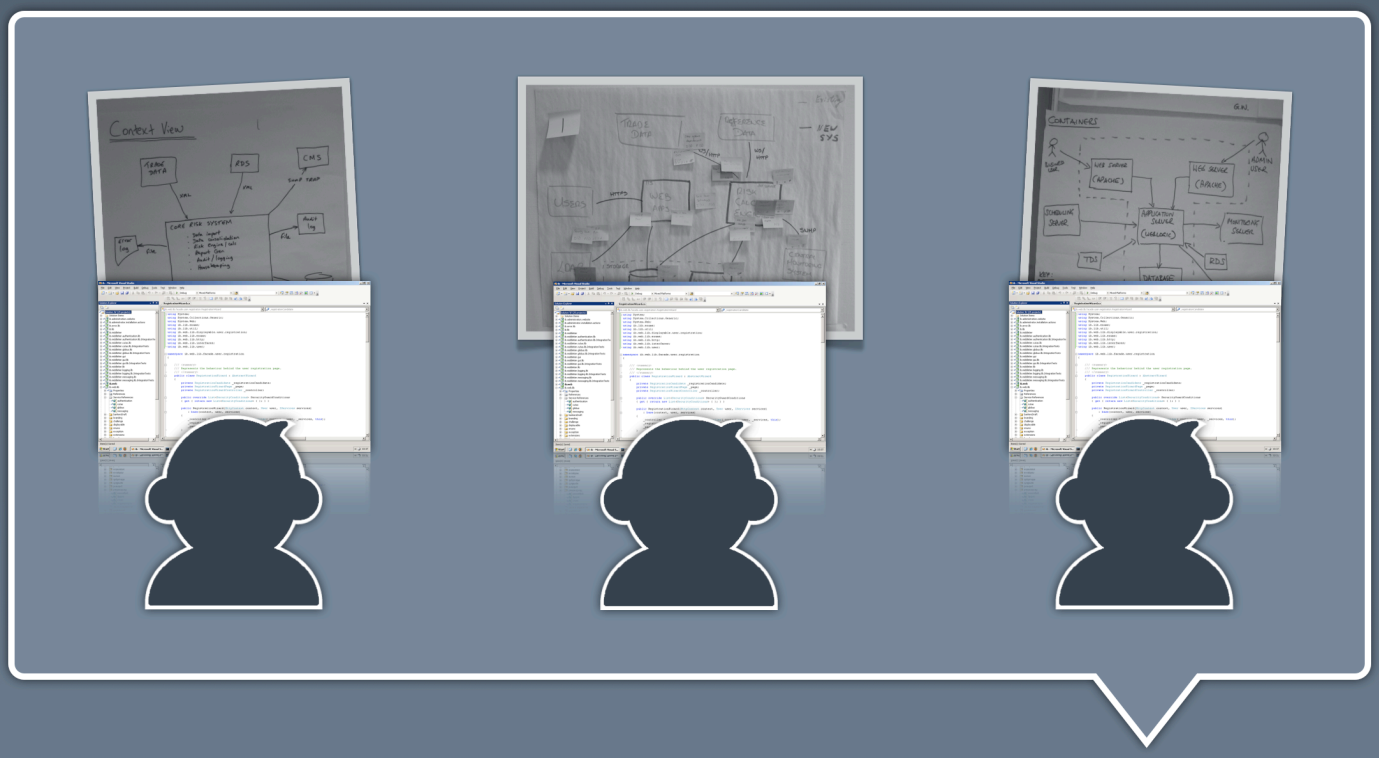
coding {the} architecture

# Software Architecture for Developers

## Software architecture, technical leadership and the balance with agility



Coding, coaching, collaboration, sketching and just enough up front design

Simon Brown

# Software Architecture for Developers

## Software architecture, technical leadership and the balance with agility

Simon Brown

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

This version was published on 2013-03-17

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

# Tweet This Book!

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

The suggested hashtag for this book is #sa4d.

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

https://twitter.com/search/#sa4d

*For Kirstie, Matthew and Oliver*

# Contents

# Preface

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 NoUML 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.

# About the author

Simon lives in Jersey (Channel Islands) and works as an independent consultant, specialising in software architecture and its role in modern software development teams. Simon regularly speaks at international software development conferences and provides consulting/training to software teams at organisations across Europe, ranging from small startups through to global blue chip companies. He is the founder of Coding the Architecture, which is a website for hands-on software architects. He still likes to write code too, primarily in .NET and Java.


More information about Simon, or to get in touch, please see simonbrown.je.

# Training

Designing software given a vague set of requirements and a blank sheet of paper is a good skill to have, although not many people get to do this on a daily basis. However, with agile methods encouraging collective ownership of the code, it's really important that everybody on the team understands the big picture. And in order to do this, you need to understand why you've arrived at the design that you have. In a nutshell, everybody on the team needs to be a software architect.

"Software Architecture for Developers" is also available as a two-day training course about pragmatic software architecture, designed by software architects that code. It will show you what "just enough" up front design is, how it can be applied to your software projects and how to communicate the big picture through a collection of simple effective sketches. It's aimed at software developers and architects regardless of whether you're building software in Java, .NET or something else.


**Join us for a mixture of presentation, discussion and deliberate practice**

See http://www.softwarearchitecturefordevelopers.com for more details or if you'd like to join us for a mixture of presentation, discussion and deliberate practice. This book accompanies the training course.

# I The prelude

# 1 The frustrated architect

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

## 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", "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 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.

## From frustration and beyond

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?

Read on to find out…

# II What is software architecture?

# 2 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. These are 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 and is about the decomposition of a product into a collection of components 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 design decisions. Crucially, it's also about communicating that vision 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. Put simply, the process of architecting is about introducing technical leadership.

# 3 Types of architecture

There are many different types of architecture within the IT industry alone. Here, in no particular order, is a list of those that people most commonly come up with 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" always appears near the bottom 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 above 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.

# 4 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).

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. In essence, application architecture is predominantly about software and the organisation of the code.

## System architecture

System architecture takes this up one level if you think of a "software system" as being composed of multiple applications. Your average Internet facing website (if there is such a thing) might be made up of multiple architectural tiers; a web-tier, a middle-tier and a database. Since each of these could be implemented using a completely different set of technologies, they are effectively different applications, each with their own application architecture. For the overall software system to function, thought needs to be put into bringing all of those separate applications together.

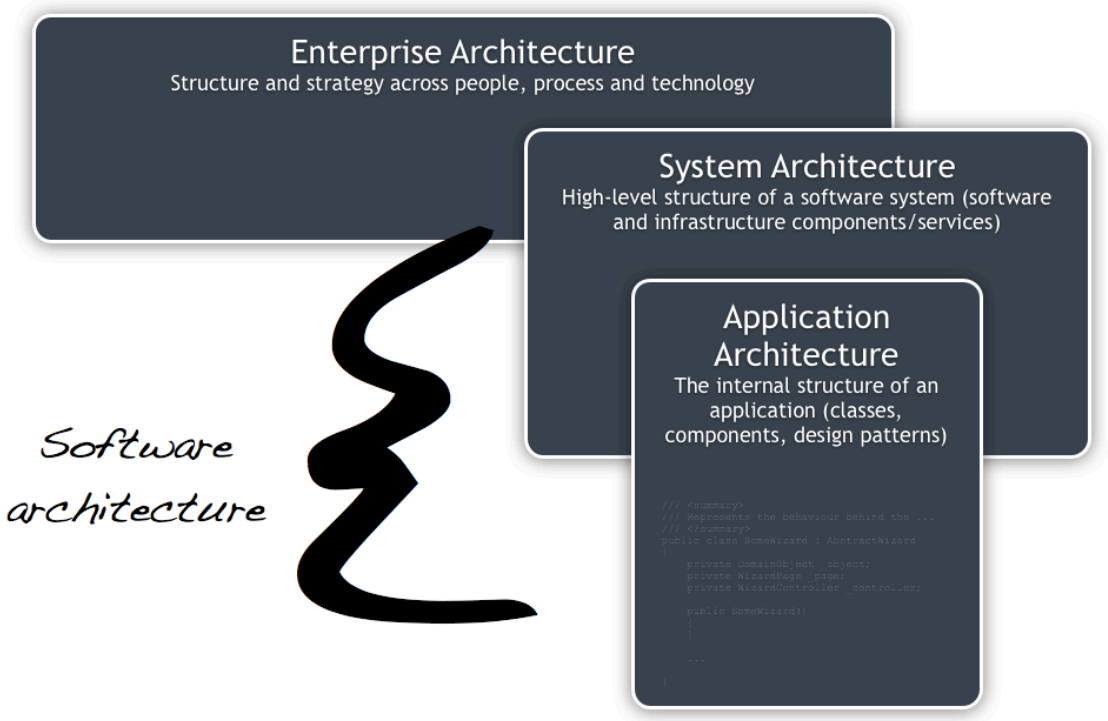
The building blocks are a mix of software and hardware, including things like programming languages and software frameworks through to servers and infrastructure. Compared to application architecture, system architecture is described in terms of higher levels of abstraction; from components and services through to sub-systems. It's also worth bearing in mind that most software systems don't live in isolation and they usually need to communicate with other systems. For this reason, system architecture is also about the interactions with other elements in the environment.

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.




**Software 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. For this reason, software architecture is, ironically, about software *and* infrastructure.

# 5 Architecture vs design

If architecture is about structure and vision, then what's design about? If you're creating a vision 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 an 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 Booch then goes on to say that…

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

This makes sense because creating a vision to solve a problem is essentially a design exercise. However, for some reason, there's a distinction being made about not all design being "architecture".

Finally, he clarifies this with the following statement.

> Architecture represents the significant design decisions that shape a system, where significant 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 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 overall structure of the software system, in terms of its components and their interactions

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 an afternoon.

## Understanding significance

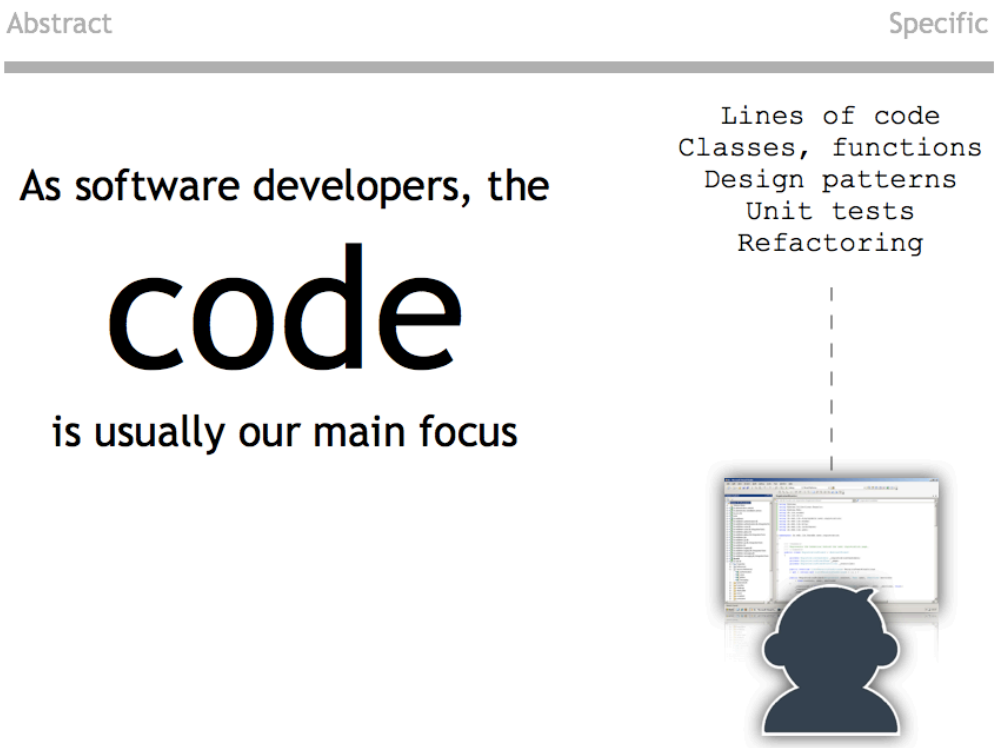
It's often worth taking a step back and considering what's significant with your own software system. For example, many teams use a relational database, the choice of which might be deemed as significant. In order to reduce the amount of rework required in the event of a change in database technology, 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 loose coupling, high cohesion and a good separation of concerns. Additionally, with the ORM in place, the choice of database can (probably) be refactored out 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 refactor 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, but again you've made another significant decision by introducing additional layering, complexity and cost. Significant decisions don't necessarily disappear, but they can be moved elsewhere. Part of the process of architecting a software system is about understanding what is significant and why.

# 6 What is the big picture?

Whenever we talk about software architecture, undoubtedly we'll end up talking about the "big picture", but what exactly does this mean?



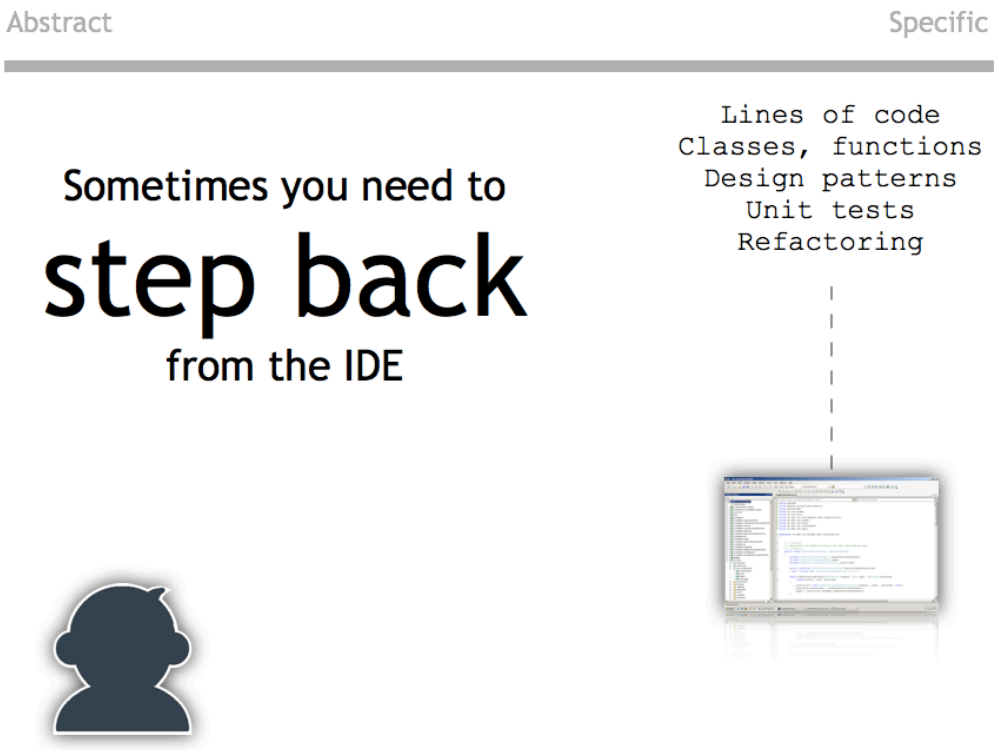

**As a developer, the code is usually our main focus**

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 practices that help us build better software. If you have a team of people that 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.

This doesn't mean that the 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.




**Sometimes you need to step back from your IDE**

All of these things can be thought of as the "big picture" and generally require you to look at your software system from different angles.

# 7 Strategy rather than code

There are many different types of architecture and many names to describe the same thing, but is enterprise architecture is the next logical step for people on the software architecture career path?

## Application architecture

Application architecture is what we're probably most familiar with as software developers. 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).

## 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 .NET Silverlight client accessing web services on a Java EE middle-tier, which itself consumes data from an Oracle database. Each of these will have their own application architecture. However, you also have the overall structure of the end-to-end software system at a high-level. This is system architecture and is about the organisation of software and infrastructure components. Additionally, software systems don't live in isolation, so system architecture also includes the concerns around interoperability and integration with other systems within the environment.

## Software architecture

For me, software architecture is the combination of application and system architecture, covering everything from the internal organisation of the code through to the high-level structure of a software system in terms of its software and infrastructure components. See What is software architecture? for a more detailed overview.

## Enterprise architecture

Enterprise architecture generally refers to the sort of work that happens centrally and across an organisation. It looks at how to organise and utilise people, process and technology to make an organisation work effectively and efficiently. This is in very stark contrast to application and system 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.

## Is enterprise architecture the next step for my career?

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 application and system architecture, taking a very different view of technology and its application across an organisation. Enterprise architecture is about a higher level of abstraction; breadth rather than depth and strategy rather than code.

# 8 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 IDE 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 project and system, why don't we simply acknowledge and place some focus on it?

## The benefits of software architecture

What benefits can "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 rest of the project.
- 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 is necessary. These include the size of the project, the complexity of the project, 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.

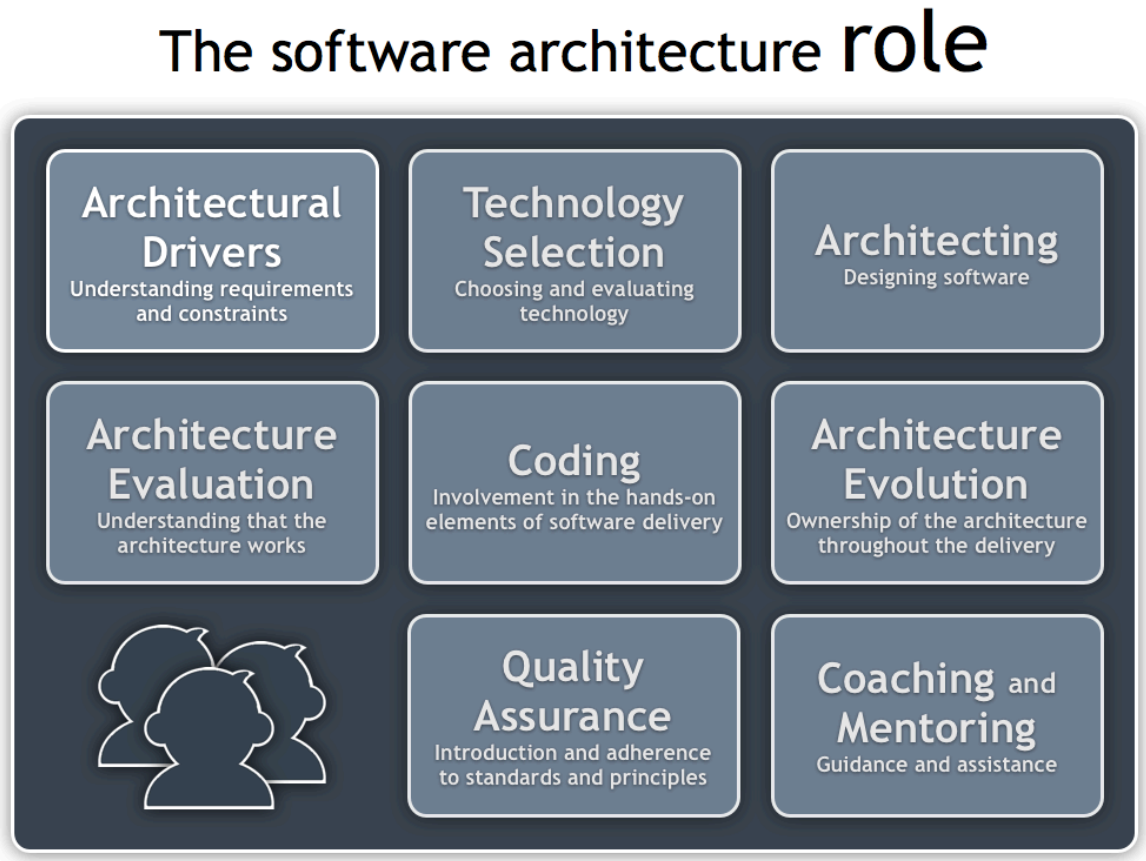
# 9 Questions

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. Can you make a list of the architectural decisions on your current software project? Is it obvious why they were deemed as significant?
5. If you step back from your IDE, what sort of things are included in *your* big picture?
6. What does the technical career path look like in your organisation? Is enterprise architecture the right path for you?
7. Is software architecture important? Why and what are the benefits? Is there enough software architecture on your software project? Is there too much?

# III The software architecture role

# 10 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.




**The software architecture role**

## 1. Architectural drivers

The first part of the role is about 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 need to be specific, measurable, achievable and testable if we are going to satisfy them. Plus, we need to make sure that all of the important non-functional requirements

are at least considered. This includes the common characteristics such performance, scalability, availability and security through to the others such as audit, extensibility, internationalisation and localisation.

Somebody needs to help the stakeholders define the non-functional requirements, refine them and challenge them when appropriate. Since most of the non-functional requirements are technical in nature, they often have a huge influence on the software architecture. For this reason alone, it makes sense that this is explicitly included as a part of the software architecture role.

## 2. Technology selection

Technology selection 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. And then there's the question of whether the technology actually works. Many projects choose to use products because they believe the hype and many have been burnt by this too, regardless of whether those products were commercial or open source. 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. Somebody needs to take ownership of the technology selection process and again it should be an explicit part of the software architecture role. If you're defining an architecture, you also need to be confident that the technology choices being made are the right ones. The software architecture role should be responsible for the successful selection of technology.

## 3. Architecting

It should come as no surprise that the process of architecting (i.e. designing software) is a part of the software architecture role. The process of architecting lets you think about how you're going to take the requirements along with any constraints imposed upon you and figure out how you're going to solve the problem. It's about creating the overall structure of the software system and creating a vision for the delivery. You need time to explicitly think about how your architecture is going to solve the problems set out by the stakeholders and your software system isn't going to do this itself! Somebody needs to take ownership of the architecture process and this falls squarely within the remit of the software architecture role.

# 4. Architecture evaluation

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. 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 lifecycle 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. Somebody needs to take ownership of the architecture evaluation and take responsibility for de-risking the architecture. Since the process of architecting is a part of the software architecture role, why shouldn't evaluation be a part of it too? Like good chefs, architects should taste what they are producing.

# 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 project, actively helping to shape and deliver 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 that commodity 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 that codes is a more effective and happier architect. You shouldn't necessarily rule out coding just because "you're an architect".

## 6. 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 project if necessary and ensuring that the architecture is being implemented consistently across the team. If an architect has defined an architecture, why shouldn't they own and evolve that architecture throughout the rest of the project too?

Proactively looking after the architecture is also about owning the high priority technical risks so that your project doesn't get cancelled and you don't get fired.

## 7. 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.

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.

## 8. Coaching and mentoring

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. Certainly 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?

## 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. Anybody undertaking the software architecture role needs to collaborate with these people to ensure that the architecture 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.

# 11 Should software architects code?

To be completed.

# 12 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 that 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 adopting a different approach. Instead of a single dedicated technical leader that stays away from the detail, many modern 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. In many cases, this 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 should 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") that 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 to control their own destiny.

# 13 Software architects are generalising specialists

To be completed.

# 14 Crossing the mythical line or bridging the gaping chasm?

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. Technology selection: choosing and evaluating technology for a new system versus adding technology to an existing system.
3. Architecting: designing a software system from scratch versus extending an existing one.
4. Architecture evaluation: proving that your architecture will work versus hoping for the best.
5. Coding: being involved in the hands-on elements of the delivery versus watching from the sidelines.

6. Architecture evolution: being continually engaged and evolving your architecture versus choosing to hand it off to an "implementation team".
7. Quality assurance: assuring quality and selecting standards versus being reviewed against them or doing nothing.
8. Coaching and mentoring: coaching the team in architecture and design versus helping people with their coding problems.

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.

# 15 Software development is not a relay sport

Software teams that are smaller and/or agile tend to be staffed with people that 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. These teams, therefore, tend to need and 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 onto the 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 architecture and the "architecture" initially created becomes detached from reality.

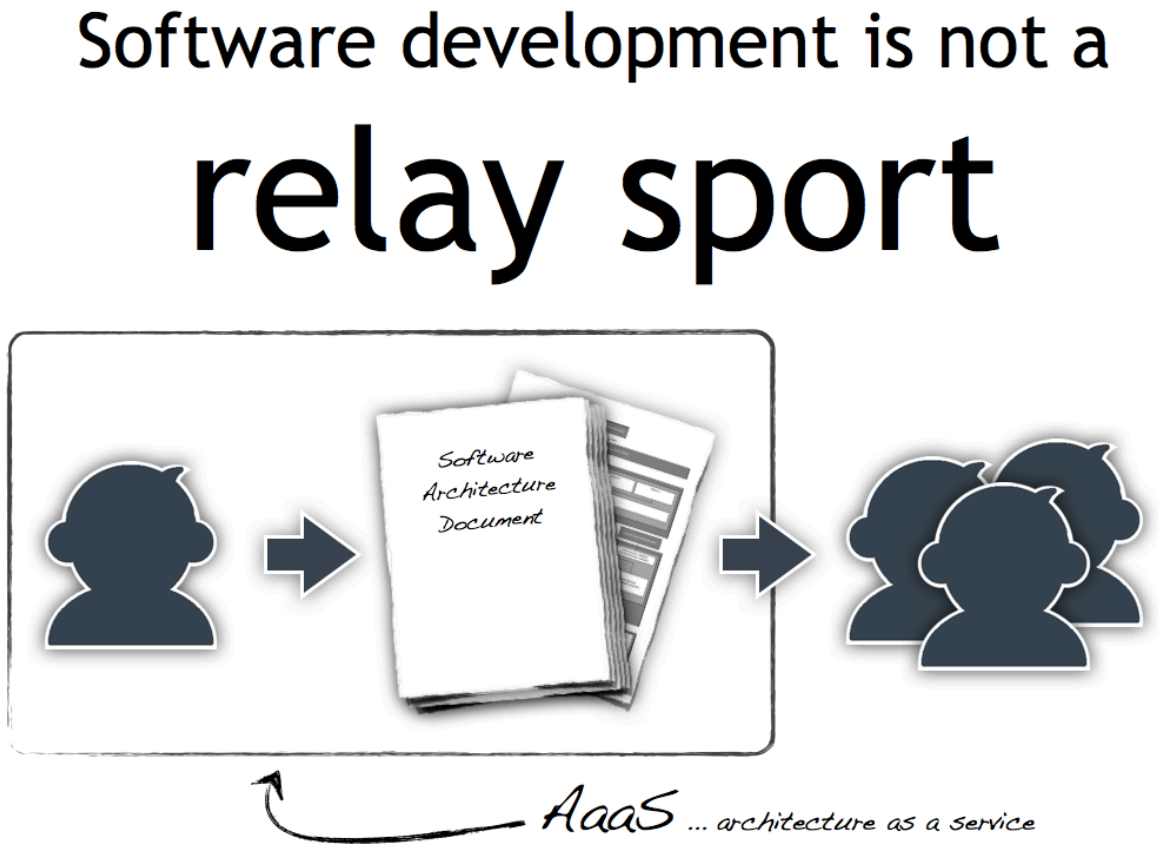
I've met a number of such architects in the past and one particular interview I held epitomises this approach to software development. After the usual "tell me about your role and recent projects" conversation, it became clear to me that this particular architect (who worked for one of the large "blue 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". Essentially, his view was that his software architecture was correct and it was the development team's problem if they couldn't get it to work for any reason. That's outrageous!

## Somebody needs to own the big picture

The software architecture role is that of a generalising specialist 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. Agile has taught us that we don't necessary have all of the information up front.

A successful software project requires the initial vision to be understood, communicated and potentially evolved throughout the entirety of the software development lifecycle. 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 design artefacts is essentially a baton that gets passed between the architect and the development team. This is inefficient, ineffective and the exchange of a document means that a lot 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!




**Software development is not a relay sport**

This problem goes away with truly self-organising teams, but most teams haven't yet reached this level of maturity. Therefore, somebody needs to take ownership of the big picture throughout the project and they also need to take responsibility for ensuring that it's delivered successfully. If somebody has designed the software, why shouldn't they own and take responsibility for it throughout the rest of the project? Software development is not a relay sport and successful delivery is not an "implementation detail".

# 16 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 or 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 you feel as if 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.

# 17 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 that 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.

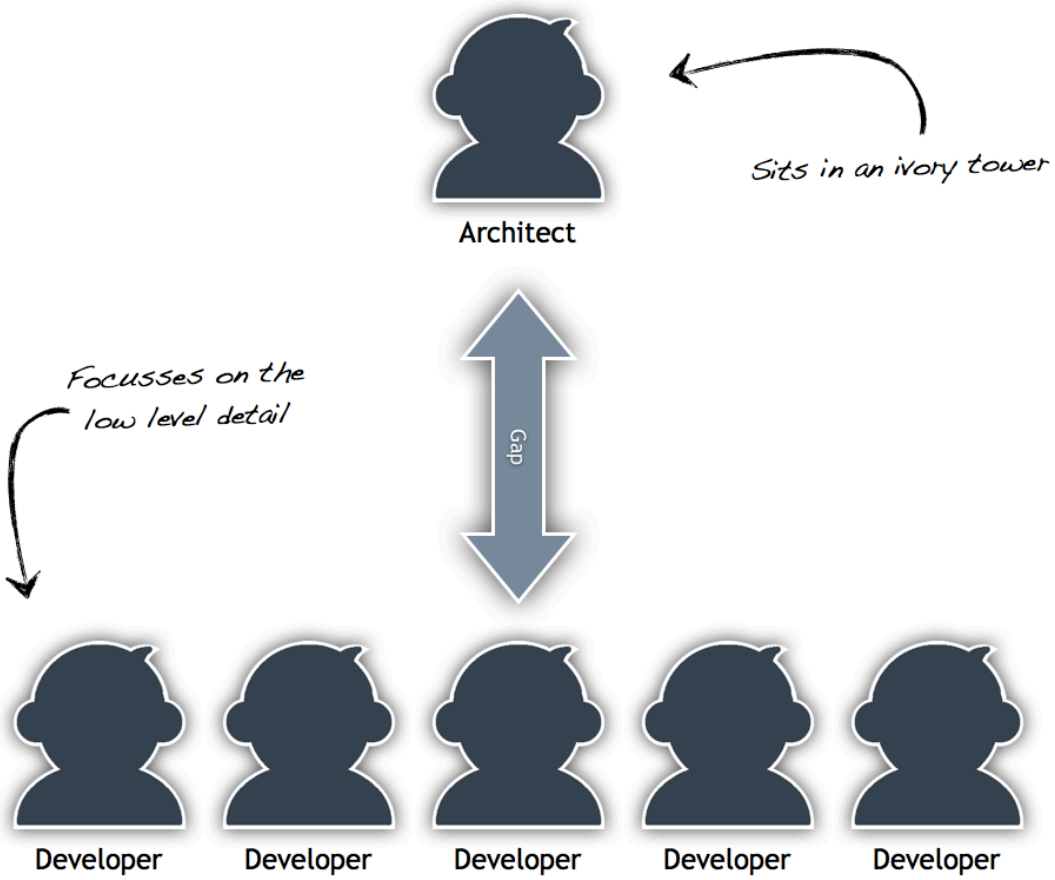
## Developers focus on the low-level detail

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 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. Typically this instantly isolates the architect by creating an exaggerated gap between them and the development team they are supposed to be working with.




**The gap between the software architect and the developers**

# 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.

## If you're a software architect:

- Be inclusive and collaborate: Get the development team involved in the software design process to help them understand the big picture and buy-in to the decisions you are making. Software development is a team activity after all.

- 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 that is responsible for the architecture and it's this person that 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.

# 18 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, testing, tools, technologies and the 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.

## 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 that are being 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 that 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?

# 19 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 I rarely see it in practice yet we talk about it a lot in the industry. This could be a side-effect of working in a consulting environment in that *my* team always changes from project to project and 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 often uses the ScrumMaster role as an example. Teams in the chaos phase tend to 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. 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 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 and that 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-functionals 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. Teams that aren't at the self-organising stage will struggle, though, if they try to run too fast. Despite people's aspirations to be agile, collective code ownership and a distribution of the architecture role is 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.

# 20 Soft skills

To be completed.

# 21 Questions

1. What's the difference between the software architecture and software developer roles?
2. 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?
3. If you're the software architect on your project, how much coding are you doing? Is this too much or too little?
4. 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?
5. Why is having a breadth *and* depth of technical knowledge as important?
6. 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?
7. 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?
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. What pitfalls have you fallen into as somebody new to the software architecture role?
12. How does the software architecture role fit into agile projects and self-organising teams?

# IV Designing software

# 22 Architectural drivers

To be completed.

# 23 Quality Attributes (non-functional requirements)

To be completed…

When you're gathering requirements, people will happily give you a wish-list of what they want their 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"?

Non-functional requirements are often thought of as the "-ilities" and are primarily about quality of service. Alternative, arguably better although less common, names for non-functional requirements are "system characteristics" or "quality attributes".

## Performance

Performance is about how something is and is usually about 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 most systems 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

### Availability

### Security

### Disaster Recovery

### Accessibility

### Monitoring

### Management

### Audit

### Flexibility

### Extensibility

### Maintainability

### Interoperability

### Legal

### Regulatory

### Compliance

### Internationalisation (i18n)

### Localisation (L10n)

## Which are important to you?

There are many non-functional requirements but they don't all have equal weighting.

# 24 Working with non-functional requirements

To be completed…

## Define

## Refine

## Challenge

# 25 Constraints

To be completed.

# 26 Principles

To be completed.

# 27 You don't need a UML tool

When tasked with the job of coming up with an architecture and design for 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.

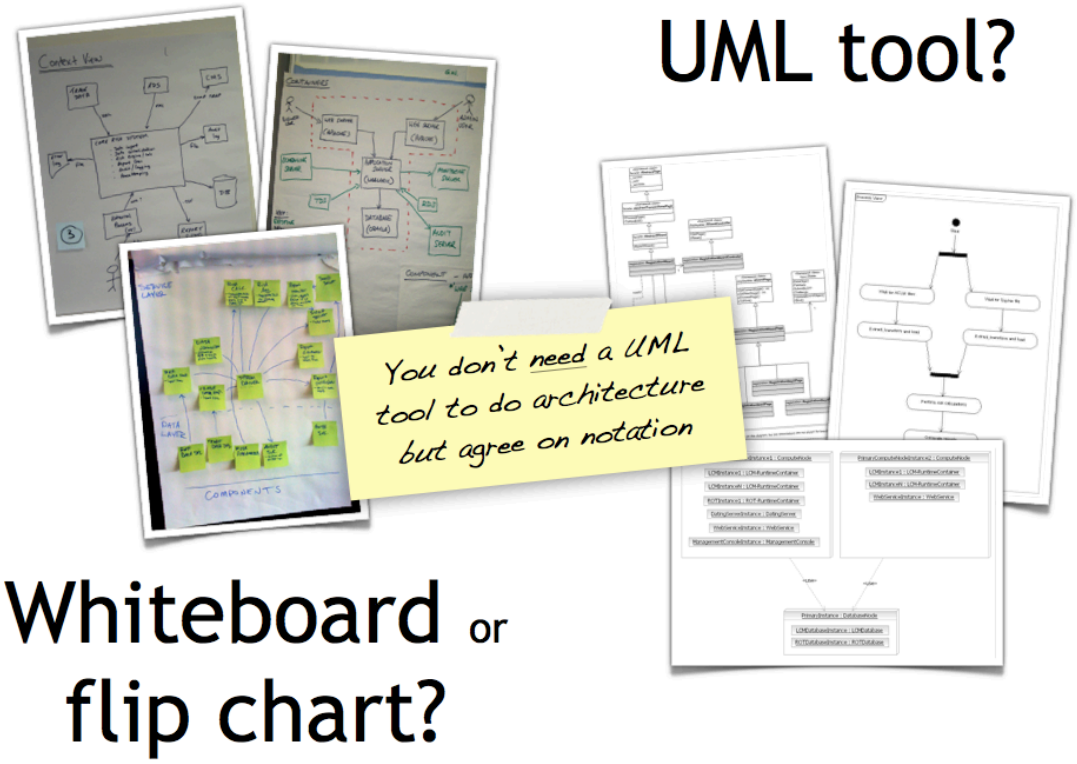
1. Diagrams only: There are many standalone UML tools and plug-ins for major IDEs that let you sketch simple UML diagrams. These are really useful if you want to be in control of your diagrams and what they portray but it's easy for such diagrams to get out of date with reality over time. Microsoft Visio or OmniGraffle with the UML templates 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, flipchart or whiteboard together with a set of post-it notes or index cards is all you need, particularly when you have a group of people that 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?




**You don’t need a UML tool**

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 post-it notes still in the “To do” column.

From a software design perspective, this 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 the 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.

## There is no silver bullet

Forget expensive tools. More often than not; a blank sheet of paper, flipchart 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. You don’t *need* UML tools to do architecture and design, but they can be useful.

# 28 Start with the big picture

One of the hardest things about the software architecture role is being asked to come up with a software design when all you're given is a set of requirements and a blank piece of paper. It's certainly one of the most fun parts of the role, but it can be a daunting prospect trying to figure out where to start and what to do. Many software teams dive straight into the code and while this can initially be very productive, some soon start to venture down the slippery slope of constant refactoring while they try to settle upon a design that works. Often, a little forethought is all that's needed to get the software development process heading in the right direction. So where do you start?

## Wide angle view (the big picture)

One of the first things I do is step back to see the big picture. What exactly is the context here? What is this all about? What is it we're building and how does it fit in with the existing environment? A really useful starting point can be to draw a simple block diagram showing your system, the other systems that it interfaces with and key actors. For example, if you're building a website, you might show the website along with the various backend systems that you need to integrate with. Detail isn't important here. It's your wide angle view showing a big picture of the system landscape.

## Standard view (containers)

Once you understand how your system fits in to the overall system landscape, a useful next step is to draw another simple block diagram showing all of the containers that make up these systems. By "container", I mean anything in which code and data will reside. This includes things like web servers, application servers, Windows services, enterprise service buses, databases, standalone applications, web browser plugins, etc. A high-level diagram like this doesn't necessary show much detail, but it only takes a few minutes to put together and provides a framework for understanding the major building blocks of your software and their interfaces.

## Telephoto view (major components/services and their interactions)

Next I start to zoom in and decompose the containers into components, services, subsystems, workflows, etc. However you decompose your system is up to you, but what you're looking for at the telephoto level is a view of the major components and their interactions. If you were building an e-commerce website, you might show components representing a catalog service, a pricing service, a payment service and so on. These are the fundamental building blocks of your system and you should be able to understand how a use case or user story can be implemented across one or more of those major 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, then perhaps you've missed something. Equally, if you have homeless components floating around outside of containers, then perhaps you've forgotten a container.

## Macro view (classes and implementation details)

For some software teams, the telephoto picture is enough detail for them to go off and successfully build the software. Others, however, need to go down to a lower level by further decomposing the major components. Typically this is low-level design where you use (for example) object oriented design principles to define how a component will work at the code level. For example, one of your components might be a web page, which you could further decompose to show how it would be implemented using a web-MVC framework. Or perhaps you want to model the business domain in some detail because it's complex in some way. You could represent this lower level of detail using anything from simple block diagrams through to formal UML class diagrams, but either way they're showing the implementation level design aspects of the system. For me, this is an optional level of detail that depends on the size/complexity of the project and size/experience of the team.

## Analysis paralysis vs refactor distractor

A logical top-down approach to software design such as this might look like common sense. And to some degree it is, but many software teams dive straight into the macro level without giving the other levels any thought. Diving into the code, tempting as it may be, can lead to designs that are over-engineered, incoherent and inconsistent balls of mud. This is especially true when you're faced with a large team where everybody is tackling system design from a macro level perspective. If you've seen teams that are constantly refactoring at the really early stages of a software project, to the point that it's distracting them from the bigger picture, they probably didn't start out with the big picture and enough vision of where they wanted to go.

A combination of design and coding will help to prevent analysis paralysis, but diving straight into the code means that you miss out on the opportunity to create a vision of how your software system should be built and you'll most likely encounter a whole host of problems further down the line. Analysis paralysis and refactor distractor are both very bad and a balance between design and coding is needed.

## Tell a consistent story

The software design process is essential and straightforward if tackled in the right way. My advice is simple; start with the big picture and work your way into the lower levels of detail, remembering that each picture should tell a different part of the same story.

# 29 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. Here are some of the typical responses I hear when I ask people why their diagrams don't show any technology decisions:

- "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.

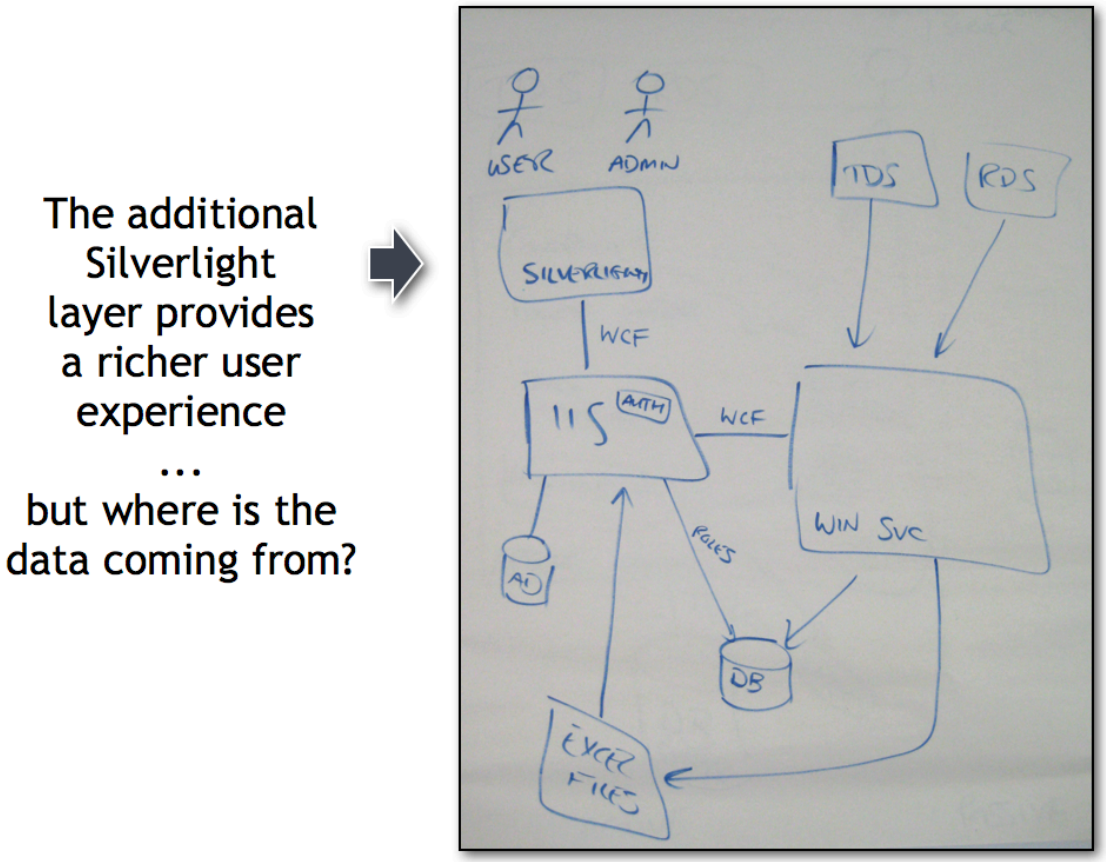
If you don't understand the trade-offs that you're making by choosing technology X over Y, you shouldn't be making those decisions. It's crucial that the people designing software systems understand technology. This is why software architects should be master builders.

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.

# 30 More layers = more complexity

One of the key functional requirements of the 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, AJAX and RIA 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 IIS as the mechanism for exposing the data, but this led to yet further questions.

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

## 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[1]. This led to a discussion about the use of SSL to secure the service, but SSL only secures the transport layer to stop data being looked at in transit. In this case, some thought needs to be given to authentication and authorisation of the service itself too.

## 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 communication between containers.

[1] I've seen organisations that have been very security conscious and have all of their public facing web servers firewalled away in DMZs, yet those same web servers access unsecured web services residing on servers within the regular corporate LAN. In some cases, assuming that I can connect to the corporate LAN, there's nothing to stop me firing up an IDE and consuming the web service for my own misuse.

# 31 Estimating and prioritisation

To be completed.

# 32 SharePoint projects need software architecture too

Although the majority of my commercial experience relates to the development of bespoke software systems, I've seen a number of SharePoint and other product/platform implementations where the basic tenets of software architecture have been forgotten or neglected. Here's a summary of why software architecture is important for SharePoint projects.

## 1. Many SharePoint implementations aren't just SharePoint

Many of the SharePoint solutions I've seen are not just simple implementations of the SharePoint product where end-users can create lists, share documents and collaborate. As with most software systems, they're a mix of new and legacy technologies, usually with complex integration points into other parts of the enterprise via web services and other integration techniques. Often, bespoke .NET code is also a part of the overall solution, either running inside or outside of SharePoint. If you don't take the "big picture" into account by understanding the environment and its constraints, there's a chance that you'll end up building the wrong thing or building something that doesn't work.

## 2. Non-functional requirements still apply to SharePoint solutions

Even if you're *not* writing any bespoke code as a part of your SharePoint solution, that doesn't mean you can ignore non-functional requirements. Performance, scalability, security, availability, disaster recovery, audit, monitoring, etc are all potentially applicable. I've seen SharePoint projects where the team has neglected to think about key non-functional requirements, even on public Internet-facing websites. As expected, the result was solutions that exhibited poor response times and/or severe security flaws (e.g. cross-site scripting). Often these issues were identified at a late stage in the (usually waterfall) project lifecycle.

## 3. SharePoint projects are complex and require technical vision too

Like any programming language, SharePoint is a complex platform and there are usually a number of different ways to solve a single problem. In order to get some consistency of approach and avoid chaos, SharePoint projects need strong technical leadership and the software architecture role is as applicable here as it is if you're writing a software system from scratch. If you've ever seen SharePoint projects where a seemingly chaotic team has eventually delivered something of a poor quality, you'll appreciate why this is important.

## 4. SharePoint solutions still need to be documented

With all of this complexity in place, I'm continually amazed to see SharePoint solutions that have no documentation. I'm not talking about hefty 200+ page documents here, but there should be at least *some* lightweight documentation that provides an overview of the solution. Some diagrams to show how the SharePoint solution works at a high-level are also useful. C4 works well for SharePoint too and some *lightweight* documentation can be a great starting point for future support, maintenance and enhancement work; particularly if the project team changes or if the project is delivered under an outsourcing agreement.

# Strong leadership and discipline aren't just for software development projects

If you're delivering software solutions then you need to make sure that you have at least one person undertaking the technical leadership role, looking after the things that I've highlighted above. If not, you're doing it wrong. As an aside, all of this applies to other platform products such as Microsoft Dynamics CRM projects, especially if you're "just tacking on an Internet-facing ASP.NET website to expose the data via the Internet".

I've mentioned this to SharePoint teams in the past and some have replied "but SharePoint isn't software development". Regardless of whether it is or isn't software development, successful SharePoint projects need strong technical leadership and discipline. SharePoint projects need software architecture.

# 33 How do you design software?

To be completed.

# 34 Questions

1. What are the major influencing factors on 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 against a well known set of architectural principles? What are they? Are they clearly understood by everybody on the team?
5. Do you need a UML tool to design software? If not, why not?
6. How to *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?

# V Visualising software

# 35 Agility requires good communication

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.

## We're adept at visualising our software development process

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 that way that your team works. As an industry, we've become pretty adept at visualising our software development process.

## But we've forgotten how to visualise the software that we're building

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.

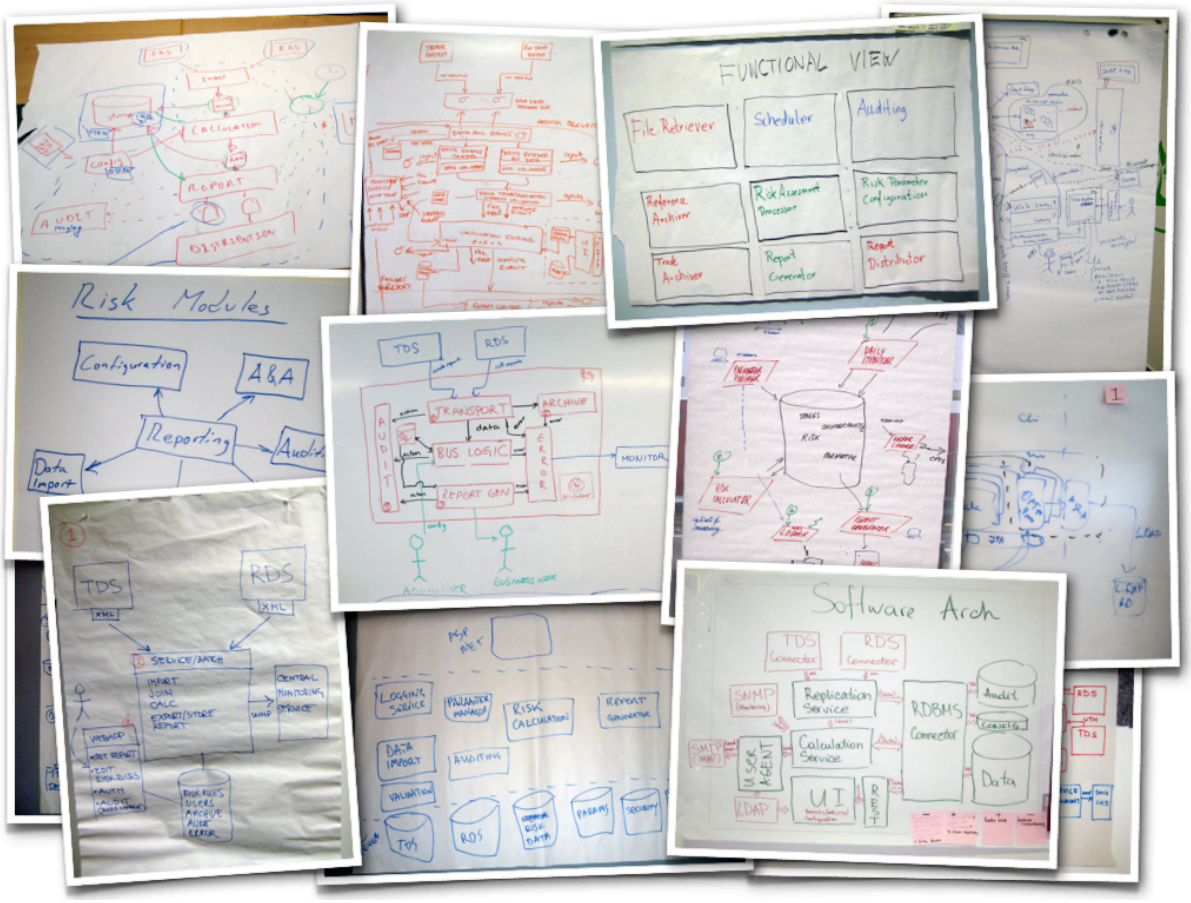
If you look back a few years, structured processes and formal notations provided a reference point for both the software design process and how to communicate the resulting designs. Examples include the Rational Unified Process (RUP), Structured Systems Analysis And Design Method (SSADM), Unified Modelling Language (UML) and so on. Although the software development industry has moved on in many ways, we seem to have forgotten some of the good things that these older processes gave us. In today's world of agile projects and lean startups, the ability to deliver fast requires good communication yet some software teams have lost the ability to effectively communicate what it is they are building. It's no surprise that these teams often seem chaotic.


**We can visualise our software development process with Kanban boards and story walls, but we've forgotten how to visualise the software that we're building**

Why is this important? If you want to ensure that everybody is contributing to the same end-goal, you need to be able to 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 effectively and efficiently.

## Abandoning UML

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 too. The example software architecture sketches (above) 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.

# 36 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 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.

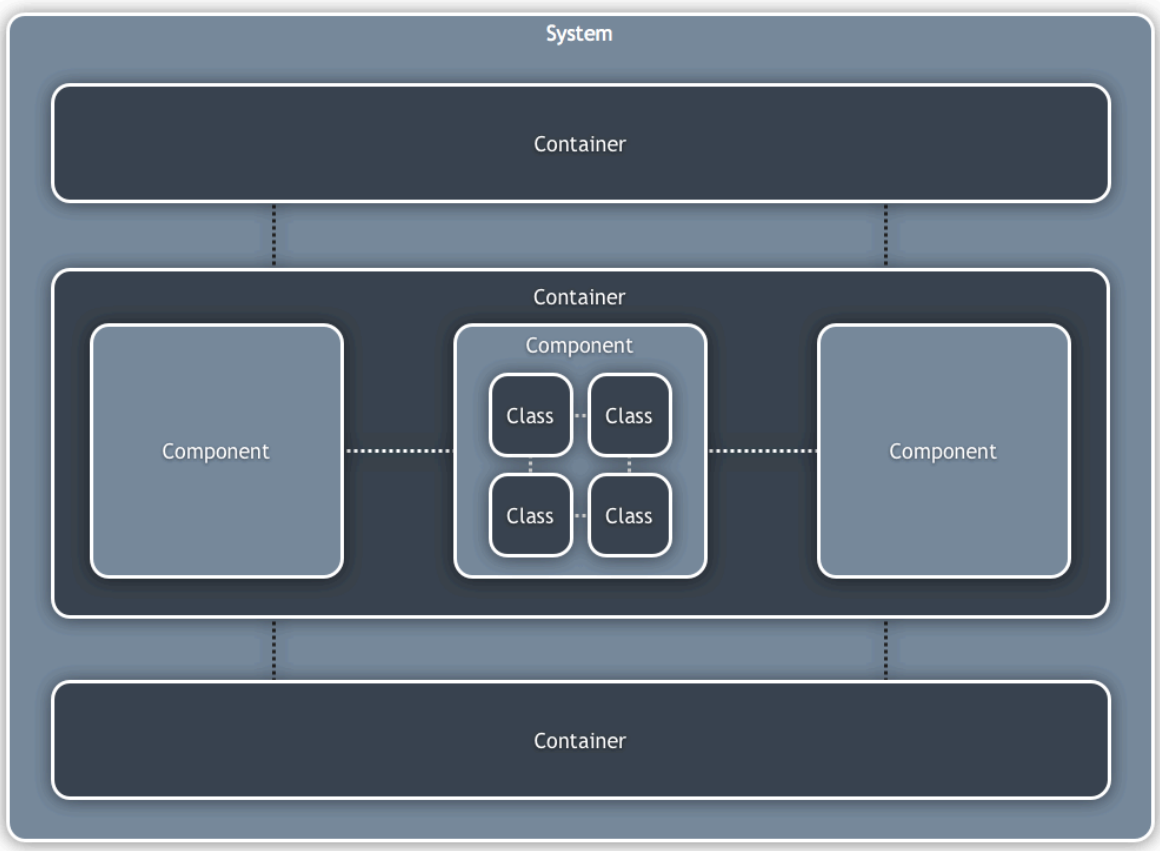
Unfortunately, drawing diagrams seems to have fallen out of favour with many software development teams but it's a skill that should be in every software developer's toolbox because it paves the way for collaborative software design and makes collective code ownership easier. Every software development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

# 37 Ineffective sketches

To be completed.

# 38 Architectural constructs

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/services 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 model with which to describe the system at a number of different levels of abstraction.

# 39 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*.

## Summarising the static view of your software

I tend to draw the following types of diagrams when summarising the static view of my software, which are based upon a set of simple architectural constructs.

1. Context: a high-level diagram that sets the scene; including key systems and actors.
2. Containers: a container represents something in which components are executed, which could be anything from a web or application server through to a rich client application or database. Drawing a diagram at this level lets you see the containers that make up your software system and the relationships between them.
3. Components: a component can be thought of as a logical grouping of one or more classes that sits inside a container. Drawing a diagram at this level lets you see the key logical components/services and their relationships.
4. Classes: if there's important information that I want to portray, I'll include some class diagrams. This is an optional level of detail and its inclusion depends on a number of factors.

## Organisational ideas, not a standard

Just to be clear, C4 is about providing some organisational ideas and guidelines rather than creating a standard. 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. Feel free to mix the concepts together; this can work really well for small systems where the containers and components diagrams can be merged. Just try to keep the diagrams as simple as possible. If you do this, 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.

# 40 Context

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.

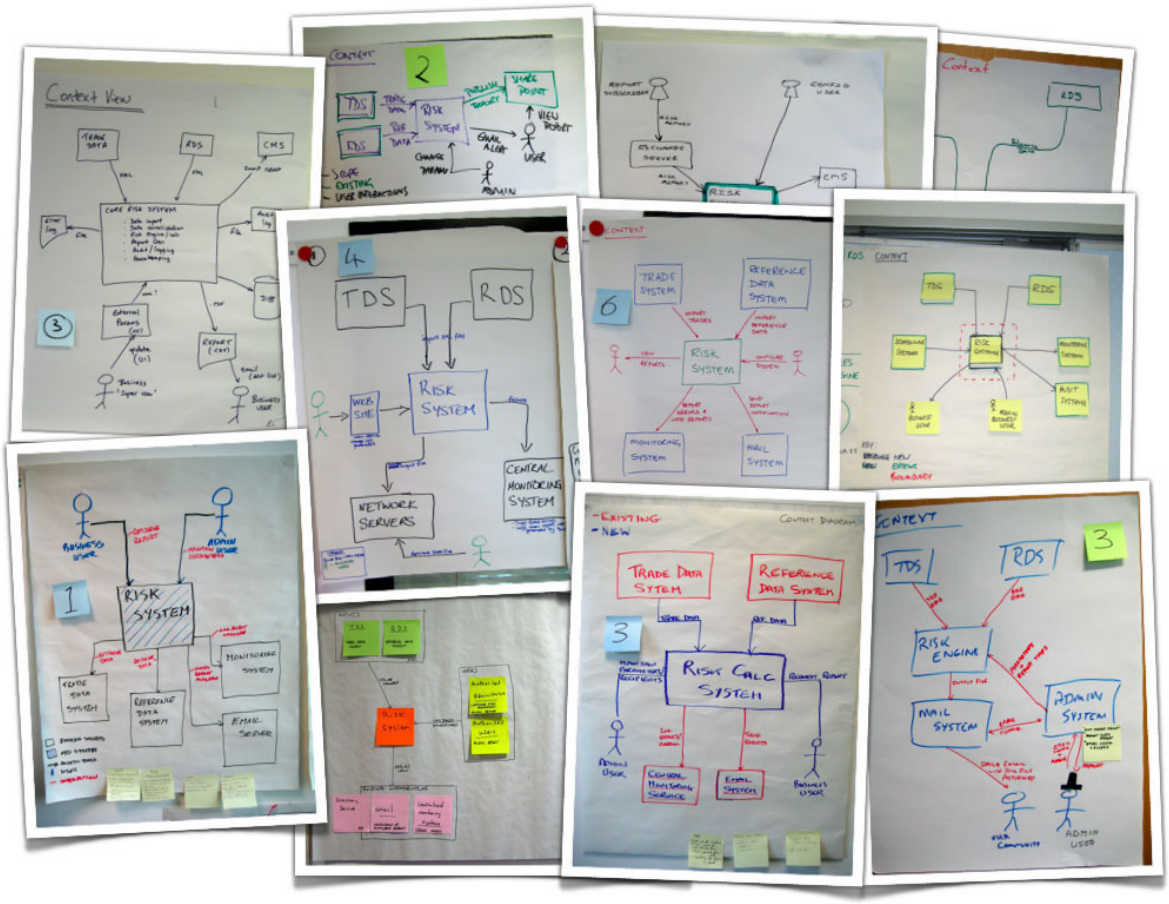
## Intent

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, along with its users and all of the other systems that it interfaces with. For example, if you were diagramming a solution to the 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 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.

# 41 Containers

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 containers diagram.

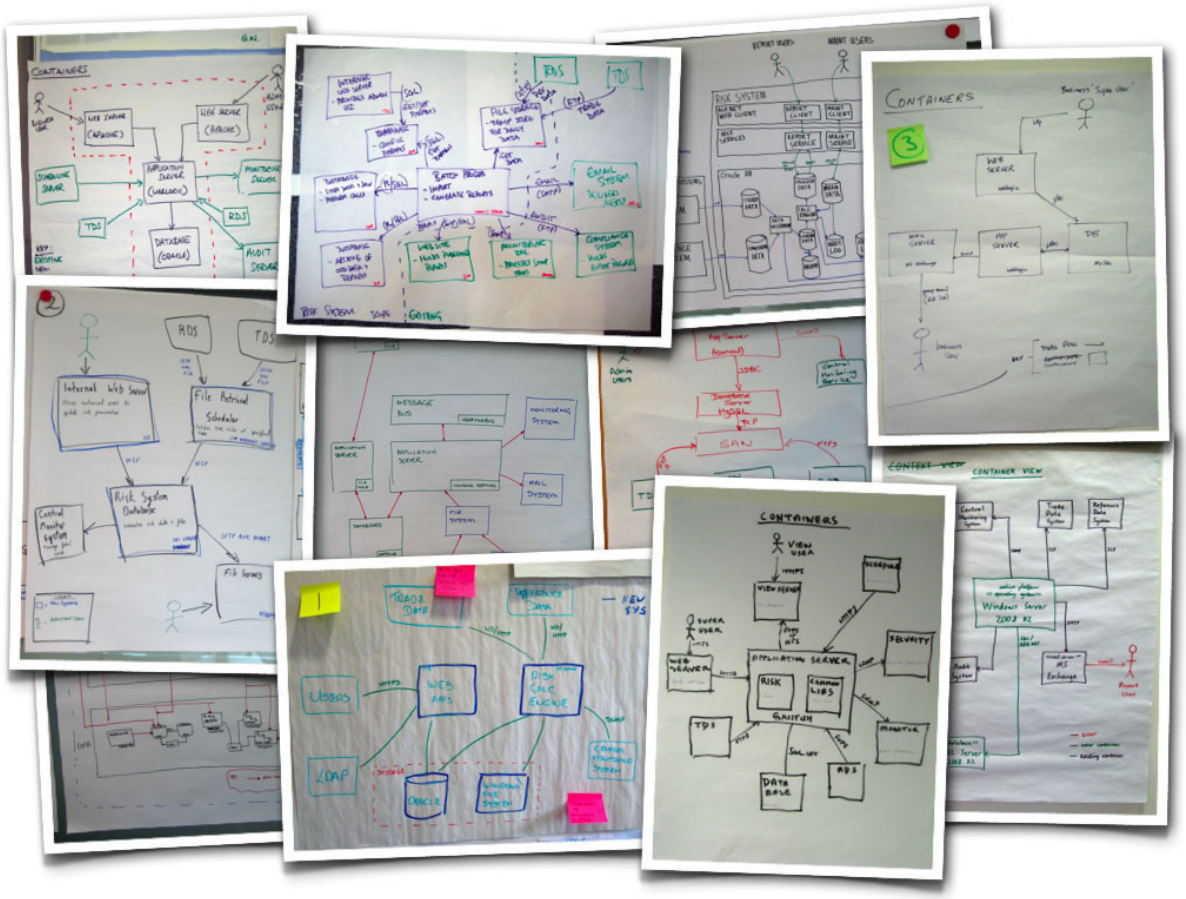
## Intent

A containers diagram helps you answer the following questions.

1. What are the high-level technology decisions?
2. How do containers communicate with one another?
3. As a developer, where do I need to write code?

## Structure

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




**Example container diagrams for the 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 or processes that make up your software system; such as:

- Web servers[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.

- 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.

### 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 containers 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 usually ends to be a cluttered component diagram showing all of the logical components that make up the 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.

# 42 Components

Following on from a container diagram showing the high-level technology decisions, I'll then start to zoom in and decompose each container further. However 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.

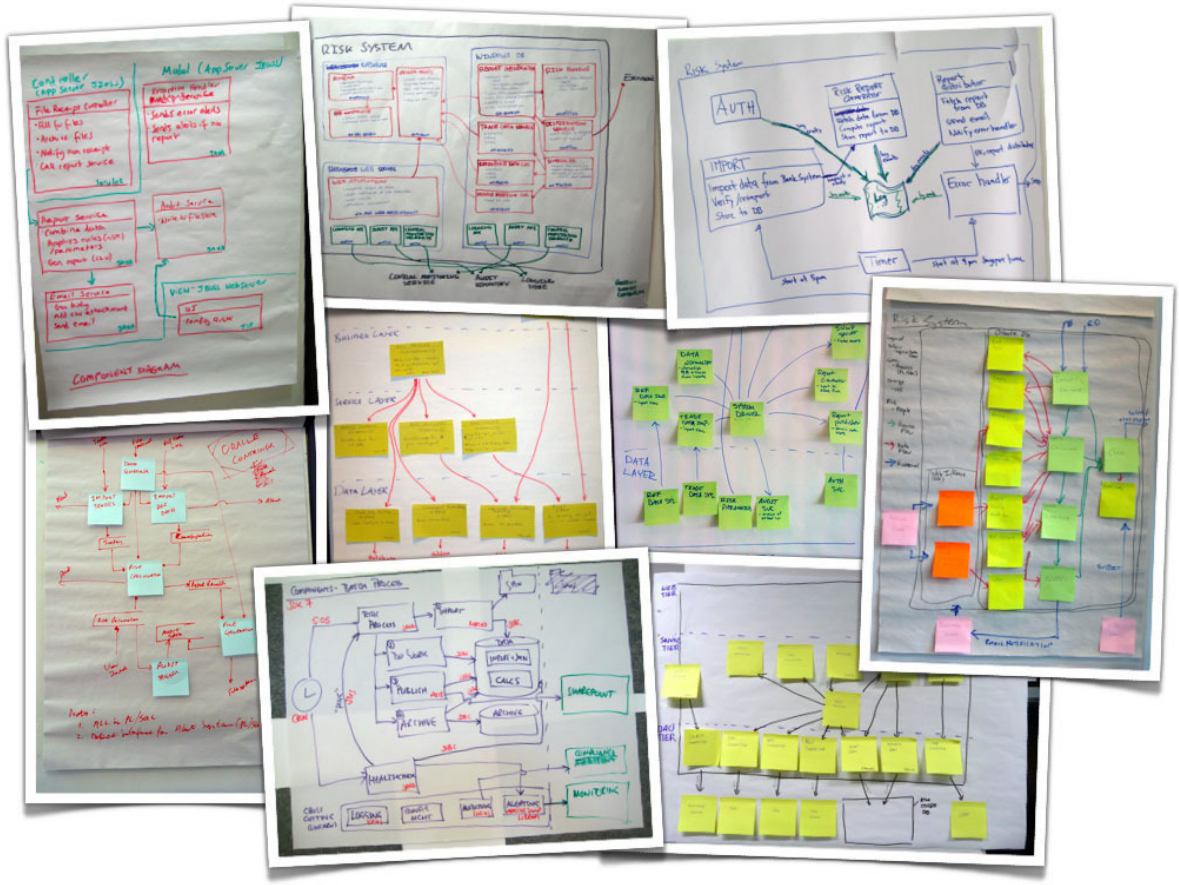
## Intent

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. Here are some examples of component diagrams if you were designing a solution to the risk system.




**Example component diagrams for the 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 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:

- The logical name of the component (e.g. "Risk calculator", "Audit component", etc)
- The technology choice for the component (e.g. Plain Old [Java|C#|Ruby|etc] Object, Enterprise JavaBean, Windows Communication Foundation service, etc)
- A very high-level statement of the component's responsibilities (e.g. either important operation names or brief sentences 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/services that reside inside each of the containers. This is useful because:

- It shows the high-level decomposition of your software system into components/services 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.

# 43 NoUML and 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, 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. Most people will interpret cylinders to be a database of some description, so make sure that if you do use different shapes on your diagrams that you include an explanation of what the various shapes are.

## Responsibilities

A really simple way to add an additional layer of information to an architecture diagram is to annotate things like systems and components with a very short statement of what their responsibilities are. Provided that this is kept short (and using a smaller font for this information works well), adding responsibilities onto a component diagram can help provide a really useful "at a glance" view of what the software system does and how it's been structured.

## Lines

Lines are an important part of most architecture sketches, acting as the glue that holds all of the systems, 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 they 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 try to, 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. 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 often don't stick well to whiteboards, so have some blu-tack handy!

As a final note, 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.

## 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 containers 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 I've seen have placed 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 and, 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.

# 44 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. Of course, the reverse is also true, which is why the people designing software should be master builders.

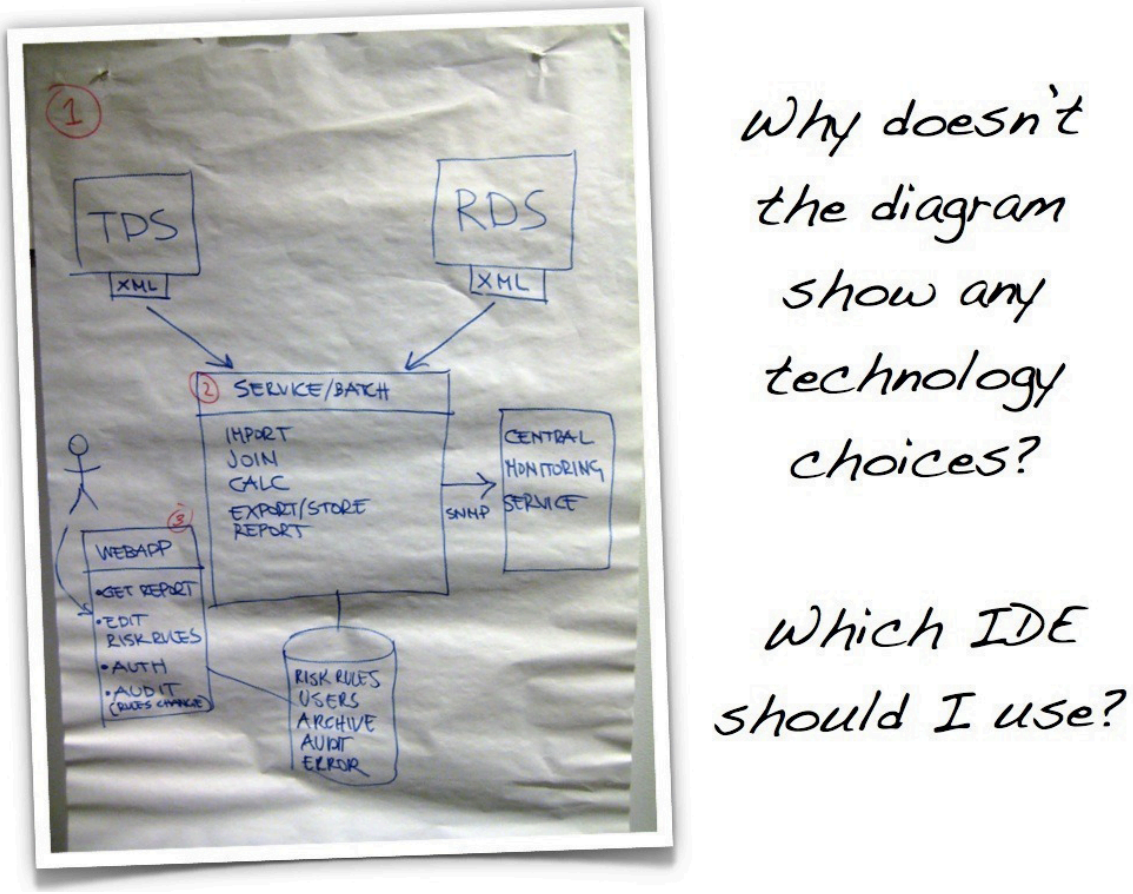
# 45 Technology choices included or omitted?

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 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.




**Which IDE would you start up?**

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

- "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".

## 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" or showing a functional decomposition with absolutely no consideration as to whether the vision is implementable. I like to see software architecture have a grounding in reality and technology choice shouldn't be an implementation detail. One way to do this is to simply show the technology choices by including them on software architecture diagrams.

## Making 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. As for technology decisions cluttering the diagrams, there are a number of strategies for dealing with this concern, including the use of a containers 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 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 that the right people are drawing them. ;-)

# 46 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.

# 47 Questions

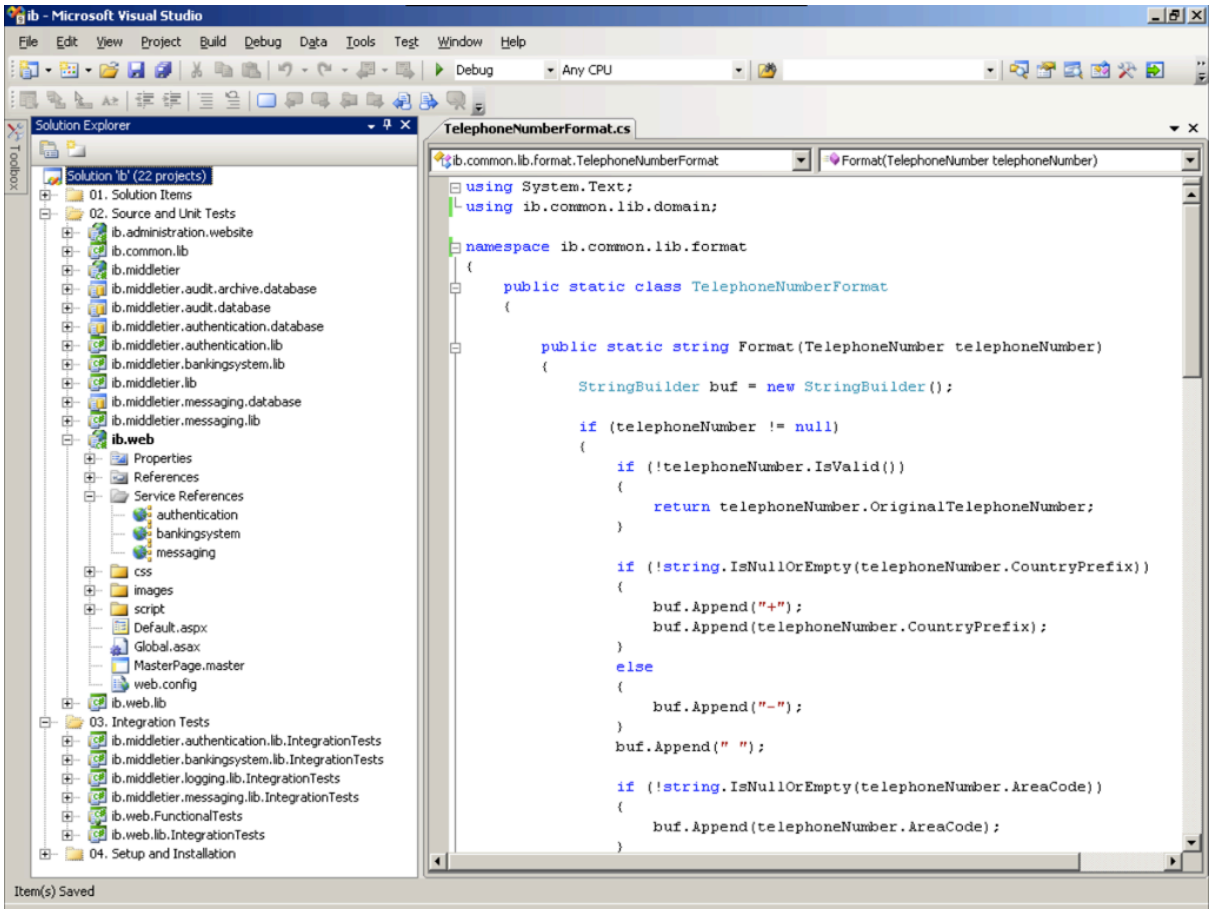
1. Are you able to explain how your software system works at various levels of abstraction? What concepts and levels of abstraction would you use to do this?
2. Do you use UML to visualise the design of your software? If so, is it effective? If not, what notation do you use?
3. Are you able to visualise 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?

# VI Documenting software

# 48 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 IDE. What do you do next and how do you start being productive?



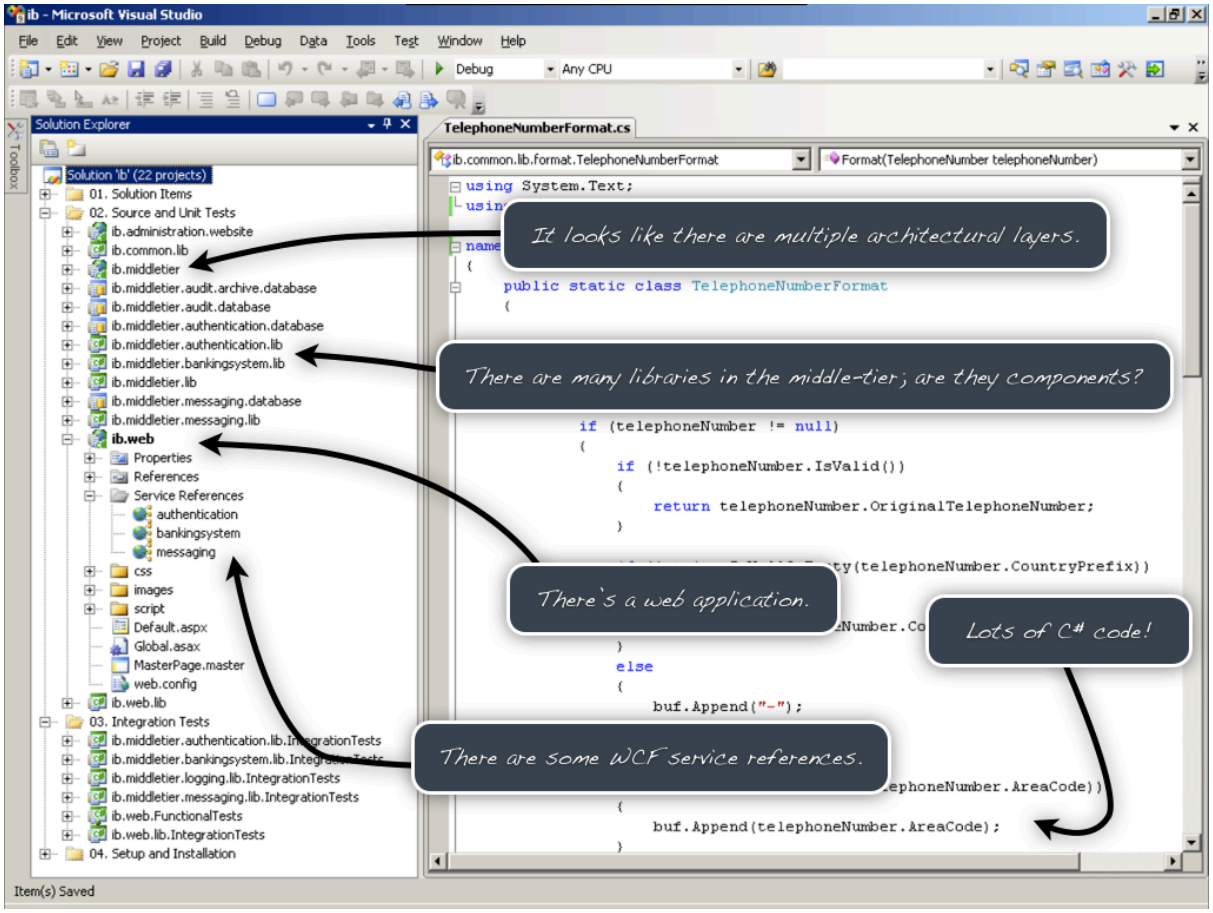

**Where do you start?**

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, 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, we can determine a number of characteristics about the software, which in this case is an 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.
- 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.




**An initial understanding**

# 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 exsiting 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 because you don't know what you don't know.

## Supplementary information

With any software system, there's essentially 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 information is what should be captured in lightweight supplementary documentation and is why you should document your software system in a way that is complementary to the code, describing what the code itself doesn't. The code tells *a* story, but it doesn't tell the whole story.

# 49 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 contraveens 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 isn't just about delivering features

End-users are just one type of stakeholder in your software project and there are usually many others.

- 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 that "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.
- 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.

# 50 Software documentation as a guidebook

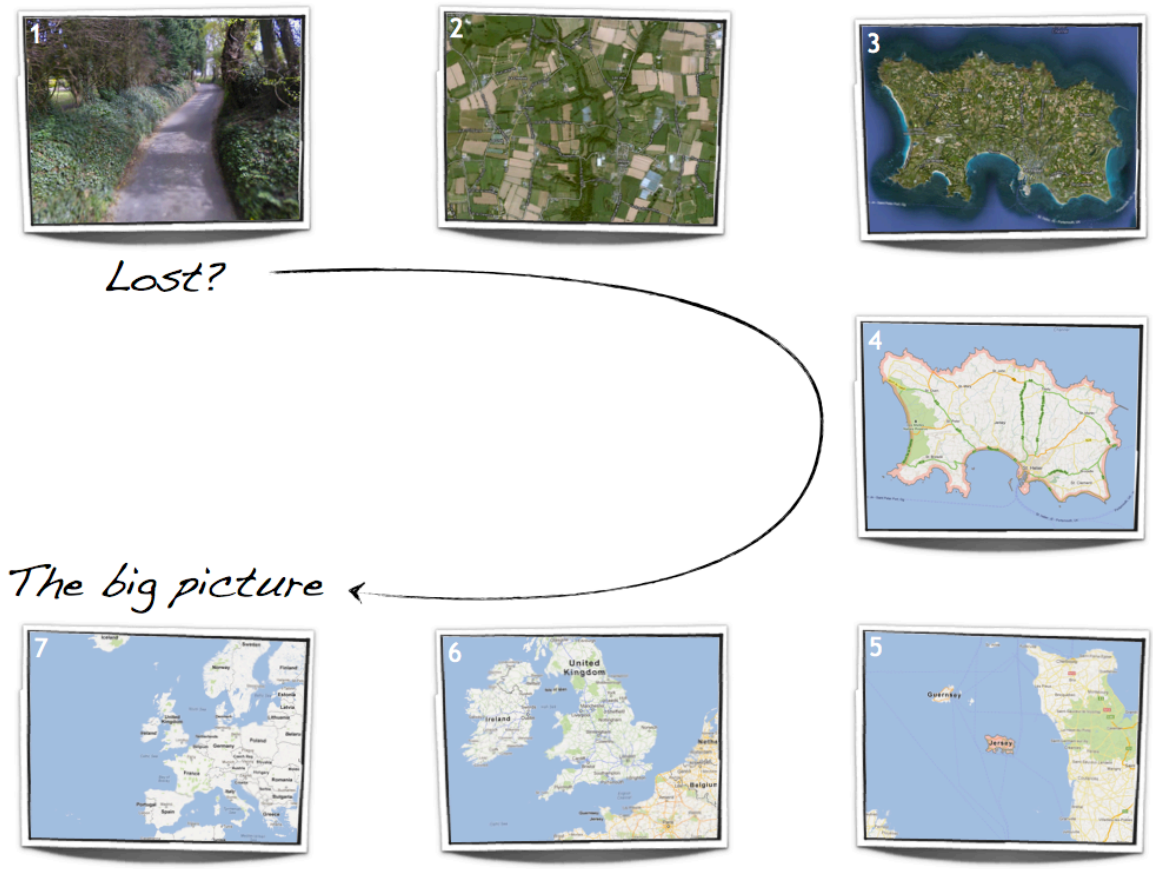
"Working software over comprehensive documentation" is what the Manifesto for Agile Software Development says and it's incredible to see how many software teams have interpreted those five words as "don't write *any* documentation". The underlying principle here is that real working software is much 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 the 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 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 software developers though, most of us 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. For example, the points of interest in a web application might 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 should because it's 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 detail will know how to find it in the codebase anyway. As with everything, there's a happy mid-point somewhere.

The following sections 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. Decision Log

# 51 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.

# 52 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, stories, etc) and, if they are significant to your 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 all of this and a single high-level UML use case diagram is a good way to do so. Instead of showing all the intricacies of the system functionality, simply show the major types of user (actors) and the features (use cases) that are important. To avoid any confusion, add a caveat underneath the diagram to indicates it only shows a high-level summary.

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.

# 53 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 attributes 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.

# 54 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 may be imposed upon you but they aren't all bad. In the words of T.S.Eliot, "Given total freedom the work is likely to sprawl". Give somebody one week to do a task and that task will take one week. Give them two weeks and that same task will take two weeks. Without constraints, there are often an infinite number of ways to solve a problem. 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

Constraints are typically forced upon you, usually by the organisation or customer that has asked for the software system to be built. 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

# Motivation

Constraints have the power to massively influence the architecture, particularly if they limit the technology that can be used to build the solution. 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. Doing this 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.

# 55 Principles

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

# 56 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 containers and components 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.

# 57 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 (e.g.) need to cache data locally?
  - How is performance/scalability/security/etc catered for?
- Has each interface been thought out 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 containers or components 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.

# 58 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 some components work. This is what the code section is for.

## 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 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.

# 59 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 the data of the software system.

## 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.

In addition this section can 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.

# 60 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.

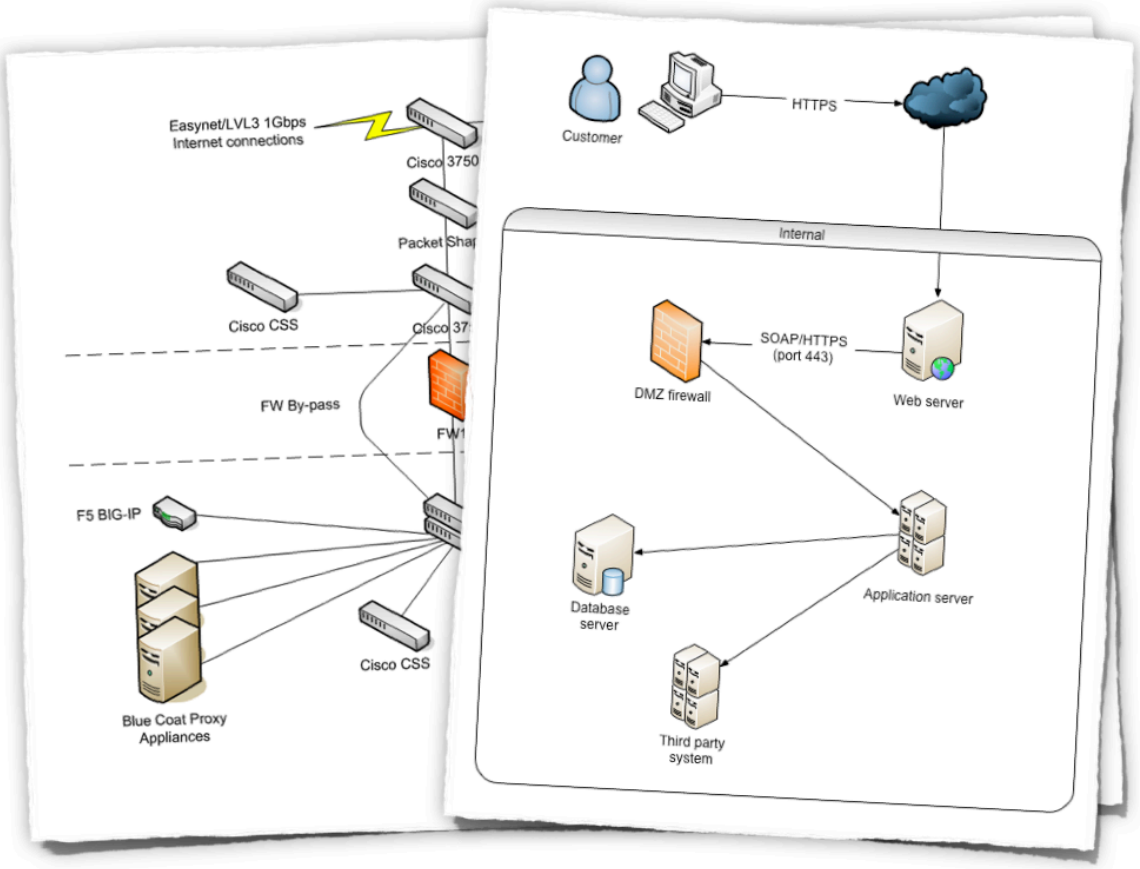
## Intent

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 if needed.

## 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 that 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.

# 61 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 are software components 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 software components 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, 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: 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.

# 62 Operation and Support

The operations and support section allows you you describe how will people 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?
- Is 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. :-)

# 63 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.

People say "nobody ever gets fired for buying IBM", but 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 that is building a software system under an outsourcing agreement for a customer.

# 64 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?

# VII Software architecture in the development lifecycle

# 65 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 no 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.

## 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.

# 66 Collaborative design

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.

# 67 Just enough up front design

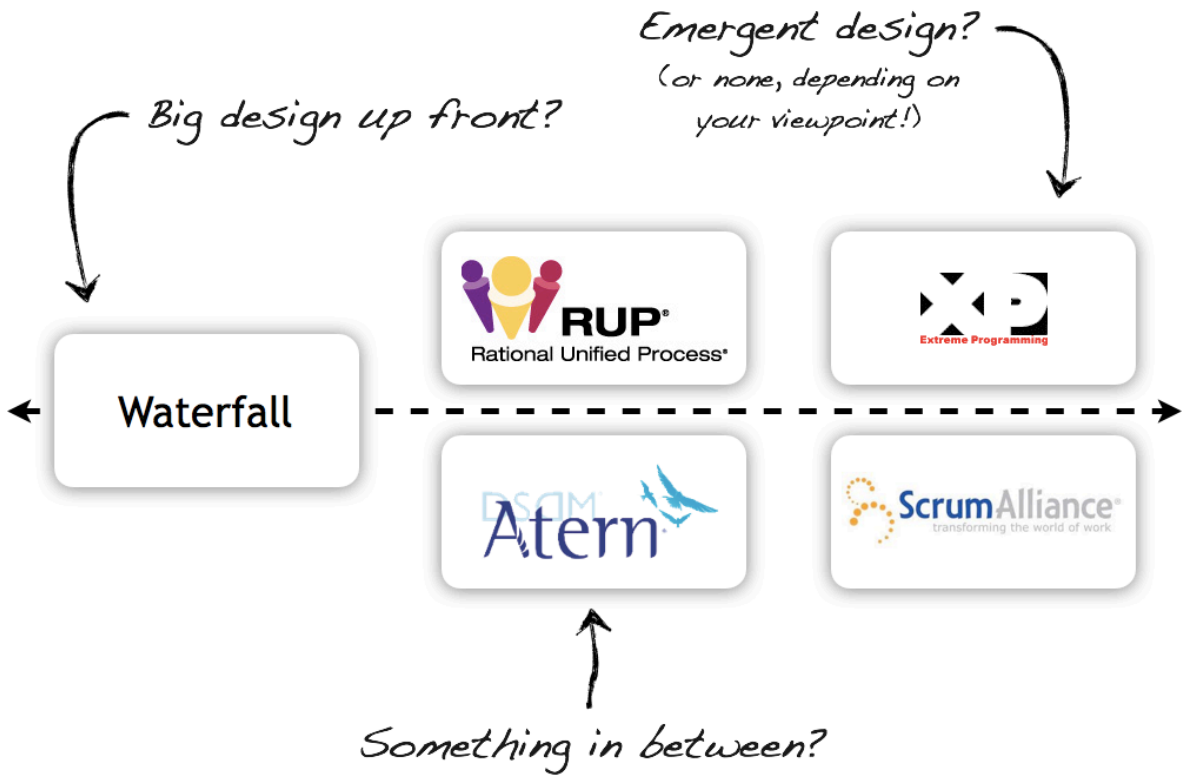
One of the major points of disagreement about software relates to how much up front design to do. People are very polarised as to when they should do design and how much they should do. From experience of working with software teams, the views basically break down like this.

- We need to do all of the software architecture up front, before we start coding features.
- 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 were to compare the common software development methodologies on account of how much up front design they advocate, you'd have something like the following diagram.




**From big design up front to emergent design**

At one end of the scale you have waterfall that, in it's typical form, suggests big design up front where everything must be decided, reviewed and signed-off before a line of code is written. And at the other end you have the agile methods 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 methods and their evangelists don't put much emphasis on the architectural aspects of agile projects, 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 project though, this could end up as "foolishly hoping for the best".

Sitting between the ends of the scale are iterative and incremental methods like the Rational Unified Process (RUP) and DSDM Atern. Both are flexible process frameworks that can be implemented by taking all or part of them. Although many implementations are heavyweight monsters that have more in common with waterfall, they can be scaled down to exhibit a combination of characteristics that let them take the centre ground on the scale. Both 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, both methodologies can be implemented with 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.

## Architecture represents the significant decisions

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.

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.

## How much is just enough?

How much architecture do you need to do then? I say that you need to do "just enough" in order to:

1. Structure: Understand how the significant elements fit together (i.e. at least decomposition down to containers and components).
2. Risks: Identify and mitigate the key risks, which may include prototyping or even changing your architecture.
3. Vision: Create a vision for the team to work with.
4. Foundations: Build firm foundations to underpin the rest of the software delivery.

*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.

# 68 Contextualising just enough up front design

In our move away from the waterfall way of building software, it's common for software teams to ask how much up front design they should be doing. "Just enough" is a good starting point but what exactly does that mean?

## Too little vs too much

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 understanding of "the big picture" within the team.
- No common understanding of "the big picture" across 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. My own definition can be summarised as follows and applies whether the software architecture role is being performed by a single person or shared amongst the team. It includes:

1. Structure: Understand how the significant elements fit together.
2. Risks: Identify and mitigate the key risks, which may include prototyping.
3. Vision: Create a vision for the team to work with.
4. Foundations: Build firm foundations to underpin the rest of the software delivery.

Although 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.

## Practice the architecture process

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 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. Anybody striving for a self-organising team where everybody is able to perform the software architecture role?

# 69 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.

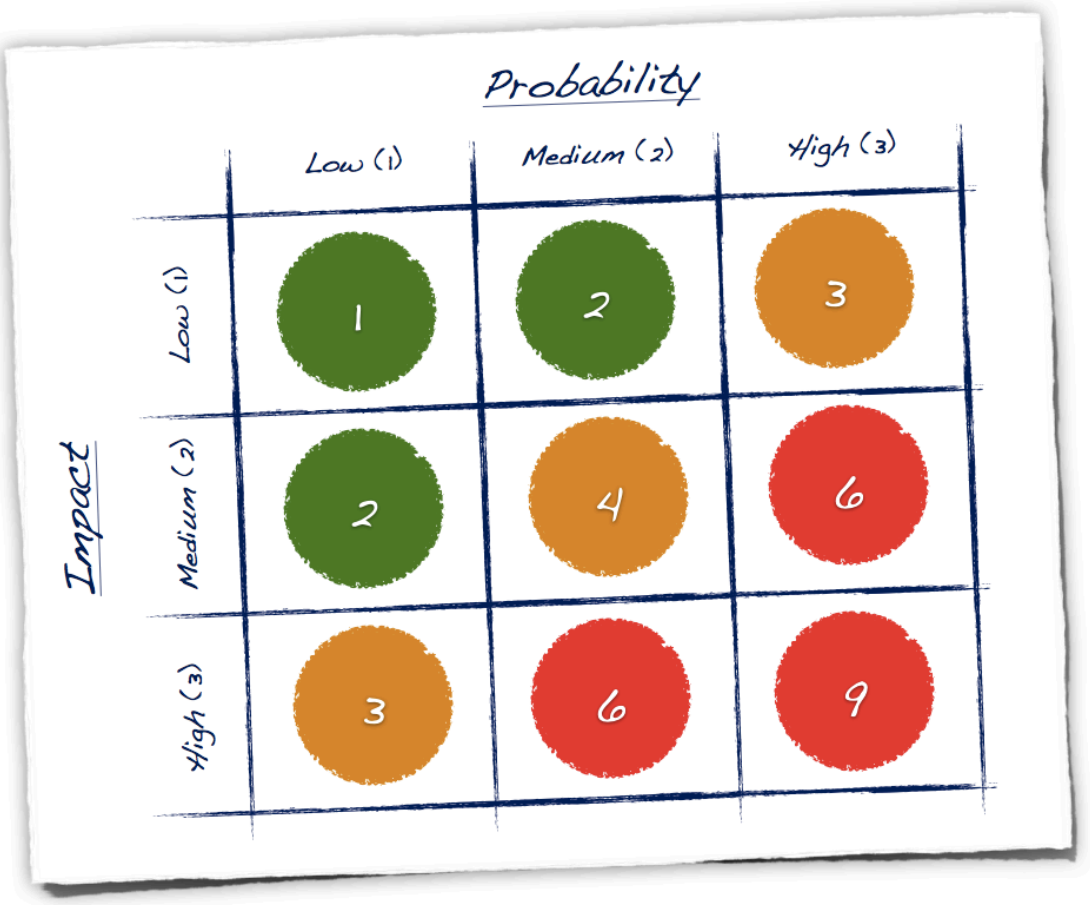
Assuming that you have a list of risks (and risk-storming is a great technique for doing this), how do you quantify each of those risks and assess their relative priorities? There are a number of well established approaches to quantifying risk; including assigning a value of low, medium or high or even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

# 70 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 flipchart 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 to individually write down the risks that they can identify. Here are some examples of the things 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

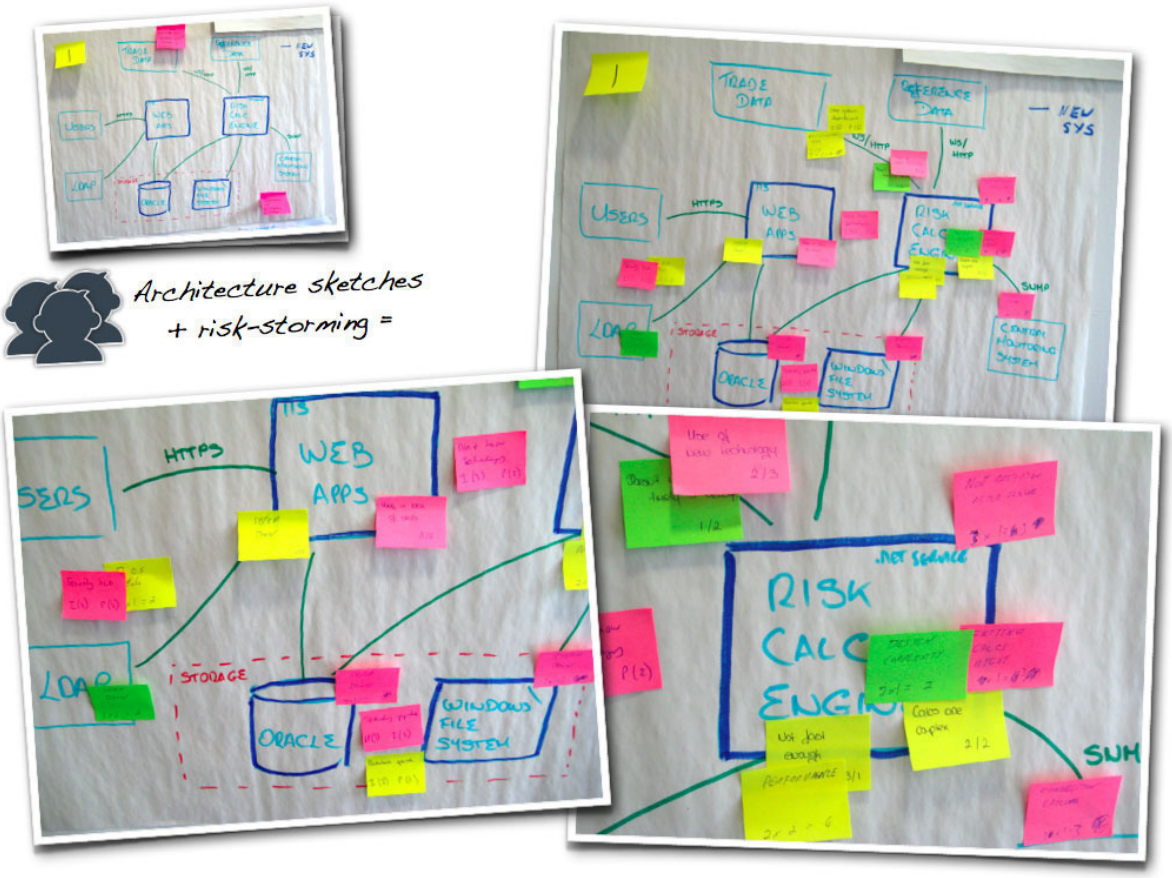
To do this, ask everybody to stand in front of the architecture diagrams and write down each risk they identify on a separate sticky note. Additionally, ask people to quantify each risk based upon probability and impact. Ideally, use different colours of sticky note to represent the different risk priorities. You can timebox this part of the exercise to 5-10 minutes to ensure that it doesn't drag on.

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 that 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. Training the team, restructuring it or hiring new team members in areas where you lack experience (e.g. with new technology).
2. 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. 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, for example in your planning sessions at the start of every iteration.

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.

# 71 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 Scrum master, is this a good idea?
9. What happens if you ignore technical risks?
10. How can you proactively deal with technical risks?

# VIII The retrospective

# 72 It's easier to ask forgiveness than it is to get permission

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.

## How do you introduce software architecture?

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".

# 73 Strategies for reintroducing software architecture

To be completed.

# IX 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 Software Architecture for Developers training course and "Agile software architecture sketches" workshop.

# 74 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.

## 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.

# X Appendix B: Software Guidebook for techtribes.je

This is a sample software guidebook for the techtribes.je website, which is a side-project of mine to provide a focal point for the tech, IT and digital sector in Jersey.

# 75 Introduction

This software guidebook provides an overview of the techtribes.je website. It includes a summary of the following:

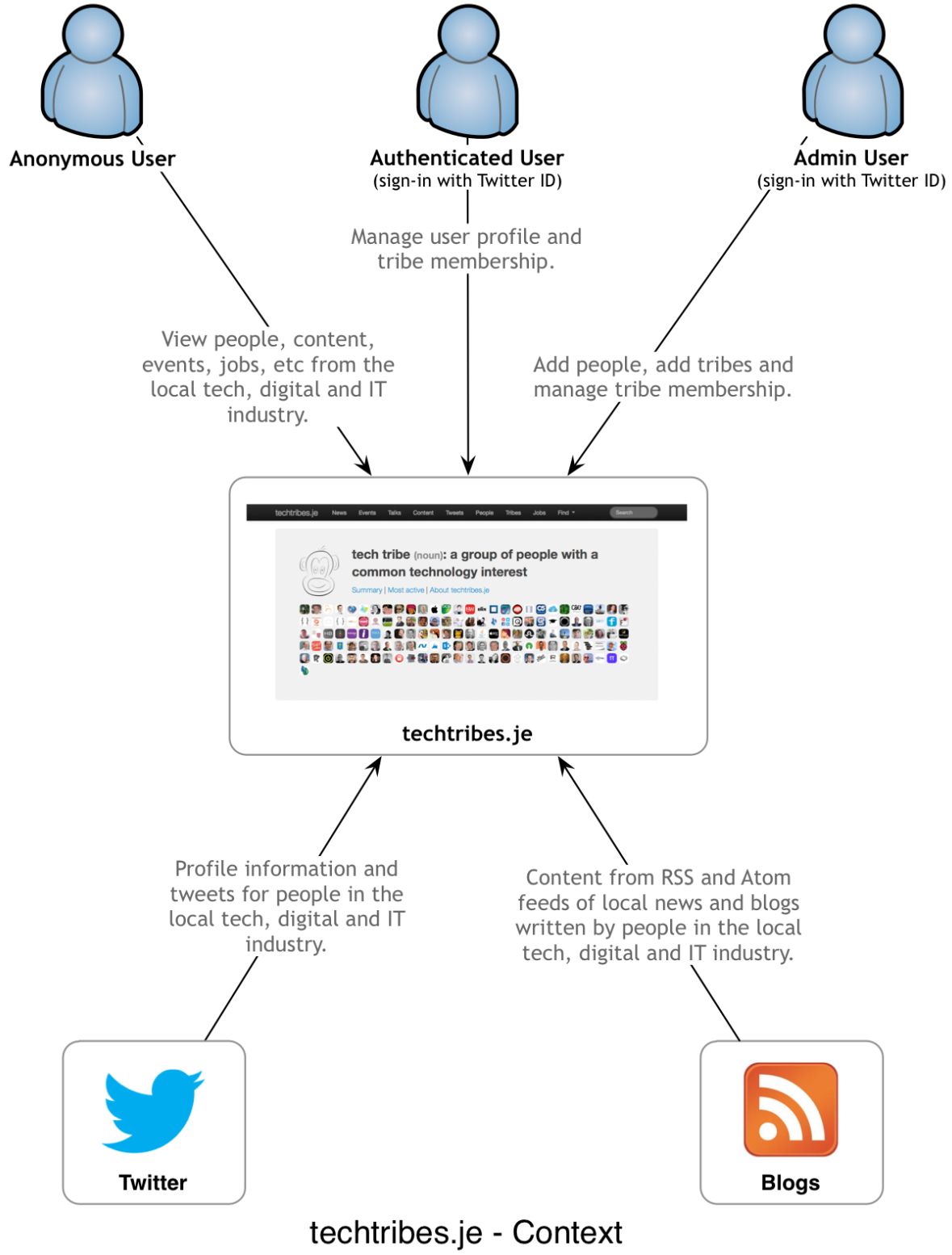
1. The requirements, constraints and principles behind the website.
2. The software architecture, including the high-level technology choices and structure of the software.
3. The infrastructure architecture and how the software is deployed.
4. Operational and support aspects of the website.

Further information about the low-level design and implementation aspects can be found in the source code[1].

[1] I'm planning to open-source the code to techtribes.je so that you can see the correlation between the code and this documentation.

# 76 Context

The techtribes.je website provides a way to find people and content related to the tech, IT and digital sector in Jersey and Guernsey. At the most basic level, techtribes.je is a content aggregator for local tweets, news, blog posts, events, talks and jobs.




**Context diagram**

The purpose of the website is to:

1. Consolidate and share local content, helping to promote it inside and outside of the local community.
2. Encourage an open, sharing and learning culture.

## Users

The techtribes.je website has three types of user:

1. **Anonymous**: anybody with a web browser can view content on the site.
2. **Authenticated**: people that have content aggregated into the website can sign-in to the website using their registered Twitter ID (if they have one) to modify some of their basic profile information.
3. **Admin**: people with "admin" access to the website can manage the people and content that is aggregated into the website.

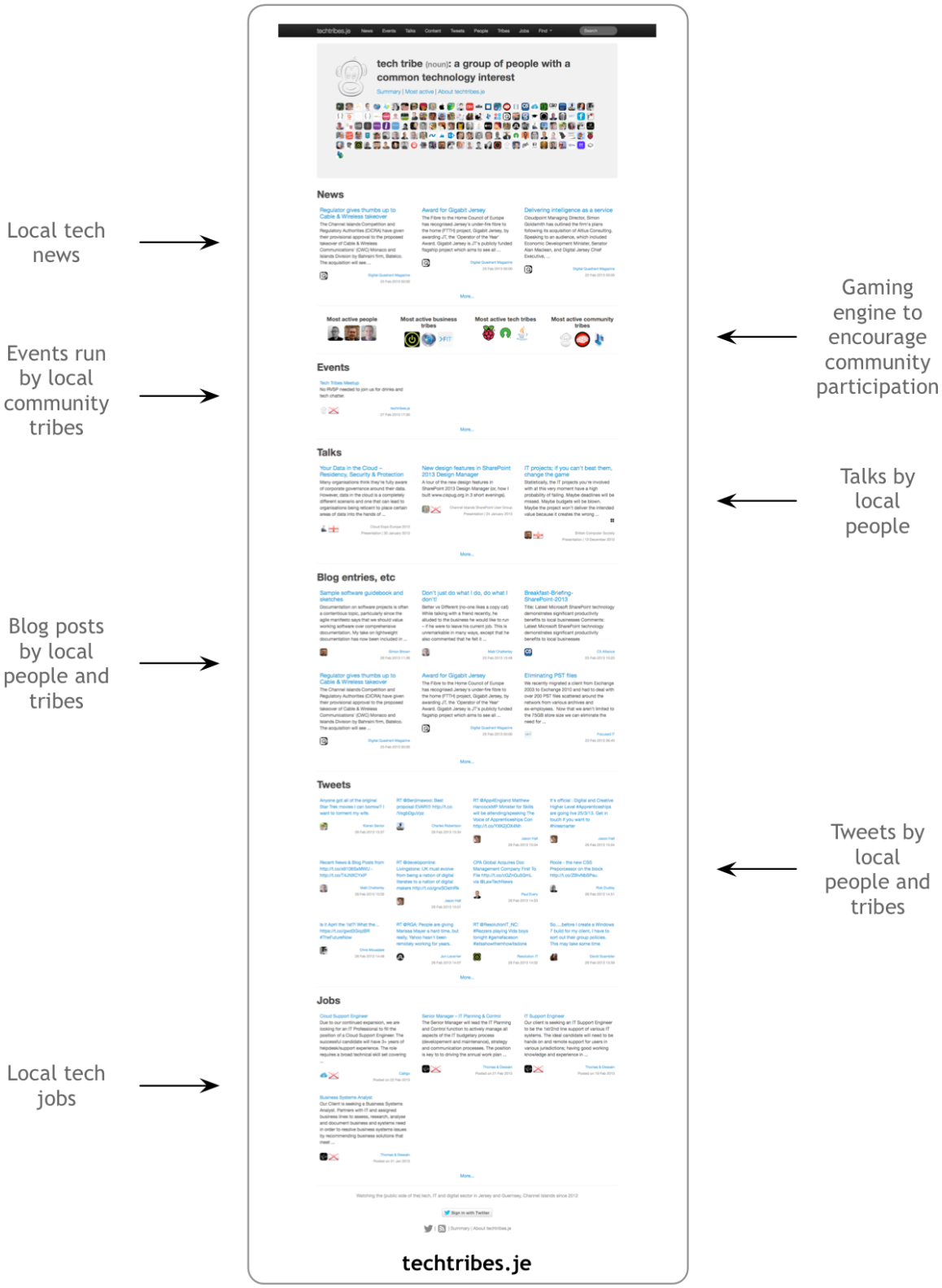
## Systems

There are two types of systems that techtribes.je integrates with:

1. **Twitter**: profile information and tweets from people in the local IT, tech and digital sector are retrieved from Twitter for aggregation into the website.
2. **Blogs**: content from blogs written by people in the local IT, tech and digital sector are retrieved via RSS or Atom feeds for aggregation into the website.

# 77 Functional Overview

This section provides a summary of the functionality provided by the techtribes.je website. If more information is required, please refer to the user stories or the source code.




**An overview of techtribes.je**

# People and Tribes

At the core of techtribes.je are people and tribes:

- **People**: these are people in the local tech, digital and IT sector.
- **Tribes**: a tribe is a group of people and there are 4 types:
  - **Business**: a business tribe represents a local company.
  - **Tech**: a tech tribe is a group of people with a common interest in a particular topic (e.g. Java, Raspberry Pi, SharePoint, etc).
  - **Media**: a media tribe is an organisation that publishes local news.
  - **Community**: a community tribe represents a local user group or other non-profit organisation.

People and tribes have some basic profile information along with a Twitter ID and one or more links to RSS/Atom feeds that techtribes.je uses to aggregate content into the website.

# Content

techtribes.je aggregates and publishes a number of different types of content, all of which is associated with either a person or a tribe.

## Blog posts and Tweets

The major function that techtribes.je performs is to aggregate blog posts and tweets from people and tribes, allowing users of the website to find that content in one place. Blog posts and tweets can be viewed in a number of ways on the website plus there's also a search feature. A single, consolidated RSS feed of *all* blog posts is published by the website too.

## News

Local tech news is simply blog posts by media tribes. Again, it can be viewed in a number of ways on the website.

## Talks

techtribes.je publishes information about the various talks that local people do at conferences, meetups and other events. Each talk has some basic information (i.e. title, abstract, date plus event details) and is associated with a person.

### Events

techtribes.je also publishes information about local tech events, meetups, user groups, etc. Each event has some basic information (i.e. title, description, date, time, event URL, etc) and is associated with a tribe.

### Jobs

Finally, techtribes.je lists local tech jobs. A job has some basic information (i.e. title, description, date posted, a URL for more information, etc) and is again associated with a tribe.

## Users

There are three types of users.

### Anonymous Users

Anonymous users represent anybody visiting techtribes.je and they have the ability to view all of the content on the website in a number of different ways.

### Authenticated Users

Local people who are listed on techtribes.je can sign-in with their Twitter ID in order to manage some basic profile information and the list of tech tribes they are a member of.

### Admin Users

Admin users are simply authenticated users that have been assigned an additional role to perform some basic administration on the website. This includes adding people and tribes to the website, plus managing tribe membership.

## Gaming Engine

The final major function that techtribes.je provides is a simple gaming engine to encourage local people and tribes to engage with other members of the community and share content more often.

### Points

Points are awarded to people and tribes for tweeting, blogging, doing talks and organising events. The points over a seven day period are calculated on an rolling basis to produce a list of who is the most active.

## Badges

In addition to points, badges are awarded to people and tribes for specific achievements. This includes simple things such as tweeting and blogging through to appearing in the top 3 on the most active list and doing a talk off-island.

# Version history

This book is a work in progress; here's a list of the essays that have been added at each version.

- 12th March 2013: Software architects should be master builders.
- 11th March 2013: Technology is not an implementation detail.
- 9th March 2013: The need for sketches and Would you code it that way?.
- 26th February 2013: Functional Overview for the sample software guidebook.
- 25th February 2013: Introduction and Context for the sample software guidebook.
- 19th February 2013: Data and Decision Log
- 18th February 2013: Deployment and Operations and Support
- 21st January 2013: Code and Infrastructure Architecture
- 17th January 2013: More layers = more complexity
- 15th January 2013: Software Architecture and External Interfaces
- 14th January 2013: Software documentation as a guidebook
- 3rd November 2012: Constraints and Principles
- 25th October 2012: Context, Functional Overview and Quality Attributes
- 23rd October 2012: The conflict between agile and architecture - myth or reality? and Technology choices included or omitted?
- 17th July 2012: Containers, Components, SharePoint projects need software architecture too
- 22nd June 2012: Moving fast requires good communication, Context, Effective sketches, Diagram review checklist
- 10th May 2012: Quantifying risk, Risk-storming
- 9th April 2012: The frustrated architect, Everybody is an architect, except when they're not, Crossing the mythical line or bridging the gaping chasm?
- 13th March 2012: The software architecture role, Software architecture introduces control, Software development is not a relay sport, Where are the software architects of tomorrow?, Contextualising just enough up front design, Financial Risk System (Appendix A)
- 11th March 2012: Software architecture is a platform for conversation, C4: context, containers, components and classes, The code doesn't tell the whole story, Mind the gap, Just enough up front design
- 3rd March 2012: You don't need a UML tool, Architectural constructs, Start with the big picture, Collaborative design, It's easier to ask forgiveness than it is to get permission
- 21st February 2012: What is architecture?, Types of architecture?, What is software architecture?, Architecture vs design, What is the big picture?, Strategy rather than code, Is software architecture important?