

# Software Architecture

*for Developers*

Technical leadership by **coding**, coaching,  
collaboration, *architecture sketching*  
and just enough up front design

Simon Brown

# **Software Architecture for Developers**

Technical leadership by coding, coaching, collaboration, architecture sketching and just enough up front design

Simon Brown

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

This version was published on 2015-12-24



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

© 2012 - 2015 Simon Brown

## **Tweet This Book!**

Please help Simon Brown by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#sa4d](#).

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

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

*For Kirstie, Matthew and Oliver*

# Contents

Preface . . . . .	i
Software architecture has a bad reputation . . . . .	i
Agile aspirations . . . . .	ii
So you think you're an architect? . . . . .	ii
The frustrated architect . . . . .	iii
About the book . . . . .	iv
Why did I write the book? . . . . .	iv
A new approach to software development? . . . . .	v
Five things every developer should know about software architecture . . . . .	vi
About the author . . . . .	viii
Software architecture training . . . . .	ix
I What is software architecture? . . . . .	1
1. What is architecture? . . . . .	2
As a noun . . . . .	3
As a verb . . . . .	3
2. Types of architecture . . . . .	4
What do they all have in common? . . . . .	5
3. What is software architecture? . . . . .	6
Application architecture . . . . .	6
System architecture . . . . .	6
Software architecture . . . . .	7
Enterprise architecture - strategy rather than code . . . . .	8

## CONTENTS

4.	<b>Architecture vs design</b> . . . . .	9
	Making a distinction . . . . .	9
	Understanding significance . . . . .	10
5.	<b>Is software architecture important?</b> . . . . .	12
	A lack of software architecture causes problems . . . . .	12
	The benefits of software architecture . . . . .	13
	Does every software project need software architecture? . . . . .	13
6.	<b>Questions</b> . . . . .	14
<b>II</b>	<b>The software architecture role</b> . . . . .	15
7.	<b>The software architecture role</b> . . . . .	16
	1. Architectural Drivers . . . . .	16
	2. Designing Software . . . . .	17
	3. Technical Risks . . . . .	17
	4. Architecture Evolution . . . . .	18
	5. Coding . . . . .	18
	6. Quality Assurance . . . . .	19
	Collaborate or fail . . . . .	19
	Technical leadership is a role, not a rank . . . . .	20
	Create your own definition of the role . . . . .	20
8.	<b>Software development is not a relay sport</b> . . . . .	22
	“Solution Architects” . . . . .	22
	Somebody needs to own the big picture . . . . .	23
9.	<b>Questions</b> . . . . .	24
<b>III</b>	<b>Designing software</b> . . . . .	25
10.	<b>Architectural drivers</b> . . . . .	26
	1. Functional requirements . . . . .	26
	2. Quality Attributes . . . . .	26
	3. Constraints . . . . .	26
	4. Principles . . . . .	27
	Understand their influence . . . . .	27

## CONTENTS

11. Questions . . . . .	28
<b>IV Communicating design . . . . .</b>	<b>29</b>
12. We have a failure to communicate . . . . .	30
Abandoning UML . . . . .	30
Agility requires good communication . . . . .	32
13. C4: context, containers, components and classes . . . . .	33
A common set of abstractions . . . . .	33
Summarising the static view of your software . . . . .	35
Common abstractions over a common notation . . . . .	35
Diagrams should be simple and grounded in reality . . . . .	36
14. Questions . . . . .	38
<b>V Documenting software . . . . .</b>	<b>39</b>
15. The code doesn't tell the whole story . . . . .	40
The code doesn't portray the intent of the design . . . . .	41
Supplementary information . . . . .	42
16. Software documentation as a guidebook . . . . .	44
1. Maps . . . . .	44
2. Sights . . . . .	46
3. History and culture . . . . .	46
4. Practical information . . . . .	47
Keep it short, keep it simple . . . . .	47
Beware of the “views” . . . . .	48
Product vs project documentation . . . . .	49
17. Questions . . . . .	50
<b>VI Agility and the essence of software architecture . . . . .</b>	<b>51</b>
18. The conflict between agile and architecture - myth or reality? . . . . .	52

## CONTENTS

Conflict 1: Team structure . . . . .	52
Conflict 2: Process and outputs . . . . .	53
Software architecture provides boundaries for TDD, BDD, DDD, RDD and clean code . . . . .	53
Separating architecture from ivory towers and big up front design . . . . .	54
<b>19. Just enough up front design . . . . .</b>	<b>56</b>
It comes back to methodology . . . . .	56
You need to do “just enough” . . . . .	58
How much is “just enough”? . . . . .	59
Firm foundations . . . . .	60
Contextualising just enough up front design . . . . .	61
<b>20. Agility . . . . .</b>	<b>63</b>
Understanding “agility” . . . . .	63
A good architecture enables agility . . . . .	64
Agility as a quality attribute . . . . .	66
Creating agile software systems in an agile way . . . . .	66
<b>21. Introducing software architecture . . . . .</b>	<b>68</b>
Software architecture needs to be accessible . . . . .	68
Some practical suggestions . . . . .	69
Making change happen . . . . .	70
The essence of software architecture . . . . .	72
<b>22. Questions . . . . .</b>	<b>74</b>
<b>VII Appendix A: Financial Risk System . . . . .</b>	<b>75</b>
<b>VIII Appendix B: Software Guidebook for techtribes.je</b>	
76	

# Preface

The IT industry is either taking giant leaps ahead or it's in deep turmoil. On the one hand we're pushing forward, reinventing the way that we build software and striving for craftsmanship at every turn. On the other though, we're continually forgetting the good of the past and software teams are still screwing up on an alarmingly regular basis.

Software architecture plays a pivotal role in the delivery of successful software yet it's frustratingly neglected by many teams. Whether performed by one person or shared amongst the team, the software architecture role exists on even the most agile of teams yet the balance of up front and evolutionary thinking often reflects aspiration rather than reality.

## **Software architecture has a bad reputation**

I tend to get one of two responses if I introduce myself as a software architect. Either people think it's really cool and want to know more or they give me a look that says "I want to talk to somebody that actually writes software, not a box drawing hand-waver". The software architecture role has a bad reputation within the IT industry and it's not hard to see where this has come from.

The thought of "software architecture" conjures up visions of ivory tower architects doing big design up front and handing over huge UML (Unified Modeling Language) models or 200 page Microsoft Word documents to an unsuspecting development team as if they were the second leg of a relay race. And that's assuming the architect actually gets involved in designing software of course. Many people seem to think that creating a Microsoft PowerPoint presentation with a slide containing a big box labelled "Enterprise Service Bus" *is* software design. Oh, and we mustn't forget the obligatory narrative about "ROI" (return on investment) and "TCO" (total cost of ownership) that will undoubtedly accompany the presentation.

Many organisations have an interesting take on software development as a whole too. For example, they've seen the potential cost savings that offshoring can bring and therefore see the coding part of the software development process as being something of a commodity. The result tends to be that local developers are pushed into the "higher value" software architecture jobs with an expectation that all coding will be undertaken by somebody else. In many cases this only exaggerates the disconnect between software architecture and software

development, with people often being pushed into a role that they are not prepared for. These same organisations often tend to see software architecture as a rank rather than a *role* too.

## Agile aspirations

“Agile” might be over ten years old, but it’s still the shiny new kid in town and many software teams have aspirations of “becoming agile”. Agile undoubtedly has a number of benefits but it isn’t necessarily the silver bullet that everybody wants you to believe it is. As with everything in the IT industry, there’s a large degree of evangelism and hype surrounding it. Start a new software project today and it’s all about self-organising teams, automated acceptance testing, continuous delivery, retrospectives, Kanban boards, emergent design and a whole host of other buzzwords that you’ve probably heard of. This is fantastic but often teams tend to throw the baby out with the bath water in their haste to adopt all of these cool practices. “Non-functional requirements” not sounding cool isn’t a reason to neglect them.

What’s all this old-fashioned software architecture stuff anyway? Many software teams seem to think that they don’t need software architects, throwing around terms like “self-organising team”, “YAGNI” (you aren’t going to need it), “evolutionary architecture” and “last responsible moment” instead. If they do need an architect, they’ll probably be on the lookout for an “agile architect”. I’m not entirely sure what this term actually means, but I assume that it has something to do with using post-it notes instead of UML or doing TDD (test-driven development) instead of drawing pictures. That is, assuming they get past the notion of only using a very high level system metaphor and don’t use “emergent design” as an excuse for foolishly hoping for the best.

## So you think you’re an architect?

It also turns out there are a number of people in the industry claiming to be software architects whereas they’re actually doing something else entirely. I can forgive people misrepresenting themselves as an “Enterprise Architect” when they’re actually doing hands-on software architecture within a large enterprise. The terminology in our industry *is* often confusing after all.

But what about those people that tend to exaggerate the truth about the role they play on software teams? Such irresponsible architects are usually tasked with being the technical leader yet fail to cover the basics. I’ve seen public facing websites go into a user acceptance testing environment with a number of basic security problems, a lack of basic performance

testing, basic functionality problems, broken hyperlinks and a complete lack of documentation. And that was just my external view of the software, who knows what the code looked like. If you’re undertaking the software architecture role and you’re delivering stuff like this, you’re doing it wrong. This *isn’t* software architecture, it’s also foolishly hoping for the best.

## The frustrated architect

Admittedly not all software teams are like this but what I’ve presented here isn’t a “straw man” either. Unfortunately many organisations do actually work this way so the reputation that software architecture has shouldn’t come as any surprise.

If we really do want to succeed as an industry, we need to get over our fascination with shiny new things and start asking some questions. Does agile need architecture or does architecture actually need agile? Have we forgotten more about good software design than we’ve learnt in recent years? Is foolishly hoping for the best sufficient for the demanding software systems we’re building today? Does any of this matter if we’re not fostering the software architects of tomorrow? How do we move from frustration to serenity?

# About the book

This book is a practical, pragmatic and lightweight guide to software architecture for developers. You'll learn:

- The essence of software architecture.
- Why the software architecture role should include coding, coaching and collaboration.
- The things that you *really* need to think about before coding.
- How to visualise your software architecture using simple sketches.
- A lightweight approach to documenting your software.
- Why there is *no* conflict between agile and architecture.
- What “just enough” up front design means.
- How to identify risks with risk-storming.

This collection of essays knocks down traditional ivory towers, blurring the line between software development and software architecture in the process. It will teach you about software architecture, technical leadership and the balance with agility.

## 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.

# Five things every developer should know about software architecture

To give you a flavour of what this book is about, here are five things that every developer should know about software architecture.

## 1. Software architecture isn't about big design up front

Software architecture has traditionally been associated with big design up front and water-fall-style projects, where a team would ensure that every last element of the software design was considered before any code was written. Software architecture is basically about the high-level structure of a software system and how you get to an understanding of it. This is about the significant decisions that influence the shape of a software system rather than understanding how long every column in the database should be.

## 2. Every software team needs to consider software architecture

Regardless of the size and complexity of the resulting product, every software team needs to consider software architecture. Why? Put simply, bad things tend to happen if they don't! If software architecture is about structure and vision, not thinking about this tends to lead to poorly structured, internally inconsistent software systems that are hard to understand, hard to maintain and potentially don't satisfy one or more of the important non-functional requirements such as performance, scalability or security. Explicitly thinking about software architecture provides you with a way to introduce technical leadership and stacks the odds of a successful delivery in your favour.

## 3. The software architecture role is about coding, coaching and collaboration

The image that many people have of software architects is of traditional “ivory tower” software architects dictating instructions to an unsuspecting development team. It doesn't need to be like this though, with modern software architects preferring an approach that favours coding, coaching and collaborative design. The software architecture role doesn't necessarily need to be undertaken by a single person plus coding is a great way to understand whether the resulting architecture is actually going to work.

## 4. You don't need to use UML

Again, traditional views of software architecture often conjure up images of huge UML (Unified Modeling Language) models that attempt to capture every last drop of detail. While creating and communicating a common vision is important, you don't need to use UML. In fact, you could argue that UML isn't a great method for communicating software architecture anyway. If you keep a few simple guidelines in mind, lightweight "boxes and lines" style sketches are an effective way to communicate software architecture.

## 5. A good software architecture enables agility

There's a common misconception that "architecture" and "agile" are competing forces, there being a conflict between them. This simply isn't the case though. On the contrary, a good software architecture enables agility, helping you embrace and implement change. Good software architectures aren't created by themselves though, and some conscious effort is needed.

# 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. I'm the author of two books about software architecture; [Software Architecture for Developers](#) (a developer-friendly guide to software architecture, technical leadership and the balance with agility) and [The Art of Visualising Software Architecture](#) (a guide to communicating software architecture with sketches, diagrams and models). I'm also the creator of the C4 software architecture model and I built [Structurizr](#), which is a web-based tool to create software architecture diagrams based upon the C4 model.

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

You can find my website at [simonbrown.je](http://simonbrown.je) and I can be found on Twitter at [@simonbrown](https://twitter.com/@simonbrown).

# Software architecture training

I provide one and two-day training courses/workshops that are practical and pragmatic guides to lightweight software architecture, covering the same content you'll find in this book. You'll learn:

- The essence of software architecture.
- Why the software architecture role should include coding, coaching and collaboration.
- The things that you *really* need to think about before coding.
- How to visualise your software architecture using simple sketches and my C4 model.
- A lightweight approach to documenting your software.
- Why there is *no* conflict between agile and architecture.
- What “just enough” up front design means.
- How to identify risks with risk-storming.



See <http://www.codingthearchitecture.com/training/> or e-mail [simon.brown@codingthearchitecture.com](mailto:simon.brown@codingthearchitecture.com) for further details.

# I What is software architecture?

In this part of the book we'll look at what software architecture is about, the difference between architecture and design, what it means for an architecture to be agile and why thinking about software architecture is important.

# 1. What is architecture?

The word “architecture” means many different things to many different people and there are many different definitions floating around the Internet. I’ve asked hundreds of people over the past few years what “architecture” means to them and a summary of their answers is as follows. In no particular order...

- 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, with both being applicable regardless of whether we're talking about constructing a physical building or a software system.

## As a noun

As a noun then, architecture can be summarised as being about structure. It's about the decomposition of a product into a collection of components/modules and interactions. This needs to take into account the whole of the product, including the foundations and infrastructure services that deal with cross-cutting concerns such as power/water/air conditioning (for a building) or security/configuration/error handling (for a piece of software).

## As a verb

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

## 2. Types of architecture

There are many different types of architecture and architects within the IT industry alone. Here, in no particular order, is a list of those that people most commonly identify when asked...

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

The unfortunate thing about this list is that some of the terms are easier to define than others, particularly those that refer to or depend upon each other for their definition. For example, what does “solution architecture” actually mean? For some organisations “solution architect” is simply a synonym for “software architect” whereas others have a specific role that focusses on designing an overall “solution” to a problem, but stopping before the level at which implementation details are discussed. Similarly, “technical architecture” is vague enough to refer to software, hardware or a combination of the two.

Interestingly, “software architecture” typically appears near the bottom of the list when I ask people to list the types of IT architecture they’ve come across. Perhaps this reflects the confusion that surrounds the term.

## What do they all have in common?

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

Take “infrastructure architecture” as an example and imagine that you need to create a network between two offices at different ends of the country. One option is to find the largest reel of network cable that you can and start heading from one office to the other in a straight line. Assuming that you had enough cable, this could potentially work, but in reality there are a number of environmental constraints and non-functional characteristics that you need to consider in order to actually deliver something that satisfies the original goal. This is where the process of architecting and having a vision to achieve the goal is important.

One single long piece of cable is *an* approach, but it’s not a very good one because of real-world constraints. For this reason, networks are typically much more complex and require a collection of components collaborating together in order to satisfy the goal. From an infrastructure perspective then, we can talk about structure in terms of the common components that you’d expect to see within this domain; things like routers, firewalls, packet shapers, switches, etc.

Regardless of whether you’re building a software system, a network or a database; a successful solution requires you to understand the problem and create a vision that can be communicated to everybody involved with the construction of the end-product. Architecture, regardless of the domain, is about [structure and vision](#).

# 3. What is software architecture?

At first glance, “software architecture” seems like an easy thing to define. It’s about the architecture of a piece of software, right? Well, yes, but it’s about more than just software.

## Application architecture

Application architecture is what we as software developers are probably most familiar with, especially if you think of an “application” as typically being written in a single technology (e.g. a Java web application, a desktop application on Windows, etc). It puts the application in focus and normally includes things such as decomposing the application into its constituent classes and components, making sure design patterns are used in the right way, building or using frameworks, etc. In essence, application architecture is inherently about the lower-level aspects of software design and is usually only concerned with a single technology stack (e.g. Java, Microsoft .NET, etc).

The building blocks are predominantly software based and include things like programming languages and constructs, libraries, frameworks, APIs, etc. It’s described in terms of classes, components, modules, functions, design patterns, etc. Application architecture is predominantly about software and the organisation of the code.

## System architecture

I like to think of system architecture as one step up in scale from application architecture. If you look at most software systems, they’re actually composed of multiple applications across a number of different tiers and technologies. As an example, you might have a software system comprised of a mobile app communicating via JSON/HTTPS to a Java web application, which itself consumes data from a MySQL database. Each of these will have their own internal application architecture.

For the overall software system to function, thought needs to be put into bringing all of those separate applications together. In other words, you also have the overall structure of the end-to-end software system at a high-level. Additionally, most software systems don’t live in isolation, so system architecture also includes the concerns around interoperability and integration with other systems within the environment.

The building blocks are a mix of software and hardware, including things like programming languages and software frameworks through to servers and infrastructure. Compared to application architecture, system architecture is described in terms of higher levels of abstraction; from components and services through to sub-systems. Most definitions of system architecture include references to software *and* hardware. After all, you can't have a successful software system without hardware, even if that hardware is virtualised somewhere out there on the cloud.

## Software architecture

Unlike application and system architecture, which are relatively well understood, the term “software architecture” has many different meanings to many different people. Rather than getting tied up in the complexities and nuances of the many definitions of software architecture, I like to keep the definition as simple as possible. For me, software architecture is simply the combination of application and system architecture.

In other words, it’s anything and everything related to the significant elements of a software system; from the structure and foundations of the code through to the successful deployment of that code into a live environment. When we’re thinking about software development as software developers, most of our focus is placed on the code. Here, we’re thinking about things like object oriented principles, classes, interfaces, inversion of control, refactoring, automated unit testing, clean code and the countless other technical practices that help us build better software. If your team consists of people who are *only* thinking about this, then who is thinking about the other stuff?

- Cross-cutting concerns such as logging and exception handling.
- Security; including authentication, authorisation and confidentiality of sensitive data.
- Performance, scalability, availability and other quality attributes.
- Audit and other regulatory requirements.
- Real-world constraints of the environment.
- Interoperability/integration with other software systems.
- Operational, support and maintenance requirements.
- Consistency of structure and approach to solving problems/implementing features across the codebase.
- Evaluating that the foundations you’re building will allow you to deliver what you set out to deliver.

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

## **Enterprise architecture - strategy rather than code**

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

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

# 4. Architecture vs design

If architecture is about [structure and vision](#), then what's design about? If you're creating a solution to solve a problem, isn't this just design? And if this is the case, what's the difference between design and architecture?

## Making a distinction

Grady Booch has a well cited definition of the difference between architecture and design that really helps to answer this question. In [On Design](#), he says that

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

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

Grady then goes on to say that...

All architecture is design but not all design is architecture.

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

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

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

- The shape of the system (e.g. client-server, web-based, native mobile client, distributed, asynchronous, etc)
- The structure of the software system (e.g. components, layers, interactions, 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.

## Understanding significance

It's often worth taking a step back and considering what's significant with your own software system. For example, many teams use a relational database, the choice of which might be deemed as significant. In order to reduce the amount of rework required in the event of a change in database technology, many teams use an object-relational mapping (ORM) framework such as Hibernate or Entity Framework. Introducing this additional ORM layer allows the database access to be decoupled from other parts of the code and, in theory, the database can be switched out independently without a large amount of effort.

This decision to introduce additional layers is a classic technique for decoupling distinct parts of a software system; promoting looser coupling, higher cohesion and a better separation of concerns. Additionally, with the ORM in place, the choice of database can probably be switched in an afternoon, so from this perspective it may no longer be deemed as architecturally significant.

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 ORM or web MVC framework and replace it with another. Of course, you could add another layer over the top of your chosen ORM to further isolate your business logic and provide the ability

to easily swap out your ORM but, again, you've made another significant decision. You've introduced additional layering, complexity and cost.

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

# 5. Is software architecture important?

Software architecture then, is it important? The agile and software craftsmanship movements are helping to push up the quality of the software systems that we build, which is excellent. Together they are helping us to write better software that better meets the needs of the business while carefully managing time and budgetary constraints. But there's still more we can do because even a small amount of software architecture can help prevent many of the problems that projects face. Successful software projects aren't just about good code and sometimes you need to step away from the code for a few moments to see the bigger picture.

## A lack of software architecture causes problems

Since software architecture is about [structure and vision](#), you could say that it exists anyway. And I agree, it does. Having said that, it's easy to see how not thinking about software architecture (and the “bigger picture”) can lead to a number of common problems that software teams face on a regular basis. Ask yourself the following questions:

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

It is possible to successfully deliver a software project by answering “no” to some of these questions, but it does require a very good team and a lot of luck. If nobody thinks about software architecture, the end result is something that typically looks like a [big ball of mud](#). Sure, it has a structure but it's not one that you'd want to work with! Other side effects could include the software system being too slow, insecure, fragile, unstable, hard to deploy, hard to maintain, hard to change, hard to extend, etc. I'm sure you've never seen or worked on software projects like this, right? No, me neither. ;-)

Since software architecture is inherent in every software system, why don't we simply acknowledge this and place some focus on it?

## The benefits of software architecture

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

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

## Does every software project need software architecture?

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

## 6. Questions

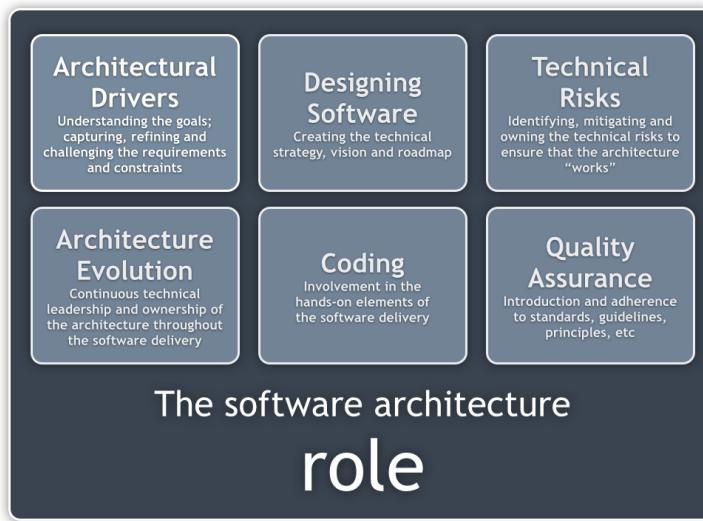
1. Do you know what “architecture” is all about? Does the rest of your team? What about the rest of your organisation?
2. There are a number of different types of architecture within the IT domain. What do they all have in common?
3. Do you and your team have a standard definition of what “software architecture” means? Could you easily explain it to new members of the team? Is this definition common across your organisation?
4. What does it mean if you describe a software architecture as being “agile”? How do you design for “agility”?
5. Can you make a list of the architectural decisions on your current software project? Is it obvious why they were deemed as significant?
6. If you step back from the code, what sort of things are included in *your* software system’s “big picture”?
7. What does the technical career path look like in your organisation? Is enterprise architecture the right path for you?
8. Is software architecture important? Why and what are the benefits? Is there enough software architecture on your software project? Is there too much?

# **II The software architecture role**

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.

# 7. The software architecture role

Becoming a software architect isn't something that simply 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 fairly well understood, "software architect" isn't. Here are the things that I consider to make up the software architecture role. Notice that I said "role" here; it's something that can be performed by a single person or shared amongst the team.



## 1. Architectural Drivers

The first part of the role is about understanding the business goals and managing the architectural drivers, which includes the requirements (both functional and non-functional) and the **constraints of the environment**. Software projects often get caught up on asking users what features they want, but rarely ask them what **non-functional requirements (or quality attributes)** that they need. Sometimes the stakeholders will tell us that "the system must be fast", but that's far too subjective. Non-functional requirements and constraints often have

a huge influence on the software architecture, so explicitly including them as a part of the software architecture role helps to ensure that they are considered and taken into account.

## 2. Designing Software

It should come as no surprise that the process of designing software is a 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 a vision for the delivery. Despite how agile you strive to be, you probably do need **some** time to explicitly think about how your architecture is going to solve the problems set out by the stakeholders because your software system isn't going to do this itself.

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 forced 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 rich client technology into a complete nightmare. Somebody needs to take ownership of the technology selection 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 the best technologies doesn't necessarily 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 have a "buy over build" strategy and use products (commercial or open source) because of the potential cost savings on offer. However, many teams also get burnt because they believe the hype from vendor websites or sales executives in expensive suits. Few people seem to ask whether the technology actually works the way it is supposed to, and fewer 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 had. All technology decisions need to be made by taking all factors into account, and all technology decisions

need to be reviewed and evaluated. This potentially includes all of the major building blocks for a software project right down to the libraries and frameworks being introduced during the development.

The question that you need to ask yourself is whether your architecture “works”. For me, an architecture works if it satisfies the non-functional requirements, works within the given environmental constraints, provides the necessary foundations for the rest of the code and works as the platform for solving the underlying business problem. One of the biggest problems with software is that it’s complex and abstract. The result being that it’s hard to visualise the runtime characteristics of a piece of software from diagrams or even the code itself. Furthermore, I don’t always trust myself to get it right first time. Your mileage may vary though!

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 failure. Like good chefs, architects should taste what they are producing. In a nutshell, this 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. Architecture Evolution

More often than not, software is designed and then the baton is passed over to a development team, effectively [treating software development as a relay sport](#). This is counterproductive because the resulting software architecture 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 an architect has created an architecture, why shouldn’t they 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. Coding

Most of the best software architects I know have a software development background, but for some reason many organisations don’t see this 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?

Many software architects are **master builders**, so it makes sense to keep those skills up to date. In addition, 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. Many companies have policies that prevent software architects from engaging in coding activities because their architects are “too valuable to undertake commodity 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?

Of course, there are situations where it’s not practical to get involved at the code level. For example, a large project generally means a bigger “big picture” to take care of 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”.

## 6. 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 or conformance is up to you, but the technical vision needs to be followed.

It’s safe to say that most projects don’t do enough quality assurance, and therefore you need to figure out what’s important and make sure that it’s sufficiently assured. For me, the important parts of a project are anything that is architecturally significant, business critical, complex or highly visible. You need to be pragmatic though and realise that you can’t necessarily assure everything.

## Collaborate or fail

It’s **unusual for a software system to reside in isolation** and there are a number of people that probably need to contribute to the overall architecture process. This ranges from the

immediate development team who need to understand and buy in to the architecture, right through to the extended team of those people who will have an interest in the architecture from a security, database, operations, maintenance or support point of view. If you're undertaking the software architecture role, you'll need to collaborate with such people to ensure that the resulting software system will successfully integrate with its environment. If you don't collaborate, expect to fail.

## Technical leadership is a role, not a rank

The software architecture role is basically 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. Often large organisations 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, I *can* describe what my view of the software architecture role is. Designing software might be the fun part of the role, but a successful software project is about much more.

## Create your own definition of the role

In my experience, although many software teams *do* understand the need for the software architecture role, they often don't have a defined terms of reference for it. Without this, you run the risk of the role not being performed in part or in whole.

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. I regularly ask software teams whether they have a defined terms of reference for 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 and we have little hope of growing the software architects of tomorrow.

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.

## **8. Software development is not a relay sport**

Software teams that are smaller and/or agile tend to be staffed with people who are generalising specialists; people that have a core specialism along with more general knowledge and experience. In an ideal world, these cross-discipline team members would work together to run and deliver a software project, undertaking everything from requirements capture and architecture through to coding and deployment. Although many software teams strive to be self-organising, in the real world they tend to be larger, more chaotic and staffed only with specialists. Therefore, these teams typically need, and often do have, somebody in the technical leadership role.

### **“Solution Architects”**

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. When you throw “not invented here” syndrome into the mix, there’s often a tendency for that receiving team to not take ownership of the solution and the “architecture” initially created becomes detached from reality.

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”, 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. In my view, this was an outrageous thing to say and it made

him look like an ass during the interview. His approach was also AaaS ... “Architecture as a Service”!

## Somebody needs to own the big picture

The [software architecture role](#) is a generalising specialist role, and different to your typical software developer. It’s certainly about steering the ship at the start of a software project, which includes things like managing the non-functional requirements and putting together a software design that is sensitive to the context and environmental factors. 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 the initial vision to be created, 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. Let’s hope that the development team never needs to ask any questions about the design or its intent!

This problem goes away with truly self-organising teams, but most teams haven’t yet reached this level of maturity. Somebody needs to take ownership of the big picture throughout the delivery and they also need to take responsibility for ensuring that the software is delivered successfully. Software development is not a relay sport and successful delivery is not an “implementation detail”.

## 9. Questions

1. What's the difference between the software architecture and software developer roles?
2. What does the software architecture role entail? Is this definition based upon your current or ideal team? If it's the latter, what can be done to change your team?
3. Why is it important for anybody undertaking the software architecture role to understand the technologies that they are using? Would you hire a software architect who didn't understand technology?
4. If you're the software architect on your project, how much coding are you doing? Is this too much or too little?
5. If, as a software architect, you're unable to code, how else can you stay engaged in the low-level aspects of the project. And how else can you keep your skills up to date?
6. Why is having a breadth *and* depth of technical knowledge as important?
7. Do you think you have all of the required soft skills to undertake the software architecture role? If not, which would you improve, why and how?
8. Does your current software project have enough guidance and control from a software architecture perspective? Does it have too much?
9. Why is collaboration an important part of the software architecture role? Does your team do enough? If not, why?
10. Is there enough coaching and mentoring happening on your team? Are you providing or receiving it?
11. How does the software architecture role fit into agile projects and self-organising teams?
12. What pitfalls have you fallen into as somebody new to the software architecture role?
13. Is there a well-defined “terms of reference” for the software architecture in your team or organisation? If so, does everybody understand it? If not, is there value in creating one to make an architect's role and responsibilities explicit?

# **III Designing software**

This part of the book is about the overall process of designing software, specifically looking at the things that you should really think about before coding.

# 10. Architectural drivers

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. 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 might call this being agile, but I call it foolish. Even a rough, short list of features or user stories (e.g. a [Scrum product backlog](#)) is essential. Requirements drive architecture.

## 2. Quality Attributes

[Quality attributes](#) are represented by the non-functional requirements and reflect levels of service such as performance, scalability, availability, security, etc. These are mostly technical in nature and can have a huge influence over the resulting architecture, particularly if you're building "high performance" systems or you have desires to operate at "Google scale". The technical solutions to implementing non-functional requirements 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.

## 3. Constraints

We live in the real world and the real world has [constraints](#). For example, the organisation that you work for probably has a raft of constraints detailing what you can and can't do with respect to technology choice, deployment platform, etc.

## 4. Principles

Where constraints are typically imposed upon you, [principles](#) are the things that you want to adopt in order to introduce consistency and clarity into the resulting codebase. These may be development principles (e.g. code conventions, use of automated testing, etc) or architecture principles (e.g. layering strategies, architecture patterns, etc).

### Understand their influence

Understanding the requirements, 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 drivers include complex non-functional requirements or major constraints such as restrictions over the 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.

Software architecture is about the significant design decisions, [where significance is measured by cost of change](#). A high-level understanding of the requirements, 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.

# 11. Questions

1. What are the major factors that influence the resulting architecture of a software system? Can you list those that are relevant to the software system that you are working on?
2. What are non-functional requirements and why are they important? When should you think about non-functional requirements?
3. Time and budget are the constraints that most people instantly relate to, but can you identify more?
4. Is your software development team working with a well-known set of architectural principles? What are they? Are they clearly understood by everybody on the team?
5. How do *you* approach the software design process? Does your team approach it in the same way? Can it be clearly articulated? Can you help others follow the same approach?

# **IV Communicating design**

This part of the book is about visualising software architecture using a collection of lightweight, yet effective, sketches.

# 12. We have a failure to communicate

If you're working in an agile software development team at the moment, take a look around at your environment. Whether it's physical or virtual, there's likely to be a story wall or Kanban board visualising the work yet to be started, in progress and done.

Why? Put simply, visualising your software development process is a fantastic way to introduce transparency because anybody can see, at a glance, a high-level snapshot of the current progress. Couple this with techniques like [value stream mapping](#) and you can start to design some complex Kanban boards to reflect the way that your team works. As an industry, we've become pretty adept at visualising our software development process.

However, it seems we've forgotten how to visualise the actual software that we're building. I'm not just referring to post-project documentation, this also includes communication *during* the software development process.

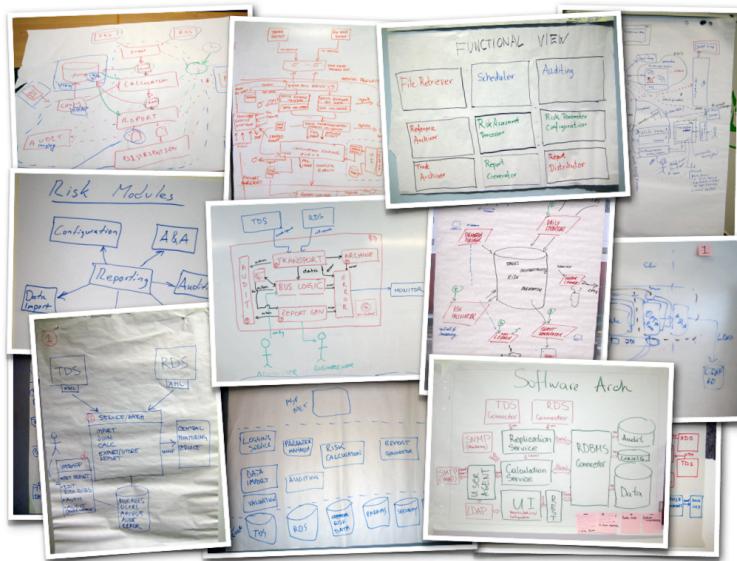
Understanding software architecture is not the same as being able to communicate it. Those architecture diagrams that you have on the wall of your office; do they reflect the system that is actually being built or are they conceptual abstractions that bear no resemblance to the structure of the code. Having run architecture katas with thousands of people over a number of years, I can say with complete confidence that visualising the architecture of a software system is a skill that very few people have. People can draw diagrams, but those diagrams often leave much to the imagination. Almost nobody uses a formal diagramming notation to describe their solutions too, which is in stark contrast to my experience of working with software teams a decade ago.

## Abandoning UML

If you cast your mind back in time, structured processes provided a reference point for both the software design process and how to communicate the resulting designs. Some well-known examples include the Rational Unified Process (RUP) and Structured Systems Analysis And Design Method (SSADM). Although the software development industry has moved on in many ways, we seem to have forgotten some of the good things that these prior approaches gave us.

As an industry, we do have the Unified Modelling Language (UML), which is a formal standardised notation for communicating the design of software systems. However, while

you can argue about whether UML offers an effective way to communicate software designs or not, that's often irrelevant because many teams have already thrown out UML or simply don't know it. Such teams typically favour informal boxes and lines style sketches instead but often these diagrams don't make much sense unless they are accompanied by a detailed narrative, which ultimately slows the team down. Next time somebody presents a software design to you focussed around one or more informal sketches, ask yourself whether they are presenting what's on the sketches or whether they are presenting what's in their head instead.



Boxes and lines sketches can work very well, but there are many pitfalls associated with communicating software architecture in this way

Abandoning UML is all very well but, in the race for agility, many software development teams have lost the ability to communicate visually. The example software architecture sketches (pictured) illustrate a number of typical approaches to communicating software architecture and they suffer from the following types of problems:

- Colour coding is usually not explained or is often inconsistent.
- The purpose of diagram elements (i.e. different styles of boxes and lines) is often not explained.
- Key relationships between diagram elements are sometimes missing or ambiguous.
- Generic terms such as “business logic” are often used.

- Technology choices (or options) are usually omitted.
- Levels of abstraction are often mixed.
- Diagrams often try to show too much detail.
- Diagrams often lack context or a logical starting point.

Boxes and lines sketches *can* work very well, but there are many pitfalls associated with communicating software architecture in this way. My approach is to use [a collection of simple diagrams](#) each showing a different part of the same overall story, [paying close attention to the diagram elements](#) if I'm not using UML.

## Agility requires good communication

Why is this important? In today's world of agile delivery and lean startups, many software teams have lost the ability to communicate what it is they are building and it's no surprise that these same teams often seem to lack technical leadership, direction and consistency. If you want to ensure that everybody is contributing to the same end-goal, you need to be able to effectively communicate the vision of what it is you're building. And if you want agility and the ability to move fast, you need to be able to communicate that vision efficiently too.

# 13. C4: context, containers, components and classes

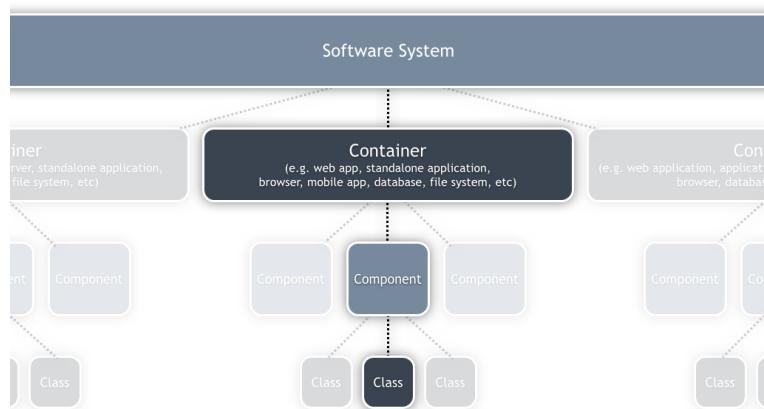
The code for any software system is where most of the focus remains for the majority of the software development life cycle, and this makes sense because the code is the ultimate deliverable. But if you had to explain to somebody how that system worked, would you start with the code?

Unfortunately [the code doesn't tell the whole story](#) and, in the absence of documentation, people will typically start drawing boxes and lines on a whiteboard or piece of paper to explain what the major building blocks are and how they are connected. When describing software through pictures, we have a tendency to create a single uber-diagram that includes as much detail as possible at every level of abstraction simultaneously. This may be because we're anticipating questions or because we're a little too focussed on the specifics of how the system works at a code level. Such diagrams are typically cluttered, complex and confusing. Picking up a tool such as Microsoft Visio, Rational Software Architect or Sparx Enterprise Architect usually adds to the complexity rather than making life easier.

A better approach is to create a number of diagrams at varying levels of abstraction. A number of simpler diagrams can describe software in a much more effective way than a single complex diagram that tries to describe *everything*.

## A common set of abstractions

If software architecture is about the structure of a software system, it's worth understanding what the major building blocks are and how they fit together at differing levels of abstraction.



A **software system** is made up of one or more **containers**,  
each of which contains one or more **components**,  
which in turn are implemented by one or more **classes**.

#### A simple model of architectural constructs

Assuming an OO programming language, the way that I like to think about structure is as follows ... a software system is made up of a number of containers, which themselves are made up of a number of components, which in turn are implemented by one or more classes. It's a simple hierarchy of logical building blocks that can be used to model most software systems.

- **Classes:** for most of us in an OO world, classes are the smallest building blocks of our software systems.
- **Components:** a component can be thought of as a logical grouping of one or more classes. For example, an audit component or an authentication service that is used by other components to determine whether access is permitted to a specific resource. Components are typically made up of a number of collaborating classes, all sitting behind a higher level contract.
- **Containers:** a container represents something in which components are executed or where data resides. This could be anything from a web or application server through to a rich client application or database. Containers are typically executables that are started as a part of the overall system, but they don't have to be separate processes in their own right. For example, I treat each Java EE web application or .NET website as a separate container regardless of whether they are running in the same physical web server process. The key thing about understanding a software system from a containers

perspective is that any inter-container communication is likely to require a remote interface such as a SOAP web service, RESTful interface, Java RMI, Microsoft WCF, messaging, etc.

- **Systems:** a system is the highest level of abstraction and represents something that delivers value to somebody. A system is made up of a number of separate containers. Examples include a financial risk management system, an Internet banking system, a website and so on.

It's easy to see how we could take this further, by putting some very precise definitions behind each of the types of building block and by modelling the specifics of how they're related. But I'm not sure that's particularly useful because it would constrain and complicate what it is we're trying to achieve here, which is to simply understand the structure of a software system and create a simple set of abstractions with which to describe it.

## Summarising the static view of your software

Visualising this hierarchy is then done by creating a collection of system context, container, component and (optionally) class diagrams to summarise the static structure of a software system:

1. **Context:** A high-level diagram that sets the scene; including key system dependencies and actors.
2. **Container:** A container diagram shows the high-level technology choices, how responsibilities are distributed across them and how the containers communicate.
3. **Component:** For each container, a component diagram lets you see the key logical components and their relationships.
4. **Classes:** This is an *optional* level of detail and I will draw a small number of high-level UML class diagrams if I want to explain how a particular pattern or component will be (or has been) implemented. The factors that prompt me to draw class diagrams for parts of the software system include the complexity of the software plus the size and experience of the team. Any UML diagrams that I do draw tend to be sketches rather than comprehensive models.

## Common abstractions over a common notation

This simple sketching approach works for me and many of the software teams that I work with, but it's about providing some organisational ideas and guidelines rather than creating

a prescriptive standard. The goal here is to help teams communicate their software designs in an effective and efficient way rather than creating another comprehensive modelling notation.

UML provides both a common set of abstractions **and** a common notation to describe them, but I rarely find teams who use either effectively. I'd rather see teams able to discuss their software systems with a common set of abstractions in mind rather than struggling to understand what the various notational elements are trying to show. For me, a common set of abstractions is more important than a common notation.

Most maps are a great example of this principle in action. They all tend to show roads, rivers, lakes, forests, towns, churches, etc but they often use different notation in terms of colour-coding, line styles, iconography, etc. The key to understanding them is exactly that - a key/legend tucked away in a corner somewhere. We can do the same with our software architecture diagrams.

It's worth reiterating that informal boxes and lines sketches provide flexibility at the expense of diagram consistency because you're creating your own notation rather than using a standard like UML. My advice here is to be conscious of [colour-coding](#), [line style](#), [shapes](#), [etc](#) and let a consistent notation evolve naturally within your team. Including a simple key/legend on each diagram to explain the notation will help. Oh, and if naming really *is* the hardest thing in software development, try to avoid a diagram that is simply a collection of labelled boxes. Annotating those boxes with responsibilities helps to avoid ambiguity while providing a nice "at a glance" view.

## Diagrams should be simple and grounded in reality

There seems to be a common misconception that "architecture diagrams" must only present a high-level conceptual view of the world, so it's not surprising that software developers often regard them as pointless. Software architecture diagrams should be grounded in reality, in the same way that the software architecture process should be about coding, coaching and collaboration rather than ivory towers. Including technology choices (or options) is usually a step in the right direction and will help prevent diagrams looking like an ivory tower architecture where a bunch of conceptual components magically collaborate to form an end-to-end software system.

A single diagram can quickly become cluttered and confused, but a collection of simple diagrams allows you to effectively present the software from a number of different levels of abstraction. This means that illustrating your software can be a quick and easy task that

requires little ongoing effort to keep those diagrams up to date. You never know, people might even understand them too.

# 14. Questions

1. Are you able to explain how your software system works at various levels of abstraction? What concepts and levels of abstraction would you use to do this?
2. Do you use UML to visualise the design of your software? If so, is it effective? If not, what notation do you use?
3. Are you able to visualise the software system that you're working on? Would everybody on the team understand the notation that you use and the diagrams that you draw?
4. Should technology choices be included or omitted from “architecture” diagrams?
5. Do you understand the software architecture diagrams for your software system (e.g. on the office wall, a wiki, etc)? If not, what could make them more effective?
6. Do the software architecture diagrams that you have for your software system reflect the abstractions that are present in the codebase? If not, why not? How can you change this?

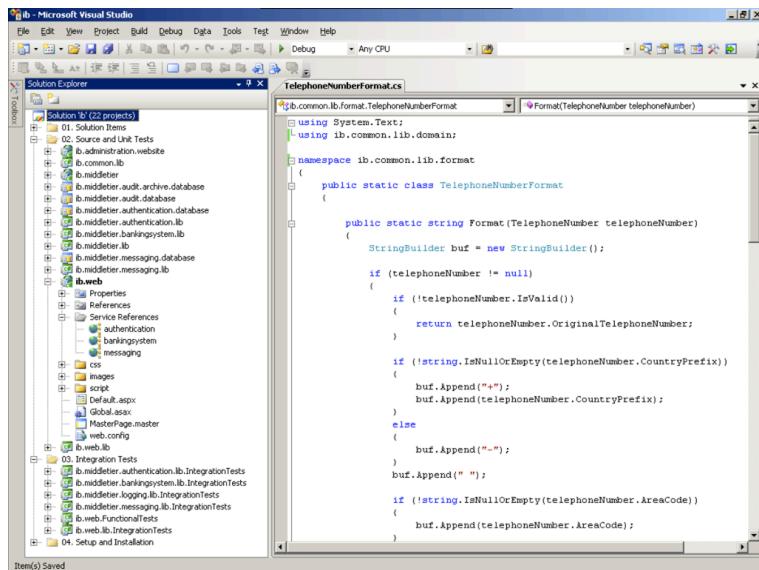
# **V Documenting software**

This part of the book is about that essential topic we love to hate - writing documentation!

# 15. The code doesn't tell the whole story

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

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



The screenshot shows a Microsoft Visual Studio interface with the following details:

- Solution Explorer:** Shows a solution named "ib" containing 22 projects. Projects include "01. Solution", "02. Infrastructure and Unit Tests", "03. Integration Tests", and several "b.middletier" and "b.web" components.
- Code Editor:** Displays a file named "ib.common.lib.format.TelephoneNumberFormat.cs". The code defines a static class `TelephoneNumberFormat` with a single static method `Format`. This method takes a `PhoneNumber` object and uses a `StringBuilder` to construct a formatted string. It handles cases where the phone number is null or invalid, and appends country and area codes if they are present.

```
using System.Text;
using ib.common.lib.domain;

namespace ib.common.lib.format
{
    public static class TelephoneNumberFormat
    {
        public static string Format(PhoneNumber phoneNumber)
        {
            StringBuilder buf = new StringBuilder();

            if (phoneNumber != null)
            {
                if (!phoneNumber.IsValid())
                {
                    return phoneNumber.OriginalPhoneNumber;
                }

                if (!string.IsNullOrEmpty(phoneNumber.CountryPrefix))
                {
                    buf.Append("+");
                    buf.Append(phoneNumber.CountryPrefix);
                }
                else
                {
                    buf.Append(" - ");
                }
                buf.Append(" ");
            }

            if (!string.IsNullOrEmpty(phoneNumber.AreaCode))
            {
                buf.Append(phoneNumber.AreaCode);
            }
        }
    }
}
```

Where do you start?

If nobody has the time to walk you through the codebase, you can start to make your own assumptions based upon the limited knowledge you have about the project, the business

domain, your expectations of how the team builds software and your knowledge of the technologies in use.

For example, you might be able to determine something about the overall architecture of the software system through how the codebase has been broken up into sub-projects, directories, packages, namespaces, etc. Perhaps there are some naming conventions in use. Even from the previous static screenshot of Microsoft Visual Studio, we can determine a number of characteristics about the software, which in this case is an (anonymised) Internet banking system.

- The system has been written in C# on the Microsoft .NET platform.
- The overall .NET solution has been broken down into a number of Visual Studio projects and there's a .NET web application called "ib.web", which you'd expect since this is an Internet banking system ("ib" = "Internet Banking").
- The system appears to be made up of a number of architectural tiers. There's "ib.web" and "ib.middletier", but I don't know if these are physical or logical tiers.
- There looks to be a naming convention for projects. For example, "ib.middletier.authentication.lib", "ib.middletier.messaging.lib" and "ib.middletier.bankingsystem.lib" are class libraries that seem to relate to the middle-tier. Are these simply a logical grouping for classes or something more significant such as higher level components and services?
- With some knowledge of the technology, I can see a "Service References" folder lurking underneath the "ib.web" project. These are Windows Communication Foundation (WCF) service references that, in the case of this example, are essentially web service clients. The naming of them seems to correspond to the class libraries within the middle-tier, so I think we actually have a distributed system with a middle-tier that exposes a number of well-defined services.

## The code doesn't portray the intent of the design

A further deep-dive through the code will help to prove your initial assumptions right or wrong, but it's also likely to leave you with a whole host of questions. Perhaps you understand what the system *does* at a high level, but you don't understand things like:

- How the software system fits into the existing system landscape.
- Why the technologies in use were chosen.
- The overall structure of the software system.
- Where the various components are deployed at runtime and how they communicate.

- How the web-tier “knows” where to find the middle-tier.
- What approach to logging/configuration/error handling/etc has been adopted and whether it is consistent across the codebase.
- Whether any common patterns and principles are in use across the codebase.
- How and where to add new functionality.
- How security has been implemented across the stack.
- How scalability is achieved.
- How the interfaces with other systems work.
- etc

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

## **Supplementary information**

With any software system, there's another layer of information sitting above the code that provides answers to these types of questions and more.



There's an additional layer of information above the code

This type of information is complementary to the code and should be captured somewhere, for example in lightweight supplementary documentation to describe what the code itself doesn't. The code tells *a* story, but it doesn't tell the whole story.

# 16. Software documentation as a guidebook

“Working software over comprehensive documentation” is what the [Manifesto for Agile Software Development](#) says and it’s incredible to see how many software teams have interpreted those five words as “don’t write *any* documentation”. The underlying principle here is that real working software is much more valuable to end-users than a stack of comprehensive documentation but many teams use this line in the agile manifesto as an excuse to not write any documentation at all. Unfortunately [the code doesn’t tell the whole story](#) and not having a source of supplementary information about a complex software system can slow a team down as they struggle to navigate the codebase.

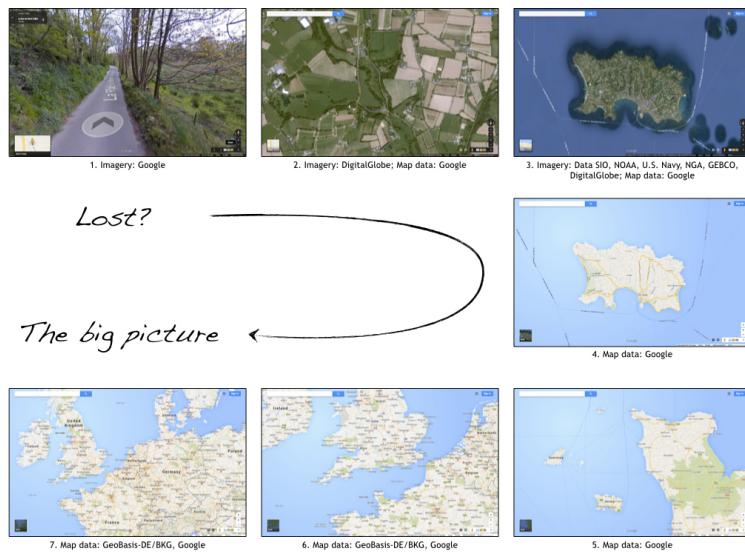
I’m also a firm believer that many software teams have a duty to deliver some supplementary documentation along with the codebase, especially those that are building the software under an outsourcing and/or offshoring contract. I’ve seen IT consulting organisations deliver highly complex software systems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

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

## 1. Maps

Let’s imagine that I teleported you away from where you are now and dropped you in a quiet, leafy country lane somewhere in the world (picture 1). Where are you and how do

you figure out the answer to this question? You could shout for help, but this will only work if there are other people in the vicinity. Or you could simply start walking until you recognised something or encountered some civilisation, who you could then ask for help. As geeks though, we would probably fire up the maps application on our smartphone and use the GPS to pinpoint our location (picture 2).



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

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

If I were to open up the codebase of a complex software system and highlight a random line of code, exploring is fun but it would take a while for you to understand where you were

and how the code fitted into the software system as a whole. Most integrated development environments have a way to navigate the code by namespace, package or folder but often the physical structure of the codebase is different to the logical structure. For example, you may have many classes that make up a single component, and many of those components may make up a single deployable unit.

Diagrams can act as maps to help people navigate a complex codebase and this is one of the most important parts of supplementary software documentation. Ideally there should be a small number of simple diagrams, each showing a different part of the software system or level of abstraction. My [C4 approach](#) is how I summarise the static structure of a software system but there are others including the use of UML.

## 2. Sights

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

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

## 3. History and culture

If you do ever [visit Jersey](#), and you really should because it *is* beautiful, you may see some things that look out of kilter with their surroundings. For example, we have a lovely granite stone castle on the south coast of the island called [Elizabeth Castle](#) that was built in the

16th century. As you walk around admiring the architecture, eventually you'll reach the top where it looks like somebody has dumped a large concrete cylinder, which is not in keeping with the intricate granite stonework generally seen elsewhere around the castle. As you explore further, you'll see signs explaining that the castle was refortified during the German occupation in the second world war. Here, the history helps explain why the castle is the way that it is.

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

## 4. Practical information

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

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

## Keep it short, keep it simple

Exploring is great fun but ultimately it takes time, which we often don't have. Since [the code doesn't tell the whole story](#), *some* supplementary documentation can be very useful, especially if you're handing over the software to somebody else or people are leaving and joining the team on a regular basis. My advice is to [think about this supplementary documentation as a guidebook](#), which should give people enough information to get started and help them accelerate the exploration process. Do resist the temptation to go into too much technical detail though because the technical people that will understand that level of detail will know how to find it in the codebase anyway. As with everything, there's a happy mid-point somewhere.

The following headings describe what you might want to include in a software guidebook:

1. Context
2. Functional Overview
3. Quality Attributes

4. Constraints
5. Principles
6. Software Architecture
7. External Interfaces
8. Code
9. Data
10. Infrastructure Architecture
11. Deployment
12. Operation and Support
13. Development Environment
14. Decision Log

## Beware of the “views”

Many typical software architecture document templates aren’t actually too bad as a starting point for supplementary documentation, but often the names of the various sections confuse people. If you glance over the list of section headings that I’ve just presented, you might be wondering where the typical software architecture “views” are.

If you’ve not seen these before, there are a number of different ways to look at a software system. Examples include [IEEE 1471](#), [ISO/IEC/IEEE 42010](#), Philippe Kruchten’s [4+1 model](#), etc. What they have in common is that they all provide different “views” onto a software system to describe different aspects of it. For example, there’s often a “logical view”, a “physical view”, a “development view” and so on.

The big problem I’ve found with many of these approaches is that it starts to get confusing very quickly if people aren’t versed in the terminology used. For example, I’ve heard people argue about what the difference between a “conceptual view” and a “logical view” is. And let’s not even start asking questions about whether technology is permitted in a logical view! Perspective is important too. If I’m a software developer, is the “development view” about the code, or is that the “implementation view”? But what about the “physical view”? I mean, code is the physical output, right? But then “physical view” means something different to infrastructure architects. But what if the target deployment environment is virtual rather than physical?

My advice is, however you write documentation, just be clear on what it is you’re trying to communicate and name the section accordingly. One option to resolve the terminology issue is to ensure that everybody on the team can point to a clear definition of what the various

architectural views are. [Software Systems Architecture](#) by Eoin Woods and Nick Rozanski comes highly recommended in this regard. Another approach is to simply rename the sections to remove any ambiguity.

## Product vs project documentation

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

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

# 17. Questions

1. We should all strive for self-documenting code, but does this tell the whole story? If not, what is missing?
2. Do you document your software systems? If so, why? If not, why not?
3. If you have lots of project-level documentation but very little product-level documentation, how do new joiners to your team understand your software system(s)? What could make their job easier?
4. What would you consider to be a minimum level of supplementary software documentation within your own environment?
5. Where do you store your supplementary documentation? (e.g. source control system, network file share, SharePoint, wiki, etc). Is this the best solution given your intended audience?

# **VI Agility and the essence of software architecture**

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 software architecture (and therefore, up front design) you should do and how to create firm foundations.

# 18. The conflict between agile and architecture - myth or reality?

The words “agile” and “architecture” are often seen as mutually exclusive but the real world often tells a different story. Some software teams see architecture as an unnecessary evil whereas others have reached the conclusion that they do need to think about architecture once again.

Architecture can be summarised as being about [structure and vision](#), with a key part of the process focussed on understanding 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 projects will have some architectural concerns and these things really should be thought about up front. Agile software projects therefore do need “architecture”, but this seems to contradict with how agile has been evangelised for the past 10+ years. Put simply, there is no conflict between agile and architecture because agile projects need architecture. So, where is the 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. This unfortunate stereotype of “[solution architects](#)” delivering large design documents to the development team before running away to cause havoc elsewhere has resulted in a backlash against having a dedicated architect on a software development team.

One of the things that agile software development teams strive towards is reducing the amount of overhead associated with communication via document hand-offs. It’s rightly about increasing collaboration and reducing waste, with organisations often preferring to create small teams of generalising specialists who can turn their hand to almost any task. 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 no stinkin’ architects”!

## Conflict 2: Process and outputs

The second conflict is between the process and the desired outputs of agile 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 and embracing change. The goal of big design up front is to settle on an understanding of everything that needs to be delivered before putting a blueprint (and usually a plan) in place.

The [agile manifesto](#) values “responding to change” over “following a plan”, but of course this doesn’t mean you shouldn’t do any planning and it seems that some agile teams are afraid of doing any “analysis” at all. The result is that in trying to avoid big up front design, agile teams often do not design up front and instead use terms like “emergent design” or “evolutionary architecture” to justify their approach. I’ve even heard teams claim that their adoption of test-driven development (TDD) negates the need for “architecture”, but these are often the same teams that get trapped in a constant refactoring cycle at some point in the future.

## Software architecture provides boundaries for TDD, BDD, DDD, RDD and clean code

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 really 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 - 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 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 doesn't complement 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 vs components vs micro-services), 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.

## Separating architecture from ivory towers and big up front design

These conflicts, in many cases, lead 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 key architectural drivers such as non-functional requirements.

Architecture is about the stuff that's 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 complex/risky/significant problems.  
**Software architecture is important.**

Big up front design typically covers these architectural concerns but it also tends to go much further, often unnecessarily so. The trick here is to differentiate what's important from what's not. Defining a high-level structure to put a vision in place is important. Drawing a countless number of detailed class diagrams before 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.

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

Considering the architectural drivers needn't take very long and can provide you with a starting point for the rest of the software design activities. Of course, this doesn't mean that the architecture shouldn't be changed, especially when you start writing code and getting feedback. The point is that you now have a framework and some boundaries to work within,

which provide some often needed vision and guidance for the team. My experience suggests that a little direction can go a long way.

# 19. Just enough up front design

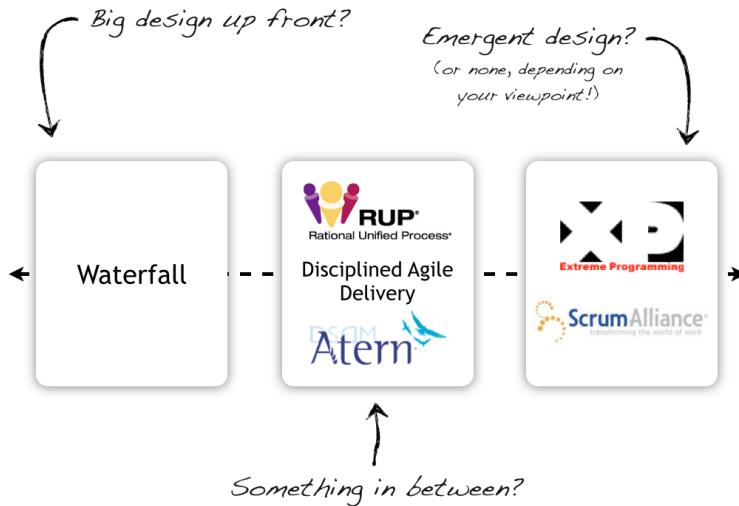
One of the major points of disagreement about software relates to how much up front design to do. People are very polarised as to when they should do design and how much they should do. From my experience of working with software teams, the views basically break down like this.

- “We need to do all of the software architecture up front, before we start coding.”
- “Software architecture doesn’t need to be done up front; we’ll evolve it as we progress.”
- “Meh, we don’t need to do software architecture, we have an excellent team.”

These different views do raise an interesting question, how much architecture do you need to do up front?

## It comes back to methodology

One of the key reasons for the disagreement can be found in how teams work, and specifically what sort of development methodology they are following. If you compare the common software development approaches on account of how much up front design they advocate, you’d have something like the following diagram.



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. And 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 methods don't say "don't do architecture", just as they don't say "don't produce any documentation". Agile is about sufficiency, moving fast, embracing change, feedback and delivering value. 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 project 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".

Sitting between the ends of the scale are methods like the [Rational Unified Process](#) (RUP), [Disciplined Agile Delivery](#) (DAD) and [DSDM Atern](#). These are flexible process frameworks that can be implemented by taking all or part of them. Although many RUP implementations have typically been heavyweight monsters that have more in common with waterfall approaches, it *can* be scaled down to exhibit a combination of characteristics that lets it take the centre ground on the scale. 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 basically 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.

## You need to do “just enough”

My approach to up front architecture and 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 in 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 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 “too little” or “too much” based upon their past experience. Here’s a summary of those thoughts from software developers I’ve met over the past few 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.
- 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).
- No thoughts on key areas of risk; such as non-functional requirements, external interfaces, etc.
- The significant problems and/or their answers haven’t been identified.
- No thought on separation of concerns, appropriate levels of abstraction, layering, modifiability, flex points, etc.
- No common understanding of the role that the architect(s) will play.
- Inconsistent approaches to solving problems.
- A lack of control and guidance for the team.

- Significant change to the architecture during the project lifecycle that could have been anticipated.
- 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 technology choices (i.e. unnecessary deferral).

## 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 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”?

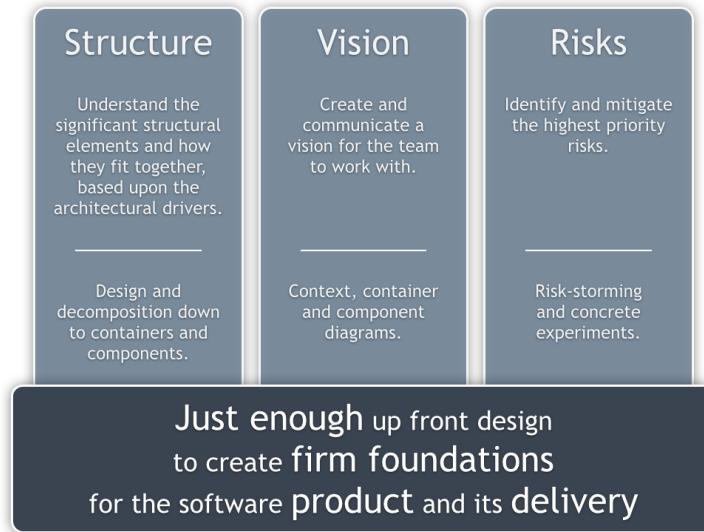
It's easy to identify with many of the answers above but “just enough” sits in that grey area somewhere between the two extremes. The key is that architecture represents the [significant decisions](#), where significance is measured by cost of change. In other words, it's the stuff that's really expensive to modify and the stuff 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 stuff that you can't easily refactor in an afternoon; such as the overall structure, core technology choices, “architectural” patterns, core frameworks and so on.

Back to RUP for a second, and it uses the term “architecturally significant”, advising that you should figure out what might be significant to your architecture. What might be significant? Well, it’s anything that’s costly to change, is complex (e.g. tricky non-functional requirements or constraints) or is new. In reality, these are the things with a higher than normal risk of consequences if you don’t get them right. It’s worth bearing in mind that the significant elements are often subjective too and can vary depending on the experience of the team.

## 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 methodology. 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. I say that you need to do whatever works for you based upon your own context.

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 architecture do you need to do then? I say that you need to do “just enough” 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:** Design and decomposition down to **containers** and **components**.

### 2. Vision

- **What:** Create and communicate a vision for the team to work with.
- **How:** **Context**, **container** and **component** diagrams.

### 3. Risks

- **What:** Identify and mitigate the highest priority risks.
- **How:** **Risk-storming** and **concrete experiments**.

This minimal set of software architecture practices will 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* architecture 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 on my training course. With this in place, ask two or more 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, agree on diagram notation and include examples of good diagrams, 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 and help 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. Are you striving for a self-organising team where everybody is able to perform the software architecture role?

# 20. Agility

In my experience, people tend to use the word “agile” to refer to a couple of things. The first is when talking about [agile approaches](#) to software development; moving fast, embracing change, releasing often, getting feedback and so on. The second use of the word relates to the agile mindset and how people work together in agile environments. This is usually about team dynamics, systems thinking, psychology and other things you might associate with creating high performing teams.

Leaving the latter “fluffy stuff” aside, for me, labelling a software architecture as being “agile” means that it can react to change within its environment, adapting to the ever changing requirements that people throw at it. This isn’t necessarily the same as the software architecture that an agile team will create. Delivering software in an agile way doesn’t guarantee that the resulting software architecture will be agile. In fact, in my experience, the opposite typically happens because teams are more focussed on delivering functionality rather than looking after their architecture.

## Understanding “agility”

To understand how much agility you need from your software architecture, it’s worth looking at what agility means. John Boyd, a fighter pilot in the US Air Force, came up with a concept that he called the [OODA loop](#) - Observe, Orient, Decide and Act. In essence, this loop forms the basis for the decision making process. Imagine that you are a fighter pilot in a dogfight with an adversary. In order to outwit your opponent in this situation, you need to observe what’s happening, orient yourself (e.g. do some analysis), decide what to do and then act. In the heat of the battle, this loop needs to be executed as fast as possible to avoid being shot down by your opponent. Boyd then says that you can confuse and disorient your opponent if you can get inside their OODA loop, by which he means execute it faster than they can. If you’re more agile than your opponent, you’re the one that will come out on top.

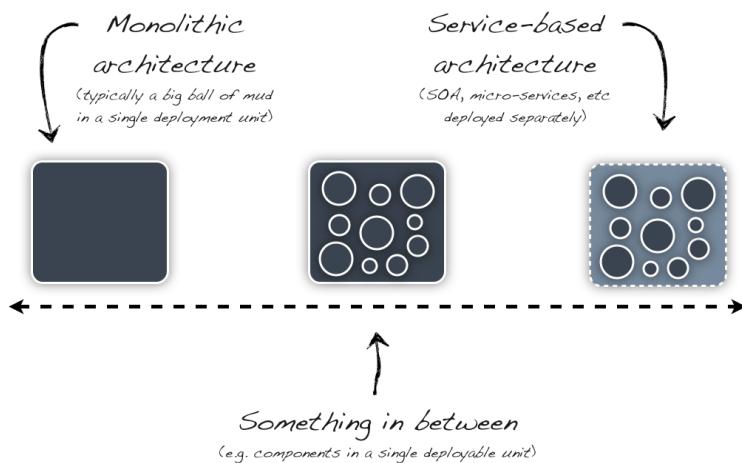
In a paper titled “[What Lessons Can the Agile Community Learn from A Maverick Fighter Pilot?](#)”, Steve Adolph, from the University of British Columbia, takes Boyd’s concept and applies it to software development. The conclusion drawn is that agility is relative and time-based. If your software team can’t deliver software and keep pace with changes in the environment, your team is not agile. If you’re working in a large, slow moving organisation

that rarely changes, you can probably take months to deliver software and still be considered “agile” by the organisation. In a lean startup, that’s likely to not be the case.

## A good architecture enables agility

The driver for having this discussion is that a good software architecture enables agility. Although [Service-Oriented Architecture \(SOA\)](#) is seen as a dirty term within some organisations due to over-complex, bloated and bodged implementations, there’s a growing trend of software systems being made up of tiny [microservices](#), where each service only does one thing but does that thing very well. A microservice may typically be less than one hundred lines of code. If change is needed, services can be rewritten from scratch, potentially in a different programming language. This style of architecture provides agility in a number of ways. Small, loosely coupled components/services can be built, modified and tested in isolation, or even ripped out and replaced depending on how requirements change. This style of architecture also lends itself well to a very flexible and adaptable deployment model, since new components/services can be added and scaled if needed.

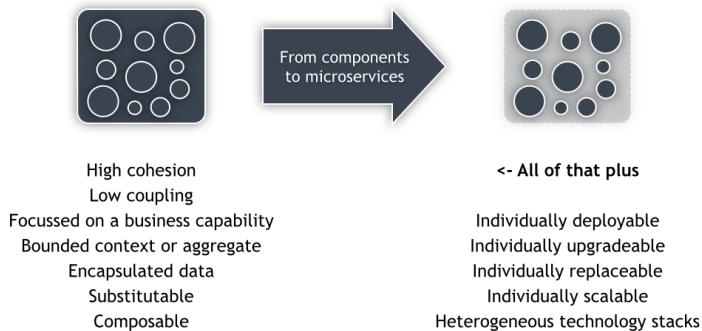
However, nothing in life is ever free. Building a software system like this takes time, effort and discipline. Many people don’t need this level of adaptability and agility either, which is why you see so many teams building software systems that are much more monolithic in nature, where everything is bundled together and deployed as a single unit. Although simpler to build, this style of architecture usually takes more effort to adapt in the face of changing requirements because functionality is often interwoven across the codebase.



### Different software architectures provide differing levels of agility

In my view, both architectural styles have their advantages and disadvantages, with the decision to build a monolithic system vs one composed of microservices coming back to the trade-offs that you are willing to make. As with all things in the IT industry, there's a middle ground between these extremes. With pragmatism in mind, you can always opt to build a software system that consists of a number of small well-defined components, yet is still deployed as a single unit. The Wikipedia page for [Component-based development](#) has a good summary and a “component” might be something like a risk calculator, audit logger, report generator, data importer, etc. The simplest way to think about a component is that it's a set of related behaviours behind an interface, which may be implemented using one or more collaborating classes (assuming an OO language). Good components share a number of characteristics with good classes and, of course, good microservices: high cohesion, low coupling, a well-defined public interface, good encapsulation, etc. Well-defined components provide a stepping stone to migrate to a microservice architecture more easily at a later date, if you need the benefits given the additional cost and complexity that such an architectural-style provides.

Well-defined, in-process components is a stepping stone to out-of-process components  
(i.e. microservices)



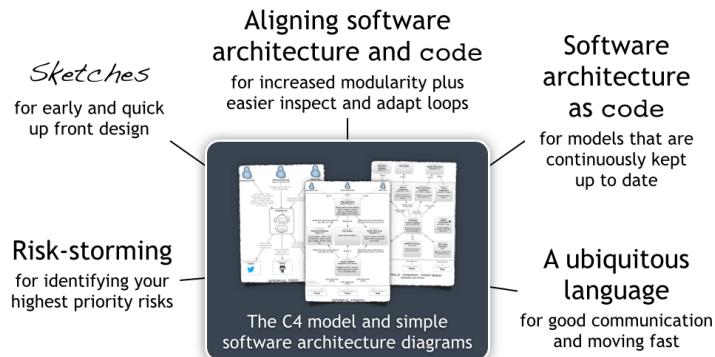
Well-defined, in-process components is a stepping stone to out-of-process components (i.e. microservices)

## Agility as a quality attribute

Understanding the speed at which your organisation or business changes is important because it can help you decide upon the style of architecture to adopt; whether that's a monolithic architecture, a microservices architecture or something in between. You need to understand the trade-offs and make your choices accordingly. Treat agility as a **quality attribute**. You don't get agility for free.

## Creating agile software systems in an agile way

Finally, and once you understand how much agility you need, is the process of architecting the solution and this is where **just enough up front design** and the essence of software architecture comes into play, at the heart of which is the **C4 model**.



## *Agility* and the **essence** of software architecture

(creating agile software systems in an agile way)

- **Sketches:** Simple software architecture sketches, based upon the C4 model, can help you visualise and communicate those early ideas quickly.
- **Ubiquitous language:** The C4 model provides a simple ubiquitous language that the whole team can use to communicate effectively and efficiently.
- **Aligning software architecture and code:** Aligning the software architecture model with the code, by adopting an architecturally-evident coding style, allows you to ensure architectural integrity and enforce modularity. This, in turn, makes your software easier to explain, understand and adapt.
- **Software architecture as code:** Representing the C4 model as code provides a way to keep those software architecture models continuously up to date, especially when architecturally-evident coding constructs are extracted from the code in an automated way.
- **Risk-storming:** Risk-storming provides a simple visual and collaborative way to identify the high-priority risks. The information from this exercise can then go into determining whether the architecture is fit for purpose and which, if any, concrete experiments need to be undertaken.

# 21. Introducing software architecture

A little software architecture discipline has a huge potential to improve the success of software teams, essentially through the introduction of 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 complex 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 from around the world 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.

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 software architecture books related to what I should do 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.

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.

## Some practical suggestions

Here are some practical suggestions for introducing software architecture.

### 1. Educate people

Simply 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.

### 2. Talk about architecture in retrospectives

If you have regular retrospectives to reflect on how your team is performing, why not simply include software architecture on the list of topics that you talk about? 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 with some [non-functional characteristics](#), then 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 consider architectural implications of the work item and conformance of the implementation with any desired architectural patterns/rules or non-functional goals.

### 4. Allocate the software architecture role to somebody

If you have a software team that doesn’t think about software architecture, simply 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 [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 coding 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 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.

Back to the original question and in essence the team was asking permission to spend some time doing the architecture stuff but they weren't getting buy-in from their management.

Perhaps their management didn'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.

Each approach tends to favour different situations, and again it depends on a number of factors. Coming back to the original question, it's possible that the first approach was used but either the message was weak or the management didn't think that mitigating the risks of not having any dedicated "architecture time" was 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 evolve 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 and before you know it you've probably started to make a world of difference. "It's easier to ask forgiveness than it is to get permission".

## The essence of software architecture

Many software teams are already using agile/lean approaches and more are following in their footsteps. For this reason, any software architecture practices adopted need to add real value otherwise the team is simply wasting time and effort. Only you can decide how much

software architecture 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!

## 22. Questions

1. Despite how agile approaches have been evangelised, are “agile” and “architecture” really in conflict with one another?
2. If you’re currently working on an agile software team, have the architectural concerns been thought about?
3. Do you feel that you have the right amount of technical leadership in your current software development team? If so, why? If not, why not?
4. How much up front design is enough? How do you know when you stop? Is this view understood and shared by the whole team?
5. Many software developers undertake coding katas to hone their skills. How can you do the same for your software architecture skills? (e.g. take some requirements plus a blank sheet of paper and come up with the design for a software solution)
6. What is a risk? Are all risks equal?
7. Who identifies the technical risks in your team?
8. Who looks after the technical risks in your team? If it’s the (typically non-technical) project manager or ScrumMaster, is this a good idea?
9. What happens if you ignore technical risks?
10. How can you proactively deal with technical risks?
11. Do you need to introduce software architecture into the way that your team works? If so, how might you do this?

# **VII Appendix A: Financial Risk System**

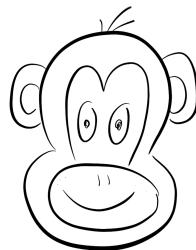
This is the financial risk system case study that is referred to throughout the book. It is also used during my [training course and workshops](#).

# VIII Appendix B: Software Guidebook for techtribes.je

[techtribes.je](#) is a side-project of mine to provide a focal point for the tech, IT and digital sector in Jersey, Channel Islands. The code behind the [techtribes.je](#) website is open source and available [on GitHub](#), while the [techtribes.je - Software Guidebook](#) can be downloaded for free from Leanpub.



## techtribes.je Software Guidebook



Simon Brown