coding {the} architecture

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

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 starting 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 waterfall-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 and I can be found on Twitter at @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 for further details.

Acknowledgements

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

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

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

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

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

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

I 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

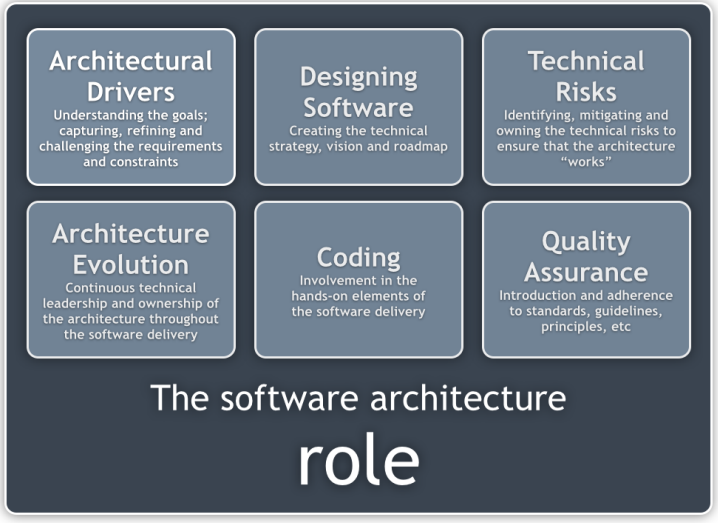
1. Do you know what “architecture” is all about? Does the rest of your team? What about the rest of your organisation?
2. There are a number of different types of architecture within the IT domain. What do they all have in common?
3. Do you and your team have a standard definition of what “software architecture” 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 to 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 necessary mean that the overall architecture will be successful. There's also the question of whether the technology choices you've made will actually work. Many teams 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. Should software architects code?

Since I created a website called Coding the Architecture, I guess the answer to this question shouldn't really come as a surprise. In my ideal view of the world, software architects *should* code. Somebody once told me that the key characteristic of a good architect is the ability to think in an abstract way. You can also think of it as the ability to not get caught up in the details all of the time. That's fine, but those boxes and lines that you're drawing do need to be coded at some point.

Writing code

My recommendation is to make coding a part of your role as a software architect. You can do this by simply being an integral part of the software development team. In other words, you have a software architecture hat and a coding hat. You don't need to be the best coder on the team, but the benefits of being hands-on and engaged in the delivery process are huge. After all, there's a difference between "knowing" and "doing".

Appreciating that you're going to be contributing to the coding activities often provides enough incentive to ensure that your designs are grounded in reality. If they aren't, you'll soon experience that pain once you understand the problems from a developer's perspective.

In addition to the obvious benefits associated with creating a software architecture that can actually be implemented by real people, contributing to the coding activities helps you build rapport with the rest of the team, which in turn helps to reduce the gap between architects and developers that you see on many software teams. To quote Rachel Davies and Liz Sedley from their Agile Coaching book:

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

Building prototypes, frameworks and foundations

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

As an alternative, you could help to build the frameworks and foundations that the rest of the team will use. Try to resist the temptation to build these things and then hand them over to the team because this approach may backfire. Software development is driven by fads and fashions much more than it really should be, so beware of building something that the rest of the team might deem to be a worthless pile of old-fashioned crap!

Performing code reviews

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

Experimenting and keeping up to date

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

Often you'll have more scope outside of work to maintain your coding skills; from contributing to an open source project through to continuously playing with the latest language/framework/API that takes your fancy. Books, blogs, podcasts, conferences and meetups will get you so far, but sometimes you just have to break out the code. I've certainly done this in the past and one of the upsides of a long commute on public transport is that it does give you time to

play with technology. Assuming you can keep your eyes open after a hard day at work, of course!

The tension between software architects and employers

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

Unfortunately many organisations seem to think that coding is the easy part of the software development process, which is why they see an opportunity to save some money and let somebody else do it, often in another country. Such organisations view cutting code as "low value". Tension therefore arises because there's a disconnect between the seniority of the software architect in the organisation and the value associated with the coding activities.

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

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

You don't have to give up coding

With this in mind, it should come as no surprise that I'm regularly asked whether software architects *can* continue to code if they are to climb the corporate career ladder. This is a shame, particularly if these people really enjoy the technical side of what they do.

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

As a software architect, you take on a great deal of responsibility for satisfying the non-functional requirements, performing technical quality assurance, making sure the software is fit for purpose, etc. It's a leadership role and coding (leading by example) is one of the very best ways to make sure the project is successful. And besides, if software architects don't remain technical, who is growing the software architects of tomorrow?

Don't code all the time

To reiterate my advice, software architects *don't* have to give up coding. However you do it, coding is a great way to keep your technology skills sharp in our ever-changing world. Many people believe that software architecture is a "post-technical" career choice but it requires deep technology skills alongside a breadth of experience and more general knowledge. It requires T-shaped people who are able to answer the question of whether their design will actually work. Leaving this as an "implementation detail" is not acceptable. Just don't code all of the time. If you're coding all of the time, who is performing the rest of the software architecture role?

9. Software architects should be master builders

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

State of the union

Software architecture has fallen out of favour over the past decade because of the association with "big up front design" and "analysis paralysis". Much of this stems from the desire to deliver software systems more efficiently, with agile approaches being a major catalyst in reducing the quantity of up front thinking that is performed by many teams. The net result is that the role of "an architect" is now often seen as redundant on software teams. Many teams are striving to be flat and self-organising, which on the face of it negates the need for a single dedicated point of technical leadership.

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

Back in time

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

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

This quote could equally be applied to us as software developers.

Did master builders actually build?

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

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

The Wikipedia page for architect says the same thing:

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

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

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

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

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

To add some additional detail:

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

Ivory towers?

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

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

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

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

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

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

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

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

Divergence of the master builder role

Much of this points to the idea that master builders didn't have much time for building, even though they possessed the skills to do so. To bring this back to the software industry; should software architects write code? My short answer is "ideally, yes" and the longer answer can be found here. Why? Because technology isn't an implementation detail and you need to understand the trade-offs of the decisions you are making.

So why don't modern building architects help with the actual process of hands-on building? To answer this, we need to look how the role has evolved over the years:

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

The Wikipedia page for structural engineering provides further information:

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

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

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

Building is now seen as an engineering discipline because of the huge body of knowledge behind it, which includes the laws of physics and being able to model/predict how materials will behave when they are used to construct buildings. By comparison, the software development industry is still relatively young and moves at an alarmingly fast pace. Buildings today are mostly built using the same materials as they were hundreds of years ago, but it seems like we're inventing a new technology every twenty minutes. We live in the era of "Internet time". Until our industry reaches the point where software can be built in the same way as a predictive engineering project, it's crucial that somebody on the team keeps up to date with technology and is able to make the right decisions about how to design software. In other words, software architects still need to play the role of structural engineer *and* architect.

Achieving the role

The final thing to briefly look at is how people achieved the role of master mason. From the Wikipedia page about stonemasonry:

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

This mirrors my own personal experience of moving into a software architecture role. It was an *evolutionary process*. Like many people, I started my career writing code under

the supervision of somebody else and gradually, as I gained more experience, started to take on larger and larger design tasks. Unlike the medieval building industry though, the software development industry lacks an explicit way for people to progress from being junior developers through to software architects. We don't have a common apprenticeship model.

Architects need to work with the teams

Herein lies the big problem for many organisations though - there aren't enough architects to go around. Although the master masons may not have had much time to work with stone themselves, they did work with the teams. I often get asked questions from software architects who, as a part of their role, are expected to provide assistance to a number of different teams. Clearly it's unrealistic to contribute to the hands-on elements of the software delivery if you are working with a number of different teams. You're just not going to have the time to write any code.

Performing a software architecture role across a number of teams is not an effective way to work though. Typically this situation occurs when there is a centralised group of architects (e.g. in an "Enterprise Architecture Group") who are treated as shared resources. From what I've read, the master masons were dedicated to a single building site at any one point in time and this is exactly the approach that we should adopt within our software development teams. If you think that this isn't possible, just take a look at how the medieval building industry solved the problem:

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

Again, it comes back to the strong apprenticeship model in place and this is exactly why coaching and mentoring should be a part of the modern software architecture role. We need to grow the software architects of tomorrow and every software development team needs their own master builder.

10. From developer to architect

The line between software development and software architecture is a tricky one. Some people will tell you that it doesn't exist and that architecture is simply an extension of the design process undertaken by developers. Others will make out it's a massive gaping chasm that can only be crossed by lofty developers who believe you must always abstract your abstractions and not get bogged down by those pesky implementation details. As always, there's a pragmatic balance somewhere in the middle, but it does raise the interesting question of how you move from one to the other.

Some of the key factors that are often used to differentiate software architecture from software design include an increase in scale, an increase in the level of abstraction and an increase in the significance of making the right design decisions. Software architecture is all about having a holistic view and seeing the "big picture" to understand how the software system works as a whole.

While this may help to differentiate software design and architecture, it doesn't necessarily help in understanding how a software developer moves into a software architecture role. Furthermore, it also doesn't help in identifying who will make a good software architect and how you go about finding them if you're hiring.

Experience is a good gauge but you need to look deeper

There are a number of different qualities that you need to look for in a software architect and their past experience is often a good gauge of their ability to undertake the role. Since the role of a software architect is varied though, you need to look deeper to understand the level of involvement, influence, leadership and responsibility that has been demonstrated across a number of different areas. In conjunction with my definition of the software architecture role, each of the parts can and should be evaluated independently. After all, the software design process seems fairly straightforward. All you have to do is figure out what the requirements are and design a system that satisfies them. But in reality it's not that simple and the software architecture role undertaken by somebody can vary wildly. For example:

1. **Architectural drivers**: capturing and challenging a set of complex non-functional requirements versus simply assuming their existence.

2. **Designing software**: designing a software system from scratch versus extending an existing one.
3. **Technical risks**: proving that your architecture will work versus hoping for the best.
4. **Architecture evolution**: being continually engaged and evolving your architecture versus choosing to hand it off to an "implementation team".
5. **Coding**: being involved in the hands-on elements of the delivery versus watching from the sidelines.
6. **Quality assurance**: assuring quality and selecting standards versus being reviewed against them or doing nothing.

Much of this comes down to the difference between taking responsibility for a solution versus assuming that it's not your problem.

The line is blurred

Regardless of whether you view the line between software development and architecture as a mythical one or a gaping chasm, people's level of experience across the software architecture role varies considerably. Furthermore, the line between software development and software architecture is blurred somewhat. Most developers don't wake up on a Monday morning and declare themselves to be a software architect. I certainly didn't and my route into software architecture was very much an evolutionary process. Having said that, there's a high probability that many software developers are *already* undertaking parts of the software architecture role, irrespective of their job title.

Crossing the line is our responsibility

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

11. Broadening the T

Despite what you may hear some people say, software architecture is *not* a "post-technical" or "non-technical" career. Drawing a handful of boxes, lines and clouds on a whiteboard and handing that off as a "software design" is not what the role is all about.

Deep technology skills

"T" is for technology, and this is exactly what good software architects need to know about. As software developers, we tend to have knowledge about things like programming language syntax, APIs, frameworks, design patterns, automated unit testing and all of the other low-level technical stuff that we use on a daily basis. And this is the same basic knowledge that software architects need too. Why? Because people in the software architecture role need to understand technology, basically so that they are able to honestly answer the following types of questions:

- Is this solution going to work?
- Is *that* how we are going to build it?

However, given the learning curve associated with being productive in different programming languages, it's common for software professionals to only know one or two technologies really well. Eventually these people get known for being a "Java developer" or an "Oracle developer", for example. I've been there and done that myself. I've also seen it happen in many other organisations. If you've ever wondered why there are so many religious battles related to programming languages, you need look no further for how many of these start.

Although we may *strive* to be open-minded, it's easy to get siloed into a single technology stack. While there's nothing particularly wrong with this, you have to be careful that you do keep an open mind. As the saying goes, "if all you have is a hammer, everything will look like a nail". Gaining experience is an essential part of the learning journey, but that same experience shouldn't constrain you. As an example, you don't *need* a relational database for every software system, but often this is the first thing that gets drawn when a team is sketching out a candidate software architecture.

Breadth of knowledge

And that brings me onto why it's important for software architects to have a breadth of technology knowledge too. Sure, they may be specialists in Java or Oracle, but the role demands more. For example, the people in the software architecture role should be able to answer the following types of questions too:

- Is the technology that we've chosen the most appropriate given the other options available?
- What are the other options for the design and build of this system?
- Is there a common architectural pattern that we should be using?
- Do we understand the trade-offs of the decisions that we're making?
- Have we catered for the desired quality attributes?
- How can we prove that this architecture will work?

Software architects are generalising specialists

Most of the best software architects I know have come from a software development background. This doesn't mean that they are the best coders on a team, but they *are* able to switch between the low-level details and the big picture. They also have a deep technical specialism backed-up with a breadth of knowledge that they've gained from years of experience in building software. But they can't (and don't) always know everything. Plus it's rare to find a software system that only uses a single technology stack. Some examples of systems with heterogeneous technology stacks that I've seen during my career include:

- A Microsoft .NET desktop client running against a number of Oracle databases.
- A Microsoft ASP.NET website pulling data from an Oracle database via a collection of Java EE web services.
- iOS and Android mobile apps pulling data from RESTful services written in Java.
- A micro-service style architecture, where various services were built using Microsoft .NET or Ruby.
- A Microsoft ASP.NET website pulling data from a Microsoft Dynamics CRM system.
- A Microsoft SharePoint website pulling data from a collection of databases via Microsoft .NET/Windows Communication Foundation services.
- A Java EE web application integrating with SAP.

- etc

Although general design knowledge, techniques, patterns and approaches often apply to a number of different technologies, not understanding how to apply them successfully at a low-level of detail can cause issues. Does this mean that the software architect should be an expert in all of the technologies that are in use on any give software system? No. Instead collaboration is key. Find somebody that *does* understand the things you don't and work with them closely. Nothing says that the software architecture role can't be shared, and often appreciating the gaps in your own knowledge is the first step to creating a more collaborative working environment. Pair programming has benefits, so why not pair architecting?

Software architecture is a technical career

Technology knowledge underpins the software architecture role, requiring a combination of deep technology skills coupled with broader knowledge. If the people designing the software and drawing the architecture diagrams can't answer the question of whether the architecture will work, they are probably the wrong people to be doing that job. Software architecture is most definitely a technical career but that's not the end of the story. Good soft skills are vital too.

12. Soft skills

The majority of the discussion about the software architecture role in this book relates back to having technical depth and breadth. But this is really only half the story. Since we're essentially talking about a leadership role, "soft skills" or "people skills" are vitally important too.

- **Leadership**: In it's simplest form, leadership is the ability to create a shared vision and to then take people on a journey to satisfy the common goal.
- **Communication**: You can have the best ideas and vision in the world, but you're dead in the water if you're unable to effectively communicate this to others. This means people inside *and* outside of the software development team, using the language and level of detail that is appropriate to the audience.
- **Influencing**: This is a key leadership skill and can be done a number of ways, from overt persuasion through to Neuro-Linguistic Programming or Jedi mind tricks. Influencing people can also be done through compromise and negotiation. Individuals may have their own ideas and agendas, which you need to deal with while keeping everybody "on-side" and motivated to get the result that you need. Good influencing requires good listening and exploring skills too.
- **Confidence**: Confidence is important, underpinning effective leadership, influence and communication. Confidence doesn't mean arrogance though.
- **Collaboration**: The software architecture role shouldn't be done in isolation, and collaboration (working with others) to come up with a better solution is a skill that is worth practicing. This means listening, being open-minded and responsive to feedback.
- **Coaching**: Not everybody will have experience in what you're trying to do and you'll need to coach people on their role, technologies, etc.
- **Mentoring**: Mentoring is about facilitating somebody's learning rather than telling them how to do something. As a leader you may be asked to mentor others on the team.
- **Motivation**: This is about keeping the team happy, up-beat and positive. The team also needs to feel motivated to follow any vision that you create as a software architect. You will face an uphill battle without the rest of the team's buy-in.
- **Facilitation**: There will often be times where you need to step back and facilitate discussions, particularly where there is conflict within the team. This requires exploration, objectivity and helping the team come up with a solution to build consensus.

- **Political**: There are always politics at play in every organisation. My mantra is to steer clear of getting involved as far as possible, but you should at least understand what's going on around you so that you can make more informed decisions.
- **Responsibility**: You can't necessarily blame the rest of the software development team for failure and it's important that you have a sense of responsibility. It's *your* problem if your software architecture doesn't satisfy the business goals, deliver the non-functional requirements or the technical quality is poor.
- **Delegation**: Delegation is an important part of any leadership role and there's a fine line between delegating everything and doing everything yourself. You should learn to delegate where appropriate but remember that it's not the *responsibility* you're delegating.

Stay positive

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

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

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

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

13. 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”.

14. Software architecture introduces control?

Software architecture introduces structure and vision into software projects but is it also about introducing control? And if so, is control a good thing or a bad thing?

Provide guidance, strive for consistency

Many of the practices associated with software architecture are about the introduction of guidance and consistency into software projects. If you've ever seen software systems where a common problem or cross-cutting concern has been implemented in a number of different ways, then you'll appreciate why this is important. A couple of examples spring to mind; I've seen a software system with multiple object-relational mapping (ORM) frameworks in a single codebase and another where components across the stack were configured in a number of different ways, ranging from the use of XML files on disk through to tables in a database. Deployment and maintenance of both systems was challenging to say the least.

Guidance and consistency can only be realised by introducing a degree of control and restraint, for example, to stop team members going off on tangents. You can't have people writing database access code in your web pages if you've specifically designed a distributed software system in order to satisfy some of your key non-functional requirements. Control can also be about simply ensuring a clear and consistent structure for your codebase; appropriately organising your code into packages, namespaces, components, layers, etc.

How much control do you need?

The real question to be answered here relates to the amount of control that needs to be introduced. At one end of the scale you have the dictatorial approach where nobody can make a decision for themselves versus the other end of the scale where nobody is getting any guidance whatsoever. I've seen both in software projects and I've taken over chaotic projects where everybody on the team was basically left to their own devices. The resulting codebase was, unsurprisingly, a mess. Introducing control on this sort of project is really hard work but it needs to be done if the team is to have any chance of delivering a coherent piece of software that satisfies the original drivers.

Control varies with culture

I've also noticed that different countries and cultures place different values on control. Some (e.g. the UK) value control and the restraint that it brings whereas others (e.g. Scandinavia) value empowerment and motivation. As an example of what this means in the real world, it's the difference between having full control over all of the technologies used on a software project (from the programming language right down to the choice of logging library) through to being happy for *anybody* in the team make those decisions.

A lever, not a button

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

- Are the team experienced?
- Has the team worked together before?
- How large is the team?
- How large is the project?
- Are the project requirements complex?
- Are there complex non-functional requirements or constraints that need to be taken into account?
- What sort of discussions are happening on a daily basis?
- Does the team or the resulting codebase seem chaotic already?
- etc

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

15. Mind the gap

Our industry has a love-hate relationship with the software architecture role, with many organisations dismissing it because of their negative experience of architects who dictate from "ivory towers" and aren't engaged with the actual task of building working software. This reputation is damaging the IT industry and inhibiting project success. Things need to change.

Developers focus on the low-level detail

If you're working on a software development project at the moment, take a moment to look around at the rest of the team. How is the team structured? Does everybody have a well defined role and responsibilities? Who's looking after the big picture; things like performance, scalability, availability, security and so on?

We all dream about working on a team where everybody is equally highly experienced and is able to think about the software at all levels, from the code right through to the architectural issues. Unfortunately the real world just isn't like that. Most of the teams that I've worked with are made up of people with different levels of experience, some of who are new to IT and others who have "been around the block a few times". As software developers, the code is our main focus but what happens if you have a team that *only* focusses on this low-level detail? Imagine a codebase where all of the latest programming language features are used, the code is nicely decoupled and testing is completely automated. The codebase can be structured and formatted to perfection but that's no use if the system has scalability issues when deployed into a live environment.

Architects dictate from their ivory towers

The software architecture role is different to a developer role. Some people view it as a step up from being a developer, and some view it as a step sideways. However you view it, the architecture role is about looking after "the big picture". Many teams do understand the importance of software architecture but will bring in somebody with the prestigious title of "Architect" only to stick them on a pedestal above the rest of the team. If anything, this instantly isolates the architect by creating an exaggerated gap between them and the development team they are supposed to be working with.

Reducing the gap

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

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

Fortunately, there are some simple ways to address this problem, from both sides. Software development is a team activity after all.

If you're a software architect:

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

If you're a software developer:

- **Understand the big picture**: Taking some time out to understand the big picture will help you understand the context in which the architectural decisions are being made and enhance your understanding of the system as a whole.

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

A collaborative approach to software architecture

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

16. Where are the software architects of tomorrow?

Agile and software craftsmanship are two great examples of how we're striving to improve and push the software industry forward. We spend a lot of time talking about writing code, automated testing, automated deployment, tools, technologies and all of the associated processes. And that makes a lot of sense because the end-goal here is delivering benefit to people through software, and working software is key. But we shouldn't forget that there are some other aspects of the software development process that few people genuinely have experience with. Think about how *you* would answer the following questions.

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

Let's face it, most software developers don't get to take a blank sheet of paper and design software from scratch all that frequently, regardless of whether that design is up front or evolutionary and whether it's a solo or collaborative exercise.

Coaching, mentoring and apprenticeships

Coaching and mentoring is an overlooked activity on most software development projects, with many team members not getting the support that they need. While technical leadership is about guiding the project as a whole, there are times when individuals need assistance. In addition to this, coaching and mentoring provides a way to enhance people's skills and to help them improve their own careers. Sometimes this assistance is of a technical nature, and sometimes it's more about the softer skills. From a technical perspective though, why shouldn't the people undertaking the software architecture role help out with the coaching and mentoring? Most architects that I know have got to where they are because they have a great deal of experience in one or more technical areas. If this is the case, why shouldn't those architects share some of their experience to help others out? An apprenticeship model is exactly how the master builders of times past kept their craft alive.

We're losing our technical mentors

The sad thing about our industry is that many developers are being forced into non-technical management positions in order to progress their careers up the corporate ladder. Ironically, it's often the best and most senior technical people who are forced away, robbing software teams of their most valued technical leads, architects and mentors. Filling this gap tomorrow are the developers of today.

Software teams need downtime

Many teams have already lost their most senior technical people, adding more work to the remainder of the team who are already struggling to balance all of the usual project constraints along with the pressures introduced by whatever is currently fashionable in the IT industry … agile, craftsmanship, cloud, rich Internet UIs, functional programming, etc. Many teams appreciate that they should be striving for improvement, but lack the time or the incentive to do it.

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

Experience of coding is easy to pick up and there are plenty of ways to practice this skill. Designing something from scratch that will be implemented by a team isn't something

that you find many teams teaching or practicing though. With many technical mentors disappearing thanks to the typical corporate career ladder, where do developers gain this experience? Where *are* the software architects of tomorrow going to come from?

17. Everybody is an architect, except when they're not

Many software teams strive to be agile and self-organising yet it's not immediately obvious how the software architecture role fits into this description of modern software development teams.

Everybody is an architect

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

Imagine if you did have a team of experienced software developers that were all able to switch in and out of the big picture. A team of genuinely hands-on architects. That team would be amazing and all of the elements you usually associate with software architecture (non-functional requirements, constraints, etc) would all get dealt with and nothing would slip through the gaps. From a *technical perspective*, this is a self-organising team.

Except when they're not

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

In "Notes to a software team leader", Roy Osherove describes his concept of "Elastic Leadership" where the leadership style needs to vary in relation to the maturity of the team.

Roy categorises the maturity of teams using a simple model, with each level requiring a different style of leadership.

1. **Survival model (chaos)**: requires a more direct, command and control leadership style.
2. **Learning**: requires a coaching leadership style.
3. **Self-organising**: requires facilitation to ensure the balance remains intact.

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

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

Does agile need architecture?

Unfortunately, many teams view the "big picture" technical skills as an unnecessary evil rather than an essential complement, probably because they've been burnt by big design up front in the past. Some are also so focussed on the desire to be "agile" that other aspects of the software development process get neglected. Chaos rather than self-organisation ensues yet such teams challenge the need for a more direct leadership approach. After all, they're striving to be agile and having a single point of responsibility for the technical aspects of the project conflicts with their view of what an agile team should look like. This conflict tends to cause people to think that agile and architecture are opposing forces - you can have one or the other but not both. It's not architecture that's the opposing force though, it's big design up front.

Agile software projects still need architecture because all those tricky concerns around complex non-functional requirements and constraints don't go away. It's just the execution of the architecture role that differs.

With collective code ownership, everybody needs to be able to work at the architecture level and so everybody *is* an architect to some degree. Teams that aren't at the self-organising stage will struggle if they try to run too fast though. Despite people's aspirations to be agile, collective code ownership and a distribution of the architecture role are likely to hinder chaotic teams rather than help them. Chaotic teams need a more direct leadership approach and they will benefit from a single point of responsibility for the technical aspects of the software project. In other words, they will benefit from a single person looking after the software architecture role. Ideally this person will coach others so that they too can help with this role.

One software architect or many? Single point of responsibility or shared amongst the team? Agile or not, the software architecture role exists. Only the context will tell you the right answer.

18. Software architecture as a consultant

Most of my career has been spent working for IT consulting companies, where I've either built software systems *for* our customers under an outsourcing arrangement or *with* our customers as a part of a mixed customer-supplier team (this is often called "body-shopping"). Although undertaking the software architecture role within a consulting context is fundamentally the same as undertaking the role in any other context, there are some potential gotchas to be aware of.

Domain knowledge

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

And this raises an interesting question. A deep knowledge of the business domain only comes from working within that domain for an extended period of time but most consultants move between different customers, teams and business domains on a regular basis. Is it therefore fair to expect consultants to possess deep domain knowledge?

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

usually saw their deep business domain knowledge as a key differentiator or unique selling point when compared to other consultants.

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

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

Authority

The degree of control that the software architecture role needs to introduce depends on the type of software development team that you work with. Often the team can present another set of challenges though, especially if you're working as a consultant software architect with a team of your customer's in-house developers.

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

The software architecture role is about technical leadership and part of this means that you need to get the whole team heading in the same direction. Dictating instructions to a team of software developers isn't likely to be very effective if you're not their immediate line manager, which is often the case if you're supplementing a customer team. This is where the soft skills come into play, particularly those related to building relationships, creating trust and motivating the team. I've found that being a hands-on, coding architect goes a long way to getting a successful outcome too.

19. 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.

20. 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.

21. Quality Attributes (non-functional requirements)

When you're gathering requirements, people will happily give you a wish-list of what they want a software system to do and there are well established ways of capturing this information as user stories, use cases, traditional requirements specifications, acceptance criteria and so on. What about those pesky "non-functional requirements" though?

Non-functional requirements are often thought of as the "-ilities" and are primarily about quality of service. Alternative, arguably better yet less commonly used names for non-functional requirements include "system characteristics" or "quality attributes". A non-exhaustive list of the common quality attributes is as follows.

Performance

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

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

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

Scalability

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

Availability

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

Security

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

Disaster Recovery

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

Accessibility

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

Monitoring

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

Management

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

Audit

There's often a need to keep a log of events (i.e. an audit log) that led to a change in data or behaviour of a software system, particularly where money is involved. Typically such logs need to capture information related to who made the change, when the change was made and why the change was made. Often there is a need retain the change itself too (i.e. before and after values).

Flexibility

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

Extensibility

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

Maintainability

Maintainability is often cited as a requirement but what does this actually mean? As software developers we usually strive to build "maintainable" software but it's worth thinking about who will be maintaining the codebase in the future. Maintainability is hard to quantify, so I'd rather think about the architecture and development principles that we'll be following instead because they are drivers for writing maintainable code.

Legal, Regulatory and Compliance

Some industries are strictly governed by local laws or regulatory bodies, and this can lead to additional requirements related to things like data retention or audit logs. As an example, most finance organisations (investment banks, retail banks, trust companies, etc) must adhere to a number of regulations (e.g. anti-money laundering) in order to retain their ability to operate in the market.

Internationalisation (i18n)

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

Localisation (L10n)

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

Which are important to you?

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

22. Working with non-functional requirements

Regardless of what you call them, you'll often need to put some effort into getting the list of non-functional requirements applicable to the software system that you're building.

Capture

I've spent most of my 15+ year career in software development working in a consulting environment where we've been asked to build software for our customers. In that time, I can probably count on one hand the number of times a customer has explicitly given us information about the non-functional requirements. I've certainly received a large number of requirements specifications or functional wish-lists, but rarely do these include any information about performance, scalability, security, etc. In this case, you need to be proactive and capture them yourself.

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

Refine

Once you've started asking those tricky questions related to non-functional requirements, or you've been fortunate enough to receive *some* information about them, you'll probably need to refine them.

On the few occasions that I've received a functional requirements specification that did include some information about non-functional requirements, they've usually been unhelpfully vague. As an example, I once received a 125 page document from a potential customer that detailed the requirements of the software system. The majority of the pages covered the functional requirements in quite some detail and the last half page was reserved for the non-functional requirements. It said things like:

- **Performance**: The system must be fast.

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

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

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

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

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

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

Challenge

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

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

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

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

23. Constraints

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

Time and budget constraints

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

Technology constraints

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

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

- **Target deployment platform**: The target deployment platform is usually one of the major factors that influences the technology decisions you make when building a greenfield software system. This includes embedded devices, the availability of Microsoft Windows or Linux servers and the cloud. Yes, even this magical thing that we call the cloud has constraints. As an example, every "platform as a service" (PaaS) offering is different, and most have restrictions on what your software can and can't do with things like local disk access. If you don't understand these constraints, there's a huge danger that you'll be left with some anxious rework when it comes to deployment time.
- **Technology maturity**: Some organisations are happy to take risks with bleeding edge technology, embracing the risks that such advancements bring. Other organisations are much more conservative in nature.
- **Open source**: Likewise, some organisations still don't like using open source unless it has a name such as IBM or Microsoft associated with it. I once worked on a project for a high-street bank who refused to use open source, yet they were happy to use a web server from a very well-known technology brand. The web server was the open source Apache web server in disguise. Such organisations simply like having somebody to shout at when things stop working. Confusion around open source licenses also prevents some organisations from fully adopting open source too. You may have witnessed this if you're ever tried to explain the difference between GPL and LGPL.
- **Vendor "relationships"**: As with many things in life, it's not what you know, it's *who* you know. Many partnerships are still forged on the golf course by vendors who wine and dine Chief Technology Officers. If you've ever worked for a large organisation and wondered why your team was forced to use something that was obviously substandard, this might be the reason!
- **Past failures**: Somewhere around the year 2000, I walked into a bank with a proposal to build them a solution using Java RMI - a technology to allow remote method calls across Java virtual machines. This was met with great resistance because the bank had "tried it before and it doesn't work". That was the end of that design and no amount of discussion would change their mind. Java RMI was banned in this environment due to a past failure. We ended up building a framework that would send serialized Java objects over HTTP to a bunch of Java Servlets instead (a workaround to reinvent the same wheel).
- **Internal intellectual property**: When you need to find a library or framework to solve some problem that you're facing, there's a high probability that there's an open source or commercial product out there that suits your needs. This isn't good enough for some people though and it's not uncommon to find organisations with their own internal logging libraries, persistence frameworks or messaging infrastructures that you *must*

use, despite whether they actually work properly. I recently heard of one organisation that built their own CORBA implementation.

People constraints

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

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

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

Organisational constraints

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

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

Are all constraints bad?

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

Constraints can be prioritised

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

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

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

Listen to the constraints

Every software system will be subject to one or more constraints, and part of the software architecture role is to seek these out, understand why they are being imposed and let them help you shape the software architecture. Failing to do this may lead to some nasty surprises.

24. Principles

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

Development principles

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

- **Coding standards and conventions**: "We will adopt our in-house coding conventions for [Java|C#|etc], which can be found on our corporate wiki."
- **Automated unit testing**: "Our goal is to achieve 80% code coverage for automated unit tests across the core library, regardless of whether that code is developed using a test-first or test-last approach."
- **Static analysis tools**: "All production and test code must pass the rules defined in [Checkstyle|FxCop|etc] before being committed to source code control."
- **etc**

Architecture principles

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

- **Layering strategy**: A layered architecture usually results in a software system that has a high degree of flexibility because each layer is isolated from those around it. For example, you may decompose your software system into a UI layer, a business layer and a data access layer. Making the business layer completely independent of the data access layer means that you can (typically) switch out the data access implementation without affecting the business or UI layers. You can do this because the

data access layer presents an abstraction to the business layer rather than the business layer directly dealing with the data storage mechanism itself. If you want to structure your software this way, you should ensure that everybody on the development team understands the principle. "No data access logic in the UI components or domain objects" is a concrete example of this principle in action.

- **Placement of business logic**: Sometimes you want to ensure that business logic always resides in a single place for reasons related to performance or maintainability. In the case of Internet-connected mobile apps, you might want to ensure that as much processing as possible happens on the server. Or if you're integrating with a legacy back-end system that already contains a large amount of business logic, you might want to ensure that nobody on the team attempts to duplicate it.
- **High cohesion**, **low coupling**, SOLID, **etc**: There are many principles related to the separation of concerns, focussing on building small highly cohesive building blocks that don't require too many dependencies in order to do their job.
- **Stateless components**: If you're building software that needs to be very scalable, then designing components to be as stateless as possible is one way to ensure that you can horizontally scale-out your system by replicating components to share the load. If this is your scalability strategy, everybody needs to understand that they must build components using the same pattern. This will help to avoid any nasty surprises and scalability bottlenecks in the future.
- **Stored procedures**: Stored procedures in relational databases are like Marmite - you either love them or you hate them. There are advantages and disadvantages to using or not using stored procedures, but I do prefer it when teams just pick one approach for data access and stick to it. There are exceptions to every principle though.
- **Domain model - rich vs anaemic**: Some teams like having a very rich domain model in their code, building systems that are very object-oriented in nature. Others prefer a more anaemic domain model where objects are simply data structures that are used by coarse-grained components and services. Again, consistency of approach goes a long way.
- **Use of the HTTP session**: If you're building a website, you may or may not want to use the HTTP session for storing temporary information between requests. This can often depend on a number of things including what your scaling strategy is, where session-backed objects are actually stored, what happens in the event of a server failure, whether you're using sticky sessions, the overhead of session replication, etc. Again, everybody on the development team should understand the desired approach and stick to it.
- **Always consistent vs eventually consistent**: Many teams have discovered that they often need to make trade-offs in order to meet complex non-functional requirements.

For example, some teams trade-off data consistency for increased performance and/or scalability. Provided that we *do* see all Facebook status updates, does it really matter if we all don't see them *immediately*? Your context will dictate whether immediate or delayed consistency is appropriate, but a consistent approach is important.

Beware of "best practices"

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

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

25. Technology is not an implementation detail

I regularly run training classes where I ask small groups of people to design a simple financial risk system. In most cases, the resulting software architecture diagrams don't show any technology decisions. Here are some of the typical responses to the question of why this is the case:

- "the [risk system] solution is simple and can be built with any technology".
- "we don't want to force a solution on developers".
- "it's an implementation detail".
- "we follow the 'last responsible moment' principle".

I firmly believe that technology choices should be included on architecture diagrams but there's a separate question here about why people don't feel comfortable making technology decisions. Saying that "it *can* be built with any technology" doesn't mean that it *should*. Here's why.

1. Do you have complex non-functional requirements?

True, most software systems *can* be built with pretty much any technology; be it Java, .NET, Ruby, Python, PHP, etc. If you look at the data storage requirements for most software systems, again, pretty much any relational database is likely to be able to do the job. Most software systems are fairly undemanding in terms of the non-functional characteristics, so any mainstream technology is likely to suffice.

But what happens if you *do* have complex non-functional requirements such as high performance and/or scalability? Potentially things start to get a little trickier and you should really understand whether your technology (and architecture) choices are going to work. If you don't consider your non-functional requirements, there's a risk that your software system won't satisfy its goals.

2. Do you have constraints?

Many organisations have constraints related to the technologies that can be used and the skills (people) that are available to build software. Some even dictate that software should be bought and/or customised rather than built. Constraints can (and will) influence the software architecture that you come up with. Challenge them by all means, but ignore them and you risk delivering a software system that doesn't integrate with your organisation's existing IT environment.

3 Do you want consistency?

Imagine you're building a software system that stores data in a relational database. Does it matter how individual members of the development team retrieve data from and store data to the database when implementing features? I've seen a Java system where there were multiple data access techniques/frameworks adopted in the same codebase. And I've seen a SharePoint system where various components were configured in different ways. Sometimes this happens because codebases evolve over time and approaches change, but often it's simply a side-effect of everybody on the development team having free rein to choose whatever technology/framework/approach they are most familiar with.

People often ask me questions like, "does it really matter which logging framework we choose?". If you want everybody on the development team to use the same one, then yes, it does. Some people are happy to allow anybody on the development team to download and use any open source library that they want to. Others realise this *can* lead to problems if left unchecked. I'm not saying that you should stifle innovation, but you should really only have a single logging, dependency injection or object-relational mapping framework in a codebase.

A lack of a consistent approach can lead to a codebase that is hard to understand, maintain and enhance. Increasing the number of unique moving parts can complicate the deployment, operation and support too.

Deferral vs decoupling

It's worth briefly talking about deferring technology decisions and waiting until "the last responsible moment" to make a decision. Let's imagine that you're designing a software system where there aren't any particularly taxing non-functional requirements or constraints.

Does it matter which technologies you choose? And shouldn't a good architecture let you change your mind at a later date anyway?

Many people will say, for example, that it really doesn't matter which relational database you use, especially if you decouple the code that you write from a specific database implementation using an object-relational mapping layer such as Hibernate, Entity Framework or ActiveRecord. If you don't have any significant non-functional requirements or constraints, and you truly believe that all relational databases are equal, then it probably doesn't matter which you use. So yes, you *can* decouple the database from your code and defer the technology decision. But don't forget, while your choice of database is no longer a significant decision, your choice of ORM is. You can decouple your code from your ORM by introducing another abstraction layer, but again you've made a significant decision here in terms of the structure of your software system.

Decoupling is a great approach for a number of reasons plus it *enables* technology decisions to be deferred. Of course, this doesn't mean that you *should* defer decisions though, especially for reasons related to the presence of non-functional requirements and constraints.

Every decision has trade-offs

Much of this comes back to the fact that every technology has it's own set of advantages and disadvantages, with different options not necessarily being swappable commodities. Relational database and web application frameworks are two typical examples of a technology space that is often seen as commoditised. Likewise with many cloud providers, but even these have their trade-offs related to deployment, monitoring, management, cost, persistent access to disk and so on.

At the end of the day, every technology choice you make will have a trade-off whether that's related to performance, scalability, maintainability, the ability to find people with the right experience, etc. Understanding the technology choices can also assist with high-level estimating and planning, which is useful if you need to understand whether you can achieve your goal given a limited budget.

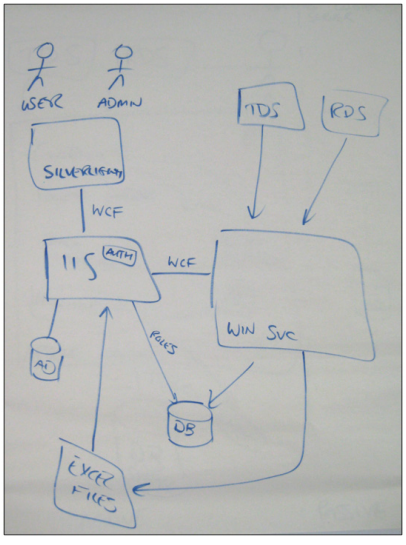
If you don't understand the trade-offs that you're making by choosing technology X over Y, you shouldn't be making those decisions. It's crucial that the people designing software systems understand technology. This is why software architects should be master builders.

Technology isn't just an "implementation detail" and the technology decisions that you make are as important as the way that you decompose, structure and design your software system. Defer technology decisions at your peril.

26. More layers = more complexity

One of the key functional requirements of the financial risk system case study that we run through on the training course is that the solution should be able to distribute data to a subset of users on a corporate LAN. Clearly there are 101 different ways to solve this problem, with one of the simplest being to allow the users to access the data via an internal web application. Since only a subset of the users within the organisation should be able to see the data, any solution would need some sort of authentication and authorisation on the data.

Given the buzz around Web 2.0 and Rich Internet Applications in recent times, one of the groups on a training course decided that it would be nice to allow the data to be accessed via a Microsoft Silverlight application. They'd already thought about building an ASP.NET application but liked the additional possibilities offered by Silverlight, such as the ability to slice and dice the data interactively. Another driving factor for their decision was that the Silverlight client could be delivered "for free" in that it would take "just as long to build as an ASP.NET application". "For free" is a pretty bold claim, especially considering that they were effectively adding an extra architectural layer into their software system. I sketched up the following summary of their design to illustrate the added complexity.



While I don't disagree that Silverlight applications aren't hard to build, the vital question the group hadn't addressed was where the data was going to come from. As always, there are options; from accessing the database directly through to exposing some data services in a middle-tier. The group had already considered deploying some Windows Communication Foundation (WCF) services into the IIS web server as the mechanism for exposing the data, but this led to yet further questions.

1. What operations do you need to expose to the Silverlight client?
2. Which technology binding and protocol would you use?
3. How do you ensure that people can't plug in their own bespoke WCF client and consume the services?
4. How do you deploy and test it?
5. etc

Non-functional requirements

In the context of the case study, the third question is important. The data should only be accessible by a small number of people and we really don't want to expose a web service that anybody with access to a development tool could consume.

Most security conscious organisations have their self-hosted public facing web servers firewalled away in a DMZ, yet I've seen some software systems where those same secured web servers subsequently access unsecured web services residing on servers within the regular corporate LAN. Assuming that I can connect a laptop to the corporate LAN, there's usually nothing to stop me firing up a development tool such as Microsoft Visual Studio, locating the service definition (e.g. a WSDL file) and consuming the web service for my own misuse. In this case, thought needs to be given to authentication and authorisation of the data service as well as the Silverlight client. A holistic view of security needs to be taken.

Time and budget - nothing is free

Coming back to the claim that building a Silverlight client won't take longer than building an ASP.NET application; this isn't true because of the additional data services that need to be developed to support the Silverlight client. In this situation, the benefits introduced by the additional rich client layer need to be considered on the basis that additional complexity is also being introduced. All architecture decisions involve trade-offs. More moving parts means

more work designing, developing, testing and deploying. Despite what vendor marketing hype might say, nothing is ever free and you need to evaluate the pros and cons of adding additional layers into a design, particularly if they result in additional inter-process communication.

27. Collaborative design can help and hinder

Let's imagine that you've been tasked with building a 3-tier web application and you have a small team that includes people with specialisms in web technology, server-side programming and databases. From a resourcing point of view this is excellent because collectively you have experience across the entire stack. You shouldn't have any problems then, right?

The effectiveness of the overall team comes down to a number of factors, one of them being people's willingness to leave their egos at the door and focus on delivering the best solution given the context. Sometimes though, individual specialisms can work against a team; simply through a lack of experience in working as a team or because ego gets in the way of the common goal. If there's a requirement to provide a way for a user to view and manipulate data on our 3-tier web application, you'll probably get a different possible approach from each of your specialists.

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

Experience influences software design

Our own knowledge, experience and preferences tend to influence how we design software, particularly if it's being done as a solo activity. In the absence of communication, we tend to make assumptions about where components will sit and how features will work based upon our own mental model of how the sofware should be designed. Getting these assumptions out into the open as early as possible can really help you avoid some nasty surprises before

it's too late. One of the key reasons I prefer using a whiteboard to design software is because it encourages a more collaborative approach than somebody sitting on their own in front of their favourite modelling tool on a laptop. If you're collaborating, you're also communicating and challenging each other.

Like pair programming, collaborating is an effective way to approach the software design process, particularly if it's done in a lightweight way. Collaboration increases quality plus it allows us to discuss and challenge some of the common assumptions that we make based our own knowledge, experience and preferences. It also paves the way for collective ownership of the code, which again helps to break down the silos that often form within software development teams. Everybody on the team will have different ideas and those different ideas need to meet.

28. Software architecture is a platform for conversation

If you're writing software as a part of your day-to-day job, then it's likely that your software isn't going to live in isolation. We tend to feel safe in our little project teams, particularly when everybody knows each other and team spirits are high. We've even built up development processes around helping us communicate better, prioritise better and ultimately deliver better software. However, many software projects are still developed in isolation by teams that are locked away from their users and their operational environments.

The success of agile methods has shown us that we need to have regular communication with the end-users or their representatives so we can be sure we're building software that will meet their needs. But what about all of those other stakeholders? Project teams might have a clear vision about *what* the software should do but often you'll hear phrases like this, often late in the delivery cycle.

- "Nobody told us you needed a production database created on this server."
- "We can't upgrade to [Java 7|.NET 4] on that server until system X is compatible."
- "We don't have spare production licenses."
- "Sorry, that contravenes our security policy."
- "Sorry, we'll need to undertake some operational acceptance testing before we promote your application into the production environment."
- "How exactly are *we* supposed to support this application?"
- "I don't care if you have a completely automated release process … I'm not giving you the production database credentials for your configuration files."
- "We need to run this past the risk and compliance team."
- "There's no way your system is going on the public cloud."

Software development isn't just about delivering features

The people who use your software are just one type of stakeholder. There are usually many others including:

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

These are just some of the stakeholders that may have an interest in your architecture, but there are probably others depending on your organisation and the way that it works. If you think you can put together a software architecture in an ivory tower on your own, you're probably doing it wrong. Software architectures don't live in isolation and the software design process is a platform for conversation. A five minute conversation now could help capture those often implied architectural drivers and improve your chance of a successful delivery.

29. 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.

30. 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.

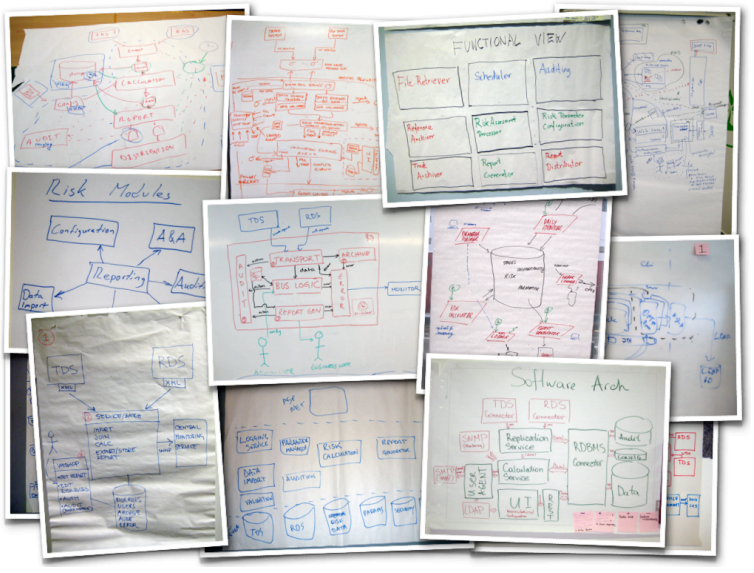
Understanding software architecture is not the same as being able to communicate it. Those architecture diagrams that you have on the wall of your office; do they reflect the system that is actually being built or are they conceptual abstractions that bear no resemblance to the structure of the code. 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.

31. The need for sketches

I usually get a response of disbelief or amusement when I tell people that I travel around to teach people about software architecture and how to draw pictures. To be fair, it's not hard to see why. Software architecture already has a poor reputation and the mention of "pictures" tends to bring back memories of analysis paralysis and a stack of UML diagrams that few people truly understand. After all, the software development industry has come a long way over the past decade, particularly given the influence of the agile manifesto and the wide range of techniques it's been responsible for spawning.

Test driven development vs diagrams

Test-driven development (TDD) is an example and it's one of those techniques that you either love or hate. Without getting into the debate of whether TDD is the "best way" to design software or not, there are many people out there that do use TDD as a way to design software. It's not for everybody though and there's nothing wrong with sketching out some designs on a whiteboard with a view to writing tests after you've written some production code. Despite what the evangelists say, TDD isn't a silver bullet.

I'm very much a visual person myself and fall into latter camp. I like being able to visualise a problem before trying to find a solution. Describe a business process to me and I'll sketch up a summary of it. Talk to me about a business problem and I'm likely to draw a high-level domain model. Visualising the problem is a way for me to ask questions and figure out whether I've understood what you're saying. I also like sketching out solutions to problems, again because it's a great way to get everything out into the open in a way that other people can understand quickly.

Why should people learn how to sketch?

Why is this a good skill for people to learn? Put simply, agility (and therefore moving fast) requires good communication. Sketching is a fantastic way to communicate a lot of information in a relatively short amount of time yet it's a skill that we don't often talk about in the software industry any more. There are several reasons for this:

1. Many teams instantly think of UML but they've dropped it as a communication method or never understood it in the first place. After all, apparently UML "isn't cool".
2. Many teams don't do class design in a visual way anymore because they prefer TDD instead.

Sketching isn't art

When I say "sketching", I mean exactly that. At the age of 12 I was told that I would fail if I was to take Art as a subject at GCSE (high school) level, so ironically I can't draw. But it's not the ability to create a work of art that's important. Rather, it's the ability to get to bottom of something quickly and to summarise the salient points in a way that others can understand. It's about communicating in a simple yet effective and efficient way.

Sketches are not comprehensive models

Just to be clear, I'm not talking about detailed modelling, comprehensive UML models or model-driven development. This is about effectively and efficiently communicating the software architecture of the software that you're building through one or more simple sketches. This allows you to:

- Help everybody understand the "big picture" of what is being built.
- Create shared vision of what you're building within the development team.
- Provide a focal point for the development team (e.g. by keeping the sketches on the wall) so that everybody in the development team remains focussed on what the software is and *how* it is being built.
- Provide a point of focus for those technical conversations about how new features should be implemented.
- Provide a map that can be used by software developers to navigate the source code.
- Help people understand how what they are building fits into the "bigger picture".
- Help you to explain what you're building to people outside of the development team (e.g. operations and support staff, non-technical stakeholders, etc).
- Fast-track the on-boarding of new software developers to the team.
- Provide a starting point for techniques such as risk-storming.

Rather than detailed class design, my goal for software architecture sketches is to ensure that the high-level structure is understood. It's about creating a vision that everybody on the team can understand and commit to. Context, containers and components diagrams are usually sufficient.

Sketching can be a collaborative activity

As a final point, sketching can be a collaborative activity, particularly if done using a whiteboard or flip chart rather than a modelling tool. This fits much more with the concept of collaborative self-organising teams that many of us are striving towards but it does require that everybody on the team understands how to sketch.

Unfortunately, drawing diagrams seems to have fallen out of favour with many software development teams but it's a skill that should be in every software developer's toolbox because it paves the way for collaborative software design and makes collective code ownership easier. Every software development team can benefit from a few high-level sketches.

32. Ineffective sketches

Over the past few years, I've found that many software development teams struggle to visualise and communicate the software architecture of the systems they build. I see three major reasons for this.

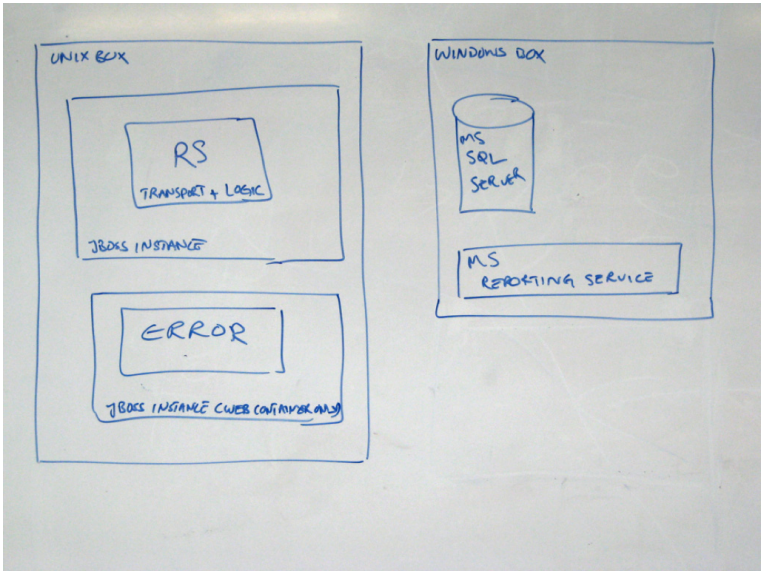
1. In their haste to adopt agile approaches, many software teams have thrown out the baby with the bath water - modelling and documentation have been thrown out alongside traditional plan-driven processes and methodologies.
2. Teams that did see the value in documents and diagrams have typically abandoned the Unified Modeling Language (UML) (assuming that they used it in the first place, of course) in favour of an approach that is more lightweight and pragmatic. My anecdotal evidence, based upon meeting and speaking to thousands of software developers, suggests that as many as nine out of ten software developers don't use UML.
3. There are very few people out there who teach software teams how to effectively visualise, model and communicate software architecture. And, based upon running workshops for some computer science students, this includes universities.

If you look around the offices of most software development teams for long enough, you're sure to find some sketches, either on whiteboards or scraps of paper sitting on desks. Sketches *are* a great way to capture and present software architecture but they usually lack the formality and rigour of UML diagrams. This isn't necessarily a bad thing, but the diagrams do need to be comprehensible and this is where things start to get tricky. Having run software architecture sketching workshops for thousands of people over the past few years, I can say without doubt that the majority of people *do* find this a very hard thing to do. The small selection of photos that follow are taken from these workshops, where groups of people have tried to communicate their software solution to the financial risk system case study. Have a look at each in turn and ask yourself whether they communicate the software architecture of the solution in an effective way. Some of the diagrams make use of colour, so apologies if you're reading this on a black and white e-book reader.

The shopping list

Regardless of whether this is *the* software architecture diagram or one of a collection of software architecture diagrams, this diagram doesn't tell you much about the solution.

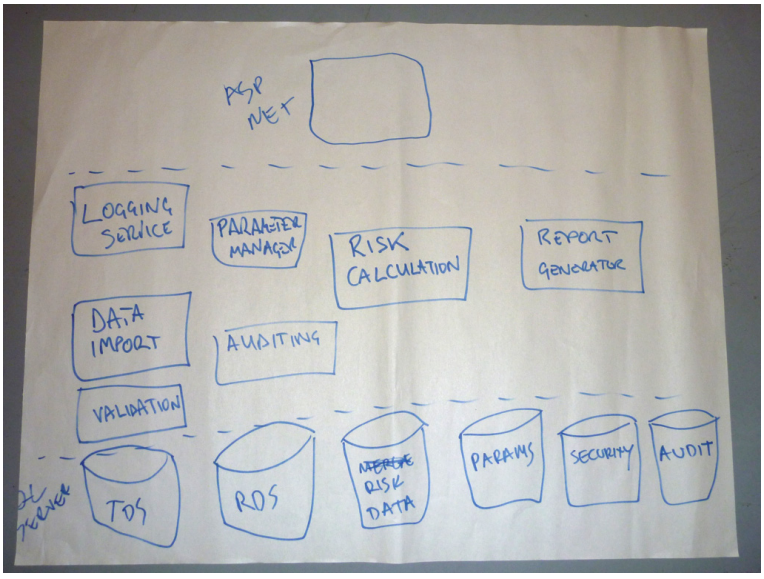
Essentially it's just a shopping list of technologies.



There's a Unix box and a Windows box, with some additional product selections that include JBoss (a Java EE application server) and Microsoft SQL Server. The problem is, I don't know what those products are doing and there seems to be a connection missing between the Unix box and the Windows box. Since responsibilities and interactions are not shown, this diagram probably would have been better presented as a bulleted list.

Boxes and no lines

When people talk about software architecture, they often refer to "boxes and lines". This next diagram has boxes, but no lines.

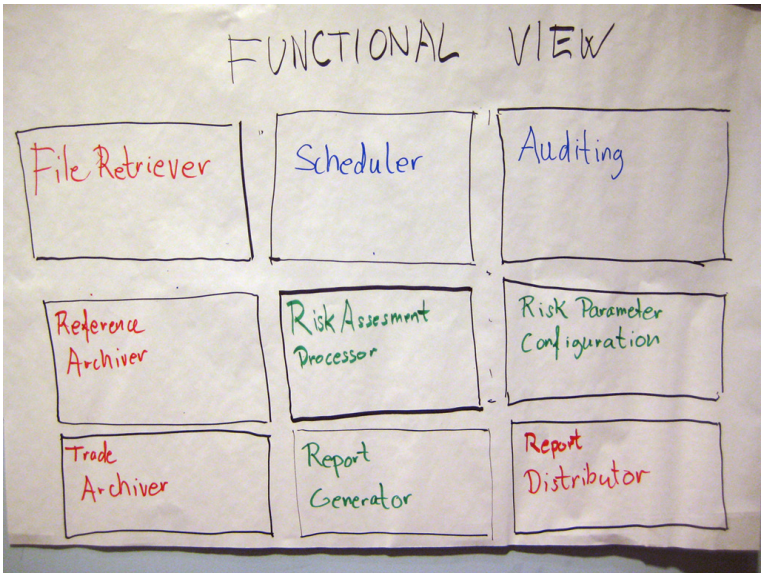


This is a three-tier solution (I think) that uses the Microsoft technology stack. There's an ASP.NET web thing at the top, which I assume is being used for some sort of user interaction, although that's not shown on the diagram. The bottom section is labelled "SQL Server" and there are lots of separate "database cans". To be honest though, I'm left wondering whether these are separate database servers, schemas or tables.

Finally, in the middle, is a collection of boxes, which I assume are things like components, services, modules, etc. From one perspective, it's great to see how the middle-tier of the overall solution has been decomposed into smaller chunks and these are certainly the types of components/services/modules that I would expect to see for such a solution. But again, there are no responsibilities and no interactions. Software architecture is about structure, which is about things (boxes) and how they interact (lines). This diagram has one, but not the other. It's telling a story, but not the whole story.

The "functional view"

This is similar to the previous diagram and is very common, particularly in large organisations for some reason.

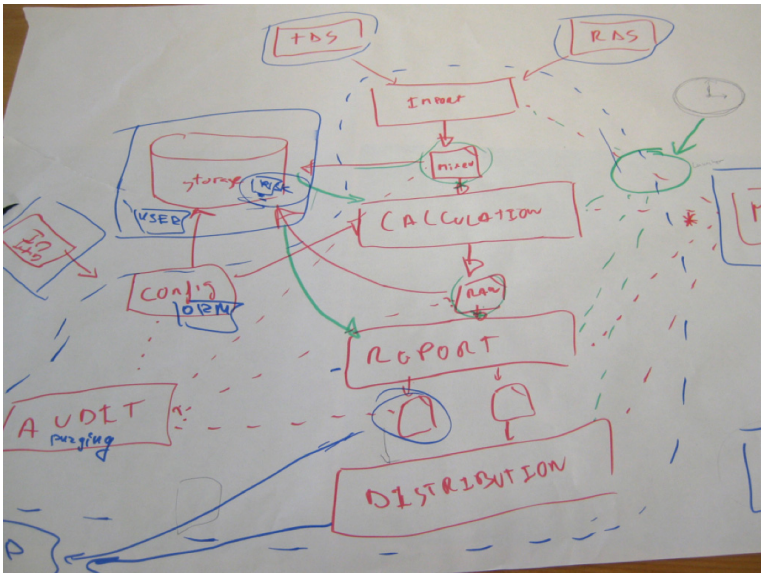


Essentially the group that produced this diagram has simply documented their functional decomposition of the solution into things, which I again assume are components, services, modules, etc but I could be wrong. Imagine a building architect drawing you a diagram of your new house that simply had a collection of boxes labelled "Cooking", "Eating", "Sleeping", "Relaxing", etc.

This diagram suffers from the same problem as the previous diagram (no responsibilities and no interactions) plus we additionally have a colour coding to decipher. Can you work out what the colour coding means? Is it related to input vs output functions? Or perhaps it's business vs infrastructure? Existing vs new? Buy vs build? Or maybe different people simply had different colour pens! Who knows. I often get asked why the central "Risk Assessment Processor" box has a noticeably thicker border than the other boxes. I honestly don't know, but I suspect it's simply because the marker pen was held at a different angle.

The airline route map

This is one of my all-time favourites. It was also the *one* and *only* diagram that this particular group used to present their solution.



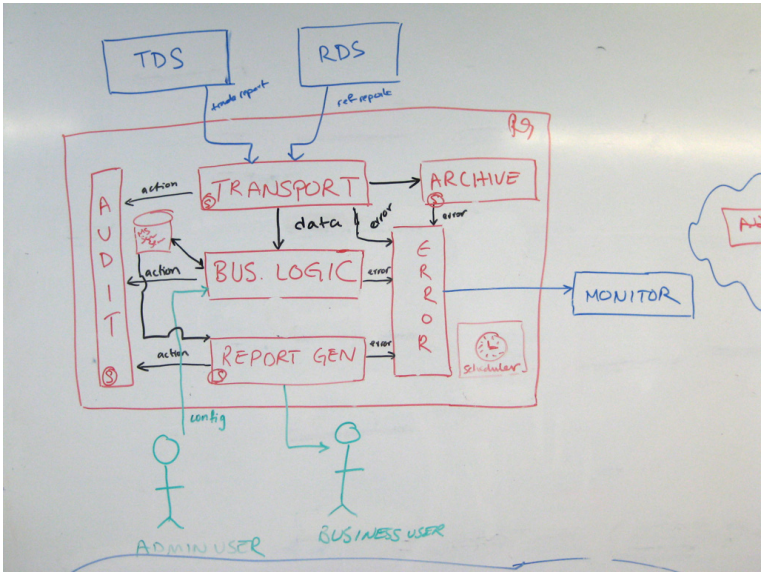
The central spine of this diagram is great because it shows how data comes in from the source data systems (TDS and RDS) and then flows through a series of steps to import the data, perform calculations, generate reports and finally distribute them. It's a super-simple activity diagram that provides a nice high-level overview of what the system is doing. But then it all goes wrong.

I think the green circle on the right of the diagram is important because everything is pointing to it, but I'm not sure why. And there's also a clock, which I assume means that something is scheduled to happen at a specific time.

The left of the diagram is equally confusing, with various lines of differing colours and styles zipping across one another. If you look carefully you'll see the letters "UI" upside-down. Perhaps this diagram makes more sense if you fold it like an Origami challenge?

Generically true

This is another very common style of diagram. Next time somebody asks you to produce a software architecture diagram of a system, present them this photo and you're done!

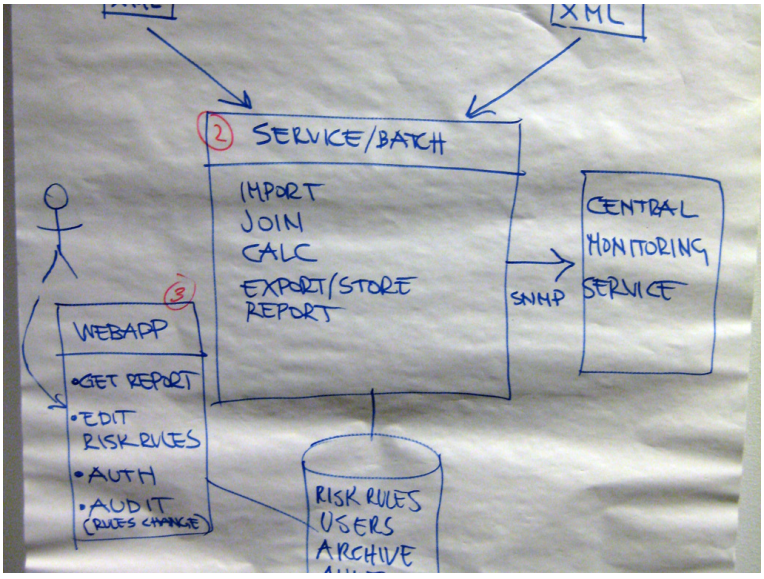


It's a very "Software Architecture 101" style of diagram where most of the content is generic. Ignoring the source data systems at the top of the diagram (TDS and RDS), we have boxes generically labelled transport, archive, audit, report generation, error handling and arrows labelled error and action. Oh and look at the box in the centre - it's labelled "business logic". Do you ever build software that implements "business logic"?

There are a number of ways in which this diagram can be made more effective, but simply replacing the "business logic" box with "financial risk calculator" at least highlights the business domain in which we are operating. In Screaming Architecture, Uncle Bob Martin says that the organisation of a codebase should scream something about the business domain. The same is true of software architecture diagrams.

The "logical view"

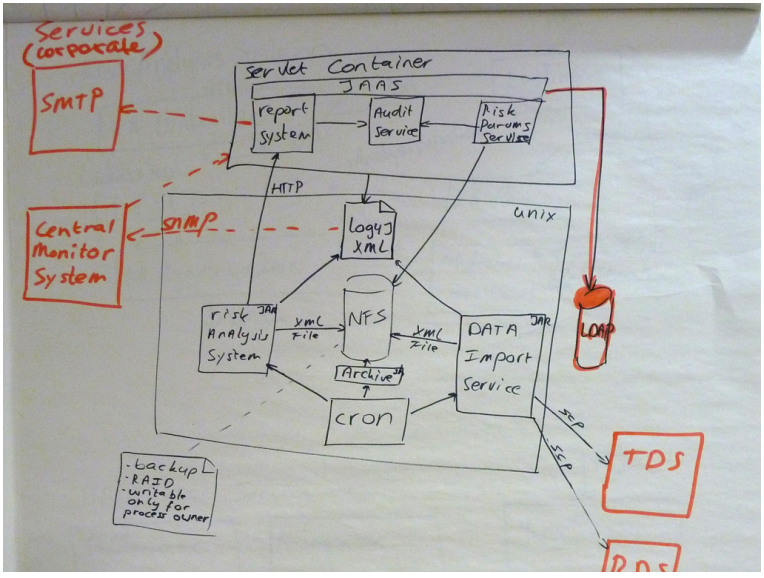
This diagram is also relatively common. It shows the overall shape of the software architecture (including responsibilities, which I really like) but the technology choices are left to your imagination.



There's a common misconception that "software architecture" diagrams should be "logical" in nature rather than include technology choices, especially before any code is written. After all, I'm often told that the financial risk system "is a simple solution that can be built with any technology", so it doesn't really matter anyway. I disagree this is the case and the issue of including or omitting technology choices is covered in more detail elsewhere in the book.

Deployment vs execution context

This next one is a Java solution consisting of a web application and a bunch of server-side components. Although it provides a simple high-level overview of the solution, it's missing some information and you need to make some educated guesses to fill in the blanks.

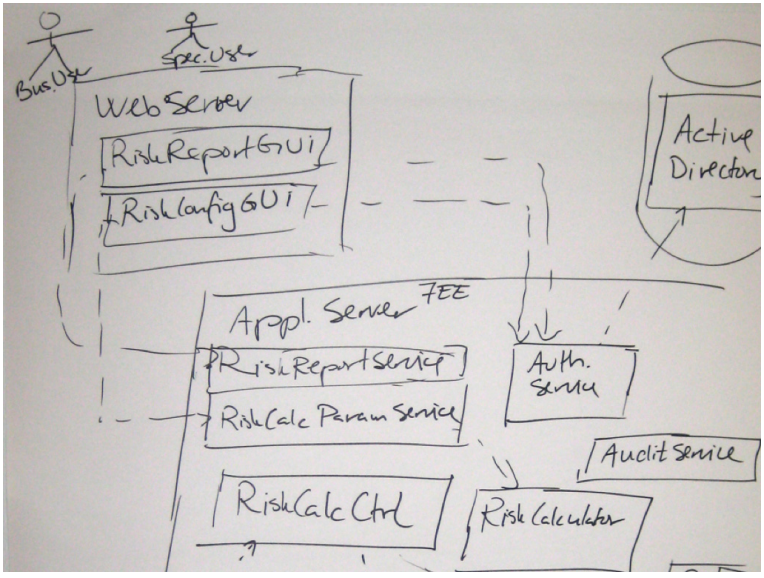


If you look at the Unix box in the centre of the diagram, you'll see two smaller boxes labelled "Risk Analysis System" and "Data Import Service". If you look closely, you'll see that both boxes are annotated "JAR", which is the deployment mechanism for Java code (Java ARchive). Basically this is a ZIP file containing compiled Java bytecode. The equivalent in the .NET world is a DLL.

And herein lies the ambiguity. What happens if you put a JAR file on a Unix box? Well, the answer is not very much other than it takes up some disk space. And cron (the Unix scheduler) doesn't execute JAR files unless they are really standalone console applications, the sort that have a "public static void main" method as a program entry point. By deduction then, I think both of those JAR files are actually standalone applications and that's what I'd like to see on the diagram. Rather than the deployment mechanism, I want to understand the execution context.

Too many assumptions

This next diagram tells us that the solution is an n-tier Java EE system, but it omits some important details.



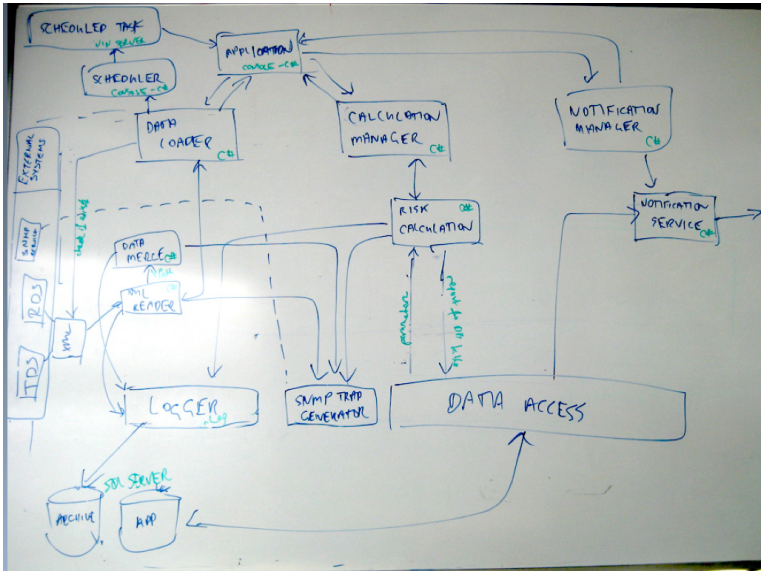
The lines between the web server and the application server have no information about how this communication occurs. Is it SOAP? RESTful services? XML over HTTP? Remote method invocation? Windows Communication Foundation? Asynchronous messaging? It's not clear and this concerns me for three reasons:

1. **Constraints**: If you're working in an environment with existing constraints, the technology choices may have be made for you already. For example, perhaps you have standards about inter-process communication or firewalls that only permit certain types of traffic through them.
2. **Non-functional requirements**: The choice of technology and protocol may have an impact on whether you meet your non-functional requirements, particularly if you are dealing with high performance, scalability or security.
3. **Complexity**: I've worked with software teams who have never created an n-tier architecture before and they are often under the illusion that this style of architecture can be created "for free". In the real world, more layers means more complexity.

Granted there are many options and often teams don't like committing early without putting together some prototypes. No problem, just annotate those lines on the diagram with the list of potential options instead so we can at least have a better conversation.

Homeless Old C# Object (HOCO)

If you've heard of "Plain Old C# Objects" (POCOs) or "Plain Old Java Objects" (POJOs), this is the homeless edition. This diagram mixes up a number of different levels of detail.

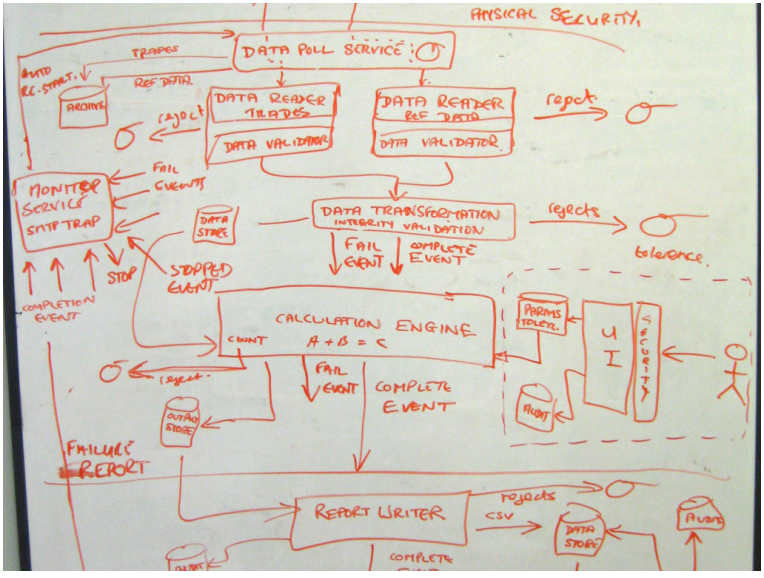


In the bottom left of the diagram is a SQL Server database, and at the top left of the diagram is a box labelled "Application". Notice how that same box is also annotated (in green) "Console-C#". Basically, this system seems to be made up of a C# console application and a database. But what about the other boxes?

Well, most of them seem to be C# components, services, modules or objects and they're much like what we've seen on some of the other diagrams. There's also a "data access" box and a "logger" box, which could be frameworks or architectural layers. Do all of these boxes represent the same level of granularity as the console application and the database? Or are they actually *part* of the application? I suspect the latter, but the lack of boundaries makes this diagram confusing. I'd like to draw a big box around most of the boxes to say "all of these things live inside the console application". I want to give those boxes a home. Again, I do want to understand how the system has been decomposed into smaller components, but I also want to know about the execution context too.

Choose your own adventure

This is the middle part of a more complex diagram.



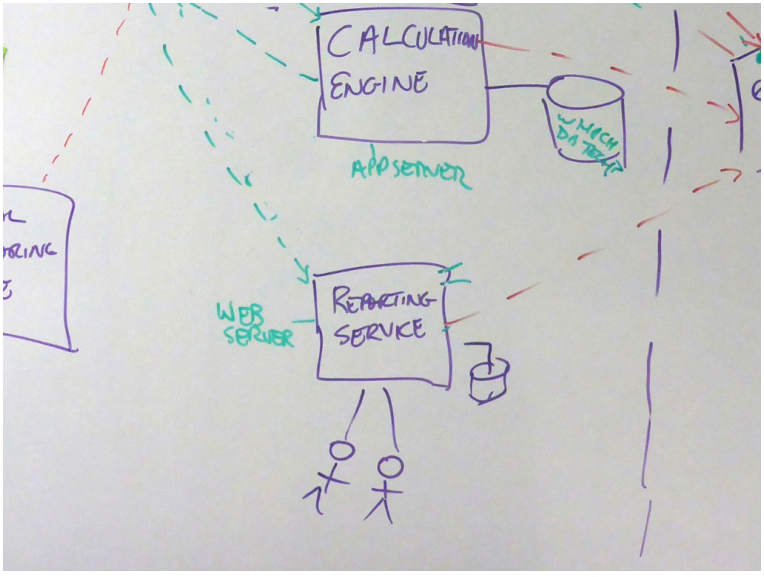
It's a little like those "choose your own adventure" books that I used to read as a kid. You would start reading at page 1 and eventually arrive at a fork in the story where you decide what should happen next. If you want to attack the big scary creature you've just encountered, you turn to page 47. If you want to run away like a coward, it's page 205 for you. You keep making similar choices and eventually, and annoyingly, your character ends up dying and you have to start over again.

This diagram is the same. You start at the top and weave your way downwards through what is a complex asynchronous and event-driven style of architecture. You often get to make a choice - should you follow the "fail event" or the "complete event"? As with the books, all paths eventually lead to the (SNMP) trap on the left of the diagram.

The diagram is complex, it's trying to show everything and the single colour being used doesn't help. Removing some information and/or using colour coding to highlight the different paths through the architecture would help tremendously.

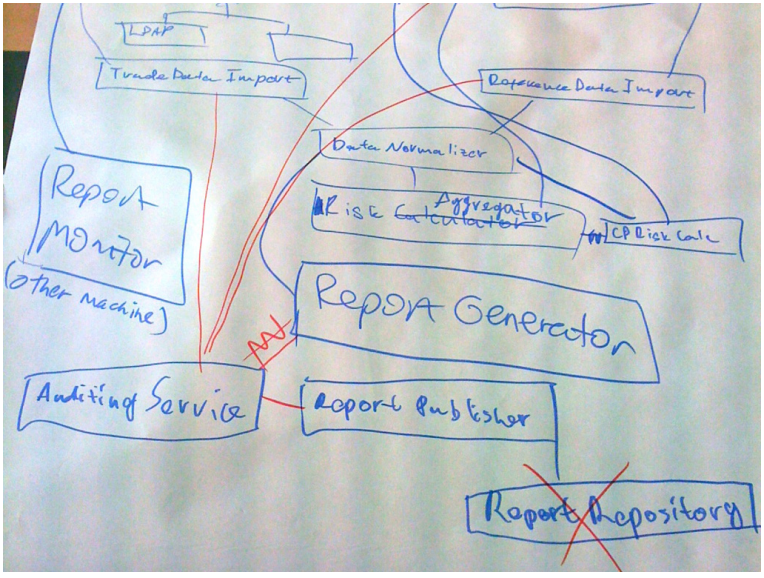
Stormtroopers

To pick up on something you may have noticed from previous sketches, I regularly see diagrams that include unlabelled users/actors. Essentially they are faceless clones.



Should have used a whiteboard!

The final diagram is a great example of why whiteboards are such useful bits of kit!



Creating effective sketches

These example diagrams typify what I see when I initially work with software teams to help them better communicate software architectures visually. Oh, and don't think that using Microsoft Visio will help! It often makes things worse because now people have the tool to struggle with too. A quick Google image search will uncover a plethora of similar block diagrams that suffer from many of the same problems we've seen already. I'm sure you will have seen diagrams like this within your own organisations too.

Using UML would avoid many of these pitfalls but it's not something that many people seem enthusiastic about learning these days. Simple and effective software architecture sketches are well within the reach of everybody though. All it takes is some simple advice and a common set of abstractions.

33. C4: context, containers, components and classes

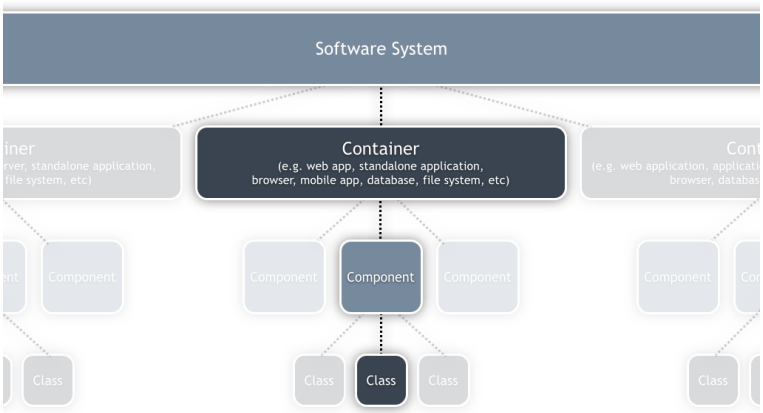
The code for any software system is where most of the focus remains for the majority of the software development life cycle, and this makes sense because the code is the ultimate deliverable. But if you had to explain to somebody how that system worked, would you start with the code?

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 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.

34. Context diagram

A context diagram can be a useful starting point for diagramming and documenting a software system, allowing you to step back and look at the big picture.

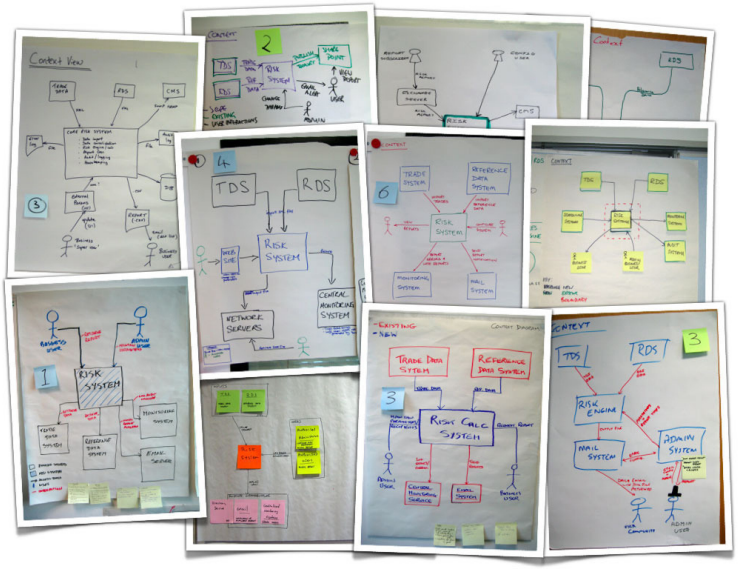
Intent

A context diagram helps you to answer the following questions.

1. What is the software system that we are building (or have built)?
2. Who is using it?
3. How does it fit in with the existing IT environment?

Structure

Draw a simple block diagram showing your system as a box in the centre, surrounded by its users and the other systems that it interfaces with. For example, if you were diagramming a solution to the financial risk system, you would draw the following sort of diagram. Detail isn't important here as it's your wide angle view showing a big picture of the system landscape. The focus should be on people and systems rather than technologies and protocols.



Example context diagrams for the financial risk system (see appendix)

These example diagrams show the risk system sitting in the centre, surrounded by its users and the other IT systems that the risk system has a dependency on.

Users, actors, roles, personas, etc

These are the users of the system. There are two main types of user for the risk system:

- Business user (can view the risk reports that are generated)
- Admin user (can modify the parameters used in the risk calculation process)

IT systems

Depending on the environment and chosen solution, the other IT systems you might want to show on a context diagram for the risk system include:

- Trade Data System (the source of the financial trade data)
- Reference Data System (the source of the reference data)
- Central Monitoring System (where alerts are sent to)
- Active Directory or LDAP (for authenticating and authorising users)

- Microsoft SharePoint or another content/document management system (for distributing the reports)
- Microsoft Exchange (for sending e-mails to users)

Interactions

It's useful to annotate the interactions (user <-> system, system <-> system, etc) with some information about the purpose rather than simply having a diagram with a collection of boxes and ambiguous lines connecting everything together. For example, when I'm annotating user to system interactions, I'll often include a short bulleted list of the important use cases/user stories to summarise how that particular type of user interacts with the system.

Motivation

You might ask what the point of such a simple diagram is. Here's why it's useful:

- It makes the context explicit so that there are no assumptions.
- It shows what is being added (from a high-level) to an existing IT environment.
- It's a high-level diagram that technical and non-technical people can use as a starting point for discussions.
- It provides a starting point for identifying who you potentially need to go and talk to as far as understanding inter-system interfaces is concerned.

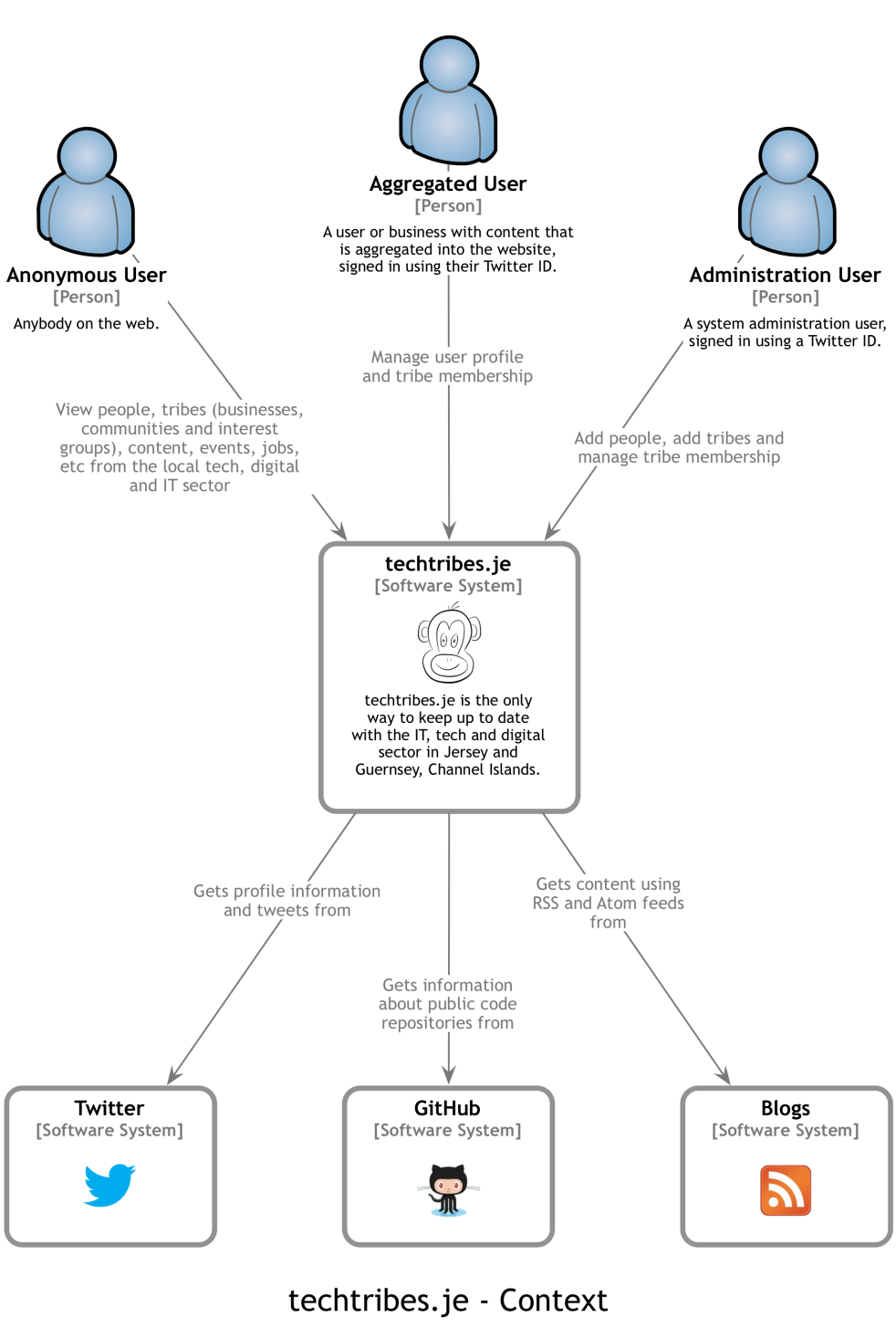
A context diagram doesn't show much detail but it does help to set the scene and is a starting point for other diagrams. Finally, a context diagram should only take a couple of minutes to draw, so there really is no excuse not to do it.

Audience

- Technical and non-technical people, inside and outside of the immediate software development team.

Example

Let's look at an example. The techtribes.je website provides a way to find people, tribes (businesses, communities, interest groups, etc) and content related to the tech, IT and digital sector in Jersey and Guernsey, the two largest of the Channel Islands. At the most basic level, it's a content aggregator for local tweets, news, blog posts, events, talks, jobs and more. Here's a context diagram that provides a visual summary of this.


Aggregated User
[Person]
A user or business with content that is aggregated into the website, signed in using their Twitter ID.
Anonymous User
[Person]
Anybody on the web.
Administration User
[Person]
A system administration user, signed in using a Twitter ID.
Manage user profile and tribe membership
View people, tribes (businesses, communities and interest groups), content, events, jobs, etc from the local tech, digital and IT sector
Add people, add tribes and manage tribe membership
techtribes.je
[Software System]
techtribes.je is the only way to keep up to date with the IT, tech and digital sector in Jersey and Guernsey, Channel Islands.
Gets profile information and tweets from
Gets information about public code repositories from
Gets content using RSS and Atom feeds from
Twitter
[Software System]
GitHub
[Software System]
Blogs
[Software System]
techtribes.je - Context

Again, detail isn't important here as this is your zoomed out view. The focus should be on people (actors, roles, personas, etc) and software systems rather than technologies, protocols and other low-level details.

35. Container diagram

Once you understand how your system fits in to the overall IT environment with a context diagram, a really useful next step can be to illustrate the high-level technology choices with a container diagram.

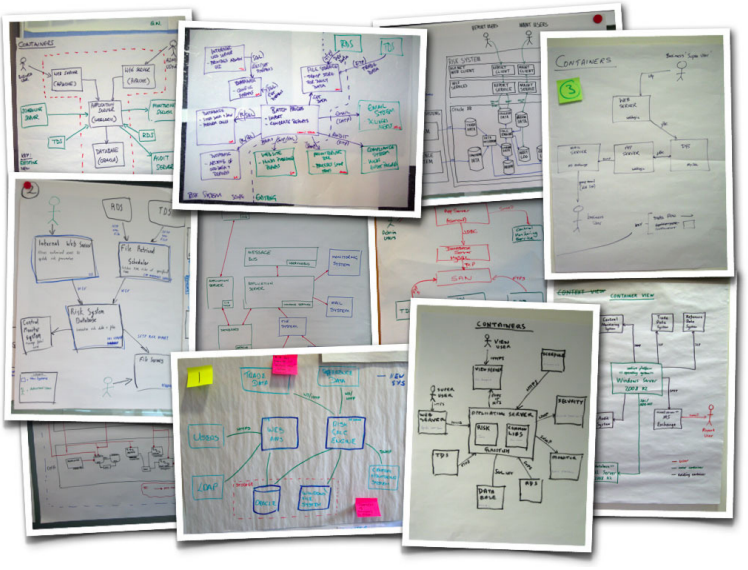
Intent

A container diagram helps you answer the following questions.

1. What is the overall shape of the software system?
2. What are the high-level technology decisions?
3. How are responsibilities distributed across the system?
4. How do containers communicate with one another?
5. As a developer, where do I need to write code in order to implement features?

Structure

Draw a simple block diagram showing your key technology choices. For example, if you were diagramming a solution to the financial risk system, depending on your solution, you would draw the following sort of diagram.



Example container diagrams for the financial risk system (see appendix)

These example diagrams show the various web servers, application servers, standalone applications, databases, file systems, etc that make up the risk system. To enrich the diagram, often it's useful to include some of the concepts from the context diagram diagram, such as users and the other IT systems that the risk system has a dependency on.

Containers

By "containers", I mean the *logical* executables, applications or processes that make up your software system; such as:

- Web servers and applications[1] (e.g. Apache HTTP Server, Apache Tomcat, Microsoft IIS, WEBrick, etc)
- Application servers (e.g. IBM WebSphere, BEA/Oracle WebLogic, JBoss AS, etc)
- Enterprise service buses and business process orchestration engines (e.g. Oracle Fusion middleware, etc)
- SQL databases (e.g. Oracle, Sybase, Microsoft SQL Server, MySQL, PostgreSQL, etc)
- NoSQL databases (e.g. MongoDB, CouchDB, RavenDB, Redis, Neo4j, etc)

[1]If multiple Java EE web applications or .NET websites are part of the same software system, they are usually executed in separate classloaders or AppDomains so I show them as separate containers because they are independent and require inter-process communication (e.g. remote method invocation, SOAP, REST, etc) to collaborate.

- Other storage systems (e.g. Amazon S3, etc)
- File systems (especially if you are reading/writing data outside of a database)
- Windows services
- Standalone/console applications (i.e. "public static void main" style applications)
- Web browsers and plugins
- cron and other scheduled job containers

For each container drawn on the diagram, you could specify:

- **Name**: The logical name of the container (e.g. "Internet-facing web server", "Database", etc)
- **Technology**: The technology choice for the container (e.g. Apache Tomcat 7, Oracle 11g, etc)
- **Responsibilities**: A very high-level statement or list of the container's responsibilities. You could alternatively show a small diagram of the key components that reside in each container, but I find that this usually clutters the diagram.

If you're struggling to understand whether to include a box on a containers diagram, simply ask yourself whether that box will be (or can be) deployed on a separate piece of physical or virtual hardware. Everything that you show on a containers diagram *should* be deployable separately. This doesn't mean that you must deploy them on separate infrastructure, but they should be able to be deployed separately.

Interactions

Typically, inter-container communication is inter-process communication. It's very useful to explicitly identify this and summarise how these interfaces will work. As with any diagram, it's useful to annotate the interactions rather than simply having a diagram with a collection of boxes and ambiguous lines connecting everything together. Useful information to add includes:

- The purpose of the interaction (e.g. "reads/writes data from", "sends reports to", etc).
- Communication method (e.g. Web Services, REST, Java Remote Method Invocation, Windows Communication Foundation, Java Message Service).
- Communication style (e.g. synchronous, asynchronous, batched, two-phase commit, etc)
- Protocols and port numbers (e.g. HTTP, HTTPS, SOAP/HTTP, SMTP, FTP, RMI/IIOP, etc).

System boundary

If you do choose to include users and IT systems that are outside the scope of what you're building, it can be a good idea to draw a box around the appropriate containers to explicitly demarcate the system boundary. The system boundary corresponds to the single box that would appear on a context diagram (e.g. "Risk System").

Motivation

Where a context diagram shows your software system as a single box, a container diagram opens this box up to show what's inside it. This is useful because:

- It makes the high-level technology choices explicit.
- It shows where there are relationships between containers and how they communicate.
- It provides a framework in which to place components (i.e. so that all components have a home).
- It provides the often missing link between a very high-level context diagram and (what is usually) a very cluttered component diagram showing all of the logical components that make up the entire software system.

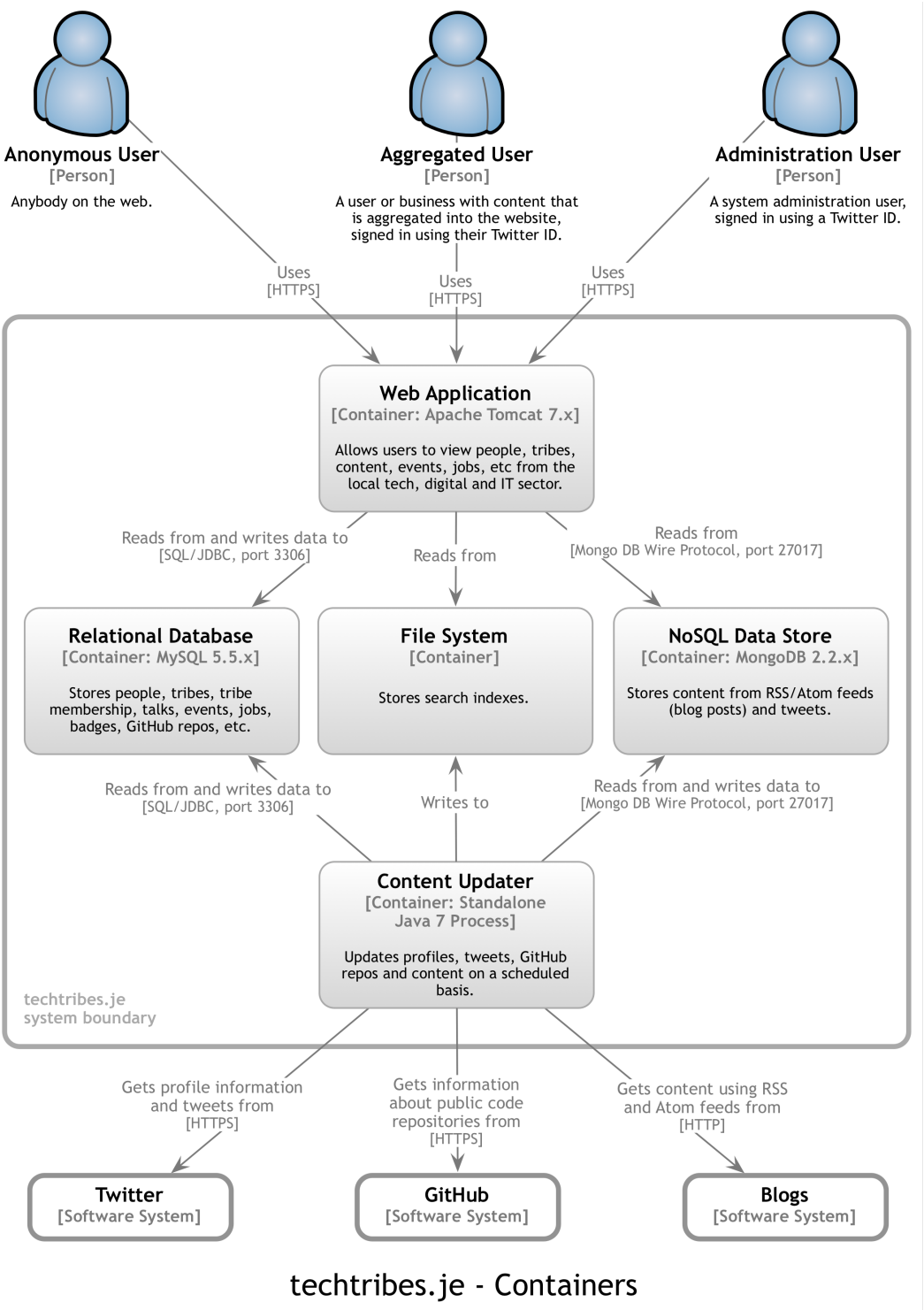
As with a context diagram, this should only take a couple of minutes to draw, so there really is no excuse not to do it either.

Audience

- Technical people inside and outside of the immediate software development team; including everybody from software developers through to operational and support staff.

Example

The following diagram shows the logical containers that make up the techtribes.je website.


Anonymous User
[Person]
Anybody on the web.
Aggregated User
[Person]
A user or business with content that is aggregated into the website, signed in using their Twitter ID.
Administration User
[Person]
A system administration user, signed in using a Twitter ID.
Uses
[HTTPS]
Uses
[HTTPS]
Uses
[HTTPS]
Web Application
[Container: Apache Tomcat 7.x]
Allows users to view people, tribes, content, events, jobs, etc from the local tech, digital and IT sector.
Reads from and writes data to
[SQL/JDBC, port 3306]
Reads from
Reads from
[Mongo DB Wire Protocol, port 27017]
Relational Database
[Container: MySQL 5.5.x]
Stores people, tribes, tribe membership, talks, events, jobs, badges, GitHub repos, etc.
File System
[Container]
Stores search indexes.
NoSQL Data Store
[Container: MongoDB 2.2.x]
Stores content from RSS/Atom feeds (blog posts) and tweets.
Reads from and writes data to
[SQL/JDBC, port 3306]
Writes to
Reads from and writes data to
[Mongo DB Wire Protocol, port 27017]
Content Updater
[Container: Standalone Java 7 Process]
Updates profiles, tweets, GitHub repos and content on a scheduled basis.
techtribes.je
system boundary
Gets profile information and tweets from
[HTTPS]
Gets information about public code repositories from
[HTTPS]
Gets content using RSS and Atom feeds from
[HTTP]
Twitter
[Software System]
GitHub
[Software System]
Blogs
[Software System]
techtribes.je - Containers

Put simply, techtribes.je is made up of an Apache Tomcat web server that provides users with information, and that information is kept up to date by a standalone content updater process. All data is stored either in a MySQL database, a MongoDB database or the file system. It's worth pointing out that this diagram says nothing about the number of physical instances of each container. For example, there could be a farm of web servers running against a MongoDB cluster, but this diagram doesn't show that level of information. Instead, I show physical instances, failover, clustering, etc on a separate deployment diagram. The containers diagram shows the high-level shape of the software architecture and how responsibilities are distributed across it. It also shows the major technology choices and how the containers communicate with one another. It's a simple, high-level technology focussed diagram that is useful for software developers and support/operations staff alike.

36. Component diagram

Following on from a container diagram showing the high-level technology decisions, I'll then start to zoom in and decompose each container further. How you decompose your system is up to you, but I tend to identify the major logical components and their interactions. This is about partitioning the functionality implemented by a software system into a number of distinct components, services, subsystems, layers, workflows, etc. If you're following a "pure Object Oriented" or Domain-Driven Design approach, then this may or may not work for you.

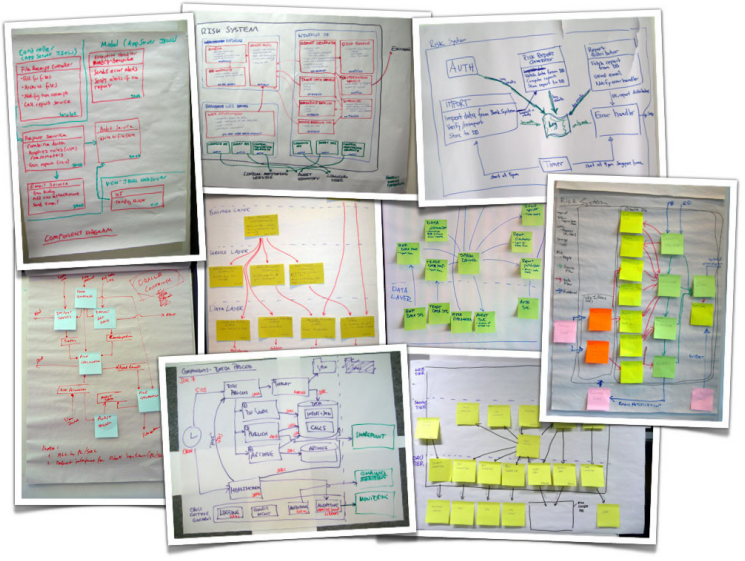
Intent

A component diagram helps you answer the following questions.

1. What components/services is the system made up of?
2. It is clear how the system works at a high-level?
3. Do all components/services have a home (i.e. reside in a container)?

Structure

Whenever people are asked to draw "architecture diagrams", they usually end up drawing diagrams that show the logical components that make up their software system. That is basically what this diagram is about, except we only want to see the components that reside within a *single* container at a time. Here are some examples of component diagrams if you were designing a solution to the financial risk system.



Example component diagrams for the financial risk system (see appendix)

Whenever I draw a component diagram, it typically only shows the components that reside within a single container. This is by no means a rule though and, for small software systems, often you can show all of the components across all of the containers on a single diagram. If that diagram starts to become too cluttered, maybe it's time to break it apart.

Components

If you were designing a solution to the financial risk system, you might include components like:

- Trade data system importer
- Reference data system importer
- Risk calculator
- Authentication service
- System driver/orchestrator
- Audit component
- Notification component (e.g. e-mail)
- Monitoring service
- etc

These components are the coarse-grained building blocks of your system and you should be able to understand how a use case/user story/feature can be implemented across one or more of these components. If you can do this, then you've most likely captured everything. If, for example, you have a requirement to audit system access but you don't have an audit component or responsibilities, then perhaps you've missed something.

For each of the components drawn on the diagram, you could specify:

- **Name**: The name of the component (e.g. "Risk calculator", "Audit component", etc).
- **Technology**: The technology choice for the component (e.g. Plain Old [Java|C#|Ruby|etc] Object, Enterprise JavaBean, Windows Communication Foundation service, etc).
- **Responsibilities**: A very high-level statement of the component's responsibilities (e.g. either important operation names or a brief sentence describing the responsibilities).

Interactions

To reiterate the same advice given for other types of diagram, it's useful to annotate the interactions between components rather than simply having a diagram with a collection of boxes and ambiguous lines connecting them all together. Useful information to add the diagram includes:

- The purpose of the interaction (e.g. "uses", "persists trade data through", etc)
- Communication style (e.g. synchronous, asynchronous, batched, two-phase commit, etc)

Motivation

Decomposing your software system into a number of components is software design at a slightly higher level of abstraction than classes and the code itself. An audit component might be implemented using a single class backing onto a logging framework (e.g. log4j, log4net, etc) but treating it as a distinct component lets you also see it for what it is, which is a key building block of your architecture. Working at this level is an excellent way to understand how your system will be internally structured, where reuse opportunities can be realised, where you have dependencies between components, where you have dependencies between components and containers, and so on. Breaking down the overall problem into a number of separate parts also provides you with a basis to get started with some high-level estimation, which is great if you've ever been asked for ballpark estimates for a new project.

A component diagram shows the logical components that reside inside each of the containers. This is useful because:

- It shows the high-level decomposition of your software system into components with distinct responsibilities.
- It shows where there are relationships and dependencies between components.
- It provides a framework for high-level software development estimates and how the delivery can be broken down.

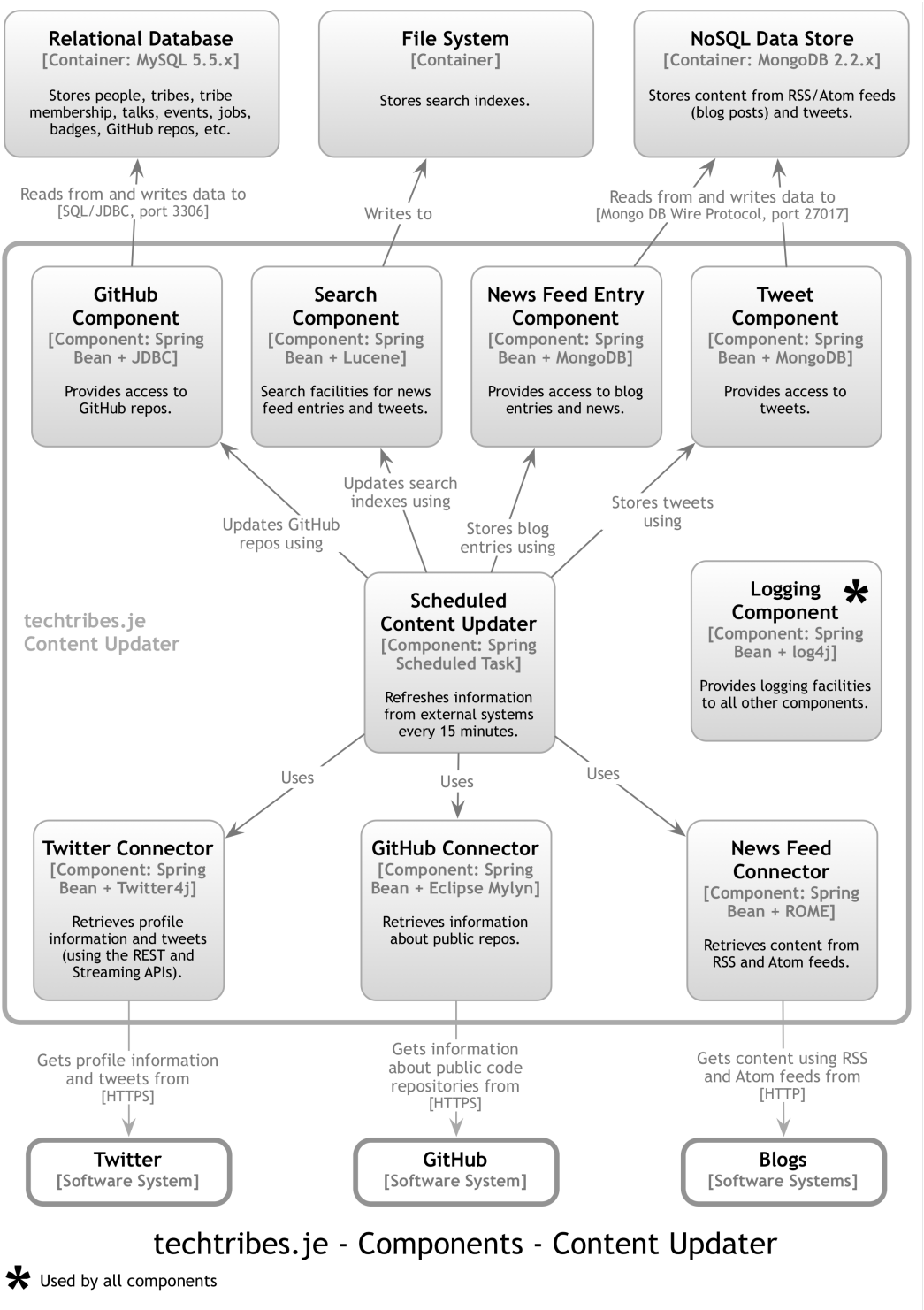
Designing a software system at this level of abstraction is something that can be done in a number of hours or days rather than weeks or months. It also sets you up for designing/coding at the class and interface level without worrying about the overall high-level structure.

Audience

- Technical people within the software development team.

Example

As illustrated by the container diagram, techtribes.je includes a standalone process that pulls in content from Twitter, GitHub and blogs. The following diagram shows the high-level internal structure of the content updater in terms of components.


Relational Database
[Container: MySQL 5.5.x]
Stores people, tribes, tribe membership, talks, events, jobs, badges, GitHub repos, etc.
File System
[Container]
Stores search indexes.
NoSQL Data Store
[Container: MongoDB 2.2.x]
Stores content from RSS/Atom feeds (blog posts) and tweets.
Reads from and writes data to
[SQL/JDBC, port 3306]
Writes to
Reads from and writes data to
[Mongo DB Wire Protocol, port 27017]
GitHub Component
[Component: Spring Bean + JDBC]
Provides access to GitHub repos.
Search Component
[Component: Spring Bean + Lucene]
Search facilities for news feed entries and tweets.
News Feed Entry Component
[Component: Spring Bean + MongoDB]
Provides access to blog entries and news.
Tweet Component
[Component: Spring Bean + MongoDB]
Provides access to tweets.
Updates GitHub repos using
Updates search indexes using
Stores blog entries using
Stores tweets using
techtribes.je
Content Updater
Scheduled Content Updater
[Component: Spring Scheduled Task]
Refreshes information from external systems every 15 minutes.
Logging Component
[Component: Spring Bean + log4j]
Provides logging facilities to all other components.
Uses
Uses
Uses
Twitter Connector
[Component: Spring Bean + Twitter4j]
Retrieves profile information and tweets (using the REST and Streaming APIs).
GitHub Connector
[Component: Spring Bean + Eclipse Mylyn]
Retrieves information about public repos.
News Feed Connector
[Component: Spring Bean + ROME]
Retrieves content from RSS and Atom feeds.
Gets profile information and tweets from
[HTTPS]
Gets information about public code repositories from
[HTTPS]
Gets content using RSS and Atom feeds from
[HTTP]
Twitter
[Software System]
GitHub
[Software System]
Blogs
[Software Systems]
techtribes.je - Components - Content Updater
Used by all components

This diagram shows how the content updater is divided into components, what each of those components are, their responsibilities and the technology/implementation details.

The Logging Component is used by everything, but I didn't want to draw the lines to it from every component because the resulting diagram looks very cluttered. Instead, I've used an asterisk to denote this.

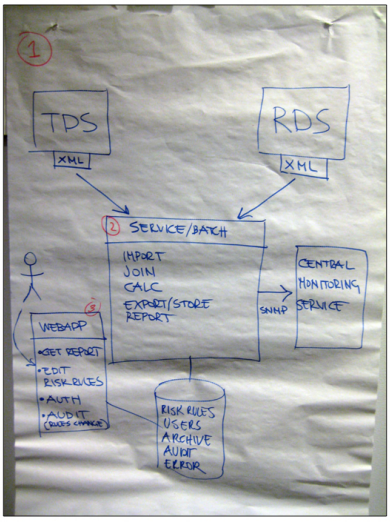
37. Technology choices included or omitted?

Think back to the last software architecture diagram that you saw. What did it look like? What level of detail did it show? Were technology choices included or omitted? In my experience, the majority of architecture diagrams omit any information about technology, instead focussing on illustrating the functional decomposition and major conceptual elements. Why is this?

Drawing diagrams during the design process

One of the main reasons for drawing software architecture diagrams is to communicate ideas during the process of designing software, much like you'd see blueprints drawn-up during the early stages of a building project.

I regularly run training classes where I ask small groups of people to design a simple financial risk system and here's a photo of an architecture diagram produced during one of those classes. Solution aside, the diagram itself is fairly typical of what I see. It shows a conceptual design rather than technical details.



Asking people why their diagrams don't show any technology decisions results in a number of different responses:

- "the [financial risk system] solution is simple and can be built with any technology".
- "we don't want to force a solution on developers".
- "it's an implementation detail".
- "we follow the 'last responsible moment' principle".

Drawing diagrams retrospectively

If you're drawing software architecture diagrams retrospectively, for documentation after the software has been built, there's really no reason for omitting technology decisions. However, others don't necessarily share this view and I often hear the following comments:

- "the technology decisions will clutter the diagrams".
- "but everybody knows that we only use ASP.NET against an Oracle database".

Architecture diagrams should be "conceptual"

It seems that regardless of whether diagrams are being drawn before, during or after the software has been built, there's a common misconception that architecture diagrams should

be conceptual in nature.

One of the reasons that software architecture has a bad reputation is because of the stereotype of ivory tower architects drawing very high-level pictures to describe their grandiose visions. I'm sure you've seen examples of diagrams with a big box labelled "Enterprise Service Bus" connected to a cloud, or perhaps diagrams showing a functional decomposition with absolutely no consideration as to whether the vision is implementable. If you truly believe that software architecture diagrams should be fluffy and conceptual in nature, my advice is to hire people that don't know about technology. That should do the trick.

Back to the real world, I like to see software architecture have a grounding in reality and technology choice shouldn't be an implementation detail. One way to ensure that technology is considered is to simply show the technology choices by including them on software architecture diagrams.

Make technology choices explicit

Including technology choices on software architecture diagrams removes ambiguity, even if you're working in an environment where all software is built using a standard set of technologies and patterns. Imagine that you're designing a software system. Are you really doing this without thinking about *how* you're actually going to implement it? Are you really thinking in terms of conceptual boxes and functional decomposition? If the answer to these questions is "not really", then why not add this additional layer of information onto the diagrams. Doing so provides a better starting point for conversations, particularly if you have a choice of technologies to use. Forcing people to include technology choices on their software architecture diagrams also tends to lead to much richer and deeper conversations that are grounded in the real-world. A fluffy conceptual diagram tends to make a lot of assumptions, but factoring in technology forces the following types of questions to be asked:

- "how *does* this component communicate with that component if it's running in separate process?"
- "how does this component get initiated, and where does that responsibility sit?"
- "why does this process need to communicate with that process?"
- "why is this component going to be implemented in technology X rather than technology Y"
- etc

As for technology decisions cluttering the diagrams, there are a number of strategies for dealing with this concern, including the use of a container diagram to separately show the major technology decisions.

Technology choices can help bring an otherwise ideal and conceptual software design back down to earth so that it is grounded in reality once again, while communicating the entirety of the big picture rather than just a part of it. Oh, and of course, the other side effect of adding technology choices to diagrams, particularly during the software design process, is that it helps to ensure the right people are drawing them.

38. Would you code it that way?

It's a common misconception that software architecture diagrams need to be stuck in the clouds, showing high-level concepts and abstractions that present the logical rather than the physical. But it doesn't have to be this way and bringing them back down to earth often makes diagrams easier to explain and understand. It can also make diagrams easier to draw too.

To illustrate why thinking about the implementation can help the diagramming process, here are a couple of scenarios that I regularly hear in my training classes.

Shared components

Imagine that you're designing a 3-tier software system that makes use of a web server, an application server and a database. While thinking about the high-level components that reside in each of these containers, it's not uncommon to hear a conversation like this:

- **Attendee**: "Should we draw the logging component outside of the web server and the application server, since it's used by both?"
- **Me**: "Would you code it that way? Will the logging component be running outside of both the web server and application server? For example, will it really be a separate standalone process?"
- **Attendee**: "Well … no, it would probably be a shared component in a [JAR file|DLL|etc] that we would deploy to both servers."
- **Me**: "Great, then let's draw it like that too. Include the logging component inside of each server and label it as a shared component with an annotation, stereotype or symbol."

If you're going to implement something like a shared logging component that will be deployed to a number of different servers, make sure that your diagram reflects this rather than potentially confusing people by including something that might be mistaken for a separate centralised logging server. If in doubt, always ask yourself how you would code it.

Layering strategies

Imagine you're designing a web application that is internally split up into a UI layer, a services layer and a data access layer.

- **Attendee**: "Should we show that all communication to the database from the UI goes through the services layer?"
- **Me**: "Is that how you're going to implement it? Or will the UI access the database directly?"
- **Attendee**: "We were thinking of perhaps adopting the CQRS pattern, so the UI could bypass the services layer and use the data access layer directly."
- **Me**: "In that case, draw the diagram as you've just explained, with lines from the UI to both the services and data access layers. Annotate the lines to indicate the intent and rationale."

Again, the simple way to answer this type of question is to understand how you would code it.

Diagrams should reflect reality

If you're drawing diagrams to retrospectively communicate a software system then the question becomes "is that *how* we coded it?". The principle is the same though. Diagrams should present abstractions that reflect reality rather than provide conceptual representations that don't exist. You should be able to see how the diagram elements are reflected in the codebase and vice versa. If you can understand how you would code it, you can understand how to visualise it.

39. Software architecture vs code

Software architecture and coding are often seen as mutually exclusive disciplines, despite us referring to higher level abstractions when we talk about our code. You've probably heard others on your team talking about components, services and layers rather than objects or classes when they're having architecture discussions. Take a look at the codebase though. Can you clearly see these abstractions or does the code reflect some other structure? If so, why is there no clear mapping between the architecture and the code? Why do those architecture diagrams that you have on the wall say one thing whereas your code says another?

Abstraction allows us to reduce detail and manage complexity

Let's imagine that you've inherited an undocumented existing codebase, which is somewhere in the region of two million lines of Java code, perhaps broken up into almost one hundred thousand Java classes. And let's say that you've been given the task of creating some software architecture diagrams to help describe the system to the rest of the team. Where do you start?

If you have enough time and patience, drawing a class diagram of the codebase is certainly an option. Although by the time you've finished drawing the diagram, it's likely to be out of date. Automating this process with a static analysis or diagramming tool isn't likely to help matters either. The problem here is there's too much information to comprehend.

Instead, what we tend to do is look for related groups of classes and instead draw a diagram showing those. These related groups of classes are usually referred to as modules, components, services, layers, packages, namespaces, subsystems, etc.

The same can be said when you're doing some up front design for a new software system. Although you could start by sketching out class diagrams, this is probably diving into the detail too quickly. My approach is to perform an initial level of decomposition by designing down to the level of components (or services, modules, etc) that each has a specific set of responsibilities.

There are a number of benefits to thinking about a software system in terms of components, but essentially it allows us to think and talk about the software as a small number of high-level abstractions rather than the hundreds and thousands of individual classes that make up

most enterprise object-oriented systems. Abstractions help us to reason about a big and/or complex software system.

We talk about components but write classes

Although we might refer to things like components when we're describing a software system, and indeed many of us consider our software systems to be built from a number of collaborating components, that structure isn't usually reflected in the code. This is one of the reasons why there is a disconnect between software architecture and coding as disciplines - the architecture diagrams on the wall say one thing, but the code says another.

When you open up a codebase, it will often reflect some other structure due to the organisation of the code. The mapping between the architectural view of a software system and the code are often very different. This is sometimes why you'll see people ignore architecture diagrams and say, "the code is the only single point of truth". George Fairbanks names this the "Model-code gap" in his book titled Just Enough Software Architecture.

The premise is that while we think about our software systems as being constructed of components, modules, services, etc, we don't have these same concepts in the programming languages that we use. For example, does Java have a "component", "module" or "layer" keyword? No, our Java systems are built from a collection of classes and interfaces, typically organised into a number of packages. It's this mismatch between architectural concepts and the code that can hinder our understanding.

An architecturally-evident coding style

George's answer to this is simple - we should just use an architecturally-evident coding style. In other words, we should drop hints into our codebase so that the code reflects the architectural intent. In concrete terms, this could be achieved by:

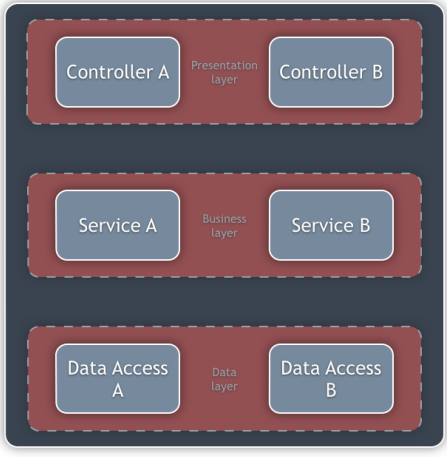
- **Naming conventions**: If you're implementing something that you think of as a component, ensure that the name of something (i.e. class, interface, package, namespace, etc) includes the word "component".
- **Packaging conventions**: In addition, perhaps you group everything related to a single component into a single package/namespace.
- **Metadata**: Alternatively, why not include metadata in the code so that parts of it can be traced back to the architectural vision. In real terms, you could use Java Annotations or C# Attributes to signify classes as being architecturally important.

This all sounds very sensible and relatively easy to do but, in my experience, I rarely see teams doing this. Instead we do something different.

Package by layer

Let's assume that we're building a web application based upon the Web-MVC pattern. There are a number of ways that we can organise our source code. Packaging code by layer is typically the default approach because, after all, that's what the books, tutorials and framework samples tell us to do. If you do a search on the web for tutorials related to Spring MVC or ASP.NET MVC, for example, you'll likely see this in the example code used. I spent most of my career building software systems in Java and I too used the same packaging approach for the majority of the codebases that I worked on.

Here we're organising code by grouping things of the same type. In other words, you'll have a package for domain classes, one for web controllers/views, one for "business services", one for data access, another for integration points and so on. I'm using the Java terminology of a "package" here, but the same is applicable to namespaces in C#, etc.



Package by layer (horizontal slicing)

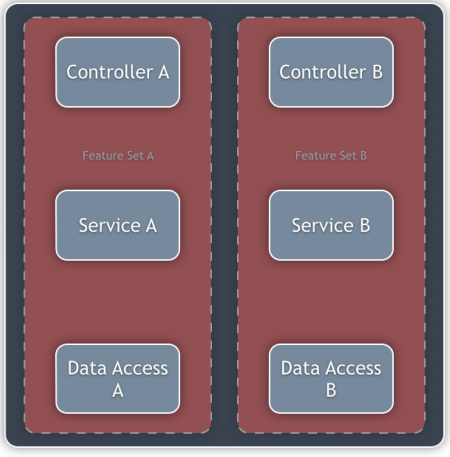
Layers are the primary organisation mechanism for the code. Terms such as "separation of concerns" are thrown around to justify this approach and layered architectures are generally thought of as a "good thing". Need to switch out the data access mechanism? No problem, everything is in one place. Each layer can also be tested in isolation to the others around

it, using appropriate mocking techniques, etc. The problem with layered architectures is that they often turn into a big ball of mud because, in Java anyway, you need to mark your classes as public for much of this to work. And once you mark classes as public, without discipline, code in any other layer of your architecture can use them.

Organising a codebase by layer makes it easy to see the overall structure of the software but there are trade-offs. For example, you need to delve inside multiple layers (e.g. packages, namespaces, etc) in order to make a change to a feature or user story. Also, many codebases end up looking eerily similar given the fairly standard approach to layering within enterprise systems. In Screaming Architecture, Uncle Bob Martin says that if you're looking at a codebase, it should scream something about the business domain. I think the same should be said for a software architecture diagram.

Package by feature

Packaging by layer isn't the only answer though and, instead of organising code by horizontal slice, "package by feature" seeks to do the opposite by organising code by vertical slice.



Package by feature (vertical slicing)

Now everything related to a single feature (or feature set) resides in a single place. You can still have a layered architecture, but the layers reside inside the feature packages. In other words, layering is the secondary organisation mechanism. The often cited benefit to package

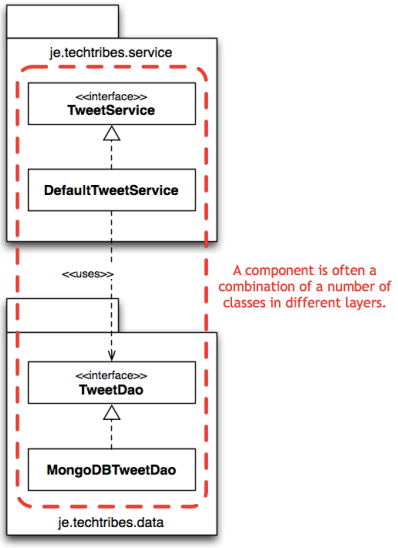
by feature is that it's "easier to navigate the codebase when you want to make a change to a feature", but this is a minor thing given the power of modern IDEs.

What you can do now though is hide feature specific classes and keep them out of sight from the rest of the codebase by marking them as package protected. The big question though is what happens when that new feature set C needs to access data from features A and B? Again, in Java, you'll need to start making classes publicly accessible from outside of the packages and the big ball of mud will likely again emerge.

The model-code gap

Although there's nothing particularly wrong with packaging code using either of these approaches, this code structure often never quite reflects the abstractions that we think about when we view the system from an architecture perspective. If you're using an object-oriented programming language, do you talk about "objects" when you're having architecture discussions? In my experience, the answer is "no". I typically hear people referring to concepts such as components and services instead. But what are these components and services? And where are they in the code?

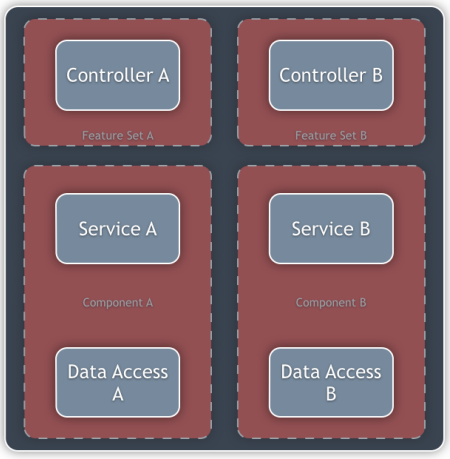
Often, a "component" that we see on an architecture diagram is actually implemented by a combination of classes across a number of different layers. To use an example, my component diagram for the techtribes.je Content Updater shows a "Tweet Component" that provides a way to store and access tweets in a MongoDB store. The diagram suggests that it's a single black box component, but my initial implementation was very different. The following diagram illustrates why.



For my initial implementation, I'd taken a "package by layer" approach and broken my tweet component down into a separate service and data access object. This is a great example of where the code doesn't quite reflect the architecture - the tweet component is a single box on an architecture diagram but implemented as a collection of classes across a layered architecture when you look at the code. Imagine having a large, complex codebase where the architecture diagrams tell a different story from the code. The easy way to fix this is to simply redraw the component diagram to show that it's really a layered architecture made up of services collaborating with data access objects. The result is a much more complex diagram but it also feels like that diagram is starting to show too much detail.

Packaging by component

The other option is to change the code to match the architectural vision and intent. And that's what I did. I reorganised the code to be packaged by component rather than packaged by layer. In essence, I merged the services and data access objects together into a single package so that I was left with a public interface and a package protected implementation. This is a hybrid approach with increased modularity and an architecturally-evident coding style as the primary goals.

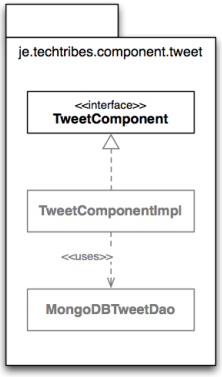


Package by component

The basic premise here is that I want my codebase to be made up of a number of coarse-grained components, with some sort of presentation layer (web UI, desktop UI, API, standalone app, etc) built on top. A "component" in this sense is a combination of the business and data access logic related to a specific thing (e.g. domain concept, bounded context or aggregate from Domain-Driven Design, etc).

If a new feature set C needs to access data related to A and B, it is forced to go through the public interface of components A and B. No direct access to the data access layer is allowed, and you can enforce this if you use Java's access modifiers properly. Again, "architectural layering" is a secondary organisation mechanism. For this to work, you have to stop using the public keyword by default.

Here's what the restructured TweetComponent looks like.



Each sub-package of je.techtribes.component houses a separate component, complete with its own *internal* layering and configuration. As far as possible, all of the internals are package scoped. You could potentially pull each component out and put it in its own project or source code repository to be versioned separately. This approach will likely seem familiar to you if you're building something that has a very explicit loosely coupled architecture such as a distributed system made up of distinct components or microservices.

Layers are an implementation detail

I'm fairly confident that most people are still building something more monolithic in nature though, despite thinking about their system in terms of components. I've certainly packaged *parts* of monolithic codebases using a similar approach in the past but it's tended to be fairly ad hoc. Let's be honest, organising code into packages isn't something that gets a lot of brain-time, particularly given the refactoring tools that we have at our disposal. Organising code by component lets you explicitly reflect the concept of "a component" from the architecture into the codebase.

With a package by component approach, the components are the architecturally significant structural building blocks and layers are now a component implementation detail rather than being the primary organisation mechanism.

Aligning software architecture and code

There's often very little mapping from the architecture into the code and back again. Effectively and efficiently visualising a software architecture can help to create a good shared vision within the team, which can help it go faster. Having a **simple and explicit mapping**

from the architecture to the code can help even further, particularly when you start looking at collaborative design and collective code ownership. Furthermore, it helps bring software architecture firmly back into the domain of the development team, which is ultimately where it belongs. Don't forget though, the style of architecture you're using needs to be reflected on your software architecture diagrams; whether that's layers, components, microservices or something else entirely.

Designing a software system based around components isn't "the one true way" but if you *are* building monolithic software systems and think of them as being made up of a number of smaller components, ensure that your codebase reflects this. Consider organising your code by component (rather than by layer or feature) to make the mapping between software architecture and code explicit. If it's hard to explain the structure of your software system, change it.

40. You don’t need a UML tool

When tasked with the job of designing a new software system, one of the first questions some people ask relates to the tooling that they should use. Such discussions usually focus around the Unified Modelling Language (UML) and whether their organisation has any licenses for some of the more well-known UML tools.

There are many types of UML tool

Unfortunately this isn’t an easy question to answer because there are lots of commercial and open source tools that can help you to do software architecture and design, all tending to approach the problem from a different perspective. At a high-level, they can be categorised as follows.

1. **Diagrams only**: There are many standalone UML tools and plug-ins for major IDEs that let you sketch simple UML diagrams. These are really useful if you want to be in control of your diagrams and what they portray but it’s easy for such diagrams to get out of date with reality over time. Microsoft Visio or OmniGraffle with UML templates installed are good starting points if you have access to them.
2. **Reverse engineering**: There are standalone UML tools and IDE plug-ins that allow you to create UML diagrams from code. This is great because you can quickly get the code and diagrams in sync, but often these diagrams become cluttered quickly because they typically include all of the detail (e.g. every property, method and relationship) by default.
3. **Round-trip engineering**: Many reverse engineering tools also allow you to do round-trip engineering, where changes made to the model are reflected in the code and vice versa. Again, this helps keeps code and diagrams in sync.
4. **Model-driven**: There are a few model-driven architecture (MDA) tools that let you drive the implementation of a software system from the model itself, usually by annotating the diagrams with desired characteristics and behaviours using languages such as Executable UML (xUML) or Object Constraint Language (OCL). These tools can offer a full end-to-end solution but you do need to follow a different and often rigid development process in order to benefit from them.

The simplest thing that could possibly work

Even this short summary of the categories of tools available makes for an overwhelming number of options. Rational Software Architect? Visio? PowerPoint? OmniGraffle? WebSequenceDiagrams.com? Which do you pick?!

The thing is though, you don't *need* a UML tool in order to architect and design software. I've conducted a number of informal polls during my conference talks over the past few years and only 10-20% of the audience said that they regularly used UML in their day to day work. Often a blank sheet of paper, flip chart or whiteboard together with a set of sticky notes or index cards is all you need, particularly when you have a group of people who want to undertake the design process in a collaborative way. Have you ever tried to get three or four people collaborating around a laptop screen?

Agile methods have been using this low-tech approach for capturing user stories, story walls and Kanban boards for a while now. In many cases, it's the simplest thing that could possibly work but nothing beats the pure visibility of having lots of stuff stuck to a whiteboard in the middle of your office. Unlike a Microsoft Project plan, nobody can resist walking past and having a look at all those sticky notes still in the "To do" column.

From a software design perspective, using a similarly low-tech approach frees you from worrying about the complexities of using the tooling and bending formal notation, instead letting you focus on the creative task of designing software. Simply start by sketching out the big picture and work down to lower levels of detail where necessary. Just remember that you need to explicitly think about things like traceability between levels of abstraction, conventions and consistency if you don't use a tool. For example, UML arrows have meaning and without a key it might not be obvious whether your freehand arrows are pointing towards dependencies or showing the direction that data flows. You can always record your designs in a more formal way using a UML tool later if you need to do so.

Uses for UML

The main reason for using informal boxes and lines diagrams over UML to visualise software architecture is that, in my opinion, UML isn't often a good fit for what I want to communicate. The information presented on my context, container and component diagrams *can* be achieved with a mix of use case, component and deployment diagrams but I personally don't find that the resulting diagrams are as easy to interpret given the notation. My C4 approach for visualising software architectures might not make use of UML then, but I still do use it on the software projects that I work on.

The tools surrounding UML allow it to be used in a number of ways, including fully fledged comprehensive models with their associated repositories through to diagrams that are reverse engineered from existing code. UML can also be used as a simple diagramming notation, either sketched on a whiteboard or within tools such as Microsoft Visio or OmniGraffle that have installable UML templates. Here's a summary of what I use UML for:

- **Processes and workflows**: If I'm building software that automates a process or is very workflow based, I'll often draw a simple UML activity diagram to represent it. UML activity diagrams seem to be ignored by many people but I find that the simple flow chart style of notation works well for a broad range of audiences.
- **Runtime behaviour**: My C4 approach is really only focussed on visualising the static structure of a software system, but often it's useful to present the system from a runtime perspective. UML sequence and collaboration diagrams are usually used to show how a number of classes collaborate at runtime to implement a particular user story, use case, feature, etc. These diagrams are still very useful even if you're not doing design down to the class level. Instead of showing a collection of collaborating classes, you can show collaborating containers or components instead.
- **Domain model**: I'll use a UML class diagram if I want to visualise a domain model, with the resulting diagrams typically only showing the most important attributes and relationships. I usually hide the method compartment of all classes on such diagrams.
- **Patterns and principles**: I'll often need to explain how patterns or principles are implemented within the codebase (e.g. in the Code section of a software guidebook), and a UML class diagram is the obvious way to do this. My advice here is keep the diagram simple and don't feel pressured into showing every last tiny piece of detail.
- **State charts**: UML state diagrams are a great way to visualise a state machine and the notation is fairly straightforward. Again, I find that people tend to forget UML state diagrams exist.
- **Deployment**: A UML deployment diagram can be a useful way to show how your containers or components are deployed. Often such a diagram is better presented as an informal boxes and lines diagram, but the option is there.

There is no silver bullet

Forget expensive tools. More often than not; a blank sheet of paper, flip chart or whiteboard is all you need, particularly when you have a group of people that want to undertake the design process in a collaborative way. Unfortunately there's no silver bullet when it comes to design tools though because everybody and every organisation works in a different way. Once

you're confident that you understand how to approach software architecture and design, only then is it time to start looking at software tools to help improve the design process.

The use of UML doesn't need to be an "adopt all or nothing" choice. A few well placed UML diagrams can really help you to present the more complex and detailed elements of a software system. If you're unfamiliar with UML, perhaps now is a good opportunity to make yourself aware of the various diagrams that are available. You don't *need* UML tools to do architecture and design, but they do have their uses. You don't need to use every type of diagram though!

41. Effective sketches

The Unified Modelling Language (UML) is a formal, standardised notation for communicating the design of software systems although many people favour boxes and lines style sketches instead. There's absolutely nothing wrong with this but you do trade-off diagram consistency for flexibility. The result is that many of these informal sketches use diagramming elements inconsistently and often need a narrative to accompany them.

If you are going to use "NoUML" diagrams (i.e. anything that *isn't* UML), here are some things to think about, both when you're drawing sketches on a whiteboard and if you decide to formalise them in something like Microsoft Visio afterwards.

Titles

The first thing that can really help people to understand a diagram is including a title. If you're using UML, the diagram elements will provide some information as to what the context of the diagram is, but that doesn't really help if you have a collection of diagrams that are all just boxes and lines. Try to make the titles short and meaningful. If the diagrams should be read in a specific order, make sure this is clear by numbering them.

Labels

You're likely to have a number of labels on your diagrams; including names of software systems, components, etc. Where possible, avoid using acronyms and if you do need to use acronyms for brevity, ensure that they are documented in a project glossary or with a key somewhere on the diagram. While the regular project team members might have an intimate understanding of common project acronyms, people outside or new to the project probably won't.

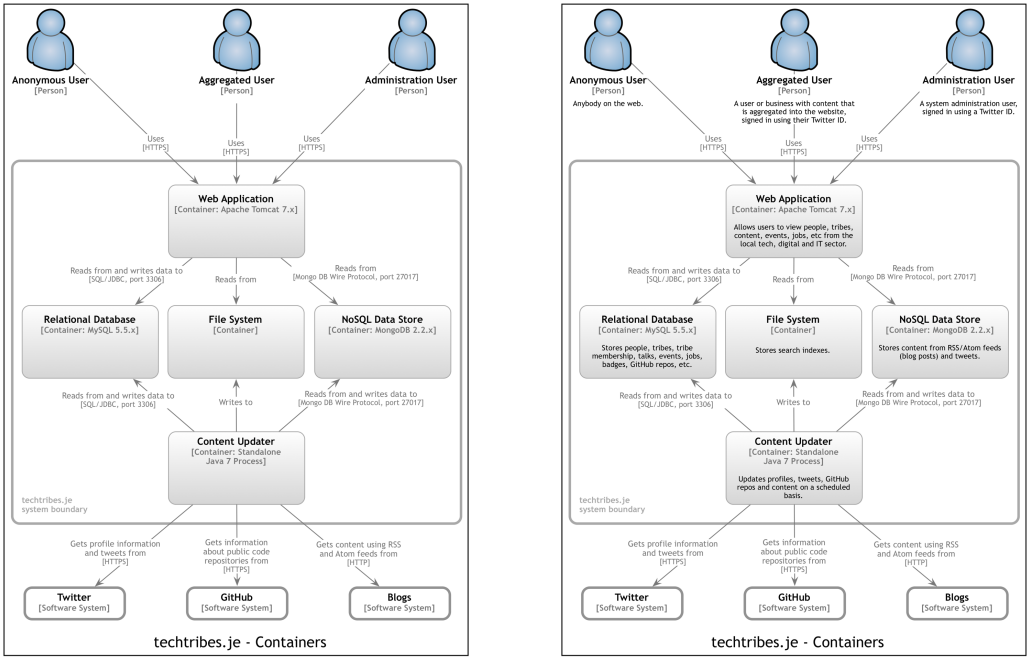
The exceptions here are acronyms used to describe technology choices, particularly if they are used widely across the industry. Examples include things like JMS (Java Message Service), POJO (plain old Java object) and WCF (Windows Communication Foundation). Let your specific context guide whether you need to explain these acronyms and if in doubt, play it safe and use the full name or include a key.

Shapes

Most boxes and lines style sketches that I've seen aren't just boxes and lines, with teams using a variety of shapes to represent elements within their software architecture. For example, you'll often see cylinders on a diagram and many people will interpret them to be a database of some description. Make sure that you include an explanation to confirm whether this is the case or not.

Responsibilities

If naming is one of the hardest things in software development, resist the temptation to have a diagram full of boxes that only contain names. A really simple way to add an additional layer of information to, and remove any ambiguity from, an architecture diagram is to annotate things like systems and components with a very short statement of what their responsibilities are. A bulleted list (7 Â± 2 items) or a short sentence work well. Provided it's kept short (and using a smaller font for this information can help too), adding responsibilities onto diagrams can help provide a really useful "at a glance" view of what the software system does and how it's been structured. Take a look at the following diagrams - which do you prefer?



Adding responsibilities to diagram elements can remove ambiguity

Lines

Lines are an important part of most architecture sketches, acting as the glue that holds all of the boxes (systems, containers, components, etc) together. The big problem with lines is exactly this though, they tend to be thought of as the things that hold the other, more significant elements of the diagram together and don't get much focus themselves. Whenever you're drawing lines on sketches, ensure you use them consistently and that they have a clear purpose. For example:

- **Line style** (solid, dotted, dashed, etc): Is the line style relevant and, if so, what does it mean?
- **Arrows**: Do arrows point in the direction of dependencies (e.g. like UML "uses" relationships) or do they indicate the direction in which data normally flows?

Often annotations on the lines (e.g. "uses", "sends data to", "downloads report from", etc) can help to clarify the direction in which arrows are pointing, but watch out for any lines that have arrows on both ends!

Colour

Software architecture diagrams don't have to be black and white. Colour can be used to provide differentiation between diagram elements or to ensure that emphasis is/isn't placed on them. If you're going to use colour, and I recommend that you should, particularly when sketching, make sure that it's obvious what your colour coding scheme is by including a reference to those colours in a key. Colour can make a world of difference. All you need are some different coloured whiteboard/marker pens and a little imagination.

Borders

Adding borders (e.g. double lines, coloured lines, dashed lines, etc) around diagram elements can be a great way to add emphasis or to group related elements together. If you do this, make sure that it's obvious what the border means, either by labelling the border or by including an explanation in the diagram key.

Layout

Using electronic drawing tools such as Microsoft Visio or OmniGraffle makes laying out diagram elements easier since you can move them around as much as you want. Many people prefer to design software while stood in front of a whiteboard or flip chart though, particularly because it provides a better environment for collaboration. The trade-off here is that you have to think more about the layout of diagram elements because it can become a pain if you're having to constantly draw, erase and redraw elements of your diagrams when you run out of space.

Examples of where sticky notes and index cards have been used instead of drawing boxes

Sticky notes and index cards can help to give you some flexibility if you use them as a substitute for drawing boxes. And if you're using a Class-Responsibility-Collaboration style technique to identify candidate classes/components/services, you can use the resulting cards as a way to start creating your diagrams.

Need to move some elements? No problem, just move them. Need to remove some elements? No problem, just take them off the diagram and throw them away. Sticky notes and index cards *can* be a great way to get started with software architecture sketches, but I do tend to find that the resulting diagrams look cluttered. Oh, and sticky notes often don't stick well to whiteboards, so have some blu-tack handy!

Orientation

Imagine you're designing a 3-tier web application that consists of a web-tier, a middle-tier and a database. If you're drawing a container diagram, which way up do you draw it? Users and web-tier at the top with the database at the bottom? The other way up? Or perhaps you lay out the elements from left to right?

Most of the architecture diagrams that I see have the users and web-tier at the top, but this isn't always the case. Sometimes those same diagrams will be presented upside-down or back-to-front, perhaps illustrating the author's (potentially subconscious) view that the database

is the centre of their universe. Although there is no "correct" orientation, drawing diagrams "upside-down" from what we might consider the norm can either be confusing or used to great effect. The choice is yours.

Keys

One of the advantages of using UML is that it provides a standardised set of diagram elements for each type of diagram. In theory, if somebody is familiar with these elements, they should be able to understand your diagram. In the real world this isn't always the case, but this certainly *isn't* the case with boxes and lines sketches where the people drawing the diagrams are inventing the notation as they go along. Again, there's nothing wrong with this but make sure that you give everybody an equal chance of understanding your creations by including a small key somewhere on or nearby the diagram. Here are the sort of things that you might want to include explanations of:

- Shapes
- Lines
- Colours
- Borders
- Acronyms

You can sometimes interpret the use of diagram elements without a key (e.g. "the grey boxes seem to be the existing systems and red is the new stuff") but I would recommend playing it safe and adding a key. Even the seemingly obvious can be misinterpreted by people with different backgrounds and experience.

Diagram review checklist

The software architecture process is about introducing structure and vision into software projects, so when reviewing architecture diagrams, here are a number of things that you might want to assert to ensure that this is the case. This checklist is applicable for diagrams produced during the initial architecture process, as well as those produced to retrospectively document an existing software system.

1. I can see and understand the solution from multiple levels of abstraction.

2. I understand the big picture; including who is going to use the system (e.g. roles, personas, etc) and what the dependencies are on the existing IT environment (e.g. existing systems).
3. I understand the logical containers and the high-level technology choices that have been made (e.g. web servers, databases, etc).
4. I understand what the major components are and how they are used to satisfy the important user stories/use cases/features/etc.
5. I understand what all of the components are, what their responsibilities are and can see that all components have a home.
6. I understand the notation, conventions, colour coding, etc used on the diagrams.
7. I can see the traceability between diagrams and diagramming elements have been used consistently.
8. I understand what the business domain is and can see a high-level view of the functionality that the software system provides.
9. I understand the implementation strategy (frameworks, libraries, APIs, etc) and can almost visualise how the system will be or has been implemented.

Listen for questions

As a final note, keep an ear open for any questions raised or clarifications being made while diagrams are being drawn. If you find yourself saying things like, “just to be clear, these arrows represent data flows”, make sure that this information ends up on a key/legend somewhere.

42. C4++

The C4 model focusses on describing and communicating the static structure of a software system; from the big picture down to the components and the classes that implement them. Although this is often sufficient to describe a software system, sometimes it can be useful to draw some additional diagrams to highlight different aspects.

Enterprise context

The C4 model provides a static view of a single software system but, in the real-world, software systems never live in isolation. For this reason, and particularly if you manage a collection of software systems, it's often useful to understand how all of these software systems fit together within the bounds of an enterprise. To do this, I'll simply add another diagram that sits on top of the C4 diagrams, to show the enterprise context from an IT perspective. C4 therefore becomes C5, with this extra enterprise context diagram showing:

- The organisational boundary.
- Internal and external users.
- Internal and external systems (including a high-level summary of their responsibilities and data owned).

Essentially this becomes a high-level map of the software systems at the enterprise level, with a C4 drill-down for each software system of interest. As a caveat, I do appreciate that enterprise architecture isn't simply about technology but, in my experience, many organisations don't have an enterprise architecture view of their IT landscape. In fact, it shocks me how often I come across organisations of all sizes that lack such a holistic view, especially considering IT is usually a key part of the way they implement business processes and service customers. Sketching out the enterprise context from a technology perspective at least provides a way to think outside of the typical silos that form around IT systems.

User interface mockups and wireframes

Mocking up user interfaces with tools such as Balsamiq is a fantastic way to understand what is needed from a software system and to prototype ideas. Such sketches, mockups and wireframes will provide a view of the software system that is impossible with the C4 model.

Domain model

Most real-world software systems represent business domains that are non-trivial and, if this is the case, a diagram summarising the key domain concepts can be a useful addition. The format I use for these is a UML class diagram where each UML class represents an entity in the domain. For each entity, I'll typically include important attributes/properties and the relationships between entities. A domain model is useful regardless of whether you're following a Domain-driven design approach or not.

Sequence and collaboration diagrams

The C4 model only describes the static structure of a software system and, of course, running software isn't static. For this reason, often it's useful to create diagrams to illustrate what happens at runtime for important use cases, user stories or scenarios. To do this, I simply take the concept of sequence and collaboration diagrams from UML and apply them to the static elements in the C4 model.

For example, you could illustrate how a use case is implemented by drawing a sequence diagram of how components interact at runtime. Or you could show the interaction between containers if you have more of a microservices style of architecture, where every service is deployed in a separate container. Either way, a runtime view of the software can be useful to describe complex interactions or interactions that are not evident from reading the code alone (e.g. the send/receipt of asynchronous messages).

Business process and workflow models

Related to sequence and collaboration diagrams are process models. Sometimes I want to summarise a particular business process or user workflow that a software system implements, without getting into the technicalities of how it's implemented. A UML activity diagram or traditional flowchart is a great way to do this.

Infrastructure model

A map of your infrastructure can be a useful thing to capture because of the obvious relationship between software and infrastructure. There are a number of ways to describe infrastructure, ranging from infrastructure diagrams in Microsoft Visio through to automated scripts that manage and provision infrastructure on a cloud provider.

Deployment model

It's often useful to describe the mapping between containers and infrastructure. For example, a database-driven website could be deployed onto a single server or across hundreds of them, depending on the need to support scalability, resilience, security, etc. Again, the deployment model could be described as a bunch of automated scripts or a simple deployment diagram.

And more

The diagrams from the C4 model plus those I've listed here are usually enough for me to adequately describe how a software system is designed, built and works. I try to keep the number of diagrams I use to do this to a minimum and I advise you to do the same. Some diagrams *can* be automatically generated (e.g. an entity relationship diagram for a database schema) but if you need an A0 sheet of paper to display it, you should consider whether the diagram is actually useful. Do add more diagrams if you need to describe something that isn't listed here and if a particular diagram doesn't add any value, simply discard it.

Philippe Kruchten's 4+1 architectural view model and Software Systems Architecture by Eoin Woods and Nick Rozanski are my recommended starting points for information about the views you may want to consider in addition to those provided by the C4 model.

43. 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?

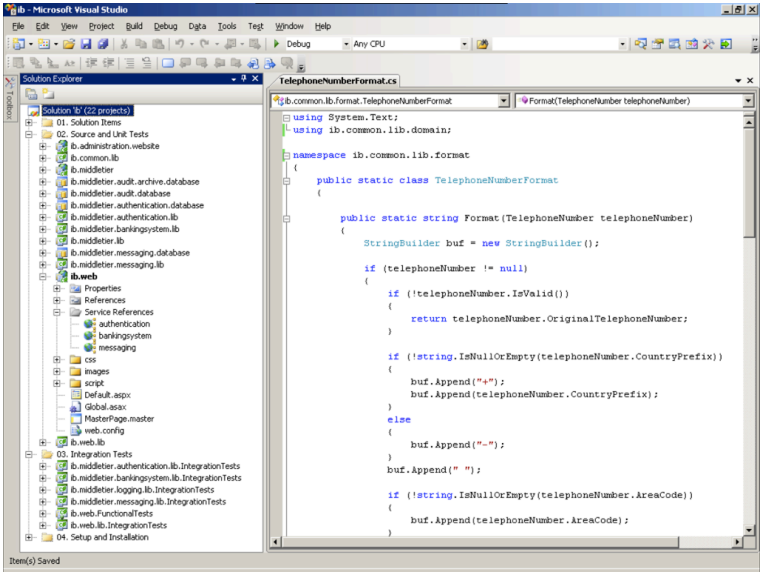
V Documenting software

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

44. 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?



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.

45. 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.

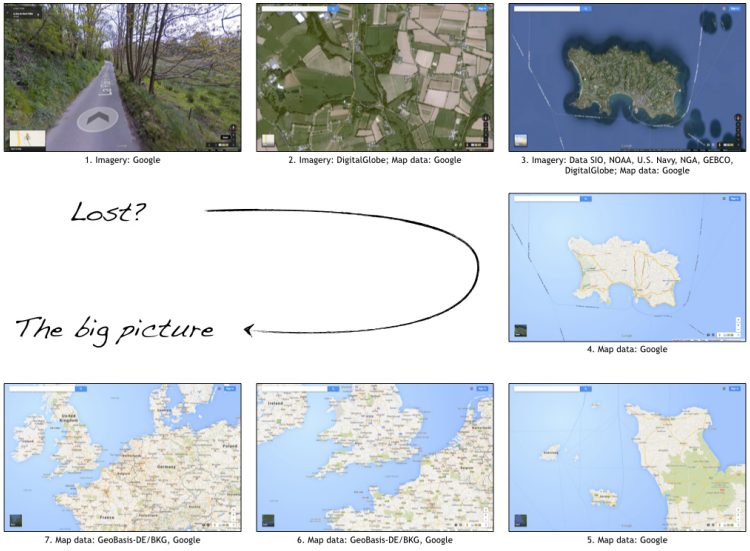
I'm also a firm believer that many software teams have a duty to deliver some supplementary documentation along with the codebase, especially those that are building the software under an outsourcing and/or offshoring contract. I've seen IT consulting organisations deliver highly complex software systems 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).



From the detail to the big picture

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 perviously 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.

46. Context

A context section should be one of the first sections of the software guidebook and simply used to set the scene for the remainder of the document.

Intent

A context section should answer the following types of questions:

- What is this software project/product/system all about?
- What is it that's being built?
- How does it fit into the existing environment? (e.g. systems, business processes, etc)
- Who is using it? (users, roles, actors, personas, etc)

Structure

The context section doesn't need to be long; a page or two is sufficient and a context diagram is a great way to tell most of the story.

Motivation

I've seen software architecture documents that don't start by setting the scene and, 30 pages in, you're still none the wiser as to why the software exists and where it fits into the existing IT environment. A context section doesn't take long to create but can be immensely useful, especially for those outside of the team.

Audience

Technical and non-technical people, inside and outside of the immediate software development team.

Required

Yes, all software guidebooks should include an initial context section to set the scene.

47. Functional Overview

Even though the purpose of a software guidebook isn't to explain what the software does in detail, it can be useful to expand on the context and summarise what the major functions of the software are.

Intent

This section allows you to summarise what the key functions of the system are. It also allows you to make an explicit link between the functional aspects of the system (use cases, user stories, etc) and, if they are significant to the architecture, to explain why. A functional overview should answer the following types of questions:

- Is it clear what the system actually does?
- Is it clear which features, functions, use cases, user stories, etc are significant to the architecture and why?
- Is it clear who the important users are (roles, actors, personas, etc) and how the system caters for their needs?
- It is clear that the above has been used to shape and define the architecture?

Alternatively, if your software automates a business process or workflow, a functional view should answer questions like the following:

- Is it clear what the system does from a process perspective?
- What are the major processes and flows of information through the system?

Structure

By all means refer to existing documentation if it's available; and by this I mean functional specifications, use case documents or even lists of user stories. However, it's often useful to summarise the business domain and the functionality provided by the system. Again, diagrams can help, and you could use a UML use case diagram or a collection of simple

wireframes showing the important parts of the user interface. Either way, remember that the purpose of this section is to provide an *overview*.

Alternatively, if your software automates a business process or workflow, you could use a flow chart or UML activity diagram to show the smaller steps within the process and how they fit together. This is particularly useful to highlight aspects such as parallelism, concurrency, where processes fork or join, etc.

Motivation

This doesn't necessarily need to be a long section, with diagrams being used to provide an overview. Where a context section summarises how the software fits into the existing environment, this section describes what the system actually does. Again, this is about providing a summary and setting the scene rather than comprehensively describing every user/system interaction.

Audience

Technical and non-technical people, inside and outside of the immediate software development team.

Required

Yes, all software guidebooks should include a *summary* of the functionality provided by the software.

48. Quality Attributes

With the functional overview section summarising the functionality, it's also worth including a separate section to summarise the quality attributes/non-functional requirements.

Intent

This section is about summarising the key quality attributes and should answer the following types of questions:

- Is there a clear understanding of the quality attributes that the architecture must satisfy?
- Are the quality attributes SMART (specific, measurable, achievable, relevant and timely)?
- Have quality attributes that are usually taken for granted been explicitly marked as out of scope if they are not needed? (e.g. "user interface elements will only be presented in English" to indicate that multi-language support is not explicitly catered for)
- Are any of the quality attributes unrealistic? (e.g. true 24x7 availability is typically very costly to implement *inside* many organisations)

In addition, if any of the quality attributes are deemed as "architecturally significant" and therefore influence the architecture, why not make a note of them so that you can refer back to them later in the document.

Structure

Simply listing out each of the quality attributes is a good starting point. Examples include:

- Performance (e.g. latency and throughput)
- Scalability (e.g. data and traffic volumes)
- Availability (e.g. uptime, downtime, scheduled maintenance, 24x7, 99.9%, etc)
- Security (e.g. authentication, authorisation, data confidentiality, etc)

- Extensibility
- Flexibility
- Auditing
- Monitoring and management
- Reliability
- Failover/disaster recovery targets (e.g. manual vs automatic, how long will this take?)
- Business continuity
- Interoperability
- Legal, compliance and regulatory requirements (e.g. data protection act)
- Internationalisation (i18n) and localisation (L10n)
- Accessibility
- Usability
- …

Each quality attribute should be precise, leaving no interpretation to the reader. Examples where this isn't the case include:

- "the request must be serviced quickly"
- "there should be no overhead"
- "as fast as possible"
- "as small as possible"
- "as many customers as possible"
- …

Motivation

If you've been a good software architecture citizen and have proactively considered the quality attributes, why not write them down too? Typically, quality attributes are not given to you on a plate and an amount of exploration and refinement is usually needed to come up with a list of them. Put simply, writing down the quality attributes removes any ambiguity both now and during maintenance/enhancement work in the future.

Audience

Since quality attributes are mostly technical in nature, this section is really targeted at technical people in the software development team.

Required

Yes, all software guidebooks should include a summary of the quality attributes/non-functional requirements as they usually shape the resulting software architecture in some way.

49. Constraints

Software lives within the context of the real-world, and the real-world has constraints. This section allows you to state these constraints so it's clear that you are working within them and obvious how they affect your architecture decisions.

Intent

Constraints are typically imposed upon you but they aren't necessarily "bad", as reducing the number of available options often makes your job designing software easier. This section allows you to explicitly summarise the constraints that you're working within and the decisions that have already been made for you.

Structure

As with the quality attributes, simply listing the known constraints and briefly summarising them will work. Example constraints include:

- Time, budget and resources.
- Approved technology lists and technology constraints.
- Target deployment platform.
- Existing systems and integration standards.
- Local standards (e.g. development, coding, etc).
- Public standards (e.g. HTTP, SOAP, XML, XML Schema, WSDL, etc).
- Standard protocols.
- Standard message formats.
- Size of the software development team.
- Skill profile of the software development team.
- Nature of the software being built (e.g. tactical or strategic).
- Political constraints.
- Use of internal intellectual property.
- etc

If constraints do have an impact, it's worth summarising them (e.g. what they are, why they are being imposed and who is imposing them) and stating how they are significant to your architecture.

Motivation

Constraints have the power to massively influence the architecture, particularly if they limit the technology that can be used to build the solution. Documenting them prevents you having to answer questions in the future about why you've seemingly made some odd decisions.

Audience

The audience for this section includes everybody involved with the software development process, since some constraints are technical and some aren't.

Required

Yes, all software guidebooks should include a summary of the constraints as they usually shape the resulting software architecture in some way. It's worth making these constraints explicit at all times, even in environments that have a very well known set of constraints (e.g. "all of our software is ASP.NET against a SQL Server database") because constraints have a habit of changing over time.

50. Principles

The principles section allows you to summarise those principles that have been used (or you are using) to design and build the software.

Intent

The purpose of this section is to simply make it explicit which principles you are following. These could have been explicitly asked for by a stakeholder or they could be principles that *you* (i.e. the software development team) want to adopt and follow.

Structure

If you have an existing set of software development principles (e.g. on a development wiki), by all means simply reference it. Otherwise, list out the principles that you are following and accompany each with a short explanation or link to further information. Example principles include:

- Architectural layering strategy.
- No business logic in views.
- No database access in views.
- Use of interfaces.
- Always use an ORM.
- Dependency injection.
- The Hollywood principle (don't call us, we'll call you).
- High cohesion, low coupling.
- Follow SOLID (Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle).
- DRY (don't repeat yourself).
- Ensure all components are stateless (e.g. to ease scaling).
- Prefer a rich domain model.
- Prefer an anaemic domain model.

- Always prefer stored procedures.
- Never use stored procedures.
- Don't reinvent the wheel.
- Common approaches for error handling, logging, etc.
- Buy rather than build.
- etc

Motivation

The motivation for writing down the list of principles is to make them explicit so that everybody involved with the software development understands what they are. Why? Put simply, principles help to introduce consistency into a codebase by ensuring that common problems are approached in the same way.

Audience

The audience for this section is predominantly the technical people in the software development team.

Required

Yes, all software guidebooks should include a summary of the principles that have been or are being used to develop the software.

51. Software Architecture

The software architecture section is your "big picture" view and allows you to present the structure of the software. Traditional software architecture documents typically refer to this as a "conceptual view" or "logical view", and there is often confusion about whether such views should refer to implementation details such as technology choices.

Intent

The purpose of this section is to summarise the software architecture of your software system so that the following questions can be answered:

- What does the "big picture" look like?
- Is there are clear structure?
- Is it clear how the system works from the "30,000 foot view"?
- Does it show the major containers and technology choices?
- Does it show the major components and their interactions?
- What are the key internal interfaces? (e.g. a web service between your web and business tiers)

Structure

I use the container and component diagrams as the main focus for this section, accompanied by a short narrative explaining what the diagram is showing plus a summary of each container/component.

Sometimes UML sequence or collaboration diagrams showing component interactions can be a useful way to illustrate how the software satisfies the major use cases/user stories/etc. Only do this if it adds value though and resist the temptation to describe how *every* use case/user story works!

Motivation

The motivation for writing this section is that it provides the maps that people can use to get an overview of the software and help developers navigate the codebase.

Audience

The audience for this section is predominantly the technical people in the software development team.

Required

Yes, all software guidebooks should include a software architecture section because it's essential that the overall software structure is well understood by everybody on the development team.

52. External Interfaces

Interfaces, particularly those that are external to your software system, are one of the riskiest parts of any software system so it's very useful to summarise what the interfaces are and how they work.

Intent

The purpose of this section is to answer the following types of questions:

- What are the key external interfaces?
  - e.g. between your system and other systems (whether they are internal or external to your environment)
  - e.g. any APIs that you are exposing for consumption
  - e.g. any files that your are exporting from your system
- Has each interface been thought about from a technical perspective?
  - What is the technical definition of the interface?
  - If messaging is being used, which queues (point-to-point) and topics (pub-sub) are components using to communicate?
  - What format are the messages (e.g. plain text or XML defined by a DTD/Schema)?
  - Are they synchronous or asynchronous?
  - Are asynchronous messaging links guaranteed?
  - Are subscribers durable where necessary?
  - Can messages be received out of order and is this a problem?
  - Are interfaces idempotent?
  - Is the interface always available or do you, for example, need to cache data locally?
  - How is performance/scalability/security/etc catered for?
- Has each interface been thought about from a non-technical perspective?
  - Who has ownership of the interface?
  - How often does the interface change and how is versioning handled?
  - Are there any service-level agreements in place?

Structure

I tend to simply list out the interfaces (in the form "From X to Y") along with a short narrative that describes the characteristics of the interface. To put the interfaces in context, I may include a simplified version of the container or component diagrams that emphasise the interfaces.

Motivation

The motivation for writing this section is to ensure that the interfaces have been considered and are understood because they're typically risky and easy to ignore. If interface details haven't been captured, this section can then act as a checklist and be a source of work items for the team to undertake.

Audience

The audience for this section is predominantly the technical people in the software development team.

Required

No, I only include this section if I'm building something that has one or more complex interfaces. For example, I wouldn't include it for a standard "web server -> database" style of software system, but I would include this section if that web application needed to communicate with an external system where it was consuming information via an API.

53. Code

Although other sections of the software guidebook describe the overall architecture of the software, often you'll want to present lower level details to explain how things work. This is what the code section is for. Some software architecture documentation templates call this the "implementation view" or the "development view".

Intent

The purpose of the code section is to describe the implementation details for parts of the software system that are important, complex, significant, etc. For example, I've written about the following for software projects that I've been involved in:

- Generating/rendering HTML: a short description of an in-house framework that was created for generating HTML, including the major classes and concepts.
- Data binding: our approach to updating business objects as the result of HTTP POST requests.
- Multi-page data collection: a short description of an in-house framework we used for building forms that spanned multiple web pages.
- Web MVC: an example usage of the web MVC framework that was being used.
- Security: our approach to using Windows Identity Foundation (WIF) for authentication and authorisation.
- Domain model: an overview of the important parts of the domain model.
- Component framework: a short description of the framework that we built to allow components to be reconfigured at runtime.
- Configuration: a short description of the standard component configuration mechanism in use across the codebase.
- Architectural layering: an overview of the layering strategy and the patterns in use to implement it.
- Exceptions and logging: a summary of our approach to exception handling and logging across the various architectural layers.
- Patterns and principles: an explanation of how patterns and principles are implemented.
- etc

Structure

Keep it simple, with a short section for each element that you want to describe and include diagrams if they help the reader. For example, a high-level UML class and/or sequence diagram can be useful to help explain how a bespoke in-house framework works. Resist the temptation to include all of the detail though, and don't feel that your diagrams need to show everything. I prefer to spend a few minutes sketching out a high-level UML class diagram that shows selected (important) attributes and methods rather than using the complex diagrams that can be generated automatically from your codebase with UML tools or IDE plugins. Keeping any diagrams at a high-level of detail means that they're less volatile and remain up to date for longer because they can tolerate small changes to the code and yet remain valid.

Motivation

The motivation for writing this section is to ensure that everybody understands how the important/significant/complex parts of the software system work so that they can maintain, enhance and extend them in a consistent and coherent manner. This section also helps new members of the team get up to speed quickly.

Audience

The audience for this section is predominantly the technical people in the software development team.

Required

No, but I usually include this section for anything other than a trivial software system.

54. Data

The data associated with a software system is usually not the primary point of focus yet it's arguably more important than the software itself, so often it's useful to document something about it.

Intent

The purpose of the data section is to record anything that is important from a data perspective, answering the following types of questions:

- What does the data model look like?
- Where is data stored?
- Who owns the data?
- How much storage space is needed for the data? (e.g. especially if you're dealing with "big data")
- What are the archiving and back-up strategies?
- Are there any regulatory requirements for the long term archival of business data?
- Likewise for log files and audit trails?
- Are flat files being used for storage? If so, what format is being used?

Structure

Keep it simple, with a short section for each element that you want to describe and include domain models or entity relationship diagrams if they help the reader. As with my advice for including class diagrams in the code section, keep any diagrams at a high level of abstraction rather than including every field and property. If people need this type of information, they can find it in the code or database (for example).

Motivation

The motivation for writing this section is that the data in most software systems tends to outlive the software. This section can help anybody that needs to maintain and support the data on an ongoing basis, plus anybody that needs to extract reports or undertake business intelligence activities on the data. This section can also serve as a starting point for when the software system is inevitably rewritten in the future.

Audience

The audience for this section is predominantly the technical people in the software development team along with others that may help deploy, support and operate the software system.

Required

No, but I usually include this section for anything other than a trivial software system.

55. Infrastructure Architecture

While most of the software guidebook is focussed on the software itself, we do also need to consider the infrastructure because software architecture is about software *and* infrastructure.

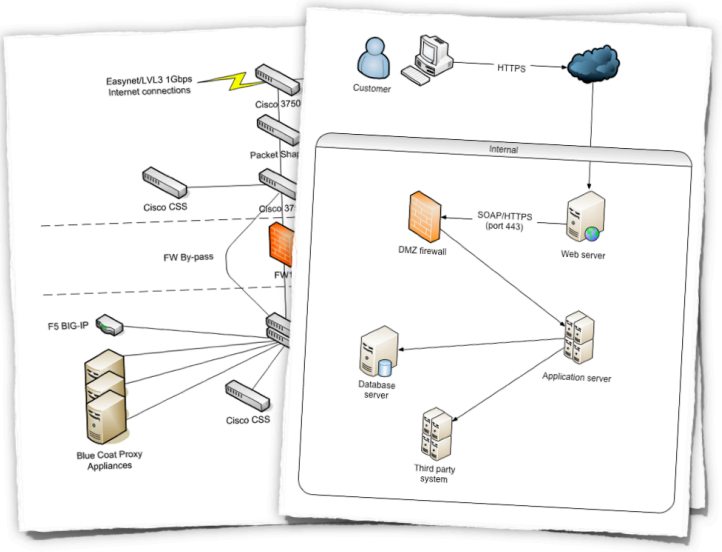
Intent

This section is used to describe the physical/virtual hardware and networks on which the software will be deployed. Although, as a software architect, you may not be involved in designing the infrastructure, you do need to understand that it's sufficient to enable you to satisfy your goals. The purpose of this section is to answer the following types of questions:

- Is there a clear physical architecture?
- What hardware (virtual or physical) does this include across all tiers?
- Does it cater for redundancy, failover and disaster recovery if applicable?
- Is it clear how the chosen hardware components have been sized and selected?
- If multiple servers and sites are used, what are the network links between them?
- Who is responsible for support and maintenance of the infrastructure?
- Are there central teams to look after common infrastructure (e.g. databases, message buses, application servers, networks, routers, switches, load balancers, reverse proxies, internet connections, etc)?
- Who owns the resources?
- Are there sufficient environments for development, testing, acceptance, pre-production, production, etc?

Structure

The main focus for this section is usually an infrastructure/network diagram showing the various hardware/network components and how they fit together, with a short narrative to accompany the diagram.



Example infrastructure diagrams, typically created in Microsoft Visio

If I'm working in a large organisation, there are usually infrastructure architects who look after the infrastructure architecture and create these diagrams for me. Sometimes this isn't the case though and I will draw them myself.

Motivation

The motivation for writing this section is to force me (the software architect) to step outside of my comfort zone and think about the infrastructure architecture. If I don't understand it, there's a chance that the software architecture I'm creating won't work or that the existing infrastructure won't support what I'm trying to do.

Audience

The audience for this section is predominantly the technical people in the software development team along with others that may help deploy, support and operate the software system.

Required

Yes, an infrastructure architecture section should be included in all software guidebooks because it illustrates that the infrastructure is understood and has been considered.

56. Deployment

The deployment section is simply the mapping between the software and the infrastructure.

Intent

This section is used to describe the mapping between the software (e.g. containers) and the infrastructure. Sometimes this will be a simple one-to-one mapping (e.g. deploy a web application to a single web server) and at other times it will be more complex (e.g. deploy a web application across a number of servers in a server farm). This section answers the following types of questions:

- How and where is the software installed and configured?
- Is it clear how the software will be deployed across the infrastructure elements described in the infrastructure architecture section? (e.g. one-to-one mapping, multiple containers per server, etc)
- If this is still to be decided, what are the options and have they been documented?
- Is it understood how memory and CPU will be partitioned between the processes running on a single piece of infrastructure?
- Are any containers and/or components running in an active-active, active-passive, hot-standby, cold-standby, etc formation?
- Has the deployment and rollback strategy been defined?
- What happens in the event of a software or infrastructure failure?
- Is it clear how data is replicated across sites?

Structure

There are a few ways to structure this section:

1. Tables: simple textual tables that show the mapping between software containers and/or components with the infrastructure they will be deployed on.

2. Diagrams: UML deployment diagrams or modified versions of the diagrams from the infrastructure architecture section showing where software will be running.

In both cases, I may use colour coding the designate the runtime status of software and infrastructure (e.g. active, passive, hot-standby, warm-standby, cold-standby, etc).

Motivation

The motivation for writing this section is to ensure that I understand how the software is going to work once it gets out of the development environment and also to document the often complex deployment of enterprise software systems.

This section can provide a useful overview, even for those teams that have adopted continuous delivery and have all of their deployment scripted using tools such as Puppet or Chef.

Audience

The audience for this section is predominantly the technical people in the software development team along with others that may help deploy, support and operate the software system.

Required

Yes, a deployment section should be included in all software guidebooks because it can help to solve the often mysterious question of where the software will be, or has been, deployed.

57. Operation and Support

The operations and support section allows you to describe how people will run, monitor and manage your software.

Intent

Most systems will be subject to support and operational requirements, particularly around how they are monitored, managed and administered. Including a dedicated section in the software guidebook lets you be explicit about how your software will or does support those requirements. This section should address the following types of questions:

- Is it clear how the software provides the ability for operation/support teams to monitor and manage the system?
- How is this achieved across all tiers of the architecture?
- How can operational staff diagnose problems?
- Where are errors and information logged? (e.g. log files, Windows Event Log, SMNP, JMX, WMI, custom diagnostics, etc)
- Do configuration changes require a restart?
- Are there any manual housekeeping tasks that need to be performed on a regular basis?
- Does old data need to be periodically archived?

Structure

This section is usually fairly narrative in nature, with a heading for each related set of information (e.g. monitoring, diagnostics, configuration, etc).

Motivation

I've undertaken audits of existing software systems in the past and we've had to spend time hunting for basic information such as log file locations. Times change and team members move on, so recording this information can help prevent those situations in the future where nobody understands how to operate the software.

Audience

The audience for this section is predominantly the technical people in the software development team along with others that may help deploy, support and operate the software system.

Required

Yes, an operations and support section should be included in all software guidebooks, unless you like throwing software into a black hole and hoping for the best!

58. Development Environment

The development environment section allows you to summarise how people new to your team install tools and setup a development environment in order to work on the software.

Intent

The purpose of this section is to provide instructions that take somebody from a blank operating system installation to a fully-fledged development environment.

Structure

The type of things you might want to include are:

- Pre-requisite versions of software needed.
- Links to software downloads (either on the Internet or locally stored).
- Links to virtual machine images.
- Environment variables, Windows registry settings, etc.
- Host name entries.
- IDE configuration.
- Build and test instructions.
- Database population scripts.
- Usernames, passwords and certificates for connecting to development and test services.
- Links to build servers.
- etc

If you're using automated solutions (such as Vagrant, Docker, Puppet, Chef, Rundeck, etc), it's still worth including some brief information about how these solutions work, where to find the scripts and how to run them.

Motivation

The motivation for this section is to ensure that new developers can be productive as quickly as possible.

Audience

The audience for this section is the technical people in the software development team, especially those who are new to the team.

Required

Yes, because this information is usually lost and it's essential if the software will be maintained by a different set of people from the original developers.

59. Decision Log

The final thing you might consider including in a software guidebook is a log of the decisions that have been made during the development of the software system.

Intent

The purpose of this section is to simply record the major decisions that have been made, including both the technology choices (e.g. products, frameworks, etc) and the overall architecture (e.g. the structure of the software, architectural style, decomposition, patterns, etc). For example:

- Why did you choose technology or framework "X" over "Y" and "Z"?
- How did you do this? Product evaluation or proof of concept?
- Were you forced into making a decision about "X" based upon corporate policy or enterprise architecture strategies?
- Why did you choose the selected software architecture? What other options did you consider?
- How do you know that the solution satisfies the major non-functional requirements?
- etc

Structure

Again, keep it simple, with a short paragraph describing each decision that you want to record. Do refer to other resources such as proof of concepts, performance testing results or product evaluations if you have them.

Motivation

The motivation for recording the significant decisions is that this section can act as a point of reference in the future. All decisions are made given a specific context and usually have trade-offs. There is usually never a perfect solution to a given problem. Articulating the decision

making process after the event is often complex, particularly if you're explaining the decision to people who are joining the team or you're in an environment where the context changes on a regular basis.

Although "nobody ever gets fired for buying IBM", perhaps writing down the fact that corporate policy forced you into using IBM WebSphere over Apache Tomcat will save you some tricky conversations in the future.

Audience

The audience for this section is predominantly the technical people in the software development team along with others that may help deploy, support and operate the software system.

Required

No, but I usually include this section if we (the team) spend more than a few minutes thinking about something significant such as a technology choice or an architectural style. If in doubt, spend a couple of minutes writing it down, especially if you work for a consulting organisation who is building a software system under an outsourcing agreement for a customer.

60. 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.

61. 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.

62. Quantifying risk

Identifying risks is a crucial part of doing "just enough up front design" and, put simply, a risk is something bad that may happen in the future, such as a chosen technology not being able to fulfil the promises that the vendor makes. Not all risks are created equal though, with some being more important than others. For example, a risk that may make your software project fail should be treated as a higher priority than something that may cause the team some general discomfort.

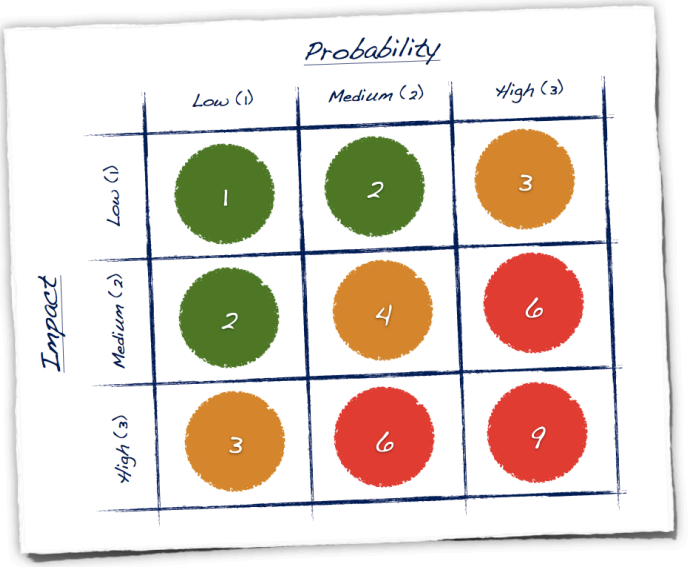
Assuming that you have a list of risks (and risk-storming is a great technique for doing this), how do you quantify each of those risks and assess their relative priorities? There are a number of well established approaches to quantifying risk; including assigning a value of low, medium or high or even a simple numeric value between 1 and 10, with higher numbers representing higher levels of risk.

Probability vs impact

A good way to think about risk is to separate out the *probability* of that risk happening from the negative *impact* of it happening.

- **Probability**: How likely is it that the risk will happen? Do you think that the chance is remote or would you be willing to bet cash on it?
- **Impact**: What is the negative impact if the risk does occur? Is there general discomfort for the team or is it back to the drawing board? Or will it cause your software project to fail?

Both probability and impact can be quantified as low, medium, high or simply as a numeric value. If you think about probability and impact separately, you can then plot the overall score on a matrix by multiplying the scores together as illustrated in the following diagram.



A probability/impact matrix for quantifying risk

Prioritising risk

Prioritising risks is then as simple as ranking them according to their score. A risk with a low probability and a low impact can be treated as a low priority risk. Conversely, a risk with a high probability and a high impact needs to be given a high priority. As indicated by the colour coding…

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

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

63. Risk-storming

Risk identification is a crucial part of doing "just enough up front design" but it's something that many software teams shy away from because it's often seen as a boring chore. Risk-storming is a quick, fun, collaborative and visual technique for identifying risk that the whole team can take part in. There are 4 steps.

Step 1. Draw some architecture diagrams

The first step is to draw some architecture diagrams on whiteboards or large sheets of flip chart paper. C4 is a good starting point because it provides a way to have a collection of diagrams at different levels of abstraction, some of which will allow you to highlight different risks across your architecture. Large diagrams are better.

Step 2. Identify the risks individually

Risks can be subjective, so ask everybody on the team (architects, developers, project managers, operational staff, etc) to stand in front of the architecture diagrams and *individually* write down the risks that they can identify, one per sticky note. Additionally, ask people to quantify each risk based upon probability and impact. Ideally, use different colours of sticky note to represent the different risk priorities. You can timebox this part of the exercise to 5-10 minutes to ensure that it doesn't drag on and this step should be done in silence, with everybody keeping their sticky notes hidden. Here are some examples of the risks to look for:

- Data formats from third-party systems change unexpectedly.
- External systems become unavailable.
- Components run too slowly.
- Components don't scale.
- Key components crash.
- Single points of failure.
- Data becomes corrupted.
- Infrastructure fails.

- Disks fill up.
- New technology doesn't work as expected.
- New technology is too complex to work with.
- etc

As with software development estimates, people's perceptions of risk can be subjective based upon their experience. If you're planning on using a new technology, hopefully somebody on the team will identify that there is a risk associated with doing this. Also, somebody might quantify the risk of using new technology relatively highly whereas others might not feel the same if they've used that same technology before. Identifying risks individually allows everybody to contribute to the risk identification process and you'll gain a better view of the risks perceived by the team rather than only from those designing the software or leading the team.

Step 3. Converge the risks on the diagrams

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

Converge the risks on the diagrams

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

Step 4. Prioritise the risks

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

- **Individual sticky notes**: Ask the person who identified the risk what their reason was and collectively agree on the probability and impact. After discussion, if either the probability or impact turns out to be "none", take the sticky note off of the architecture diagram but don't throw it away just yet.
- **Clusters of sticky notes**: If the probability and impact are the same on each sticky note, you're done. If they aren't, you'll need to collectively agree on how to quantify the risk in the same way that you agree upon estimates during a Planning Poker or Wideband Delphi session. Look at the outliers and understand the rationale behind people quantifying the risk accordingly.

Mitigation strategies

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

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

1. **Education**: Training the team, restructuring it or hiring new team members in areas where you lack experience (e.g. with new technology).
2. **Prototypes**: Creating prototypes where they are needed to mitigate technical risks by proving that something does or doesn't work. Since risk-storming is a visual technique, it lets you easily see the stripes through your software system that you should perhaps look at in more detail with prototypes.
3. **Rework**: Changing your software architecture to remove or reduce the probability/impact of identified risks (e.g. removing single points of failure, adding a cache to protect from third-party system outages, etc). If you do decide to change your architecture, you can re-run the risk-storming exercise to check whether the change has had the desired effect.

When to use risk-storming

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

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

Collective ownership

As a final point related to risks, who owns the risks on most software projects anyway? From my experience, the "risk register" (if there is one) is usually owned by the lone non-technical project manager. Does this make sense? Do they understand the technical risks? Do they really *care* about the technical risks?

A better approach is to assign ownership of technical risks to the software architecture role. By all means keep a central risk register, but just ensure that somebody on the team is actively looking after the technical risks, particularly those that will cause your project to be cancelled or you to be fired. And, of course, sharing the software architecture role amongst the team paves the way for collective ownership of the risks too.

64. Just enough up front design

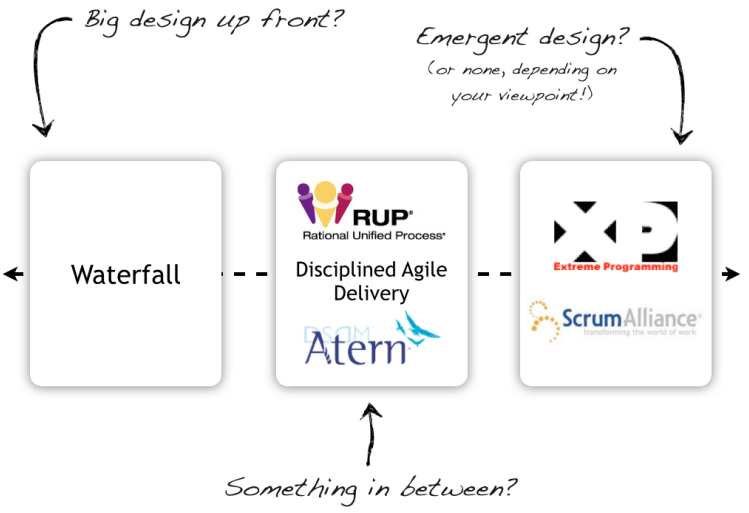
One of the major points of disagreement about software relates to how much up front design to do. People are very polarised as to when they should do design and how much they should do. From my experience of working with software teams, the views basically break down like this.

- "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"?

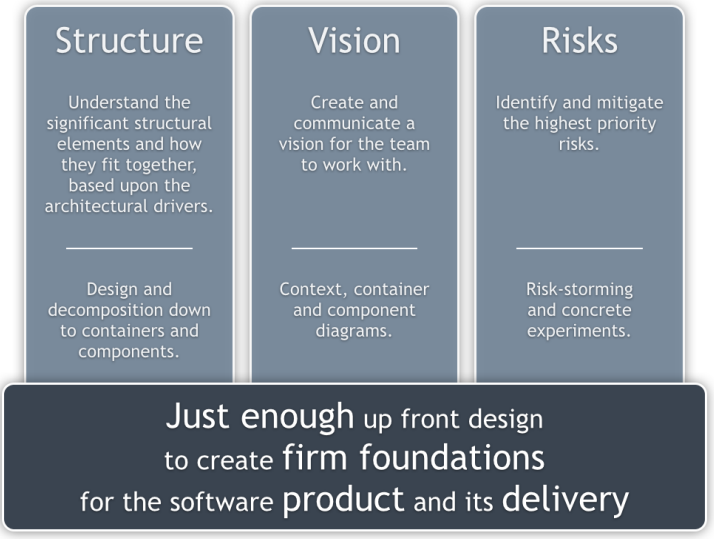
It's easy to identify with many of the answers above but "just enough" sits in that grey area somewhere between the two extremes. The key is that architecture represents the significant decisions, where significance is measured by cost of change. In other words, it's the stuff that's really expensive 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?

65. 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.

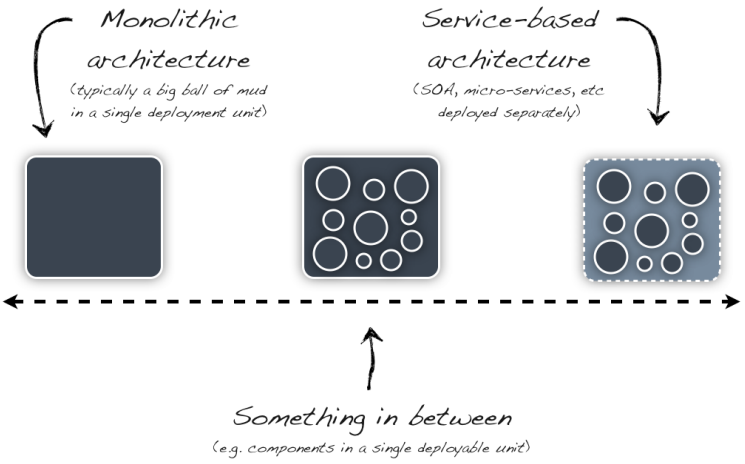
In a paper titled "What Lessons Can the Agile Community Learn from A Maverick Fighter Pilot?", Steve Adolph, from the University of British Columbia, takes Boyd's concept and applies it to software development. The conclusion drawn is that agility is relative and time-based. If your software team can't 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

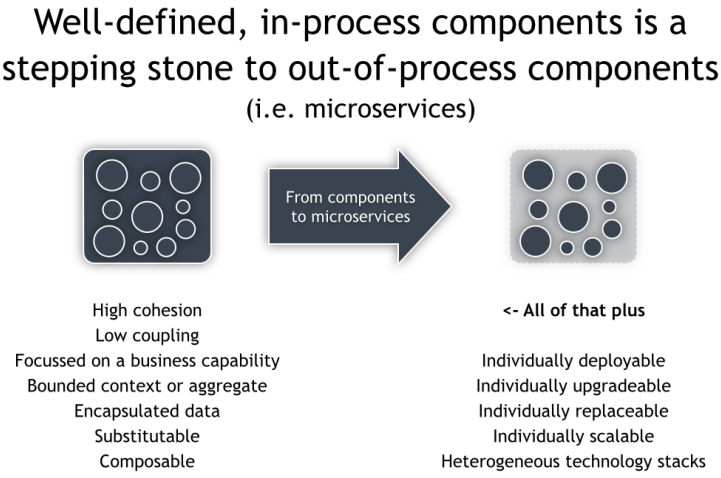
The driver for having this discussion is that a good software architecture enables agility. Although Service-Oriented Architecture (SOA) is seen as a dirty term within some organisations due to over-complex, bloated and bodged implementations, there's a growing trend of software systems being made up 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.



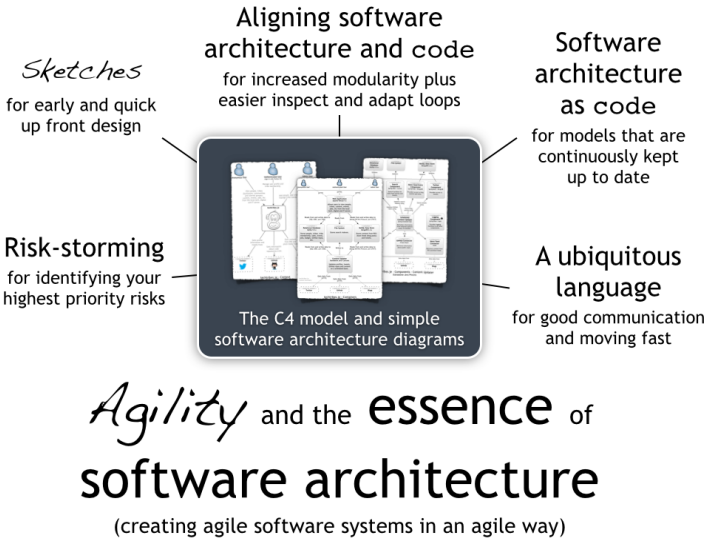
Well-defined, in-process components is a stepping stone to out-of-process components (i.e. microservices)

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.



- 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 continously up to date, especially when architeturally-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.

66. 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 doing on a daily basis. This lack of understanding is made worse because most software developers don't get to practice architecting software on a regular basis. How many software systems have you architected during your own career?

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

Software architecture needs to be accessible

As experienced practitioners, *we* have a duty to educate others but we do need to take it one step at a time. We need to remember that many people are being introduced to software architecture with potentially no knowledge of the related research that has been conducted in the past. Think about the terminology that you see and hear in relation to software architecture. How would you explain to a typical software developer what a "logical

view" is? When we refer to the "physical view", is this about the code or the physical infrastructure? Everybody on the development team needs to understand the essence of software architecture and the consequences of not thinking about it before we start talking about things like architecture description languages and evaluation methods. Information about software architecture needs to be accessible and grounded in reality.

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 continouous 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!

67. 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.

68. Financial Risk System

Background

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

Trade Data System

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

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

Reference Data System

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

Functional Requirements

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

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

Non-functional Requirements

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

Performance

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

Scalability

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

Availability

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

Failover

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

Security

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

Audit

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

Fault Tolerance and Resilience

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

Internationalization and Localization

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

Monitoring and Management

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

Data Retention and Archiving

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

Interoperability

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

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.