams, on the other hand, don't need to be told what to do. The clue is in the name; they are self-organising by definition, and can take the role upon themselves.

I would say that the same is true of the software architecture, and therefore, the technical leadership role. As I said, a team where everybody is an experienced software developer and architect would be amazing, but this isn't something I see happen on a regular basis. Most projects don't have *anybody* on the team with experience of the software architecture role, and this is evidenced by codebases that don't make sense (big balls of mud), designs that are unclear, systems that are slow and so on. This type of situation is the one I see the most and, from a technical perspective, I recommend that *one* person on the team takes responsibility for the software architecture role.

With collective code ownership, everybody needs to be able to work at the architecture level, and therefore everybody *is* an architect to some degree. Teams that aren't at the self-organising stage will struggle if they try to run too fast though. Despite people's aspirations to be agile, collective code ownership and a distribution of the architecture role are likely to hinder many teams rather than help them. Chaotic teams need a more direct leadership approach and they will benefit from a single point of responsibility for the technical aspects of the software delivery. In other words, they will benefit from a single person looking after the software architecture role. Ideally this person will coach others so that they too can eventually help with this role. One software architect or many? Single point of responsibility or shared amongst the team? The software architecture role exists on every team. Only the context will tell you the right answer.

Do agile teams need software architects?

Unfortunately, many software teams, especially those who deem themselves to be "agile", have historically viewed the software architecture role as an unnecessary evil rather than an essential complement, probably because they've been burnt by big design up front in the past. Some teams are also *so* focussed on the desire to be "agile", that other aspects of the software development process get neglected. Chaos rather than self-organisation ensues, yet such teams challenge the need for a more direct leadership approach. After all, they're striving to be agile and having a single point of responsibility for the technical aspects of the project conflicts with their view of what an agile team should look like. This conflict tends to cause people to think that agile and architecture are opposing forces - you can have one or the other, but not both. It's not architecture that's the opposing force though, it's big design up front. Agile software teams still need to consider the software architecture role because all those tricky concerns around complicated quality attributes and constraints don't go away. It's just the execution of the architecture role that differs.

Software development is not a relay sport

There are a lot of people out there, particularly in larger organisations, calling themselves "solution architects" or "technical architects", who design software and document their solutions before throwing them over the wall to a separate development team. With the solution "done", the architect will then move on to do the same somewhere else, often not even taking a cursory glimpse at how the development team is progressing. In this situation, there's often a tendency for the receiving team to not take ownership of the solution, or they ignore the architect's ideas and do something completely different.

I've met a number of such architects in the past, and one particular interview I held epitomises this approach to software development. After the usual "tell me about your role and recent projects" conversation, it became clear to me that this particular architect (who worked for one of the large "blue chip" consulting firms) would create and document a software architecture for a project before moving on elsewhere to repeat the process. After telling me that he had little or no involvement in the project after he handed over the "solution" to the development team, I asked him how he knew that his software architecture would work. Puzzled by this question, he eventually made the statement that this was "an implementation detail". He confidently viewed his software architecture as correct, and it was the development team's problem if they couldn't get it to work. I have a name for this approach; "AaaS" - "Architecture as a Service"!

The software architecture role is certainly about steering the ship at the start of a software delivery, which includes things like managing the architectural drivers, and putting together a software design that is sensitive to them. But it's also about *continuously* steering the ship, because your chosen path might need some adjustment en-route. After all, agile approaches have shown us that we don't necessarily have (and need) all of the information up front.

A successful software project requires an initial vision to be created as a starting point, communicated and potentially evolved throughout the entirety of the software development life cycle. For this reason alone, it doesn't make sense for one person to create that vision, and for another team to (try to) deliver it. When this does happen, the set of initial design artefacts is essentially a baton that gets passed between the architect and the development team. This is inefficient, ineffective and exchange of a document means that much of the decision making context associated with creating the vision is also lost.

A failure to create a starting point and set a technical direction tends to lead to chaos - poorly structured, internally inconsistent codebases (the stereotypical "big ball of mud") that are hard to understand, hard to maintain and potentially don't satisfy one or more of the important quality attributes such as performance, scalability or security. Software development is not a relay sport, and successful delivery is not an "implementation detail". Continuous technical leadership is crucial for all teams of all sizes.

Mind the gap

Finally, and on a related note, it's also worth being cautious of creating too much of a gap between you, as a software architect, and the rest of the development team. The software architecture role is different to a software developer role. Some people view it as a step up from being a developer, and some view it as a step sideways. However you view it, the architecture role is about looking after "the big picture". I've seen teams that do understand the importance of software architecture and will bring in somebody with the prestigious title of "Architect", only to put them on a pedestal above the rest of the team. If anything, this instantly isolates the architect by creating an exaggerated gap between them and the development team they are supposed to be working with.

Unfortunately, many software teams have this unnecessary gap between the development team and the architect, particularly if the architect is seen to be dictating and laying down commandments for the team to follow. This leads to several problems.

- The development team doesn't respect the architect, regardless of whether the architect's decisions are appropriate or not.
- The development team becomes demotivated.
- Important decisions fall between the gap, because responsibilities aren't well defined.
- The project eventually suffers because nobody is looking after the big picture.

Fortunately, there are some simple ways to address this problem, from both sides. Software development is a team activity after all. If you're a software architect:

- **Be inclusive and collaborate**: Get the development team involved in the software architecture process to help them understand the big picture and buy-in to the decisions you are making. This can be helped by ensuring that everybody understands the rationale and intent behind the decisions.
- **Get hands-on**: Where possible, get involved with some of the day-to-day development activities of the delivery to improve your understanding of how the architecture is being delivered. Depending on your role and team size, this might not be possible, so look at other ways of retaining some low-level understanding of what's going on such as assisting with design and code reviews. Having an understanding of how the software works at a low-level will give you a better insight into how the development team are feeling about the architecture (e.g. whether they are ignoring it!) and it will provide you with valuable information that you can use to better shape/influence your architecture. If the developers are experiencing pain, you need to feel it too.

And if you're a software developer:

- **Understand the big picture**: Taking some time out to understand the big picture will help you understand the context in which the architectural decisions are being made, and enhance your understanding of the system as a whole.
- **Challenge architectural decisions**: With an understanding of the big picture, you now have the opportunity to challenge the architectural decisions being made. Architecture should be a collaborative process and not dictated by people that aren't engaged in the project day-to-day. If you see something that you don't understand or don't like, challenge it.
- **Ask to be involved**: Many projects have an architect who is responsible for the architecture and it's this person who usually undertakes all of the "architecture work". If you're a developer and you want to get more involved, just ask. You might be doing the architect a favour!

What I've talked about here is easily applicable to small/medium project teams, but things do start to get complicated with larger teams. By implication, a larger team means a bigger software system, and a bigger software system means a bigger "big picture". Whatever the size of the team though, ensuring that the big picture isn't neglected is crucial for success, and this typically falls squarely within the remit of those people performing the software architecture role. However, most software teams can benefit from reducing the unnecessary gap between developers and architects, with the gap itself being reducible from both sides. Developers can increase their architectural awareness, while architects can improve their collaboration with the rest of the team. Make sure that *you* mind the gap, and others may follow.

8. Software architects and coding

Applying the building metaphor to software doesn't necessarily work, although in medieval times, the people who architected buildings were the select few that made it into the exclusive society of "master builders". The clue here is in the name, and a master builder really was a master of their craft. Once elevated to this status though, did the master builder continue to build, or was that task left to those less noble? Fast-forward several hundred years and it seems we're asking the same question about the software industry, and whether software architects should code.

Many people still believe the role of a software architect is solely about high-level, abstract thinking. I'm sure you've seen the terms "ivory tower architect", "PowerPoint architect", or "architecture astronaut" used to refer to people that design a solution without ever considering the detail. If we go back in time though, this isn't what the architecture role was all about.

A step back in time

If you trace the word "architect" back to its roots in Latin (*architectus*) and Greek (*arkhitekton*), it basically translates to "chief builder" and, as indicated by the name, these people were masters of their craft. In medieval times, the term "architect" was used to signify those people who were "master masons", where "mason" refers to stonemasons because stone is what the majority of the buildings were constructed from at the time. This quote summarises the role well:

> A master mason, then, is a manipulator of stone, an artist in stone and a designer in stone.

This quote could equally be applied to code and software developers.

Did master builders actually build?

The key question in all of this is whether the master builders actually built anything. If you do some research into how people achieved the role of "master mason", you'll find something similar to this:

> Although a master mason was a respected and usually wealthy individual, he first had to prove his worth by going through the ranks as a stonemason and then a supervisor, before being appointed to the highest position in his trade.

The Wikipedia page for "Architect" says the same thing:

> Throughout ancient and medieval history, most architectural design and construction was carried out by artisans—such as stone masons and carpenters, rising to the role of master builder.

Interestingly, there's no single view on how much building these master masons actually did. For example:

> How much contact he actually had with this substance is, however, debatable. The terminology may differ but, as I understand it, the basic organisation and role of the medieval master mason is similar to that of the chief architect today – perhaps a reflection of the immutable fundamentals of constructing buildings.

Only when you look at what the role entailed does this truly make sense. To use another quote:

> A mason who was at the top of his trade was a master mason. However, a Master Mason, by title, was the man who had overall charge of a building site and master masons would work under this person. A Master Mason also had charge over carpenters, glaziers etc. In fact, everybody who worked on a building site was under the supervision of the Master Mason.

To add some additional detail:

> The master mason, then, designed the structural, aesthetic and symbolic features of what was to be built; organised the logistics that supported the works; and, moreover, prioritised and decided the order of the work.

Ivory towers?

If this is starting to sound familiar, wait until you hear how the teams used to work:

> Every lesser mason followed the directions set by the master and all decisions with regard to major structural, or aesthetic, problems were his domain.

It's certainly easy to see the parallels here in the way that many software teams have been run traditionally, and it's not surprising that many agile software development teams aspire to adopt a different approach. Instead of a single dedicated technical leader that stays away from the detail, many modern software development teams attempt to share the role between a number of developers. Of course, one of the key reasons that many architects stay away from the detail is because they simply don't have the time. This typically leads to a situation where the architect becomes removed from the real-world day-to-day reality of the team, and slowly becomes detached from it. It turns out that the master masons suffered from this problem too:

> If, as seems likely, this multiplicity of tasks was normal it is hardly surprising that master masons took little part in the physical work (even had their status permitted it). Testimony of this supposition is supplied by a sermon given in 1261 by Nicholas de Biard railing against the apparent sloth of the master mason "who ordains by word alone".

This quote from Agile Coaching, by Rachel Davies and Liz Sedley, highlights a common consequence of this in the software industry:

> If you know how to program, it's often tempting to make suggestions about how developers should write the code. Be careful, because you may be wasting your time - developers are likely to ignore your coding experience if you're not programming on the project. They may also think that you're overstepping your role and interfering in how they do their job, so give such advice sparingly.

To cap this off, many people see the software architecture role as an elevated position/rank within their organisation, which further exaggerates the disconnect between developer and architect. It appears that the same is true of master masons too:

> In order to avoid the sort of struggle late Renaissance artists had to be recognised as more than mere artisans it would seem that master masons perpetuated a myth (as I see it) of being the descendants of noblemen. Further to this, by shrouding their knowledge with secrecy they created a mystique that separated them from other less 'arcane' or 'noble' professions.

Should software architects write code?

Much of this short history lesson points to the idea that master builders didn't actually have much time for building, even though they possessed the skills to do so. To bring this back to the software industry, should software architects write code?

In my ideal view of the world, yes, software architects *should* write code. Somebody once told me that the key characteristic of a good architect is the ability to think in an abstract way. You can also think of it as the ability to not get caught up in the details all of the time. And that's fine, but those abstract boxes and lines that you're drawing on the whiteboard do need to be coded at some point. As a software architect, it also definitely makes sense to keep your technical skills up to date, and coding is a great way to do this.

Most of the best software architects I know have a software development background, and they still enjoy writing code. For some reason though, many organisations don't see coding as a part of the software architecture role. Being a "hands-on software architect" doesn't necessarily mean that you *need* to get involved in the day-to-day coding tasks, but it does mean that you're continuously *engaged* in the delivery, actively helping to lead and shape it. Having said that, why shouldn't the day-to-day coding activities be a part of the software architecture role?

Writing production code

My ideal recommendation is to make coding a part of your role as a software architect. You can do this by simply being an integral part of the software development team. In other words, you have a "software architecture hat" and a "coding hat". You don't need to be the best coder on the team, but the benefits of being hands-on and engaged in the delivery process are huge. After all, there's a substantial difference between knowing what something is, and knowing how to actually do it. Appreciating that you're going to be contributing to the coding activities often provides enough incentive to ensure that your designs are grounded in reality. Coding provides a way for the architect(s) to share the software development experience with the rest of the team, which in turn helps them better understand how the architecture is viewed from a development perspective. It also allows you to keep "a pulse on the code", helping you ascertain whether the team is actively following the architectural principles or ignoring them.

Along with the obvious benefits associated with creating a software architecture that can actually be implemented by people, contributing to the coding activities helps you build rapport with the rest of the team, which in turn helps to reduce the gap between architects and developers that you see on many software teams. As implied by the previous quote from the Agile Coaching book, it's easier to lead a team if you're seen to be a part of that team.

Of course, there are situations where it's not practical to be involved at the code level. For example, a large project generally means a bigger "big picture" to take care of, which implies more technical leadership, and there may be times when you just don't have the time for coding. But, generally speaking, a software architect who codes is a more effective and happier architect. You shouldn't necessarily rule out coding just because "you're an architect".

Building prototypes, frameworks and foundations

Although the software architecture role is much easier to do if you're seen to be part of the development team, sometimes this isn't possible. One of the problems with being promoted to, or assigned as, a software architect is that you might find that you can't code as much as you'd like to. This may be down to time pressures because you have a lot of "architecture" work to do, or it might simply be down to company politics not allowing you to code. I've seen this happen. If this is the case, building prototypes and proof of concepts related to the software system in question is a great way to be involved. Again, this allows you to build some rapport with the team and it's a great way to evaluate that your architecture will work.

As an alternative, you could help to build the frameworks, libraries and foundations that the rest of the team will use. Try to resist the temptation to build these things and then hand them over to the team because you might be seen as the stereotypical "ivory tower architect", and this approach may backfire. Software development is driven by fads and fashions much more than it really should be, so also beware of building something that the rest of the team might deem to be worthless and old-fashioned, or too bleeding edge that it's unusable given the real-world time and budget constraints.

Performing code reviews

There's obviously no substitute for writing real production code, but getting involved with (or doing) the code reviews is one way to at least keep your mind fresh with technology and how it's being used. I wouldn't recommend this as a long-term strategy, and you could damage your reputation if you start nitpicking or getting involved in discussions about technologies that you have no experience with. I remember having to explain my Java code to an architect who had never written a single line of Java in his life. It wasn't a fun experience, plus I didn't feel particularly motivated to follow his "advice".

Experimenting and keeping up to date

You need to retain a certain level of technical knowledge so that you can competently design a solution using it. But if you are unable to contribute to the delivery, how do you maintain your coding skills as an architect?

Often you'll have more scope outside of work to maintain your coding skills; from contributing to an open source project through to continuously playing with the latest language/framework/API that piques your interest. Books, blogs, podcasts, conferences and meetups will get you so far, but sometimes you just have to break out the code. One of the upsides of a long commute on public transport is that it does give you time to play with technology. Assuming you can keep your eyes open after a hard day at work, of course!

Don't code all the time

However you do it, coding is a great way to keep your technology skills sharp in our ever-changing world. Many people believe that software architecture is a "post-technical" career choice, and something that you do when you don't want to write code any more. This isn't the case though. As we'll see in the next chapter, the software architecture role requires deep technology skills alongside a breadth of experience and more general knowledge. It requires "generalising specialists" who are able to answer the question of whether their design will actually work. Leaving this as an "implementation detail" is not acceptable.

With this in mind, my final piece of advice is simple: don't code all of the time! If you're coding all of the time, who is performing the rest of the software architecture role?

The tension between coding and being senior

Many organisations have policies that prevent software architects from engaging in coding activities because their architects are "too valuable to undertake the coding work". Clearly this is the wrong attitude. Why let your software architects put all that effort into designing software if you're not going to let them contribute to its successful delivery?

I've been fortunate in that I've managed to retain a large hands-on element as a part of my software architecture roles, and I've written code on most of the development projects that I've been involved in. I'm a firm believer that you're in control of creating your own opportunities. The reason I've remained hands-on is that I've always expressed to my employers that coding is a crucial part of the role - "coding is essential when designing software because I need to keep my skills up to date and understand that what I'm designing will work". Plus, I'm not ashamed to admit that coding is fun.

Unfortunately many organisations seem to think that coding is the easy part of the software development process, which is why they see an opportunity to save some money and let somebody else do it, often in another country. Such organisations unfortunately view writing code as a "low value" activity, and not something that a senior employee should be doing. Tension therefore arises because there's a disconnect between the seniority of the software architect in the organisation, and the value associated with the coding activities. With this in mind, it should come as no surprise that I'm regularly asked whether software architects can continue to code if they are to climb the corporate career ladder. This is a shame, particularly if these people really enjoy the technical side of what they do.

My take on this is yes, absolutely, you *can* continue to code. For me, it's quite frustrating to hear people say, "well, I understand that I'll have to give up coding to become an architect, or to progress further up the career path". There are lots of organisations where this is the expectation and, although frustrating, it's somewhat reassuring that I'm not the only person to have been told that coding doesn't have a place in the senior ranks of an organisation.

As a software architect, you take on a great deal of responsibility for building software that satisfies the architectural drivers, performing technical quality assurance, making sure the software is fit for purpose, etc. It's a leadership role and coding (leading by example) is one of the very best ways to make sure the delivery is successful.

In my experience, this doesn't happen in small organisations and startups, because everybody usually has to get involved in whatever is needed. No, it's those larger organisations where the tension is greatest. I spent some time working for a medium size consulting firm where my job grade put me as a peer of the management team, yet I still wrote code. In some ways it was quite an achievement to be graded as an "Executive Manager" and write code on a daily basis! Yet it often felt very uncomfortable, as other managers would regularly try to push my name into boxes on the corporate organisation chart.

Being in this situation is tricky and only you can get yourself out of it. Whether you're in an organisation where this is happening or you're looking to move on, be clear about how you view the role of coding as a software architect and be prepared to stand your ground.

Software as an engineering discipline

Although I recommend that modern software architects do remain involved in the actual process of building software, it's worth looking at why modern building architects don't help with the process of hands-on building. To answer this, we need to look how the role has evolved over the years:

> Throughout ancient and medieval history, most architectural design and construction was carried out by artisans—such as stone masons and carpenters, rising to the role of master builder. Until modern times there was no clear distinction between architect and engineer. In Europe, the titles architect and engineer were primarily geographical variations that referred to the same person, often used interchangeably.

The Wikipedia page for structural engineering provides further information:

> Structural engineering has existed since humans first started to construct their own structures. It became a more defined and formalised profession with the emergence of the architecture profession as distinct from the engineering profession during the industrial revolution in the late 19th century. Until then, the architect and the structural engineer were usually one and the same - the master builder. Only with the development of specialised knowledge of structural theories that emerged during the 19th and early 20th centuries did the professional structural engineer come into existence.

In essence, the traditional architect role has diverged into two roles. One is the structural engineer, who ensures that the building doesn't fall over. And the other is the architect, who interacts with the client to gather their requirements and design the building from an aesthetic perspective. Martin Fowler's bliki has a page that talks about the purpose of and difference between the roles.

> A software architect is seen as a chief designer, someone who pulls together everything on the project. But this is not what a building architect does. A building architect concentrates on the interaction with client who wants the building. He focuses his mind on issues that matter to the client, such as the building layout and appearance. But there's more to a building than that.

Building is now seen as an *engineering discipline* because of the hundreds of years of experience that have led to the creation of a huge body of knowledge behind it. Structural engineers now understand the laws of physics, and are able to model/predict how materials will behave when they are used to construct buildings. Buildings today are still mostly built using the same materials as they were hundreds of years ago, but structures are increasing in size, and we're using materials in new ways. Much of the underlying knowledge and principles haven't changed dramatically though.

In contrast, the software development industry is still relatively young and changes at an alarmingly fast pace. We live in the era of "Internet time", where it seems like we're inventing a new technology every twenty minutes. Until our industry reaches the point where software can be built in the same way as a predictive engineering project, and we have a standardised body of knowledge to refer to, it's crucial that somebody on the team keeps up to date with technology and is able to make the right decisions about how to design software. In other words, software architects still need to play the role of structural engineer *and* architect. Coding is a great way to keep your skills up to date, understand whether your design is implementable, and to build rapport with the rest of the team.

9. The skills and knowledge of a software architect

Despite what you may hear some people say, software architecture is not a “post-technical” or “non-technical” career choice. Simply drawing a handful of boxes and lines on a whiteboard, before telling a team to “just go and build it”, is not what software architecture is all about. The role requires a number of skills, some of which are technical, and some of which are not.

Technology skills

Most of the best software architects I know have come from a software development background, and they are still good software developers themselves. This doesn’t mean that they are the *best* coders on a team, but they *are* able to easily switch between the low-level technical details and the “big picture”. Good software architects often have a deep technical specialism, supplemented with a breadth of knowledge that they’ve gained from many years of experience of building software. Let’s look at this in more detail.

As software developers, we tend to have knowledge about things like programming language syntax, APIs, frameworks, design patterns, automated unit testing, and all of the other technical activities that we perform on a daily basis. And this is the same basic knowledge that software architects need too. Why? Because people in the software architecture role need to understand technology, so that they are able to honestly answer the following types of questions when designing software:

- What are the trade-offs associated with this technical decision?
- Is the solution going to work?
- Do the diagrams reflect how we are really going to build the software?

When you consider the learning curve associated with being truly productive in any given programming language, it’s common for professional software developers to only know one or two languages *really* well. Eventually, these same people typically become known for being, or associate themselves with being, a “Java developer” or a “.NET developer”, for example. I’ve been there and done that myself. Java is the programming language that I’ve used the most, so it’s the one that I know the best. But I have built software with other languages, including a few years using C# and the .NET platform. For a large part of my professional career though, I only ever wanted to work on “Java projects”. And I only ever applied for jobs at organisations that were seeking “Java developers”.

Although we may *strive* to be open-minded when designing software, it’s easy to get siloed into a single technology stack, based upon the experience that we have. While there’s nothing particularly wrong with this, you have to be careful that you do keep an open mind. As the saying goes, “if all you have is a hammer, everything will look like a nail”. Gaining experience is an essential part of the learning journey, but that same experience shouldn’t constrain you. Java is a great choice for building many software systems, but it’s not the only choice, and other languages may present a better option. Similarly, you don’t *need* a relational database for every software system, but often this is the first thing that gets drawn on a diagram when a team is designing a solution.

Having said this, understanding the intricacies of a particular technology stack *is* important. If, as a software architect, you’re asked to design a highly scalable server-side web application, you need to understand how to actually do this with the technologies that you know about. How do you scale server-side applications in your technology of choice? How do you replicate HTTP sessions across a cluster of servers to provide resilience in the event of server failure? Are there any performance/scalability issues associated with HTTP session replication? Or should you use an out-of-process key-value store for user session data instead? What’s the best framework to use for building this application? How does user authentication work? These are the types of questions that you need to answer, and you can’t do that if you don’t have some detailed knowledge of the technology you’re planning to use. Of course, if you’re planning on using a new technology, you likely won’t have this knowledge. But you should know how to gain it, and we’ll talk about that in the next section.

A breadth of technology knowledge is also important too, so that you can answer the following types of questions too:

- Is the technology that we’ve chosen the most appropriate given the other options available?
- What *are* the other options available?
- Is there a common architectural pattern/style that we should be using?
- Do we understand the trade-offs of the decisions that we’re making?
- Have we catered for the desired quality attributes, both now and in the future?
- What are the current best practices for web application security?

- What problems are we likely to encounter by adopting this design?
- How can we prove that this architecture will work?
- How popular is this technology at the moment, and it is easy to find developers with experience?
- Is it easy to integrate this technology with a continuous integration/delivery platform?

Although general design knowledge, techniques, patterns and approaches are often applicable across a number of different technologies, not understanding how to apply them successfully at the implementation level can cause issues.

However you look at it, technology knowledge underpins the software architecture role, requiring a combination of deep technology skills coupled with broader knowledge. If the people designing software can't answer the question of whether the architecture will work, they are probably the wrong people to be doing that job. Software architecture is most definitely a technical career choice, but that's not the end of the story.

Soft skills

Having good technology skills is only half of the story. Since we're essentially talking about a leadership role, "soft skills" (or "people skills") are vitally important too. This includes:

- **Leadership**: In it's simplest form, leadership is the ability to create a shared vision, and to then take people on a journey to achieve the common goal.
- **Communication**: You can have the best ideas and vision in the world, but you're dead in the water if you're unable to effectively communicate this to others. This means people inside and outside of the software development team; using the language, terminology and level of detail that is appropriate to the audience.
- **Influencing**: This is a key leadership skill and can be done a number of ways, from overt persuasion through to Neuro-Linguistic Programming or Jedi mind tricks. Influencing people can also be done through compromise and negotiation. Individuals may have their own ideas and agendas, which you need to deal with while keeping everybody "on-side" and motivated to get the result that you need. Good influencing requires good listening and exploring skills too.
- **Confidence**: Confidence is important, underpinning effective leadership, influence and communication. Confidence doesn't mean arrogance though.
- **Collaboration**: The software architecture role shouldn't be done in isolation, and collaboration (working with others) to come up with a better solution is a skill that is worth practicing. This means listening, being open-minded and responsive to feedback.

- **Coaching**: Not everybody will have experience in what you're trying to do, so you'll need to coach people on their role, technologies, etc.
- **Mentoring**: Mentoring is about facilitating somebody's learning rather than telling them how to do something. As a leader you may be asked to mentor others on the team.
- **Motivation**: This is about keeping the team happy, up-beat and positive. The team also needs to feel motivated to follow any vision that you create as a software architect. You will face an uphill battle without the rest of the team's buy-in.
- **Facilitation**: There will often be times where you need to step back and facilitate discussions, particularly where there is conflict within the team. This requires exploration, objectivity, and helping the team come up with a solution to build consensus.
- **Political**: There are politics at play in every organisation. My mantra is to steer clear of getting involved as far as possible, but you should at least understand what's going on around you so that you can make more informed decisions.
- **Responsibility**: You can't necessarily blame the rest of the software development team for failure, and it's important that you have a sense of responsibility. As somebody performing the software architecture role, it's *your* problem if the software architecture doesn't satisfy the architectural drivers, or the technical quality is poor.
- **Delegation**: Delegation is an important part of any leadership role, and there's a fine line between delegating everything versus doing everything yourself. You should learn to delegate where appropriate, but remember that it's not necessarily the *responsibility* you're delegating.

If you're responsible for the software architecture and technical delivery of a software system, you must have the **authority** to make decisions. If you have the responsibility but not the authority, and are therefore continually seeking permission to make decisions, you could be in for a bumpy ride.

The software architecture role, and technical leadership, is about getting the whole team heading in the same direction. Dictating instructions to a team of software developers isn't likely to be a very effective approach if you're new to the environment, or you're not their immediate line manager, which is often the case if you're supplementing a team as an outside consultant. This is where the soft skills come into play, particularly those related to building relationships, creating trust and motivating the team. As I mentioned in the previous chapter, being part of the team, and writing code where needed, goes a long way to securing a successful outcome too. It's easier to lead a team if you're seen to be a *part* of that team.

As a software architect, you're likely to be an important role model for a number of people on the development team too. The reason for this? Many of the team are probably aspiring software architects themselves. Although this is a flattering situation to be in, there are some major downsides if you take your eye off the ball.

Whether you've recognised it or not, you're in a very influential position, and the eyes of the development team are likely to be watching your every move. For this reason alone, you have the power to change the whole dynamic of the team, whether you like it or not. If you're motivated, the development team is likely to be motivated. If you're enthusiastic about the work, the rest of the team is likely to be enthusiastic about the work. If you're optimistic (and realistic!) that everything will work out, the development team will be too.

You can almost think of this as a self-supporting loop of positive energy, where your enthusiasm drives the team, and their enthusiasm drives you. This is all fantastic, but it's not hard to see the damage that can be caused by a slip-up on your behalf. Any degree of lethargy, apathy or pessimism will rub onto your team quicker than you can say "but we'll be okay" and you'll start spiralling towards a vicious circle of negativity.

We don't often talk about the softer side of being a software architect, but the soft skills are sometimes harder, and more important, than being technically strong. A happy team is a team that delivers. As a leader, it's *your* responsibility to keep the team positive, and your role in the overall team dynamics shouldn't be underplayed.

Collaborate or fail

The success of agile approaches has shown us that we need to have regular communication with the end-users, or their representatives, so we can be sure we're building software that will meet their needs. But what about all of those other stakeholders? Although software development teams might have a clear vision about *what* the software should do, there are often many other factors that need to be taken into consideration before you can release your code into a live environment. For example, you might get into the following types of discussions if you don't regularly collaborate with some of the other stakeholders that exist in your organisation:

- "Nobody told us you needed a production database created on this server."
- "We can't upgrade to [Java 11|.NET 4.7] on that server until system X is compatible."
- "We don't have spare production licenses for your database."
- "Sorry, the way you're storing passwords contravenes our security policy."
- "We need to undertake some operational acceptance testing before we promote your application into the production environment."
- "How exactly are *we* supposed to support this application?"

- “I don’t care if you have a completely automated release process … we’re not giving you the production database credentials for your configuration files.”
- “We need to run this past the risk and compliance team.”
- “Corporate policy says that you are not permitted to use public cloud services, especially when they are based in the US.”

These types of conversations can lead to making a surprising discovery, causing your team some major rework because you either didn’t understand the environment constraints, or you made some assumptions about the target deployment platform. I’ve done this myself in the past - making an assumption about the version of a programming language runtime installed on a production server, which turned out to be incorrect. There are a number of people who likely need to contribute to the overall process of delivering software, to ensure that the resulting software system will successfully integrate with its environment. This includes:

- **Current development team**: The current team need to understand the architecture, and be aware of the drivers, so that they produce a solution that is architecturally consistent and therefore “works”.
- **Future development team**: Any future development/maintenance teams need to have the same information to hand so that they understand how the solution works, and are able to modify/extend it in a consistent way.
- **Other teams**: Often your software needs to integrate with other systems within the environment, from bespoke (custom built) software systems through to off-the-shelf vendor products. It’s crucial that everybody agrees on how this will work.
- **Database administrators**: Some organisations have separate database teams that need to understand how your solution uses their database services (e.g. from design and optimisation, through to capacity planning, data security and archiving).
- **Operational/support staff**: Operational staff typically need to understand how to run and support your software, including aspects such as configuration and deployment through to monitoring and problem diagnostics.
- **Compliance, risk and audit**: Some organisations have strict regulations that they need to follow, and people in your organisation may need to certify that you’re following them too.
- **Security team**: The same is true with regards to security. Some organisations have dedicated security teams that need to review software before it is deployed into production environments.

- **Data privacy**: With the recent introduction of GDPR, many organisations now have an obligation to take a much more structured approach to how data is stored/accessed, and the purposes that data is used for. You might need to consult with people within your organisation so they can assert that your software is not breaching GDPR.

These are just some of the stakeholders that may have an interest in your architecture, but there are probably others depending on your organisation, and the way that it works. You may be mistaken if you think you can design software in isolation, without collaborating with anybody else. Software systems don't live in isolation, and the software design process should be a platform for conversation. A five minute conversation now could help capture those often implied architectural drivers, and improve your chance of a successful delivery. If you don't collaborate, expect to fail.

Domain knowledge

The other, often ignored aspect that makes a good software architect, is that of domain knowledge, or understanding "the business". A good working knowledge of the business domain is essential. If you're working within the finance industry, you should know something about how your particular part of the finance industry works (e.g. funds management, investment banking, retail banking, etc). Most business domains are far more complicated than they really should be, and even seemingly simple domains can surprise you. I remember the first time that I saw the ferry and hotel domains, which surprisingly aren't solely about booking seats on boats or rooms in hotels. Having an appreciation of the business domain helps you to better understand the goals, and therefore create successful software products.

And this raises an interesting question. A deep knowledge of the business domain only comes from working within that domain for an extended period of time. But what happens if you're working as a software architect in a consulting capacity, building software for your customers? After all, most consultants move between different customers, teams and business domains on a regular basis. Is it therefore fair to expect consultants to possess deep domain knowledge?

Most of my career has been spent working for IT consulting companies, where I've either built software systems *for* our customers under an outsourcing arrangement, or *with* our customers as a part of a mixed customer-supplier team. Although undertaking the software architecture role within a consulting context is fundamentally the same as undertaking the role in any other context, there are a couple of approaches that I've seen people take with regards to domain knowledge.

The first is to restrict yourself to working within a single business domain as a consultant, so that you do gain a deep working knowledge of the business domain. As an example, a number of the IT consulting organisations that I've worked for have specialised in the investment banking industry, with consultants moving from bank to bank within that industry. This is certainly an effective way to ensure that consultants do understand the business domain, but it's not an approach that I personally like. Some of the consultants who I've worked with in the past have actually taken offence when offered a consulting role outside of investment banking. These people usually saw their deep domain knowledge as a key differentiator, a unique selling point, when compared to other consultants.

A look at my bookshelf will reveal that my interests lie far more with technology than any business domain. If I wanted to work for a bank, I'd work for a bank rather than a consulting organisation. As a result, I'm happy to regularly switch between business domains because this provides a degree of variety that you can't necessarily get from working in a single domain. I also find it interesting to see how other industries solve similar problems, and this itself leads to a number of opportunities for the cross-pollination of ideas. The downside, of course, is that my domain knowledge of any particular domain isn't as deep as somebody who works full-time in that domain.

To prevent this being an issue, I believe that there's a skill in being able to understand enough about a new domain to become proficient quickly. And that's really my approach. If you're a undertaking the software architecture role on a consulting basis, you need razor sharp analysis skills to understand the key parts of the domain without getting trapped in a cycle of analysis paralysis.

From developer to architect

As you may have realised by now, the line between being a software developer and being a software architect is actually somewhat blurred. Most developers don't wake up at the start of a new working week and declare themselves to be a software architect. I certainly didn't, and my route into software architecture was very much an evolutionary process. Having said that, there's a high probability that many software developers are already undertaking parts of the software architecture role, irrespective of their job title.

There are a number of different qualities that you need to look for in a software architect, with past experience often being a good gauge of the ability to undertake the role. Since the software architecture role is varied though, you need to look deeper to understand the level of involvement, influence, leadership and responsibility that has been demonstrated across a number of different areas.

Following on from my definition of the software architecture role, each of the parts of the role can, and should be, evaluated independently when assessing somebody's readiness for promotion to, or hiring into, a software architecture role. There's a huge difference between throwing some technologies together and hoping for the best, versus consciously selecting those technologies on the basis that they are going to work together. In other words, the way that people undertake the software architecture role varies wildly across the industry. For example:

1. **Architectural drivers**: it's better to capture and challenge a set of complicated quality attributes, versus ignoring them or assuming their existence.
2. **Designing software**: it's often more challenging to design a software system from scratch (greenfield), versus extending an existing one (brownfield).
3. **Technical risks**: it's safer to prove that your architecture will work, versus crossing your fingers and hoping for the best.
4. **Technical leadership**: it's more proactive to be continuously engaged with, and evolve the architecture, versus choosing to hand it off to an "implementation team" and let them deal with the problems.
5. **Quality assurance**: it's more responsible to decide upon some guidelines/principles/s-tandards and to continuously assure quality, versus doing nothing and assuming that the team will "just do a good job".

Much of this comes down to the difference between taking responsibility for a solution versus assuming that it's not your problem. The level of experience somebody has in the software architecture role can vary considerably, with more experienced software architects tending to be more proactive than their less experienced peers.

There's a big difference between contributing to the architecture of a software system, and being responsible for it; with a continuum of skills, knowledge and experience needed across the different activities that make up the software architecture role. Crossing the line between software developer and software architect is up to us. As individuals we need to understand the level of our own experience, and where we need to focus our efforts to increase it, seeking help where necessary.

Coaching, mentoring and apprenticeships

I've already mentioned the stonemasons in previous chapters, and it's interesting to see how *they* achieved the role of master mason (architect). From the Wikipedia page about stonemasonry:

> Medieval stonemasons' skills were in high demand, and members of the guild, gave rise to three classes of stonemasons: apprentices, journeymen, and master masons. Apprentices were indentured to their masters as the price for their training, journeymen had a higher level of skill and could go on journeys to assist their masters, and master masons were considered freemen who could travel as they wished to work on the projects of the patrons.

This mirrors my own personal experience of moving into a software architecture role. It was an *evolutionary process*. Like many people, I started my career writing code under the supervision of somebody else and gradually, as I gained more experience, started to take on larger and larger design tasks.

From what I've read, the master masons were dedicated to a single building site at any one point in time, and I think that we are similar in the software industry, working on a single project/product at a time. Whether you retain the same mentor across projects/products is a different question:

> A mason would have an apprentice working for him. When the mason moved on to a new job, the apprentice would move with him. When a mason felt that his apprentice had learned enough about the trade, he would be examined at a Mason's Lodge.

Unlike the medieval building industry though, the software development industry currently lacks an explicit way for people to progress from being junior developers through to software architects. We don't have a common apprenticeship model. Many organisations do have internal career development frameworks and competency frameworks, but many don't.

Coaching and mentoring is an overlooked activity on many software development teams, with many team members not getting the support that they need. This is especially true when teams are trying to move too fast, perhaps because of time/budgetary constraints. While technical leadership is about guiding the team as a whole, there are times when *individuals* need assistance. In addition to this, coaching and mentoring provides a way to enhance people's skills, and to help them improve their own careers. Sometimes this assistance is of a technical nature, and sometimes it's more about the softer skills.

From a technical perspective, I think that the people undertaking the software architecture role have a duty to help out with the coaching and mentoring. Why? Most software architects that I know have got to where they are because they have a great deal of experience in one or more technical areas, and they are usually more than capable of sharing some of this experience to help other people out. An apprenticeship model is how the master builders of times past kept their craft alive, and we should consider doing the same.

The sad thing about our industry is that many developers are being forced into non-technical management positions in order to "progress" their careers up the corporate ladder. Ironically, it's often the best and most senior technical people who are forced away, robbing software teams of their most valued technical leads, architects and mentors; exactly the sort of people who are capable of mentoring. Filling this gap tomorrow are the developers of today. In some cases, teams have already lost their most senior technical people, adding more work to the remainder of the team who are already struggling to balance all of the usual project constraints along with the pressures introduced by whatever is currently fashionable in the IT industry.

Many teams appreciate that they should be striving for improvement, but lack the time or the incentive to do it. To improve, software teams need some time away from the daily grind to reflect, but they also need to retain a focus on *all* aspects of the software development process. It's really easy to get caught up in the hype of the industry, but it's worth asking whether this is more important than ensuring you have a good pragmatic grounding.

Where are the software architects of tomorrow?

Experience of coding is relatively easy to pick up, and there are plenty of ways to practice this skill. Designing something from scratch that will be implemented by a team isn't something that you find many teams teaching or practicing though. With many technical mentors disappearing thanks to the typical corporate career ladder, where do developers gain this experience?

Herein lies a big problem for many organisations though - there aren't enough software architects to go around. I'm often asked questions from software architects who, as a part of their role, are expected to provide assistance across a *number* of different teams. Performing a software architecture role across a number of teams is not an effective way to work, because you spread yourself too thinly and end up providing superficial assistance instead of real help. Typically this situation occurs when there is a centralised group of architects (e.g. in a "Central Architecture Group") who are treated as shared resources.

We spend a lot of time talking about how to improve the technical practices associated with building software; writing code, testing, deployment, etc. And that makes a lot of sense because the end-goal here is delivering benefit to people through software, and working software is key. But we shouldn't forget that there are some other aspects of the software development process that few people genuinely have experience with. Think about how *you* would answer the following questions.

1. When did you last code?
    - *Earlier today, I'm a software developer so it's part of the job.*
2. When did you last refactor?
    - *I'm always looking to make my code the best I can, and that includes refactoring if necessary. Extract method, rename, pull up, push down … I know all that stuff.*
3. When did you last test your code?
    - *We test continuously by writing automated tests either before, during or after we write any production code. We use a mix of unit, integration and acceptance testing.*
4. When did you last design something?
    - *I do it all the time, it's a part of my job as a software developer. I need to think about how something will work before coding it, whether that's by sketching out a diagram or using TDD.*
5. When did you last design a software system from scratch? I mean, take a set of vague requirements and genuinely create something from nothing?
    - *Well, there's not much opportunity on my current project, but I have an open source project that I work on in my spare time. It's only for my own use though.*
6. When did you last design a software system from scratch that would be implemented by a team of people.
    - *Umm, that's not something I get to do.*

Let's face it, most software developers don't get to take a blank sheet of paper and design software from scratch all that frequently, regardless of whether it's a solo or collaborative exercise. So where are the software architects of tomorrow going to come from? Hopefully *you* are part of the answer.

Tips for new software architects

Like any leadership role, the software architecture role can be challenging for a number of reasons, and you might not get it right first time. Here are some tips for those of you who are new to the role:

- **Understand the role**: As an industry, we generally understand what software developers, testers, product owners, scrum masters, etc do on a daily basis. The software architecture role is somewhat more ambiguous, especially in the years since the "Manifesto for Agile Software Development" was created. Ensure that you understand the role, and specifically what your day-to-day responsibilities are.

- **Find a mentor**: If possible, try to find a mentor who can help you transition into, or perform, the software architecture role. Ideally this would be somebody in the organisation that you work for, although don't rule out finding a mentor outside of your organisational boundary, such as somebody you've worked with in the past.
- **Seek advice where needed**: As smart as we think we might be, we won't, and can't, know everything, especially given the heterogeneous nature of most modern software systems. Also, the rate of change in the software development industry is immense, and it's impossible for any one person to understand everything that's happening. Instead, collaboration is key. Find somebody that *does* understand the things you don't, and work with them closely. Nothing says that the software architecture role can't be shared. Often appreciating the gaps in your own knowledge is the first step to creating a more collaborative working environment. Pair programming has benefits, so why not pair architecting? Don't try to pretend to know everything, and to seek advice from people inside and outside of your team where necessary.
- **Do understand your team**: Understanding your team (e.g. the team dynamics, any politics, individual strengths and weaknesses, etc) is a part of the leadership role. Without this knowledge, you'll find it harder to make the right decisions related to leading that team.
- **Don't ignore the team**: Don't ignore the feedback from the rest of the team, whether it's explicitly given to you, or presented a series of implicit murmurings. If you think that people have some thoughts or feelings about the architecture, especially if they are negative in nature, be sure to seek them out so that you can act on them one way or another.
- **Do exercise some control**: Be wary of the aspiration for a team to be "self organising". This can and does work for experienced teams, but I've seen far too many teams try this without having the prerequisite skills and experience. The result was a chaotic mess, with a very inconsistent looking codebase. In such situations, you need to exercise some control by introducing some boundaries, guidelines and principles for the team to follow. Ideally this should be a collaborative exercise to avoid you being seen as a dictator, although this might be needed in extreme circumstances.
- **Fix problems early**: When you do see something going awry (e.g. inconsistent ways to solve the same problem, architectural principles not being followed, etc), do try to fix these as early as possible. You need to be proactive here, as the problems won't go away on their own.
- **Do make some decisions**: Many people talk about making decisions at the "last responsible moment", and there's a fine line between this and not making any decisions at all. At times, you will need to make some assumptions and put a "stake in the sand", making a decision as a starting point.

- **Document the major decisions**: Back at the start of this book, we saw that architecture is about the significant decisions, where significance is measured by cost of change. With this in mind, when you do make decisions, consider recording them in your software architecture documentation, perhaps using something like lightweight architecture decision records.
- **Don't code all of the time**: I've already mentioned this, but be mindful about how much coding your are doing, and do reserve some time to proactively undertake the tasks associated with the software architecture role.
- **Communicate continuously**: On a similar note, be sure to communicate continuously, with people inside and outside of the team. Make sure that everybody knows what is happening, and regularly validate this by talking to people. Problems can start to occur when the team charges on with the work in silence.
- **Don't ignore the code**: Make sure that you keep a "pulse" on the code, especially if you're not actively involved in writing it. If you lose touch with the code that is being written, you'll be less able to determine whether it's fit for purpose, satisfies the architectural principles, etc.
- **Don't get stuck in the detail**: Since most good software architects are also good software developers, it's very tempting to spend more time on the coding aspects than we perhaps should. Make sure that you don't get stuck in the detail, and keep the bigger picture in mind (e.g. the architectural drivers) when making decisions.
- **Stay up to date with the software development industry**: Do stay up to date with new technologies, development trends, etc. You don't necessarily need to understand everything in detail, but you should at least have an awareness of what's current.
- **Leave your ego at the door**: Software development, and success, is a team game. It's not just about you. As somebody once told me; take responsibility for the failures, and celebrate success as a team.

III Architecting

This, the final part of the book, is about how everything else covered in the book fits into the day-to-day world of software development. We'll also answer the question of how much up front design you should do.

10. Managing technical risks

There is risk inherent with building any piece of software; whether you're building a completely new greenfield project or adding a new feature to an existing codebase. A risk is a possibility that something bad can happen, there being different types of risks, each with their own potential consequence.

I once did a few days consulting to review a multi-year software project that had a planned budget of nearly Â£1M. This isn't a huge project by any stretch of the imagination, but when I reviewed the list of risks that the project manager had put together (often called a "risk register"), I was rather shocked to see that only half a dozen risks had been identified. This is a *very* small number for a project that size. After all, risks come in all shapes and sizes, and can be associated with:

- **People**: Does the team have the appropriate number of people? Do they have the appropriate skills? Do the team members work well together? Will the leadership roles be undertaken appropriately? Can training or consultants be brought in to help address skill gaps? Will we be able to hire people with the same skills in the future, for maintenance purposes? Does the operations and support team have the appropriate skills to run and look after the software? How likely is it that a key person will leave the team/organisation?
- **Process**: Does the team understand how they are going to work? Is there a described process? Are there expected outputs and artefacts? Is there enough budget available, both now and in the future? Are there any outside events (e.g. major changes in business, mergers/acquisitions, legal or regulatory changes, etc) that could disrupt progress of the work?
- **Technology**: Will the proposed architecture be able to satisfy the key quality attributes? Will the technology work as expected? Is the technology stable enough to build upon? Etc…

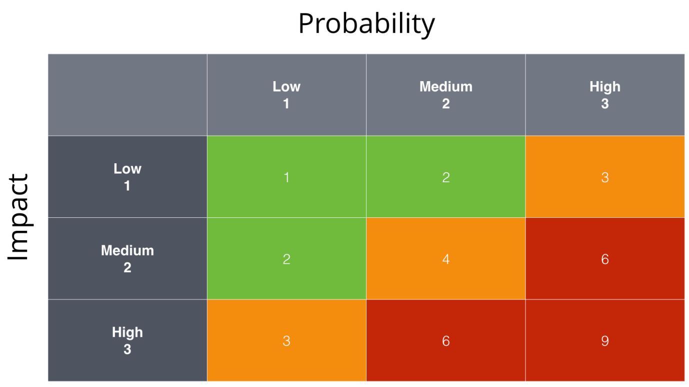
Quantifying and prioritising risks

Not all risks are equal though, with some being more important than others. For example, a risk related to your software project failing should be treated as higher priority than a risk related to the team encountering some short-term discomfort.

How do you quantify each of the risks, and assess their relative priorities then? The easiest way to do this is to map each risk onto a matrix, where you evaluate the *probability* of that risk happening against the negative *impact* of it happening.

- **Probability**: How likely is it that the risk will happen? Do you think that the chance is remote, or very likely to happen?
- **Impact**: What is the negative impact if the risk does occur? Is there general discomfort for the team, or is it back to the drawing board for some significant rework?

Both probability and impact could be quantified as low/medium/high or simply as a numeric value (1-3, 1-5, etc). The following diagram illustrates what a typical risk matrix might look like.


Probability
Impact

	Low 1	Medium 2	High 3
Low 1	1	2	3
Medium 2	2	4	6
High 3	3	6	9

A probability/impact matrix for quantifying risk

Prioritising risks is then about ranking each risk according to their score, which can be found by multiplying the numbers in the risk matrix together. A risk with a low probability and a low impact can be treated as a low priority risk. Conversely, a risk with a high probability and a high impact needs to be given a high priority. As indicated by the colour coding:

- **Green**: a score of 1 or 2; the risk is low priority.
- **Amber**: a score of 3 or 4; the risk is medium priority.
- **Red**: a score of 6 or 9; the risk is high priority.

There are other ways to quantify risks too, and I've seen more formal risk evaluation approaches used by teams building health and safety-critical software systems. I've also seen organisations use information security classification levels (e.g. public, restricted, confidential, top secret, etc) as the basis for understanding the risk associated with specific pieces of data being exposed by a security breach.

It's often difficult to prioritise which risks you should take care of and, if you get it wrong, you'll put the risk mitigation effort in the wrong place. Quantifying risks provides you with a way to focus on those risks that are most likely to cause your software project to fail, or you to be fired.

Identifying risks

Now that we understand how to quantify risks, we need to step back and look at how to identify them in the first place. One of the worst ways to identify risks is to let a single person do this on their own. Unfortunately this tends to be the norm, and in the past I've seen *non-technical* project managers take on this task too. The problem with asking a single person to identify risks is that you only get a view based upon *their* perspective, knowledge and experience. As with software development estimates, an individual's perceptions of risk will be based upon their own experience, and therefore subjective. For example, if you're planning on using a new technology, the choice of that technology may or may not be identified as a risk depending on whether the individual identifying the risks has used it in the past.

A better approach to risk identification is to make it a collaborative activity, and something that the whole team can participate in. This is essentially what we see with software estimates, when teams use techniques such as Planning Poker or its predecessor, Wideband Delphi. One approach is to run something called a pre-mortem, where you imagine that your project has failed, and you discuss/brainstorm the reasons as to why it failed, and ultimately what caused that failure.

Risk-storming

I have a technique that I created called "risk-storming", which is a quick, fun, collaborative, and visual technique for identifying risk that the whole team (architects, developers, testers, project managers, operational staff, etc) can take part in. There are 4 steps.

Step 1. Draw some architecture diagrams

The first step is to draw some architecture diagrams on whiteboards or large sheets of flip chart paper. Ideally this should be a collection of diagrams at different levels of abstraction, because each diagram (and therefore level of detail) will allow you to highlight different risks across your architecture. These diagrams should show what you are planning to build, or planning to change.

Step 2. Identify the risks individually

Since risks are subjective, step 2 is about asking everybody to look at the architecture diagrams and to *individually* write down the risks that they can identify, one per sticky note. Each risk should be quantified according to the probability and impact. Ideally, use different colour sticky notes to represent the different risk priorities (e.g. pink for high priority risks, yellow for medium, green for low).

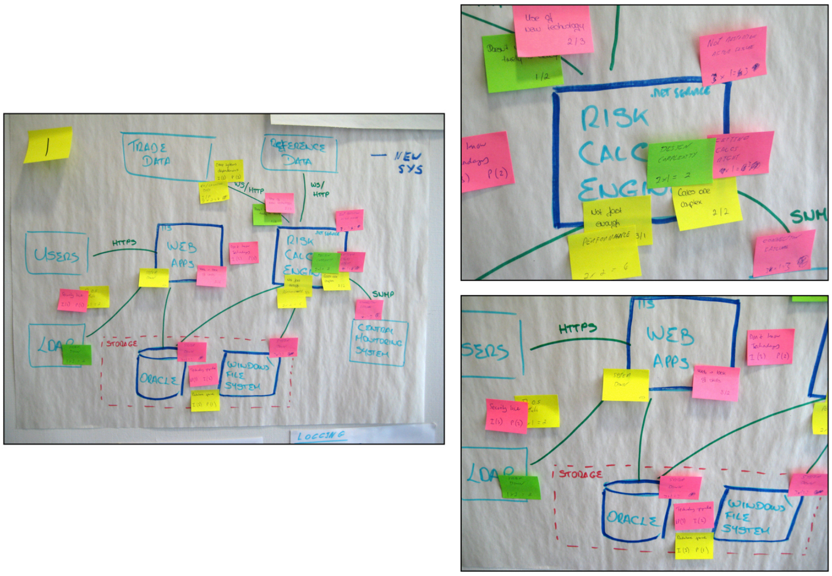
You can timebox this part of the exercise to 5-10 minutes to ensure that it doesn't drag on, and this step should be done in silence, with everybody keeping their sticky notes hidden from view so they don't influence what other people are thinking. Here are some examples of the risks to look for:

- Data formats from third-party systems change unexpectedly.
- External systems become unavailable.
- Components run too slowly.
- Components don't scale.
- Key components crash.
- Single points of failure.
- Data becomes corrupted.
- Infrastructure fails.
- Disks fill up.
- New technology doesn't work as expected.
- New technology is too complex to work with.
- The team lacks experience of the technology.
- Off-the-shelf products and frameworks don't work as advertised.
- etc

The goal of risk-storming is to seek input from as many people as possible so that you can take advantage of the collective experience of the team. If you're planning on using a new technology, hopefully somebody on the team will identify that there is a risk associated with doing this. Also, one person might quantify the risk of using that new technology relatively highly, whereas others might not feel the same if they've used that same technology before. Asking people to identify risks individually allows everybody to contribute to the risk identification process, so that you'll gain a better view of the risks perceived by the team rather than only from those designing the software or leading the team.

Step 3. Converge the risks on the diagrams

Next, ask everybody to place their sticky notes onto the architecture diagrams, sticking them in close proximity to the area where the risk has been identified. For example, if you identify a risk that one of your components will run too slowly, put the sticky note over the top of that component on the most appropriate architecture diagram.



Converge the risks on the diagrams

This part of the technique is very visual and, once complete, lets you see at a glance where the highest areas of risk are. If several people have identified similar risks you'll get a clustering of sticky notes on top of the diagrams as people's ideas converge.

Step 4. Prioritise the risks

Now you can take each sticky note (or cluster of sticky notes) and agree on how you will collectively quantify the risk that has been identified.

- **Individual sticky notes**: Ask the person who identified the risk what their reason was and collectively agree on the probability and impact. After discussion, if either the probability or impact turns out to be "none", take the sticky note off of the architecture diagram but don't throw it away just yet.

- **Clusters of sticky notes**: If the probability and impact are the same on each sticky note, you're done. If they aren't, you'll need to collectively agree on how to quantify the risk in the same way that you agree upon estimates during a Planning Poker or Wideband Delphi session. Look at the outliers, and understand the rationale behind people quantifying the risk accordingly.

When to use risk-storming

Risk-storming is a quick technique that provides a collaborative way to identify and visualise risks. As an aside, this technique can be used for anything that you can visualise; from enterprise architectures through to business processes and workflows. It can be used at the start of a software development project (when you're coming up with the initial software architecture) or throughout, during iteration planning sessions or retrospectives.

Just make sure that you keep a log of the risks that are identified, including those that you later agree have a probability or impact of "none". Additionally, why not keep the architecture diagrams with the sticky notes on the wall of your project room so that everybody can see this additional layer of information. Identifying risks is essential in preventing project failure, and it doesn't need to be a chore if you get the whole team involved.

Mitigating risks

Identifying the risks associated with your software architecture is an essential exercise, but you also need to come up with mitigation strategies, either to prevent the risks from happening in the first place, or to take corrective action if the risk does occur. Since the risks are now prioritised, you can focus on the highest priority ones first.

There are a number of mitigation strategies the are applicable depending upon the type of risk, including:

1. **Education**: Training the team, restructuring it or hiring new team members in areas where you lack experience (e.g. with new technology).
2. **Prototypes**: Creating prototypes (or proof of concepts, running concrete experiments, etc) where they are needed to mitigate technical risks by proving that something does or doesn't work. Since risk-storming is a visual technique, it lets you easily see the stripes through your software system that you should perhaps look at in more detail with prototypes.

3. **Re-work**: Changing your software architecture to remove or reduce the probability/impact of identified risks (e.g. removing single points of failure, adding a cache to protect from third-party system outages, etc). If you do decide to change your architecture, you can re-run the risk-storming exercise to check whether the change has had the desired effect, and not introduced too many other high priority risks.

Risk ownership

As a final point related to risks, in my experience, the “risk register” (if there is one) is often owned by the lone non-technical project manager. Does this make sense? Do they understand the technical risks? Do they really *care* about the technical risks?

As I hinted in part II of this book, a better approach is to assign ownership of technical risks to the software architecture role. By all means keep a central risk register, but do ensure that somebody on the team is actively looking after the technical risks. And, of course, sharing the software architecture role amongst the team paves the way for collective ownership of the risks too.

11. Software architecture in the delivery process

This, the final chapter, looks at how you introduce software architecture into the software delivery process, and seeks to answer the question of how much up front design you should do. First we need to look at some of the perceived conflicts between modern software development (i.e. "agile") and architecture.

The conflict between agile and architecture

The words "agile" and "architecture" are often seen as mutually exclusive, but the real world often tells a different story. Some software teams unfortunately see architecture as an unnecessary evil, whereas others have reached the conclusion that they do need to think about architecture once again. I regularly receive e-mails from organisations who have embarked on an agile transformation, yet they're still struggling with the technical aspects of software delivery. Typically these organisations have misinterpreted what agile is all about, and have thrown away some of the practices you might typically associate with software architecture; such as up front design, modelling and documentation.

As we learnt back in chapter 1, architecture can be summarised as being about structure and vision, with a key part of the process focussed on understanding and making the **significant design decisions**. Unless you're running the leanest of startups and you genuinely don't know which direction you're heading in, even the most agile of software teams will be building software that has some architectural concerns, and is subject to some complicated architectural drivers. Many of these things really should be thought about up front, rather than being deferred.

Agile software projects therefore do need "architecture", but this seems to contradict how agile has been evangelised across the industry since the creation of the "Manifesto for Agile Software Development" in 2001. Put simply, there is no conflict between agile and architecture because agile projects need to think about architecture. So, where is the perceived conflict then?

Conflict 1: Team structure

The first conflict between architecture and agile software development approaches is related to team structure. Traditional approaches to software architecture usually result in a dedicated software architect, triggering thoughts of ivory tower dictators who are a long way removed from the process of building software, probably because they don't write code any more. This unfortunate stereotype of "solution architects" delivering large design documents to the development team before running away to cause havoc elsewhere has unfortunately resulted in a backlash against having a dedicated architect on a software development team.

One of the things that agile and lean approaches have taught us is that software development teams should strive towards reducing the amount of overhead associated with communication via document hand-offs. It's rightly about increasing collaboration and reducing waste, with organisations now often preferring to create small teams of generalising specialists who can turn their hand to almost any task, rather than teams of specialists who communicate via the exchange of documents or models. Indeed, because of the way in which agile approaches have been evangelised, there is often a perception that agile teams *must* consist of cross-discipline team members, and simply left to self-organise. The result? Many agile teams will tell you that they "don't need architects", or that "everybody is an architect". As we saw in part II of this book, distributing the software role across all team members doesn't always work, and different approaches to technical leadership are needed depending on the team you have.

Conflict 2: Process and outputs

The second conflict is between the process of delivering software and the desired outputs of agile approaches versus those of big up front design, which is what people usually refer to when they talk about architecture. One of the key goals of agile approaches is to deliver customer value, frequently and in small chunks. It's about moving fast, getting feedback, embracing change and continually improving. Software architecture has traditionally been associated with big design up front and waterfall-style delivery, where a team would ensure that every last element of the software design was considered before any code was written. After all, the goal of big design up front is to settle upon an understanding of everything that needs to be delivered before putting a blueprint (and usually a plan) in place.

In 2001, the "Manifesto for Agile Software Development" suggested that we should value "responding to change over following a plan". Unfortunately, when taken at face value, this is often misinterpreted to mean that we shouldn't plan. In trying to avoid big up front design, teams often do *no* design up front though; instead using terms like "emergent design", "evolutionary architecture" or "last responsible moment" to justify their approach. I've even heard teams claim that their adoption of test-driven development (TDD) negates the need for "architecture". Unfortunately these are often the same teams that get trapped in a constant refactoring cycle at some point in the future too, because the overall structure of their code is a mess and doesn't properly support the architectural drivers.

The reality of architecture in an agile world

The net result, and I've seen this first hand, is that some software development teams have flipped from doing big design up front to doing no design up front. It's at this point that I again like to quote Dave Thomas, who said the following in a conference talk a number of years ago:

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

Clearly both extremes are foolish, and there's a sweet spot somewhere in between, which is relatively easy to discover if you're willing to consider that up front design is not necessarily about creating a perfect end-state. Instead, think about up front design as being about creating a **starting point** and **setting a direction** for the team, rather than creating a plan that needs to be followed.

Agile and architecture aren't in conflict. Rather than blindly following what others say, software teams need to cut through the hype and think for themselves, understanding the technical leadership style and quantity of up front design that they need given their own unique context.

Technical vs functional design

At this point, it's worth mentioning that "design" is a relatively overloaded term. So, by "design", I'm referring to the technical design aspects of the software (i.e. how the software will be built) rather than the functional (product) design aspects (i.e. what the software will do). My reason for doing this is that any up front design exercise will be undertaken in a specific context, and that context will help to determine what type of design, and how much design, is necessary.

For example, this might be an unpopular opinion, but there are actually many contexts where the functional and non-functional requirements of the software are known up front. You'll see this frequently in enterprise software development, where teams are tasked with building replacement systems, or building software that fulfils a specific purpose, such as integrating two other software systems. From my own experience, I've worked with many organisations where we've been asked to build software that specifically meets a set of functional requirements to support a new business process, or to replace an aging legacy system. Often the quality attributes were understood up front too; especially those related to performance (e.g. message latency), scalability (e.g. bank trading volumes) and security.

On the flipside, there are many contexts where the functional and non-functional requirements are not well-known. You'll see this with product companies, and particularly startups. In these contexts, rather than spending lots of up front time designing the functionality that should be supported, and understanding how it will be used, organisations will instead be much more experimental by delivering features quicky, iterating on them, and capturing feedback.

Irrespective of whether the functional requirements are well known or not, there still exists a need to undertake some basic technical design, especially relating to those things that might be deemed as significant and hard to change later; such as the choice of programming language, key technologies/libraries/frameworks, and the overall structure of the software system (e.g. monolithic deployment unit vs a collection of distributed microservices). If the functional requirements are not well-known, and you are planning to deliver features and iterate rapidly in order to experiment with the functionality, you still need a well-designed codebase to support this. The ability for a codebase to respond to change is related to the structure of that codebase. A "good", well-structured, codebase will allow you to move faster than repeatedly trying to change a "big ball of mud".

An up front design phase that focusses on the *technical design aspects*, even lasting just a few hours or days, can therefore help to create a starting point, align the team members and set an overall direction. This often missed step can add a tremendous amount of value to a team by encouraging them to understand what they are going to build and whether it is going to work. Although this may sound straightforward, and even obvious, the perceived conflict between agile approaches and architecture frequently cloud people's view of up front design.

Software architecture provides boundaries

One of the recurring questions I get asked whenever I talk to teams about software architecture is how it relates to techniques such as TDD, BDD, DDD, RDD, etc. The question really relates to whether xDD is a substitute for "software architecture", particularly within "agile environments". The short answer is no. The slightly longer answer is that the process of thinking about software architecture is about putting some boundaries in place, inside which you can build your software using whatever xDD and agile practices you like.

For me, the "why?" is simple. As we saw in chapter 2, you need to think about the architectural drivers (the things that play a huge part in influencing the resulting software architecture), including:

- **Functional requirements**: Requirements drive architecture. You need to know (even vaguely) what you're building, irrespective of how you capture and record those requirements (i.e. user stories, use cases, requirements specifications, acceptance tests, etc).
- **Quality attributes**: The non-functional requirements (e.g. performance, scalability, security, etc) are usually technical in nature and are hard to retrofit. They ideally need to be baked into the initial design, and ignoring these qualities will lead you to a software system that is either over- or under-engineered.
- **Constraints**: The real-world usually has constraints; ranging from approved technology lists, prescribed integration standards, target deployment environment, size of team, etc. Again, not considering these could cause you to deliver a software system that fights against, rather than complements your environment, adding unnecessary friction.
- **Principles**: These are the things that you want to adopt in an attempt to provide consistency and clarity to the software. From a design perspective, this includes things like your decomposition strategy (e.g. layers and components in a monolithic deployment unit vs microservices), separation of concerns, architectural patterns, etc. Explicitly outlining a starting set of principles is essential so that the team building the software starts out heading in the same direction.

Not considering these inputs, and doing a sufficient quantity of up front design, in many cases, leads to chaotic teams that lack an appropriate amount of technical leadership. The result? Software systems that look like big balls of mud and/or don't satisfy the key architectural drivers.

Architecture is about the things that are hard or costly to change. It's about the big or significant decisions, the sort of decisions that you can't easily refactor in an afternoon. This includes, for example, the core technology choices, the overall high-level structure (the big picture) and an understanding of how you're going to solve any complicated/risky/significant problems. Software architecture is important.

Big up front design typically covers these concerns but it's historically tended to go much further too, often unnecessarily so. The trick here is to differentiate what's important from what's not. Defining a high-level structure to create a technical vision is important. Drawing a countless number of detailed UML class diagrams before you start writing the code most likely isn't. Understanding how you're going to solve a tricky performance requirement is important. Understanding the length of every database column most likely isn't.

How much up front design should you do?

All of this raises a rather obvious question - how much up front design should you do? Different teams all have their own perspective on this, and I've found teams (and the individuals within those teams) to be very polarised as to when they should do design, and how much they should do. One of the key reasons for the difference in how teams think about up front design can be found in how teams work, and specifically what sort of development methodology they are following, or have followed in the past. If you compare the common software development approaches on account of how much up front design they advocate, or are perceived to advocate, you'd have something like this:

- **Big up front design**: At one end of the scale you have "waterfall" that, in it's typical form, suggests big design up front where everything must be decided, reviewed and signed-off before a line of code is written.
- **No up front design**: At the other end you have the agile approaches that, on the face of it, shy away from doing architecture.

At this point it's worth saying that this isn't actually true. Agile approaches don't say "don't do architecture", just as they don't say "don't produce any documentation". From a process perspective, agile approaches are about sufficiency, moving fast, embracing change, getting feedback, delivering value as early as possible, and continuous improvement. But since agile approaches and their evangelists don't put much emphasis on the architectural aspects of software development, many people have misinterpreted this to mean "agile says don't do any architecture".

More commonly, agile teams choose to spread the design work out across the delivery rather than doing it all up front. There are several names for this, including "evolutionary architecture" and "emergent design". Depending on the size and complexity of the software system along with the experience and maturity of the team, this could unfortunately end up as "foolishly hoping for the best".

Evolutionary architecture can definitely work, but it's not as straightforward as you would think. I've seen a number of teams who have tried this, where they only ever do enough design to satisfy the user stories/features/etc that are planned to be delivered in the current iteration. Typically teams will have a planning meeting to decide which user stories/features/etc are planned for the iteration, with some view of "priority" being used to help determine this. User stories and features can be prioritised in a number of different ways, for both business and technical reasons. And herein lies the problem.

If the business stakeholders continually force the development team to implement features in an order that makes most sense to them (i.e. those features that provide the most business value first), the development team can easily find themselves in a situation whereby they are constantly forced to move fast and focus on features, so the underlying architecture and code structure suffers as a result. Or worse, they end up discovering a complicated non-functional requirement or environmental constraint that needs to be retrofitted into their existing codebase. The manifests itself as a team that is constantly asking permission to pause so they can have a "refactoring sprint", to clean up the technical debt that has been accumulated. On the flip side, a team who focusses too much on the technical side of things will be slow to deliver business value.

With this in mind, **some up front design** sounds like a better answer than either of the extremes, which is what approaches like the Rational Unified Process (RUP), Disciplined Agile Delivery (DAD) and DSDM Atern encourage. These are flexible process frameworks that can be implemented by taking all or part of them. Although many RUP implementations have historically been heavyweight monsters that have more in common with waterfall approaches, it *can* be scaled down to be relatively nimble. DAD is basically a trimmed down version of RUP, and DSDM Atern is a similar iterative and incremental method that is also influenced by the agile movement. All three are *risk-driven methodologies* that say, "gather the majority of the key requirements at a high level, get the risky stuff out of the way, then iterate and increment". DSDM Atern even uses the term "firm foundations" to describe this. Done right, these methods can lead to a nice balance of up front design and evolutionary architecture.

"Just enough"

My approach to up front design is that you need to do "just enough". If you say this to people they either think it's an inspirational breath of fresh air that fits with all of their existing beliefs, or they think it's a complete cop out! "Just enough" works as a guideline but it's vague and doesn't do much to help people assess how much is enough. Based upon my definition of architecture, you could say that you need to do just enough up front design to give you *structure and vision*. In other words, do enough up front design so that you know what your goal is, and how you're going to achieve it. This is a better guideline, but it still doesn't provide any concrete advice.

It turns out that while "just enough" up front design is hard to quantify, many people have strong opinions on, and past experience with, "too little" or "too much" up front design. Here's a summary of what these two extremes mean to some of the software developers I've met over the years.

How much up front design is too little?

- No understanding of what and where the system boundary is.
- No common understanding of "the big picture" within the team.
- An inability to communicate the overall vision.
- Team members aren't clear or comfortable with what they need to do.
- No thought about non-functional requirements/quality attributes.
- No thought about how the constraints of the real-world environment affect the software (e.g. deployment environment, cost, etc).
- No thoughts on key areas of risk; such as complicated quality attributes, external system interfaces/integration points, etc.
- The significant problems and/or their answers haven't been identified.
- No thought or concensus on separation of concerns, appropriate levels of abstraction, layering, modularity, modifiability, flex points, etc.
- Inconsistent approaches to solving problems.
- A lack of control and guidance for the team.
- Significant change to the architecture during the delivery lifecycle that could have been anticipated with a little thought up front.
- Too many design alternatives and options, often with team members disagreeing on the solution or way forward.
- Uncertainty over whether the design will work (e.g. no prototyping was performed as a part of the design process).
- A lack of concrete technology choices made (i.e. unnecessary deferral of decisions).

How much up front design is too much?

- Too much information (i.e. long documents and/or information overload).
- A design that is too detailed at too many levels of abstraction.
- Too many diagrams.
- Writing code or pseudo-code in documentation.
- An architecture that is too rigid, with no flexibility.

- All decisions at all levels of abstraction have been made.
- Class/code level design with numerous sequence diagrams showing all possible interactions.
- Detailed entity relationship models and database designs (e.g. tables, views, stored procedures and indexes).
- Analysis paralysis and a team that is stuck focussing on minor details.
- Coding becomes a simple transformation of design artefacts to code, which is boring and demotivating for the team.
- An unbounded "design phase" (i.e. time and/or budget).
- The deadline has been reached without any coding.

How much is "just enough"?

It's easy to identify with many of the answers above, but "just enough" sits in that grey area somewhere between the two extremes. The key to figuring out how much up front design to do is to remember that, as Grady Booch says, "architecture represents the significant decisions, where significance is measured by cost of change". In other words, it's the set of decisions that are expensive to modify, and the things that you really do need to get right as early as possible. For example, qualities such as high performance, high scalability, high security and high availability generally need to be baked into the foundations early on because they are hard to retrofit into an existing codebase. The significant decisions also include the things that you can't easily refactor in an afternoon; such as the overall structure, core technology choices, "architectural" patterns, core frameworks and so on.

In reality, these are the things with a higher than normal risk of consequences if you don't get them right. As we saw with risk in the previous chapter, it's worth remembering that the significant elements are often subjective, and can vary depending on the experience of the team members.

Firm foundations

What you have here then is an approach to software development that lets you focus on what's risky in order to build sufficient foundations to move forward with. The identification of architecturally significant elements and their corresponding risks is something that should be applied to *all* software projects, regardless of the delivery methodology and/or process being followed. Some agile projects already do this by introducing a "sprint zero", although some agile evangelists will say that "you're doing it wrong" if you need to introduce an "architecture sprint" that "doesn't deliver value to users". I say that you need to do whatever works for you based upon your own context. That's actually what agile is about anyway.

Although all of this provides some guidance, the answer to "how much is just enough?" needs one of those "it depends" type answers because all software teams are different. Some teams will be more experienced, some teams will need more guidance, some teams will continually work together, some teams will rotate and change frequently, some software systems will have a large amount of essential complexity, etc.

How much up front design do you need to do then? I would recommend that you do enough up front design in order to do the following, which applies whether the software architecture role is being performed by a single person or shared amongst the team.

1. **Structure**
    - **What**: Understand the significant structural elements and how they fit together, based upon the architectural drivers.
    - **How**: Perform design and decomposition down to the level of containers (for a microservices architecture) or components (for a more traditional monolithic architecture). Software Architecture for Developers: Volume 2 and c4model.com describes my C4 model, and what I mean by "containers" and "components". These are essentially your architectural/structural building blocks though.
2. **Vision**
    - **What**: Create and communicate a vision for the team to work with.
    - **How**: Visualise your software architecture using a collection of diagrams at varying levels of abstraction. I use my C4 model, which is a way to create hierarchical diagrams of your software architecture, at different levels of abstraction.
3. **Risks**
    - **What**: Identify and mitigate the highest priority risks.
    - **How**: Perform the risk-storming exercise (or a pre-mortem), along with some concrete experiments to help de-risk the delivery process.

The other way to look at this is that you should stop doing up front design when:

- You understand the key architectural drivers.
- You understand the context and scope of what you're designing/building.
- You understand the significant design decisions (i.e. technology, modularity, etc).
- You have a way to communicate your vision to other people.
- You are confident that your design satisfies the key architectural drivers.

- You have identified, and are comfortable with, the risks associated with building the software.

This minimal set of software architecture practices will help provide you with firm foundations that underpin the rest of the software delivery, both in terms of the product being built, and the team that is building it. *Some* design usually does need to be done up front, but some doesn't, and can naturally evolve. Deciding where the line sits between mandatory and evolutionary design is the key.

Contextualising just enough up front design

In reality, the "how much up front design is enough?" question must be answered by *you* and here's my advice ... go and practice architecting a software system. Find or create a small-medium size software project scenario, and draft a very short set of high-level requirements (functional and non-functional) to describe it. This could be an existing system that you've worked on or something new and unrelated to your domain such as the financial risk system that I use during my training courses.

With this in place, ask groups of 2-3 people to come up with a solution by choosing some technology, doing some design and drawing some diagrams to communicate the vision. Timebox the activity (e.g. 90 minutes) and then hold an open review session where the following types of questions are asked about each of the solutions:

- Will the architecture work? If not, why not?
- Have all of the key risks been identified?
- Is the architecture too simple? Is it too complex?
- Has the architecture been communicated effectively?
- What do people like about the diagrams? What can be improved?
- Is there too much detail? Is there enough detail?
- Could you give this to *your* team as a starting point?
- Is there too much control? Is there not enough guidance?
- Are you happy with the level of technology decisions that have been made or deferred?

Think of this exercise as an architectural kata except that you perform a review that focusses additionally on the process you went through and the outputs rather than just the architecture itself. Capture your findings and try to distill them into a set of guidelines for how to approach the software design process in the future. Agree upon and include examples of how much detail to go down into when doing up front design, agree on the diagrams that are useful to draw, determine the common constraints within your own environment, etc. If possible, run the exercise again with the guidelines in mind to see how it changes things. One day is typically enough time to run through this exercise with a couple of design/communicate/review cycles.

No two software teams are the same. Setting aside a day to practice the software design process within your own environment will provide you with a consistent starting point for tackling the process in the future, helping you contextualise exactly what “just enough” up front design means to you and your team. An additional benefit of practicing the software design process is that it’s a great way to coach and mentor others.

Introducing software architecture

A little software architecture discipline has a huge potential to improve the success of software teams because it’s all about technical leadership. With this in mind, the final question that we need to address is how to get software teams to adopt a “just enough” approach to software architecture, to ensure they build well-structured software systems that satisfy the goals, particularly with respect to any complicated non-functional requirements and constraints. Often, this question becomes how to we *reintroduce* software architecture back into the way that software teams work.

In my view, the big problem that software architecture has as a discipline is that it’s competing with all of the shiny new things created in the software industry on a seemingly daily basis. I’ve met thousands of software developers around the globe and, in my experience, there’s a large number of them that don’t think about software architecture as much as they should. Despite the volume of educational material out there, teams lack knowledge about what software architecture really is, why it’s important, and how to do it.

People have limited time and energy for learning, but a lack of time isn’t often the reason that teams don’t understand what software architecture is all about. When I was moving into my early software architecture roles, I, like others I’ve spoken to, struggled to understand how much of what I read about in the existing software architecture books related to what I should doing on a daily basis. This lack of understanding is made worse because most software developers don’t get to practice architecting software on a regular basis. How many software systems have you architected during your own career?

Simply saying that all software teams need to think about software architecture isn’t enough to make it happen though. So, how *do* we get software teams to reintroduce software architecture?

Software architecture needs to be accessible

As experienced practitioners, *we* have a duty to educate others, but we do need to take it one step at a time. We need to remember that many people are being introduced to software architecture with potentially no knowledge of the related research that has been conducted in the past. Think about the terminology that you see and hear in relation to software architecture. How would you explain to a typical software developer what a "logical view" is? When we refer to the "physical view", is this about the code or the physical infrastructure? Everybody on the development team needs to understand the essence of software architecture, and the consequences of not thinking about it, before we start talking about things like architecture description languages and evaluation methods. Information about software architecture needs to be accessible and grounded in reality. For this reason, I'd recommend the following software architecture books if you're a software developer:

- Software Systems Architecture, 2nd edition (Nick Rozanski and Eoin Woods)
- Just Enough Software Architecture (George Fairbanks)
- Design It! (Michael Keeling)

This may seem an odd thing to say, but the people who manage software teams also need to understand the essence of software architecture, and why it's a necessary discipline. Some of the teams I've worked with over the years have been told by their management to "stop doing software architecture and get on with the coding". In many cases, the reason behind this is a misunderstanding that all up front design activities need to be dropped when adopting agile approaches. Such software development teams are usually put under immense pressure to deliver, and some up front thinking usually helps rather than hinders. Here are some practical suggestions for introducing, or reintroducing, software architecture.

1. Educate people

Run some workshops where people can learn about and understand what software architecture is all about. This can be aimed at developers or non-developers, and it will help to make sure that everybody is talking the same language. At a minimum, you should look to cover:

- What software architecture is.
- Why software architecture is important.
- The practices you want to adopt, and the value they add.

2. Talk about architecture in retrospectives

If you have regular retrospectives to reflect on how your team is performing, include software architecture on the list of topics that you talk. If you don't think that enough consideration is being given to software architecture, perhaps because you're constantly refactoring the architecture of your software, or you're having issues satisfying some of the quality attributes, think about the software architecture practices that you can adopt to help. On the flip side, if you're spending too much time thinking about software architecture or up front design, perhaps it's time to look at the value of this work, and whether any practices can be dropped or substituted.

3. Definition of done

If you have a "definition of done" for work items, add software architecture to the list. This will help ensure that you explicitly consider the architectural implications of the work item, and conformance of the implementation with any desired architectural patterns/rules/principles.

4. Allocate the software architecture role to somebody

If you have a software team that doesn't think about software architecture, allocating the software architecture role to somebody appropriate on the team may kickstart this because you're explicitly giving ownership and responsibility for the software architecture to somebody. Allocating the role to more than one person does work with some teams, but I find it better that one person takes ownership initially, with a view to sharing it with others as the team gains more experience. Some teams dislike the term "software architect" and use the term "tech lead" or "architecture owner" instead. Whatever you call it, coaching and collaboration are key.

5. Architecture katas

Words alone are not enough and the skeptics need to see that architecture is not about big design up front. This is why I run short architecture katas where small teams collaboratively architect a software solution for a simple set of requirements, producing one or more diagrams to visualise and communicate their solutions to others. This allows people to experience that up front design doesn't necessarily mean designing everything to a very low level of abstraction, and it provides a way to practice communicating software architecture.

Making change happen

Here's a relatively common question from people that understand why software architecture is good, but don't know how to introduce it into their projects.

> "I understand the need for software architecture but our team just doesn't have the time to do it because we're so busy building our product. Having said that, we don't have consistent approaches to solving problems, etc. Our managers won't give us time to do architecture. If we're doing architecture, we're not coding. How do we introduce architecture?"

It's worth asking a few questions to understand the need for actively thinking about software architecture:

1. What problems is the lack of software architecture causing now?
2. What problems is the lack of software architecture likely to cause in the future?
3. Is there a risk that these problems will lead to more serious consequences (e.g. loss of reputation, business, customers, money, etc)?
4. Has something already gone wrong?

One of the things that I tell people new to the software architecture role is that they do need to dedicate some time to doing architecture work (the big picture stuff) but a balance needs to be struck between this and the regular day-to-day development activities. If you're coding all of the time then that big picture stuff doesn't get done. On the flip side, spending too much time on "software architecture" means that you don't ever get any coding done, and we all know that pretty diagrams are no use to end-users!

"How do we introduce software architecture?" is one of those questions that doesn't have a straightforward answer because it requires changes to the way that a software team works, and these can only really be made when you understand the full context of the team. On a more general note though, there are two ways that teams tend to change the way that they work.

1. **Reactively**: The majority of teams will only change the way that they work based upon bad things happening. In other words, they'll change if and only if there's a catalyst. This could be anything from a continuous string of failed system deployments or maybe something like a serious system failure. In these cases, the team knows something is wrong, probably because their management is giving them a hard time,

and they know that something needs to be done to fix the situation. This approach unfortunately appears to be in the majority across the software industry.
2. **Proactively**: Some teams proactively seek to improve the way that they work. Nothing bad might have happened yet, but they can see that there's room for improvement to prevent the sort of situations mentioned previously. These teams are, ironically, usually the better ones that don't *need* to change, but they do understand the benefits associated with striving for continuous improvement. You could say that these teams are truly agile.

Back to the original question; I find that some teams try to seek permission to spend some time doing the architecture stuff, but they don't get buy-in from their management. Perhaps their management doesn't clearly understand the benefits of doing it, or the consequences of *not* doing it. Either way, the team didn't achieve the desired result. Whenever I've been in this situation myself, I've either taken one of two approaches.

1. Present in a very clear and concise way what the current situation is and what the issues, risks and consequences are if behaviours aren't changed. Typically this is something that you present to key decision makers, project sponsors or management. Once they understand the risks, they can decide whether mitigating those risks is worth the effort required to change behaviours. This requires influencing skills and it can be a hard sell sometimes, particularly if you're new to a team that you think is dysfunctional!
2. Lead by example, by finding a problem and addressing it. This could include, for example, a lack of technical documentation, inconsistent approaches to solving problems, too many architectural layers, inconsistent component configuration, etc. Sometimes the initial seeds of change need to be put in place before everybody understands the benefits in return for the effort. A little like the reaction that occurs when most people see automated unit testing for the first time - you need to see it to understand it.

Each approach tends to favour different situations, and again it depends on a number of factors. You'll find it hard to change how a team works when the first approach is used, but the message is weak or the management doesn't think that mitigating the risks of not having any dedicated "architecture time" is worth the financial outlay. In this particular case, I would introduce software architecture through being proactive and leading by example. Simply find a problem (e.g. multiple approaches to dealing with configuration, no high-level documentation, a confusing component structure, etc) and just start to fix it. I'm not talking about downing tools and taking a few weeks out, because we all know that trying to sell a three month refactoring effort to your management is a tough proposition. I'm talking about baby steps, where you improve the situation by breaking the problem down and addressing it a piece at a time. Take a few minutes out from your day to focus on these sort of tasks. Before you know it, you've probably started to make a world of difference. As Grace Hopper once said; "it's easier to ask forgiveness than it is to get permission".

The essence of software architecture

As we've seen, the up front design process should be about understanding the significant decisions that influence the shape of a software system, so that teams understand what they are going to build, how they are going to build it (at a high-level, anyway) and whether what they've designed will have a good chance of actually working. In summary, up front design should be about stacking the odds of success in your favour.

Any software architecture practices adopted need to add real value, otherwise the team is simply wasting time and effort. My experience suggests that a little direction can go a long way. Only you can decide how much up front design is just enough, and only you can decide how best to lead the change that you want to see in your team. Good luck with your journey!

IV Appendices

12. Appendix A: Financial Risk System

Background

A global investment bank based in London, New York and Singapore trades (buys and sells) financial products with other banks (counterparties). When share prices on the stock markets move up or down, the bank either makes money or loses it. At the end of the working day, the bank needs to gain a view of how much risk they are exposed to (e.g. of losing money) by running some calculations on the data held about their trades. The bank has an existing Trade Data System (TDS) and Reference Data System (RDS) but need a new Risk System.

Trade Data System

The Trade Data System maintains a store of all trades made by the bank. It is already configured to generate a file-based XML export of trade data at the close of business (5pm) in New York. The export includes the following information for every trade made by the bank:

- Trade ID
- Date
- Current trade value in US dollars
- Counterparty ID

Reference Data System

The Reference Data System maintains all of the reference data needed by the bank. This includes information about counterparties; each of which represents an individual, a bank, etc. A file-based XML export is also available and includes basic information about each counterparty. A new organisation-wide reference data system is due for completion in the next 3 months, with the current system eventually being decommissioned.

Functional Requirements

The high-level functional requirements for the new Risk System are as follows.

1. Import trade data from the Trade Data System.
2. Import counterparty data from the Reference Data System.
3. Join the two sets of data together, enriching the trade data with information about the counterparty.
4. For each counterparty, calculate the risk that the bank is exposed to.
5. Generate a report that can be imported into Microsoft Excel containing the risk figures for all counterparties known by the bank.
6. Distribute the report to the business users before the start of the next trading day (9am) in Singapore.
7. Provide a way for a subset of the business users to configure and maintain the external parameters used by the risk calculations.

Non-functional Requirements

The non-functional requirements for the new Risk System are as follows.

Performance

- Risk reports must be generated before 9am the following business day in Singapore.

Scalability

- The system must be able to cope with trade volumes for the next 5 years.
- The Trade Data System export includes approximately 5000 trades now and it is anticipated that there will be an additional 10 trades per day.
- The Reference Data System counterparty export includes approximately 20,000 counterparties and growth will be negligible.
- There are 40-50 business users around the world that need access to the report.

Availability

- Risk reports should be available to users 24x7, but a small amount of downtime (less than 30 minutes per day) can be tolerated.

Failover

- Manual failover is sufficient for all system components, provided that the availability targets can be met.

Security

- This system must follow bank policy that states system access is restricted to authenticated and authorised users only.
- Reports must only be distributed to authorised users.
- Only a subset of the authorised users are permitted to modify the parameters used in the risk calculations.
- Although desirable, there are no single sign-on requirements (e.g. integration with Active Directory, LDAP, etc).
- All access to the system and reports will be within the confines of the bank's global network.

Audit

- The following events must be recorded in the system audit logs:
  - Report generation.
  - Modification of risk calculation parameters.
- It must be possible to understand the input data that was used in calculating risk.

Fault Tolerance and Resilience

- The system should take appropriate steps to recover from an error if possible, but all errors should be logged.
- Errors preventing a counterparty risk calculation being completed should be logged and the process should continue.

Internationalization and Localization

- All user interfaces will be presented in English only.
- All reports will be presented in English only.
- All trading values and risk figures will be presented in US dollars only.

Monitoring and Management

- A Simple Network Management Protocol (SNMP) trap should be sent to the bank's Central Monitoring Service in the following circumstances:
    - When there is a fatal error with a system component.
    - When reports have not been generated before 9am Singapore time.

Data Retention and Archiving

- Input files used in the risk calculation process must be retained for 1 year.

Interoperability

- Interfaces with existing data systems should conform to and use existing data formats.