Software Architecture

for Developers

Volume 2

Visualise, document and explore your software architecture

Simon Brown

Visualise, document and explore your software architecture

Software Architecture for Developers - Volume 2

Simon Brown

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

This version was published on 2019-09-20

Tweet This Book!

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

The suggested hashtag for this book is #sa4d.

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

#sa4d

For Kirstie, Matthew and Oliver

Contents

About the book

I graduated from university in 1996, a time when CASE and modeling tools were popular and in common use. I remember attending a training course about the Unified Modeling Language and the SELECT tooling soon after I started my professional career. A number of the projects I worked on made extensive use of tools like SELECT and Rational Rose for diagramming and documenting the design of software systems. With buggy user interfaces and ugly diagrams, the tooling may not have been brilliant back then, but it was still very useful if used in a pragmatic way.

I'm very much a visual person. 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.

But then something happened somewhere during the early 2000s, probably as a result of the Manifesto for Agile Software Development that had been published a few years beforehand. The way teams built software started to change, with things like diagramming and documentation being thrown away alongside big design up front. I remember seeing a number of software development teams reducing the quantity of diagrams and documentation they were creating. In fact, I was often the only person on the team who really understood UML well enough to create diagrams with it.

Fast forward to 2019; creating software architecture diagrams and documentation seems to be something of a lost art. I've been running software architecture training courses for a number of years, part of which is a simple architecture kata where groups of people are asked to design a software solution and draw some architecture diagrams to describe it. Over 10,000 people and 30 countries later, I have gigabytes of anecdotal photo evidence - the majority of the diagrams use an ad hoc "boxes and lines" notation with no clear notation or semantics. Designing software is where the complexity should be, not communicating it.

This book focusses on the visual communication and documentation of software architecture. I've seen a number of debates over the years about whether software development is an art, a craft or an engineering discipline. Although I think it *should* be an engineering discipline, I believe we're a number of years away from this being a reality. So while this book

doesn't present a formalised, standardised method to communicate software architecture, it does provide a collection of lightweight ideas and techniques that thousands of people across the world find useful. The core of this is my "C4 model" for visualising software architecture, and the "software guidebook". You'll also find discussion about notation, the various uses for diagrams, the value of creating a model and tooling.

About the author

I'm an independent software development consultant specialising in software architecture; specifically technical leadership, communication and lightweight, pragmatic approaches to software architecture. In addition to being the author of Software Architecture for Developers, I'm the creator of the C4 software architecture model and I built Structurizr, which is a collection of tooling to help you visualise, document and explore your software architecture.

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

You can find my website at simonbrown.je and I can be found on Twitter at @simonbrown.

I Visualise

This part of the book is about visualising software architecture using a collection of diagrams to tell different parts of the same overall story.

1. We have a failure to communicate

We've reached an interesting point in the software development industry. Globally distributed teams are building Internet-scale software systems in all manner of programming languages, with architectures ranging from monolithic systems through to those composed of dozens of microservices. Agile and lean approaches are now no longer seen as niche ways to build software, and even the most traditional of organisations are seeking fast feedback with minimum viable products to prove their ideas. Techniques such as automated testing and continuous delivery coupled with the power of cloud computing make this a reality for organisations of any size too. But there's something still missing.

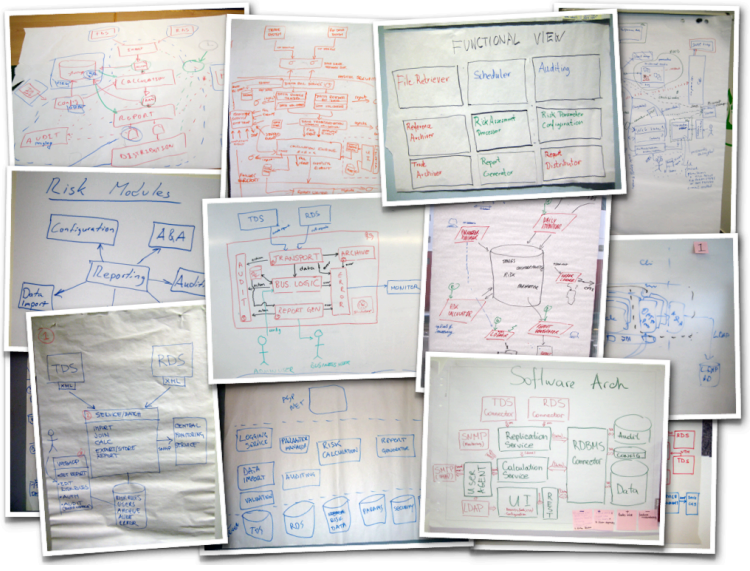
Ask somebody in the building industry to visually communicate the architecture of a building and you'll likely be presented with site plans, floor plans, elevation views, cross-section views and detail drawings. In contrast, ask a software developer to communicate the software architecture of a software system using diagrams, and you'll likely get a confused mess of boxes and lines.

I've asked thousands of software developers around the world to do just this over the past decade, and continue to do so today. The results still surprise me, with the thousands of photos taken during these software architecture sketching workshops anecdotally suggesting that effective visual communication of software architecture is a skill that's sorely lacking in the software development industry. We've forgotten how to visualise software design, both as post-project documentation and during the software development process.

1.1 What happened to SSADM, RUP, UML, etc?

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

Of course, the Unified Modelling Language (UML), a standardised notation for communicating the design of software systems, still lives on. 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.



A selection of typical "boxes and lines" diagrams

Abandoning UML is all very well but, in the race for agility, many software development teams have lost the ability to communicate visually. The example software architecture sketches (pictured) illustrate a number of typical approaches to communicating software architecture and they often suffer from a number of problems as we'll see in the next chapter.

1.2 A lightweight approach

This book aims to resolve these problems, by providing some lightweight ideas and techniques that software development teams can use to visualise and document their software. 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, with a view to:

- Help everybody understand the "big picture" of what is being built, and how this fits into the "bigger picture".
- Create a shared vision of what you're building within the development team.
- Provide a focal point for the development team to remain 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 you explain what you're building to people outside of the development team, whether they are technical or non-technical.
- Fast-track the on-boarding of new software developers into the team.

Furthermore, *any diagrams that are created need to reflect reality*. Those architecture diagrams that you have on the wall of your office at the moment; do they reflect the system that is actually being built, or are they conceptual abstractions that bear no resemblance to the structure of the code?

1.3 Moving fast requires good communication

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

1.4 Draw one or more diagrams

As I mentioned in the introduction, I've asked thousands of software developers to draw software architecture diagrams during workshops I've run. Sometimes this is done as part of a software architecture kata, where groups of people are tasked with designing a software system. Other times it's done as part of a diagramming workshop where I ask software developers to draw some pictures to describe the software architecture of a system they are currently working on. Either way, the result is the same - an ad hoc collection of "boxes and lines" diagrams.

The task is literally phrased as "draw one or more software architecture diagrams to describe your software system". As you can probably imagine, the resulting diagrams are all very different. Some diagrams show a very high-level of abstraction, others present low-level design details. Some diagrams show static structure, others show runtime and behavioural aspects. Some diagrams show technology choices, most don't.

1.5 Where do we start?

When you think about it, this result is unsurprising. Asking people what they found challenging about the exercise reveals that perhaps visual communication of software architecture isn't something that is proactively being taught. I regularly hear the following questions during the workshops:

- "What types of diagram should we draw?"
- "What notation should we use?"
- "What level of detail should we present?"
- "Who is the audience for these diagrams?"

I run this as a group-based exercise, typically with between two and five people per group. Rather than making the exercise easier, having a group of people with different backgrounds and experience tends to complicate matters, as time is wasted debating how best to complete the task. This is because, unlike the building industry, the software development industry lacks a standard, consistent way to think about, describe and visually communicate software architecture. I believe there are a number of factors that contribute to this:

1. In their haste to adopt agile approaches in recent years, many software teams have "thrown out the baby with the bath water". Modeling and documentation have been thrown out alongside traditional plan-driven processes and methodologies. That may sound a little extreme, but many of the software teams I work with only have a very limited amount of documentation for their software systems.
2. Teams that still do see the value in documents and diagrams have typically abandoned the Unified Modeling Language (UML) in favour of an approach that is more lightweight and pragmatic. I'll discuss UML later in the book, but my *anecdotal* evidence, based upon meeting and speaking to thousands of software developers, suggests that UML is *optimistically* only used by a small percentage of software developers.

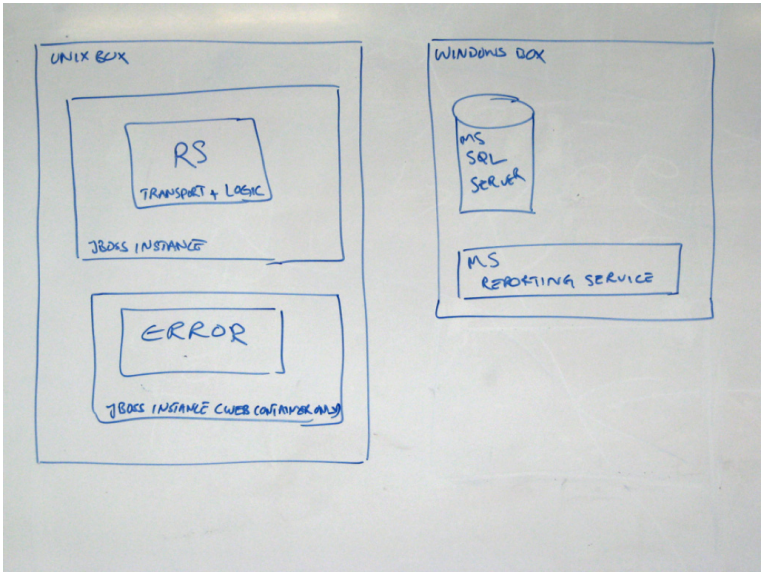
3. There are very few people out there who teach software teams how to effectively model, visualise and communicate software architecture. Based upon running a small number of workshops for computer science undergraduates, this includes lecturers at universities too.

1.6 Some examples

Let's look at some examples. The small selection of photos that follow are taken from my workshops, where groups have been asked to design a small "financial risk system" for a bank, and draw one or more diagrams to communicate the software architecture of it. The purpose of the financial risk system is to import data from two data sources (a "Trade Data System", and a "Reference Data System"), merge the datasets, perform some risk calculations and produce a Microsoft Excel compatible report for a number of business users. A subset of those business users can additionally modify some of the parameters that are used during the calculations. You can see the full set of requirements for the financial risk system in the Appendix A.

The shopping list

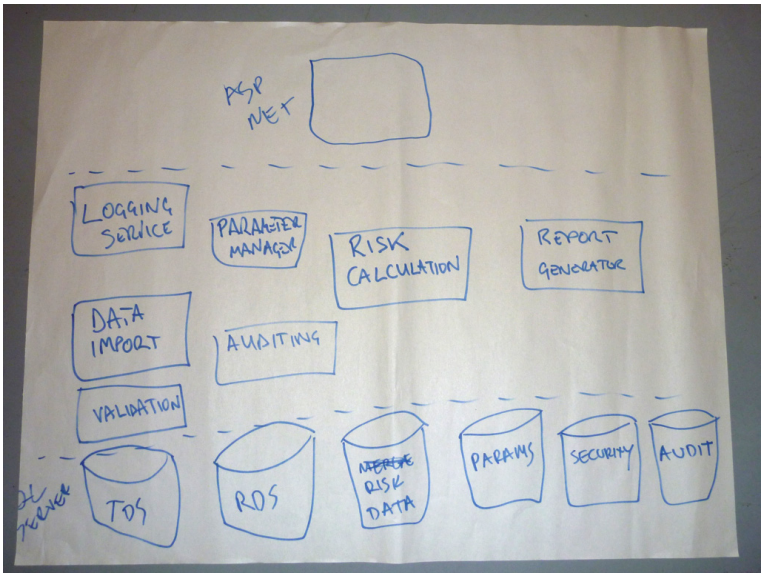
Regardless of whether this is the only software architecture diagram or one of a collection of software architecture diagrams, this diagram doesn't tell you much about the solution. Essentially it's just a shopping list of technologies.



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

Boxes and no lines

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

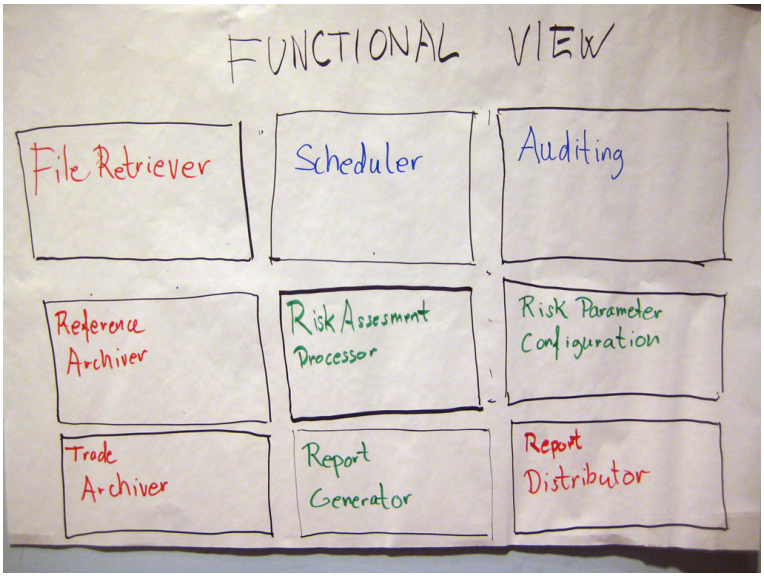


This is a three-tier solution (I think) that uses the Microsoft technology stack. There's an ASP.NET web application at the top, which I assume is being used for some sort of user interaction, although that's not shown on the diagram. The bottom section is labelled "SQL Server" and there are lots of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

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

The "functional view"

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

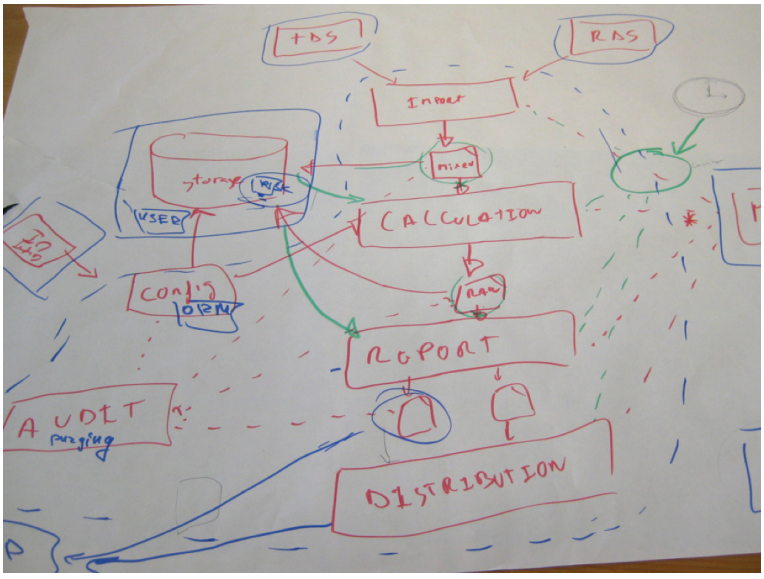


Essentially the group that produced this diagram has simply documented their functional decomposition of the solution into a number of smaller things. Imagine a building architect drawing you a diagram of your new house that simply had a collection of boxes labelled "Cooking", "Eating", "Sleeping", "Relaxing", etc or "Kitchen", "Dining Room", "Bedroom", "Lounge", etc.

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

The airline route map

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



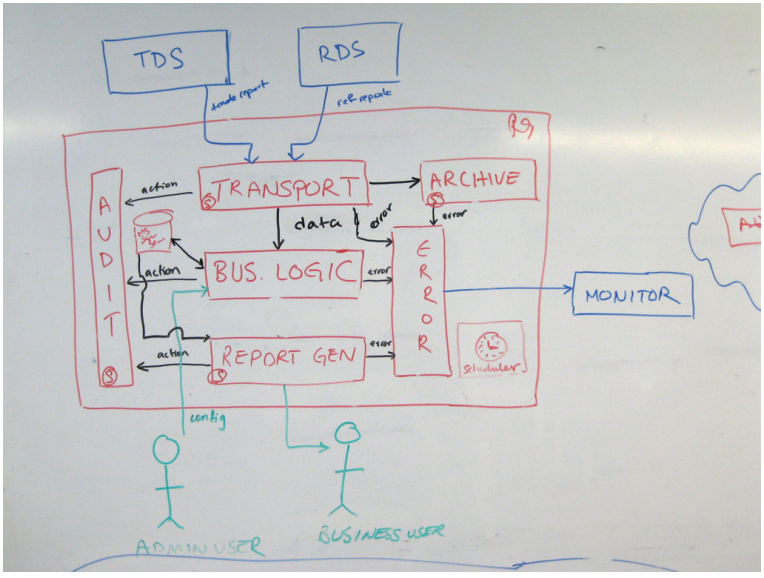
The central spine of this diagram is great because it shows how data comes in from the source data systems (TDS and RDS) and then flows through a series of steps to import the data, perform some calculations, generate reports and finally distribute them. It's a super-simple activity diagram that provides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

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

The left of the diagram is equally confusing, with various lines of differing colours and styles zipping across one another. If you look carefully you'll see the letters "UI" (User Interface) upside-down. The reason? People were writing from wherever they sat around the table.

Generically true

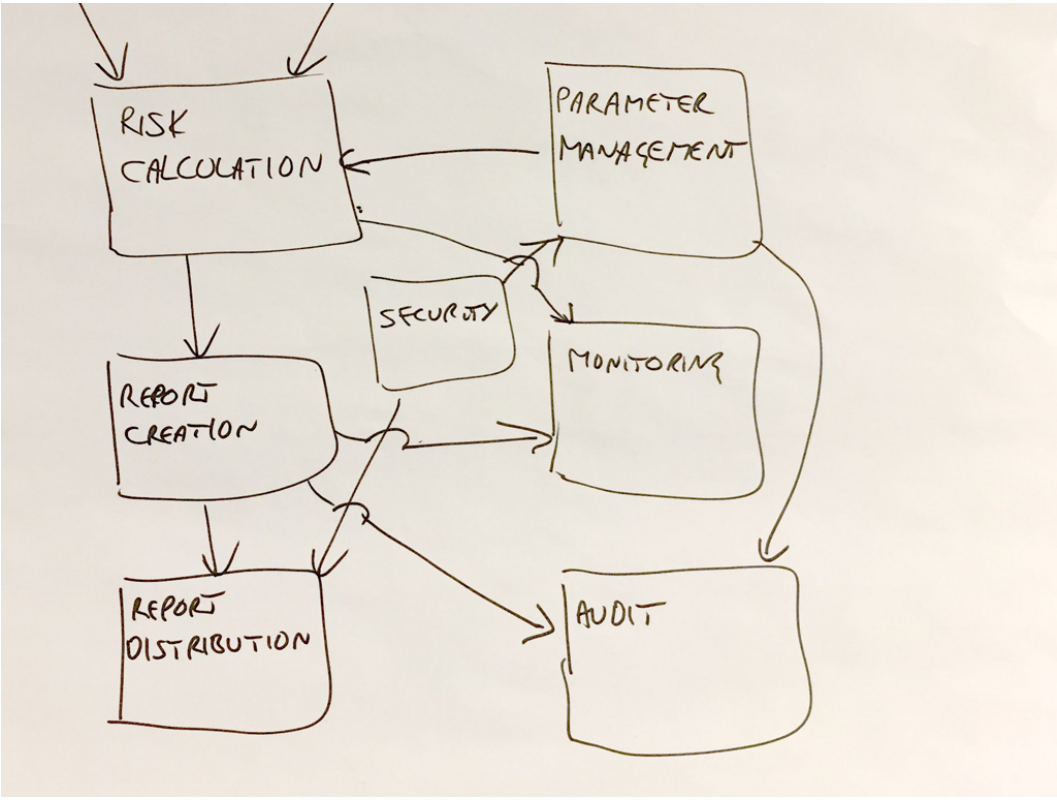
This is another very common style of diagram. Next time somebody asks you to produce a software architecture diagram of a system, present them this photo and you're done!



It's a very "Software Architecture 101" style of diagram where most of the content is generic. Ignoring the source data systems at the top of the diagram (TDS and RDS); we have boxes generically labelled "transport", "archive", "audit", "report generation", "error handling" and arrows labelled "error" and "action". And look at the box in the centre - it's labelled "business logic", which is not hugely descriptive!

The "logical view"

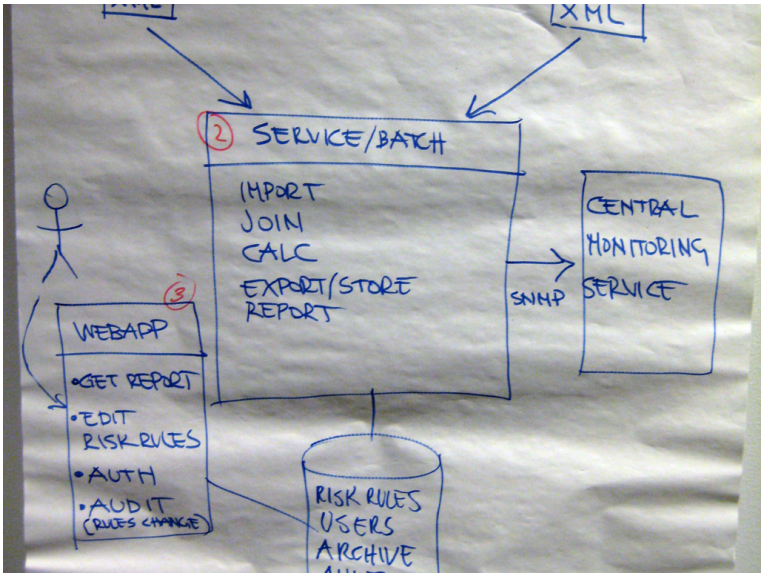
This diagram is also relatively common. It shows the logical (or conceptual, or functional) building blocks that the software system is comprised of, but offers very little information other than that.



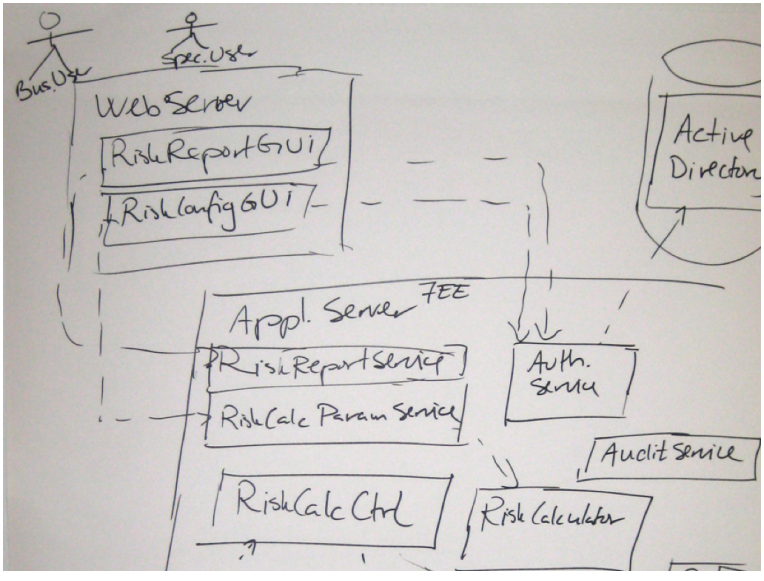
There's a common misconception that "software architecture" diagrams should be "logical" in nature rather than include any references to technology or implementation details, especially before any code is written. We'll look at this later in the book.

Missing technology details

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



And similarly, this next diagram tells us that the solution is an n-tier Java EE system but, like the previous diagram, it omits some important technology details.



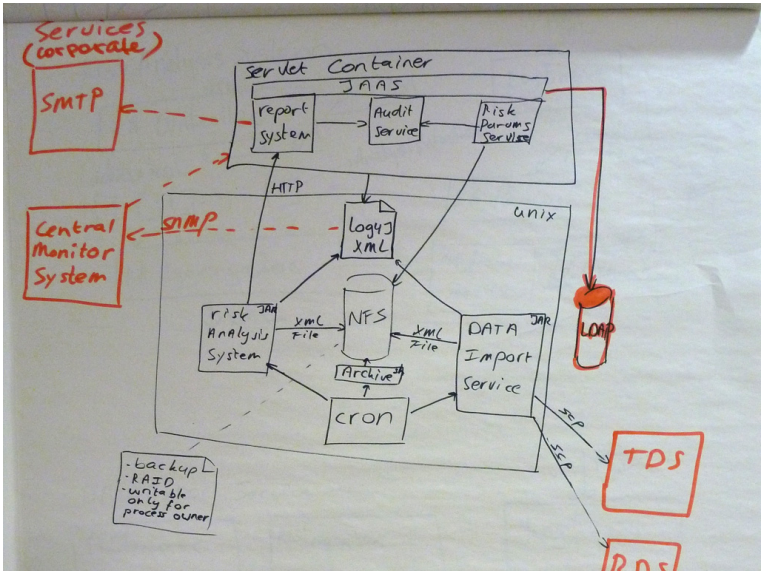
The lines between the web server and the application server have no information about how this communication occurs. Is it SOAP? A JSON web API? XML over HTTPS? Remote

method invocation? Asynchronous messaging? It's not clear.

I'm often told that the financial risk system "is a simple solution that can be built with any technology", so it doesn't really matter anyway. I disagree this is the case and the issue of including or omitting technology choices is covered in more detail elsewhere in the book.

Deployment vs execution context

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



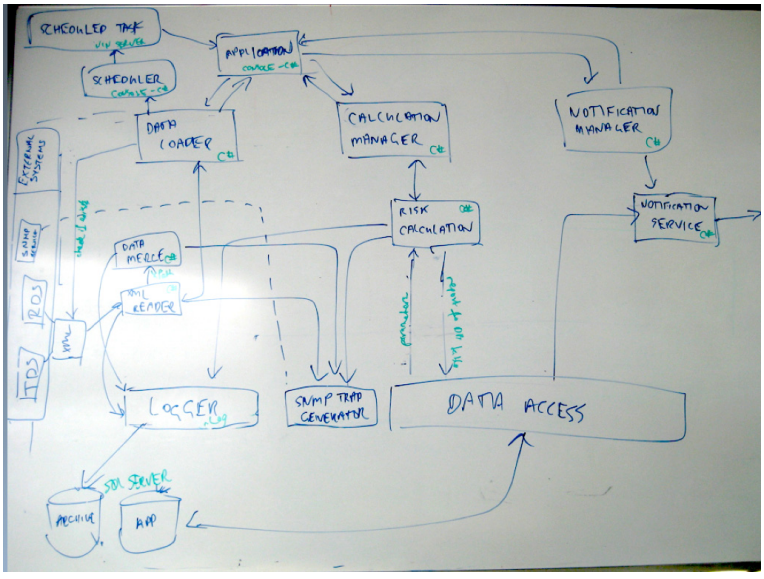
If you look at the Unix box in the centre of the diagram, you'll see two smaller boxes labelled "Risk Analysis System" and "Data Import Service". If you look closely, you'll see that both boxes are annotated "JAR", which is the deployment mechanism for Java code (Java ARchive). Basically this is a ZIP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

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

like to see on the diagram. Rather than the deployment mechanism, I want to understand the execution context.

Homeless Old C# Object (HOCO)

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

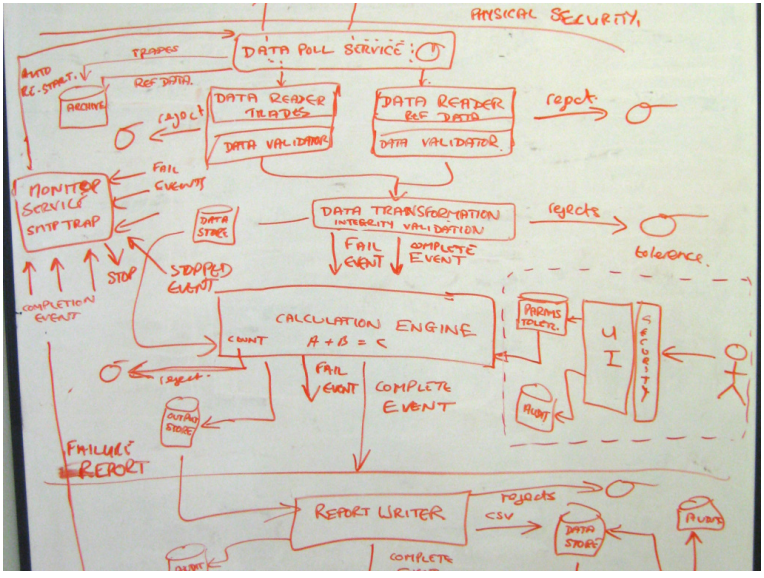


In the bottom left of the diagram is a SQL Server database, and at the top left of the diagram is a box labelled "Application". Notice how that same box is annotated (in green) "Console-C#". Basically, this system seems to be made up of a C# console application and a database. But what about the other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

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

Choose your own adventure

This is the middle part of a more complex diagram.



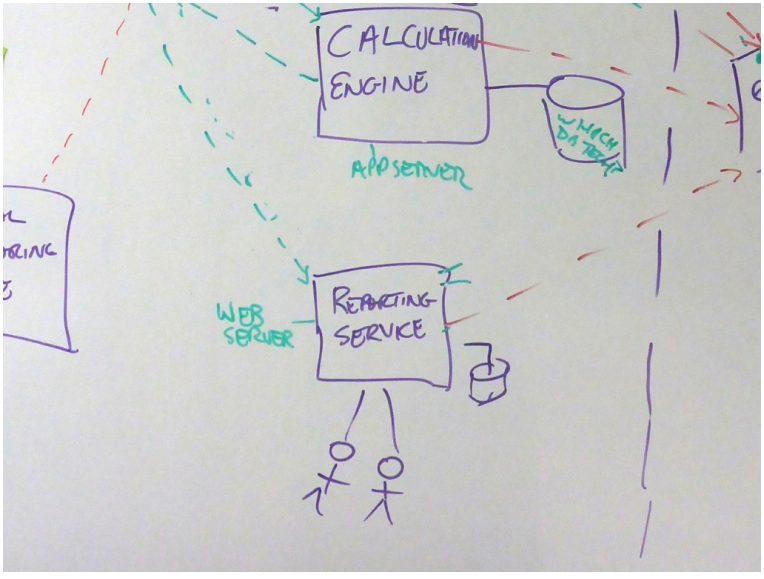
It's a little like those "choose your own adventure" books that I used to read as a kid. You would start reading at page 1 and eventually arrive at a fork in the story where you decide what should happen next. If you want to attack the big scary creature you've just encountered, you turn to page 47.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

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

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

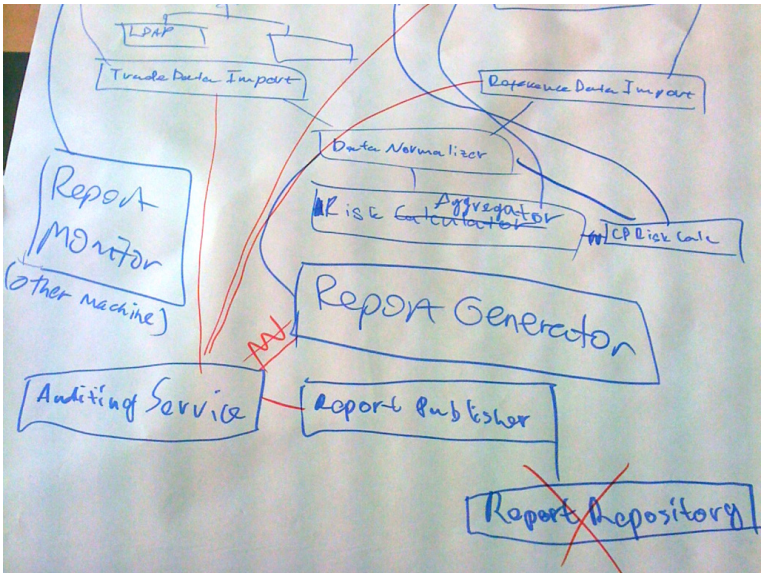
Stormtroopers

To pick up on something you may have noticed from previous diagrams, I regularly see diagrams that include unlabelled users/actors. Essentially they are faceless clones. I don't know who they are and why they are using the software.



Should have used a whiteboard!

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



1.7 Common problems

All joking aside, these diagrams do suffer from one or more of the following problems:

- Notation (e.g. colour coding, shapes, etc) is not explained or is inconsistent.
- The purpose and meaning of elements is ambiguous.
- Relationships between elements are missing or ambiguous.
- Generic terms such as “business logic” are used.
- Technology choices (or options, if doing up front design) are omitted.
- Levels of abstraction are mixed.
- Too much or too little detail.
- No context or a logical starting point.

In addition, the problems associated with a single diagram are often exacerbated when a collection of diagrams is created:

- The notation (colour coding, line styles, etc) is not consistent between diagrams.
- The naming of elements is not consistent between diagrams.
- The logical order in which to read the diagrams isn’t clear.

- There is no clear transition between one diagram and the next.

The example diagrams typify what I see during my workshops and these types problems are incredibly common. A quick Google image search will uncover a plethora of similar block diagrams that suffer from many of the same problems we've seen already. I'm sure you will have seen diagrams like this within your own organisations too.

1.8 The hidden assumptions of diagrams

One of the easiest ways to understand whether a diagram makes sense is to give it to somebody else and ask them to interpret it without providing a narrative. I'm a firm believer that diagrams should be able to stand alone, to some degree anyway. Any narrative should *complement the diagram rather than explain it*. However, I often hear groups in my workshops say the following:

- "We'll talk through the diagrams."
- "This doesn't make sense, but we'll explain it during the presentation."

The assumption that a diagram will be accompanied by a narrative creates a gap between the information captured on the paper and what remains in people's heads. Diagrams that need explaining have limited value, especially when used for the purpose of creating long-lived documentation.

2. A shared vocabulary

The diagrams we've seen so far have been an ad hoc collection of "boxes and lines". Although notation is important, one of the fundamental problems I believe we have in the software development industry is that we lack a common, shared vocabulary with which to think about and describe the software systems we build.

Next time you're sitting in a conversation about software design, listen out for how people use terms like "component", "module", "sub-system", etc. These terms are typically ambiguous. For example, the dictionary definition for the word "component" is "a part of a larger whole". Imagine that you're building a web application, which itself uses a database. Given the dictionary definition, both of the following uses of the word "component" are valid.

- "The web application is a component of the entire software system."
- "The web application is made up of a number of components."

In essence, the word "component" is being used to describe two very different levels of *abstraction.*

2.1 Common abstractions over a common notation

My goal is to see teams able to discuss the structure of their software systems with a *common set of abstractions* rather than struggling to understand what the various notational elements are trying to show. Although I would like to see the software development industry create a standard notation that we all understand (like the electrical engineering industry has with circuit diagrams, for example), I don't think we are there yet. Perhaps a common set of abstractions is more important than a common notation given our industry's current lack of maturity and engineering discipline.

Most maps are a great example of this principle in action. If you get two different maps of your local area and lay them out side by side, they will both show the major roads, rivers, lakes, forests, towns, districts, schools, churches and so on. Visually though, these maps will

probably use different notation in terms of colour-coding, line styles, iconography, etc. In other words, the maps are showing the same things (the same abstractions), but the notation varies. The key to understanding them is exactly that; a key or legend tucked away in a corner somewhere. We can do the same with our software architecture diagrams.

> Diagrams are the maps that help software developers navigate a complex codebase.

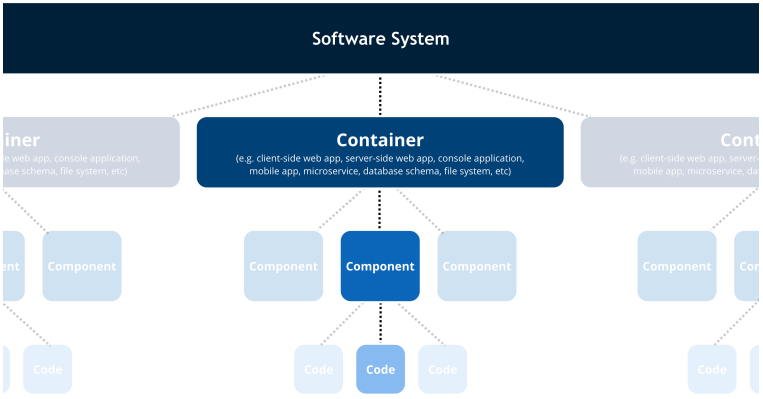
2.2 Static structure

In order to get to this point though, we need to agree upon some vocabulary. And this is the step that is usually missed during the initial iteration of my software architecture diagramming workshops. Teams charge headlong into the exercise without having a shared understanding of the terms they are using.

> "This is a component of our system", says one developer, pointing to a box on a diagram labelled "Web Application".

I've witnessed groups of people having design discussions using terms like "component" where they are clearly not talking about the same thing. Yet everybody in the group is oblivious to this. Each group needs to agree upon the vocabulary, terminology and abstractions they are going to use. The notation can then evolve.

So, notation aside (we'll cover that later in the book), my approach to tackling this problem is to introduce a shared vocabulary that we can use to describe our software. The primary aspect I'm interested in is the *static structure*. And I'm interested in the static structure from *different levels of abstraction*. Once this static structure is understood and in use, it's easy to supplement it with other information to illustrate runtime/behavioural characteristics, infrastructure, deployment models, etc.



A **software system** is made up of one or more **containers** (web applications, mobile apps, desktop applications, databases, file systems, etc), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (e.g. classes, interfaces, objects, functions, etc).

A simple model of architectural constructs used to define the static structure of a software system

I like to think of my software system as being a hierarchy of building blocks as follows:

> A **software system** is made up of one or more **containers** (web applications, mobile apps, desktop applications, databases, file systems, etc), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (e.g. classes, interfaces, objects, functions, etc). And **people** use the software systems that we build.

Level 1: Software systems

A software system is the highest level of abstraction, representing something that delivers value to its users, whether they are human or not.

Level 2: Containers

Put simply, a container represents something that hosts code or data, like an application or a data store. A container is something that needs to be running in order for the overall software system to work. In real terms, a container is something like:

- **Server-side web application**: A Java EE web application running on Apache Tomcat, an ASP.NET MVC application running on Microsoft IIS, a Ruby on Rails application running on WEBrick, a Node.js application, etc.
- **Client-side web application**: A JavaScript application running in a web browser using AngularJS, Backbone.JS, jQuery, etc.
- **Client-side desktop application**: A Windows desktop application written using WPF, a macOS desktop application written using Objective-C, a cross-platform desktop application written using JavaFX, etc.
- **Mobile app**: An Apple iOS app, an Android app, a Microsoft Windows Phone app, etc.
- **Server-side console application**: A standalone (e.g. "public static void main") application, a batch process, etc.
- **Microservice**: A single microservice, hosted in anything from a traditional web server to something like Spring Boot, Dropwizard, etc.
- **Serverless function**: A single serverless function (e.g. Amazon Lambda, Azure Function, etc).
- **Database**: A schema or database in a relational database management system, document store, graph database, etc such as MySQL, Microsoft SQL Server, Oracle Database, MongoDB, Riak, Cassandra, Neo4j, etc.
- **Blob or content store**: A blob store (e.g. Amazon S3, Microsoft Azure Blob Storage, etc) or content delivery network (e.g. Akamai, Amazon CloudFront, etc).
- **File system**: A full local file system or a portion of a larger networked file system (e.g. SAN, NAS, etc).
- **Shell script**: A single shell script written in Bash, etc.
- **etc**

A container is essentially a context or boundary inside which some code is executed or some data is stored. The name "container" was chosen because I wanted a name that didn't imply anything about the physical nature of how that container is executed[1]. For example, some web servers run multiple threads inside a single process, whereas others run single threads across multiple processes. When I'm thinking about the static structure of a software system, I don't want to concern myself with the details of whether a web application is using one operating system process or many when it's servicing requests. It's an important detail, but we can get into that later.

[1]I do appreciate that the term "container" is now in widespread use because of containerisation and technologies like Docker. Feel free to use something like "runtime context", "execution environment" or "deployable unit" instead if you'd prefer to avoid the term "container" when discussing software architecture.

Containers are separately deployable

It's also worth noting that each container should be a separately deployable thing. The physical deployment is another important detail that we will look at later, but, in theory anyway, every container can be deployed onto or run on a separate piece of infrastructure; whether that infrastructure is physical, virtual or containerised. The implication here is that communication between containers is likely to require an out-of-process or remote procedure call across the process and/or network boundary.

To give an example, let's imagine you're building a website that is comprised of two different web applications (e.g. a desktop version and a mobile version, or an end-user version serving HTML and an API endpoint serving JSON). There are a number of scenarios to consider:

1. Each web application is packaged up into separately deployable units (e.g. two Java WAR files, two ASP.NET web applications, etc). This is two containers, regardless of whether both deployable units are actually deployed into the same physical web server (a deployment optimisation).
2. Although you think about the two web applications as being logically separate, they are actually *inseparable* because they are packaged as a single deployment unit (e.g. a single Java WAR file or ASP.NET web application). This is a single container.

The same is true with relational database schemas. I would treat two separate schemas as two separate containers, irrespective of whether they are deployed into the same database server or not.

As a final note, put simply, a container refers to an execution context and it's a really *runtime* construct. This means that libraries or modules (e.g. JAR files, DLL files, .NET assemblies, etc) should not be considered as containers unless they are runnable on their own, like a Java or Spring Boot application that is packaged into an executable JAR file, for example.

Level 3: Components

The word "component" is a hugely overloaded term in the software development industry, but I like to think of a component as being a grouping of related functionality encapsulated behind a well-defined interface. With the C4 model, components are *not* separately deployable units. Instead, it's the container that's the deployable unit. In other words, all of the components inside a container typically execute in the same process space.

Aspects such as how those components are packaged (e.g. one component vs many components per JAR file, DLL, shared library, etc) is an orthogonal concern and, from my perspective, doesn't affect how we think about components.

Level 4: Code

Finally, components are made up of one or more code elements constructed with the basic building blocks of the programming language that you're using; classes, interfaces, enums, functions, objects, etc.

2.3 Components vs code?

A component is a way to step up one level of abstraction from the code-level building blocks that you have in the technology you're using. For example:

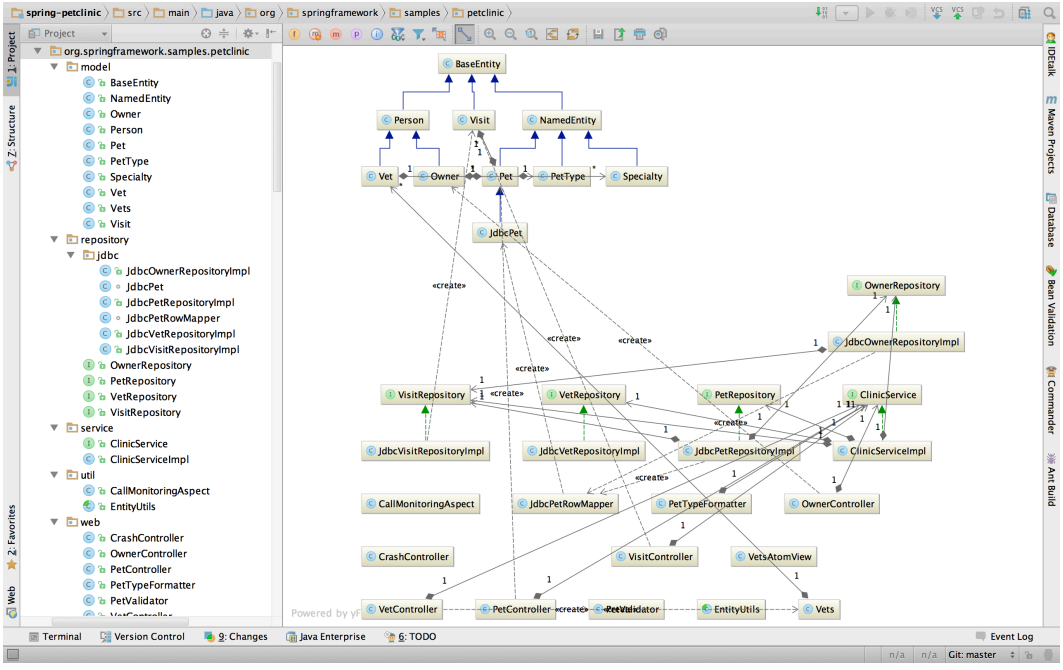
- **Object-oriented programming languages (e.g. Java, C#, C++, etc)**: A component is made up of classes and interfaces.
- **Procedural programming languages (e.g. C)**: A component could be made up of a number of C files in a particular directory.
- **JavaScript**: A component could be a JavaScript module, which is made up of a number of objects and functions.
- **Functional programming languages**: A component could be a module (a concept supported by languages such as F#, Haskell, etc), which is a logical grouping of related functions, types, etc.
- **Relational database**: A component could be a logical grouping of functionality; based upon a number of tables, views, stored procedures, functions, triggers, etc.

If you're using an object-oriented programming language, your components will be implemented using one or more classes. Let's look at a quick example to better define what a component is in the context of some code.

The Spring PetClinic application is a sample codebase that illustrates how to build a Java web application using the Spring MVC framework. From a non-technical perspective, it's a software system designed for an imaginary pet clinic that stores information about pets and their owners, visits made to the clinic, and the vets who work there. The system is only designed to be used by employees of the clinic. From a technical perspective, the Spring PetClinic system consists of a web application and a relational database.

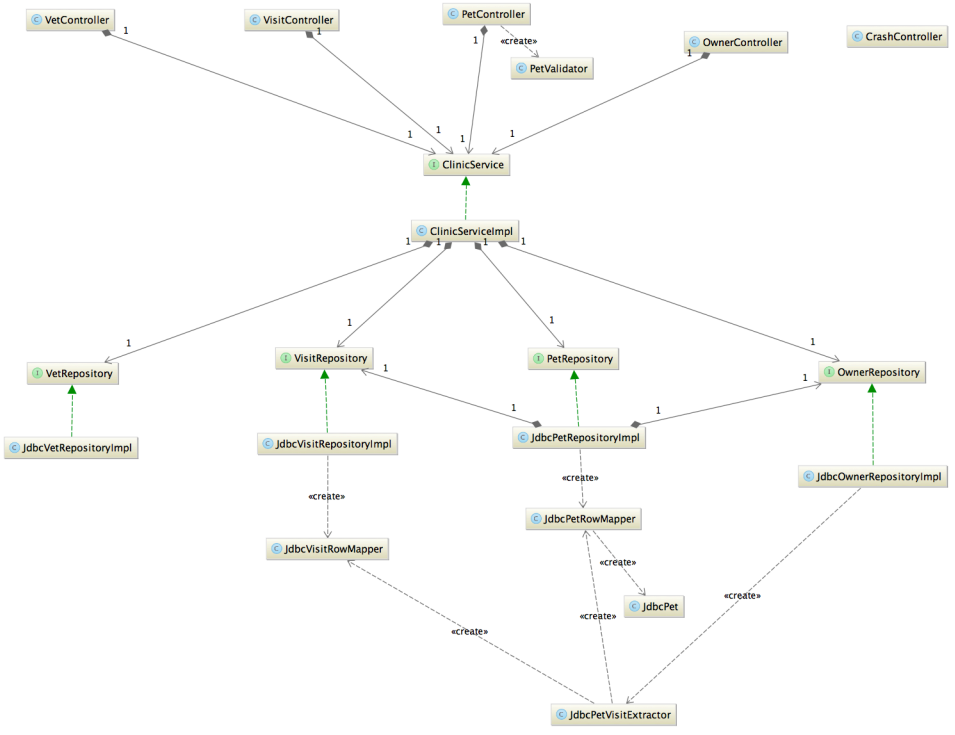
The version of the web application we'll look at here is a typical layered architecture consisting of a number of web MVC controllers, a service containing "business logic" and some repositories for data access. There are also some domain and util classes too. If you

download a copy of the GitHub repository, open it in your IDE of choice and visualise it by reverse-engineering a UML class diagram from the code, you'll get something like this.

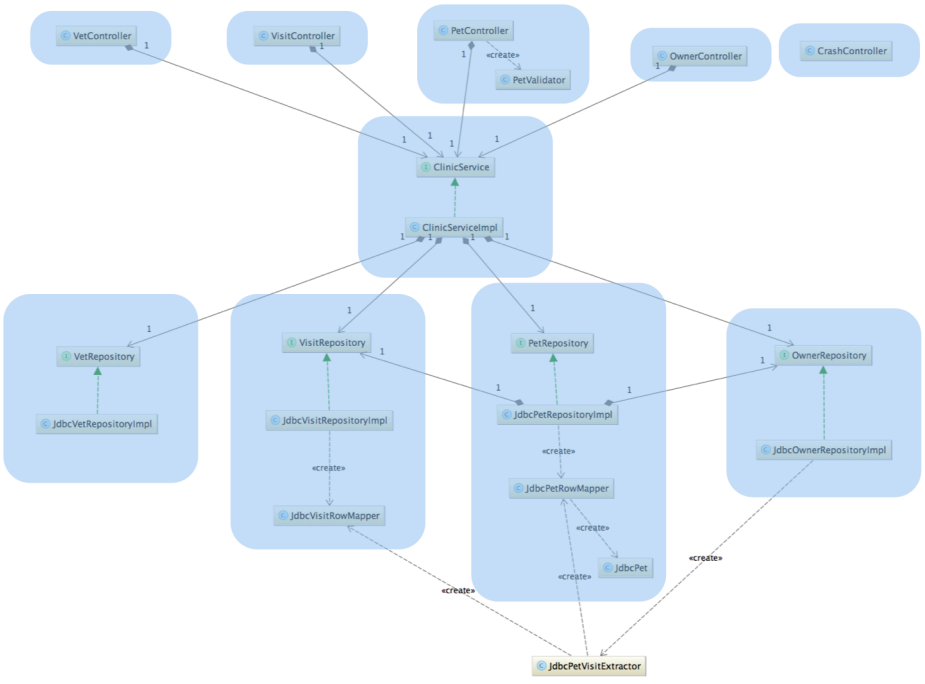


As you would expect, this diagram is showing you all of the Java classes and interfaces that make up the Spring PetClinic web application, plus all of the relationships between them. The properties and methods are hidden on the diagram because they add too much noise to the picture. This isn't a complex codebase by any stretch of the imagination but, by showing classes and interfaces, the diagram is showing too much detail.

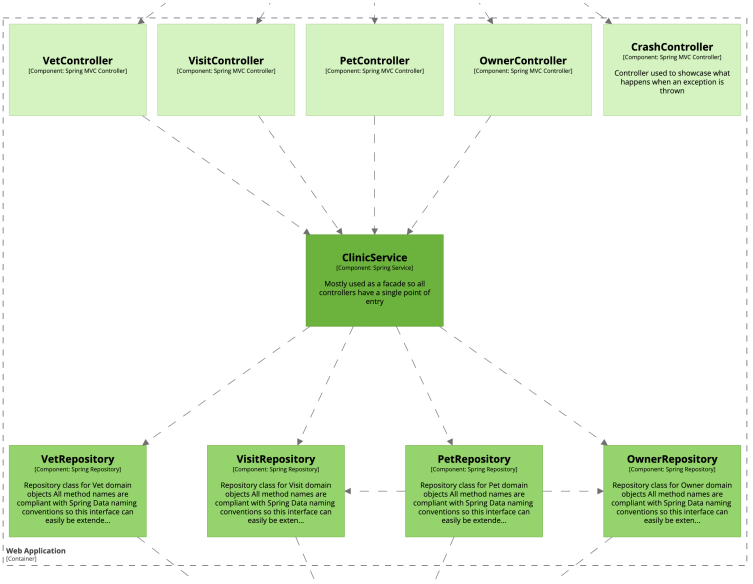
Let's remove those classes that aren't useful to having an "architecture" discussion about the system. In other words, let's only show those classes/interfaces that have some significance from a static structure perspective. In concrete terms, for this specific codebase, it means excluding the model (domain) and util classes.



After a little rearranging, we now have a simpler diagram with which to reason about the software architecture. We can also see the architectural layers again (controllers, services and repositories). But this diagram is still showing *code-level elements* (i.e. classes and interfaces). In order to zoom up one level, we need to identify which of these code-level elements can be grouped together to form "components". The strategy for grouping code-level elements into components will vary from codebase to codebase (and we'll discuss this later in the book) but, for this codebase, the strategy might look like this.



Each of the blue boxes represents what I would consider to be a "component" in this codebase. In summary, each of the web controllers is a separate component, along with the result of combining the remaining interfaces and their implementation classes. If we remove the code level noise, we get a picture like this.



In essence, we're grouping the classes and interfaces into components, which is a unit of related functionality. You will likely have shared code (e.g. abstract base classes, supporting classes, helper classes, utility classes, etc) that are used across many components, such as the `JdbcPetVisitExtractor` in this example. Some can be refactored and moved "inside" a particular component, but some of them are inevitable.

Although this example illustrates a traditional layered architecture, the same principles are applicable regardless of how you package your code (e.g. by layer, feature or component) or the architectural style in use (e.g. layered, hexagonal, ports and adapters, etc). My aim in all of this is to minimise, and in fact *remove*, the gap between how software developers think about components from a logical and physical perspective. Components should be real things, evident in the code, rather than logical constructs that are used in architecture discussions only.

2.4 Modules and subsystems?

If you're familiar with the definition of software architecture from books such as Software Architecture in Practice, you will have noticed that I don't use the term "module" as a part of the static structure definition. A module typically refers to an implementation unit (e.g. a library or some other collection of programming elements) that may be combined with other modules into a component, which itself is instantiated to create component instances

at runtime. While this model makes sense, I find it adds an additional level of detail that is usually unnecessary when thinking about a software system from a "big picture" perspective. For this reason, I've deliberately avoided using the term "module" and instead focus on the identification of coarser-grained components within the static structure.

I've also avoided using the term "subsystem", which some people use to refer to a collection of related components or a functional slice of a software system. The problem I have with the term "subsystem" is that it's often difficult to map this concept onto a real-world codebase. If the concept of components and modules, or systems and subsystems, is useful, then feel free to build that into the shared vocabulary that you create.

2.5 Microservices?

Given the degree of hype and discussion around microservices at the moment, it's worth being explicit about how to describe microservices using the vocabulary we've defined so far. Broadly speaking, there are two options.

1. Microservices as software systems

If your software system has a dependency upon a number of microservices that are outside of your control (e.g. they are owned and/or operated by a separate team), I would treat these microservices as external *software systems* that you can't see inside of.

2. Microservices as containers

On the other hand, if the microservices are a part of a software system that you are building (i.e. you own them), I would treat them as *containers*, along with any data stores that those microservices use (these are separate containers). In the same way that a modular monolithic application is a container with a number of components running inside it, a microservice is a container with a (smaller) number of components running inside it. The actual number of components will depend upon the implementation strategy. It could range from the very simple (i.e. one, where a microservice is a container with a single component running inside) through to something like a mini-layered or hexagonal architecture.

2.6 Serverless?

I tend to treat the serverless concepts (e.g. Amazon Lambdas, Azure Functions) in the same way as microservices. If you're building a software system comprised of a number of

serverless functions, think of them as containers because they are all separately deployable.

2.7 Platforms, frameworks and libraries?

You might also be wondering where platforms, frameworks and libraries fit into all of this. After all, platforms and frameworks are usually something that you build your software on top of, while libraries are things that your software uses. In most cases, these are really just technology choices that components make use of, and are therefore implementation details rather than components in their own right. For example, the components in the Spring PetClinic web application *use* the Spring framework, but I wouldn't show the Spring framework as a component. The same is true of the logging library and database driver, etc.

In most cases, good design[2] will encourage you to wrap up components from platforms, frameworks and libraries into your own components. Having said that, there are times when your software might use components provided by platforms, frameworks and libraries of course. Understanding how you use such components is the key to understanding how they fit into a static model of your software.

2.8 Create your own shared vocabulary

I've illustrated the vocabulary that I use here, and it works for the majority of organisations I work with. But, of course, there are no universal rules. Sometimes, rather than introducing something new, it's easier for an organisation to stick with the vocabulary they are already using, ensuring that it is explicitly defined and understood by everybody.

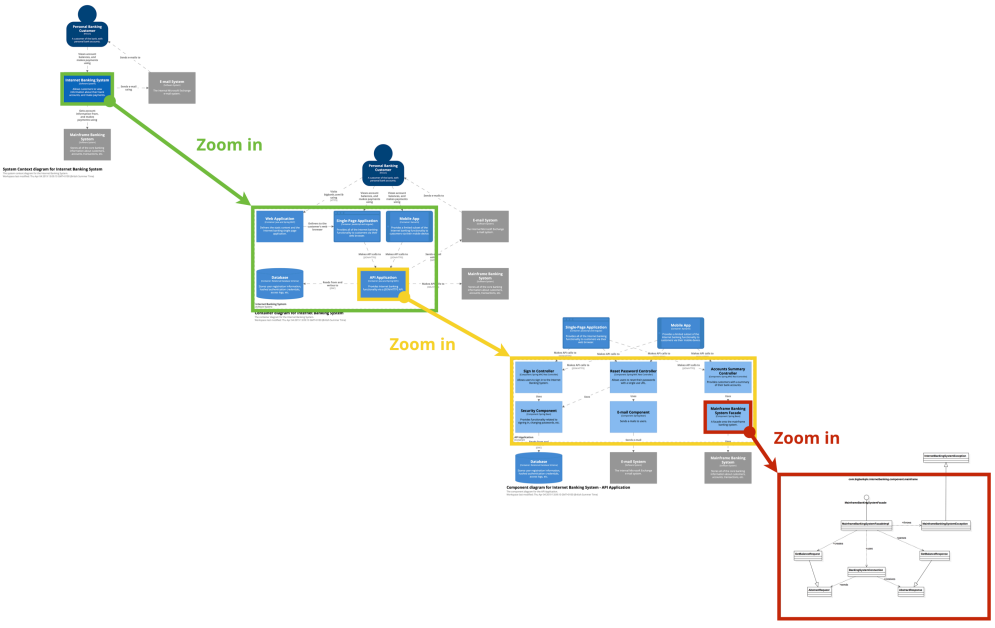
As an example, one organisation I worked with builds software in C to run on mobile devices. Instead of "container" and "component", they use the terms "component" and "module" respectively. In this case, a "component" refers to an executable built in C, which in turn is made up of a number of modules. Although the terminology is different, we still have a hierarchical structure that can be used to describe a software system at a number of different levels of abstraction.

[2]Many software architects will often try to decouple their code from that provided by platforms, frameworks and libraries so that decisions can deferred and technologies changed if necessary.

3. The C4 model

With a shared vocabulary in mind, we can now move on to draw some diagrams at varying levels of abstraction to visualise the static structure of a software system. I call this the "C4 model"; (System) Context, Containers, Components and Code.

1. **System Context**: A System Context diagram provides a starting point, showing how the software system in scope fits into the world around it.
2. **Containers**: A Container diagram zooms into the software system in scope, showing the high-level technical building blocks (containers) and how they interact.
3. **Components**: A Component diagram zooms into an individual container, showing the components inside it.
4. **Code**: A code (e.g. UML class) diagram can be used to zoom into an individual component, showing how that component is implemented.

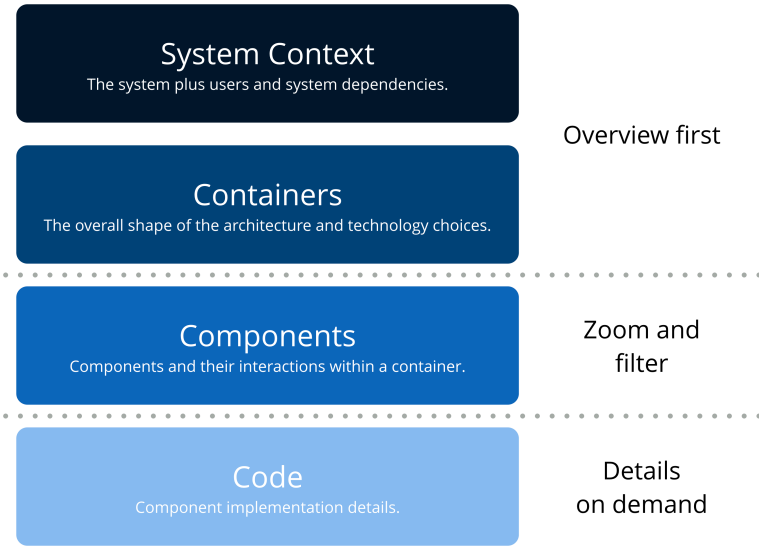


A summary of the C4 model

3.1 Hierarchical maps of your code

You can think of the C4 diagrams as being a set of maps for a software system, which provides you with the ability to zoom in and out at varying levels of detail. Shneiderman's mantra is a simple concept for understanding and visualising large quantities of data, but it fits really nicely with the C4 model because it's hierarchical.

> Overview first, zoom and filter, then details-on-demand



Shneiderman's mantra

System Context and Containers: Overview first

My starting point for understanding any software system is to draw a system context diagram. This helps me to understand the scope of the system, who is using it and what the key system dependencies are. It's usually quick to draw and quick to understand.

Next I'll open up the system and draw a diagram showing the containers (web applications, mobile apps, standalone applications, databases, file systems, message buses, etc) that make up the system. This shows the overall shape of the software system, how responsibilities have been distributed and the key technology choices that have been made.

Components: Zoom and filter

As developers, we often need more detail, so I'll then zoom into each (interesting) container in turn and show the "components" inside it. This is where I show how each application has been decomposed into components, along with a brief note about key responsibilities and technology choices of those components. Hand-drawing the diagrams can become tedious, which is why you should ideally look at tooling to help automate it instead.

Code: Details on demand

I might optionally progress deeper into the hierarchy to show the code-level elements (e.g. classes, interfaces, objects, functions, etc) that make up a particular component. Ultimately though, this detail resides in the code and, as software developers, we can get that on demand via our IDEs.

Different diagrams, different stories

Next time you're asked to create some software architecture diagrams (whether that's to understand an existing system, present a system overview, or do some software archaeology), my advice is to keep Shneiderman's mantra in mind. Start at the top and work into the detail, creating a story that gets deeper into the detail as it progresses. The different levels of diagrams allow you to tell different stories to different audiences; some of who will be technical, some not.

As a quick note, the C4 model is *not a description of a design process*, it's just a collection of diagrams that you can use to describe the static structure of a software system. That said, and we'll cover this later, while the C4 model describes diagrams covering four levels of abstraction, you don't necessarily need to create every diagram at every level. My recommendation is that all teams create System Context and Container diagrams, and really think about whether Component and Code diagrams provide enough benefit considering the cost of creating and keeping them up to date.

4. Level 1: System Context diagram

A System 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.

4.1 Intent

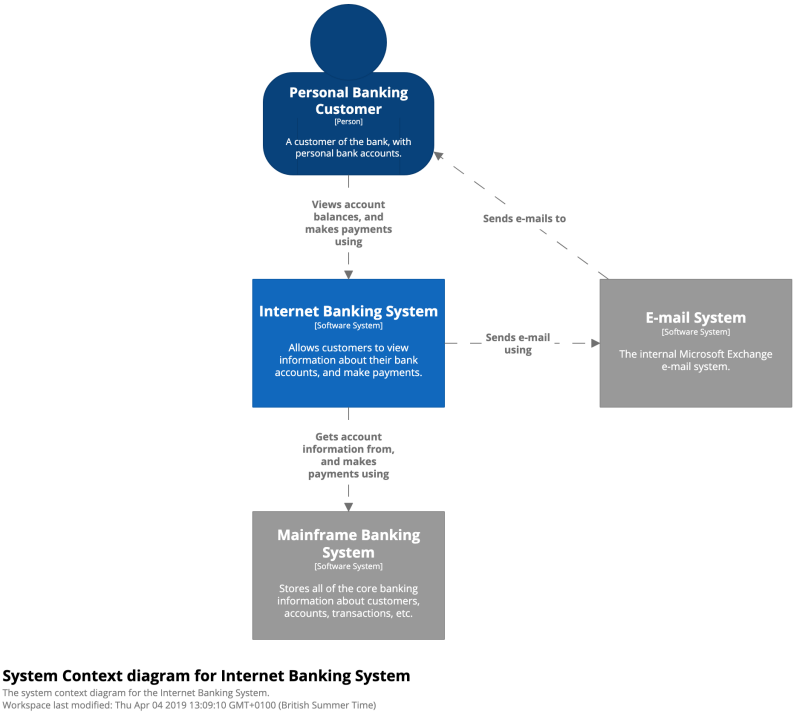
A System Context diagram helps you to answer the following questions.

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 environment?

4.2 Structure

Draw a block diagram showing your software system as a box in the centre, surrounded by its users and the other software systems that it interacts with. Detail isn't important here as this is your zoomed-out view showing a big picture of the software system and the immediate world around it. The focus should be on people (actors, roles, personas, etc) and software systems rather than technologies, protocols and other low-level details. It's the sort of diagram that you could show to non-technical people.

Let's look at an example. This is a System Context diagram for a fictional Internet Banking System. It shows the people who use it, and the other software systems that the Internet Banking System has a relationship with.



A System Context diagram for the Internet Banking System

In summary, Personal Customers of the bank use the Internet Banking System to view information about their bank accounts, and to make payments. The Internet Banking System itself uses the bank's existing Mainframe Banking System to do this, and uses the bank's existing E-mail System to send e-mails to customers.

4.3 Elements

A System Context diagram usually includes two types of elements; people and software systems.

People

These are the people who use your software system. Whether you model them as individual people, users, roles, actors or personas is your choice. Typically I'll capture the following information about people:

- **Name**: The name of the person, user, role, actor or persona.
- **Description**: A short description of the person, their role, responsibilities, etc.

Software systems

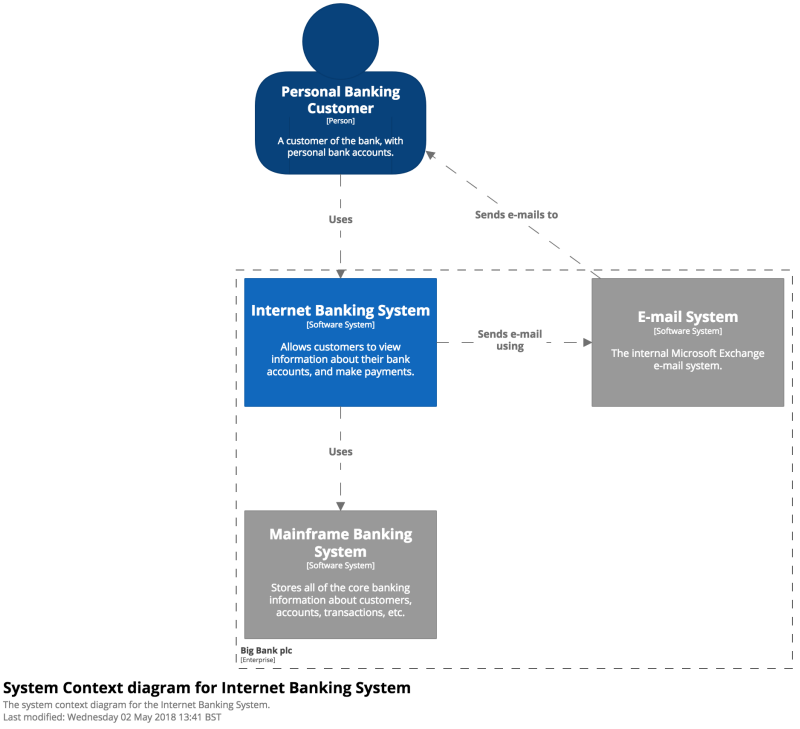
These are the other software systems that your software system interacts with. Typically these software systems sit outside the scope or boundary of your own software system, and you don't have responsibility or ownership of them. The Internet Banking example is very clear in this respect. As a team building the Internet Banking System, we don't own or have any responsibility over the bank's existing Mainframe Banking System, or the E-mail System. For this reason, they are included on the diagram to illustrate that they are dependencies of the Internet Banking System, and not something we are building ourselves.

Again, I'll capture the following information about each software system:

- **Name**: The name of the software system.
- **Description**: A short description of the software system, its responsibilities, etc.

Enterprise boundary

Optionally, I may want to capture some information about the location of people and/or software systems relative to my point of reference. If I'm building a software system inside an organisational/enterprise boundary, that software system may interact with people and software systems outside that boundary. In this slightly modified example, the dashed line represents the boundary of the bank, and is used to illustrate what's inside vs what's outside of the bank.



A System Context diagram for the Internet Banking System, additionally showing the enterprise boundary

4.4 Interactions

Try to annotate every interaction between elements on the diagram with some information about the purpose of that interaction. This avoids creating a diagram where a collection of boxes are somehow connected via a set of ambiguous lines. Again, try to keep this relatively high-level, and don't feel that you need to include lots of technical details (e.g. protocols, data formats, etc).

4.5 Motivation

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

- It makes the context and scope of the software system explicit so that there are no assumptions.

- It shows what is being added (from a high-level) to an existing 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 System Context diagram doesn't show much detail but it does help to set the scene, and is a starting point for other diagrams. I will often draw this diagram during a requirements gathering workshop, to ensure that everybody understands the scope of what we've been tasked to build. It can be a great requirements gathering and analysis tool.

4.6 Audience

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

4.7 Required or optional?

All software systems should have a System Context diagram.

5. Level 2: Container diagram

Once you understand how your software system fits in to the overall environment with a System Context diagram, a useful next step is to illustrate the high-level technology choices with a Container diagram.

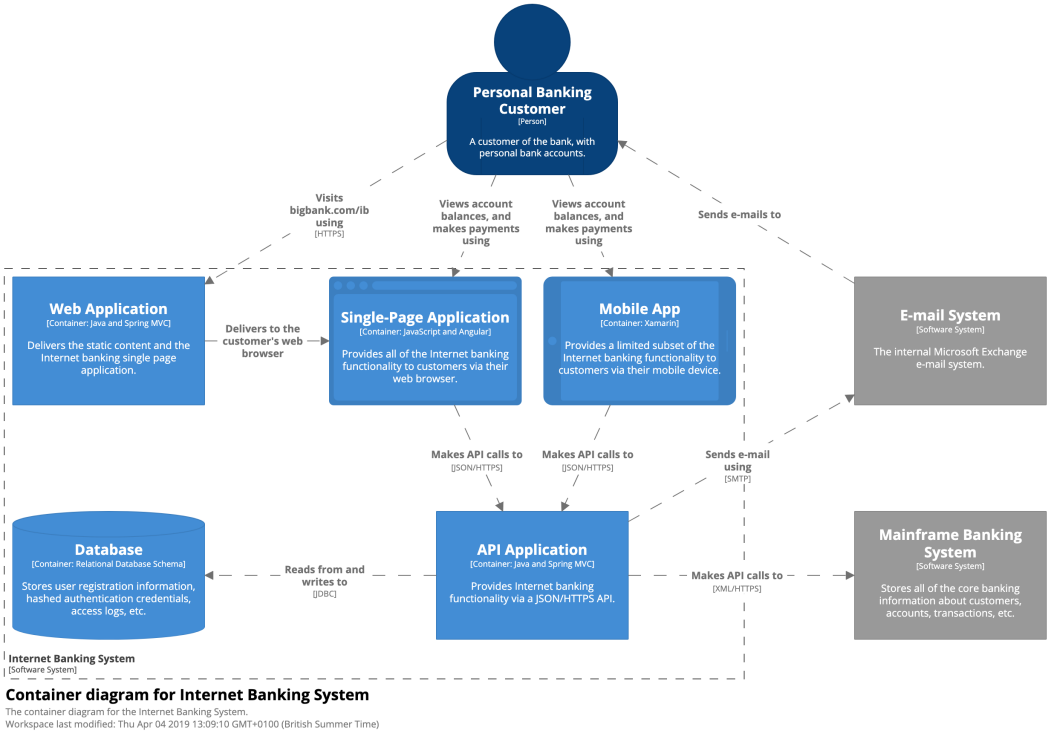
5.1 Intent

The Container diagram shows the high-level shape of the software architecture and how responsibilities are distributed across it. It also shows the major technology choices, how they are used, and how containers communicate with each other. It's a high-level *technology focussed* diagram that is useful for software developers and support/operations staff alike. A container diagram helps you answer the following questions:

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

5.2 Structure

Draw a block diagram showing the high-level technical elements (containers) that your software system consists of. This is an example Container diagram for the fictional Internet Banking System.



A Container diagram for the Internet Banking System

It shows that the Internet Banking System (the dashed box) is made up of five containers: a server-side Web Application, a Single-Page Application, a Mobile App, a server-side API Application, and a Database. The Web Application is a Java/Spring MVC web application that serves static content (HTML, CSS and JavaScript), including the content that makes up the Single-Page Application. The Single-Page Application is an Angular application that runs in the customer's web browser, providing all of the Internet banking features. Alternatively, customers can use the cross-platform Xamarin Mobile App, to access a subset of the Internet banking functionality.

Both the Single-Page Application and Mobile App use a JSON/HTTPS API, which is provided by another Java/Spring MVC application running on the server. The API Application gets user information from the Database (a relational database schema). The API Application also communicates with the existing Mainframe Banking System, using a proprietary XML/HTTPS interface, to get information about bank accounts or make transactions. The API Application also uses the existing E-mail System if it needs to send e-mails to customers.

It's worth pointing out that this diagram says nothing about the number of physical

instances of each container. For example, the API Application could be running on a farm of Apache Tomcat servers, with the relational database running on an Oracle cluster, but this diagram doesn't show that level of information. Instead, I show physical instances, failover, clustering, etc on a separate Deployment diagram that illustrates the mapping of containers onto infrastructure.

How much detail?

If you're drawing a Container diagram during an up-front design exercise, you might not have some of the technical details to hand. That's fine, just add what you know. If, on the other hand, you're drawing a diagram to document an existing system, it's more likely that you'll be able to add some of the finer details; such as protocols, port numbers, etc. The choice is yours, add as much detail as you feel is necessary.

5.3 Elements

A container diagram usually includes three types of elements; people, software systems and containers.

Containers

A container typically represents an application or data store. I'll capture the following information about each container:

- **Name**: The name of the container (e.g. "Internet-facing web server", "Database", etc).
- **Technology**: The implementation technology (e.g. Spring MVC application on Apache Tomcat 8, ASP.NET web application on Microsoft IIS 8.5, etc).
- **Description**: A short descriptive statement, or perhaps a list of the container's key responsibilities or entities/tables/files/etc that are being stored.

File systems and log files vs data storage

If I'm describing a software system where an application stores business critical data on a file system, I will include a "File System" container on the diagram, to make this explicit. In contrast, although many types of containers will write log files to a file system (and this is undoubtedly important), I typically omit this detail for brevity.

Data storage as a service

A frequently asked question is whether services like Amazon S3, Amazon RDS, or Azure SQL Database should be shown on a container diagram. After all, these are external services that we don't own or run ourselves.

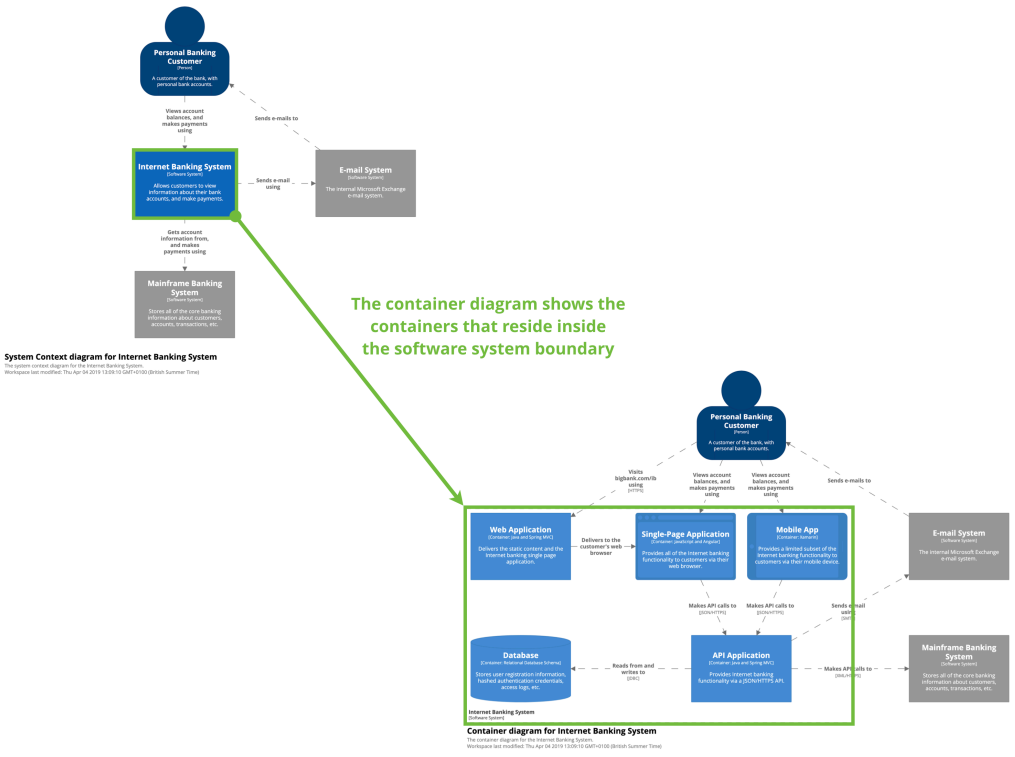
If you're building a software system that is using Amazon S3 for storing data, it's true that you don't run S3 yourself, but you do have ownership and responsbility for the buckets you are using. Similarly with Amazon RDS, you have (more or less) complete control over any database schemas that you create. For this reason, I *will* usually show such containers on a container diagram. They are an integral part of your software architecture, although they are hosted elsewhere.

People and software systems

If you look back at the example Container diagram, you'll notice that it has the same people and software systems as the example System Context diagram. I usually include the same collection of people (users, actors, personas, etc) and software systems because it helps to tell the overall story of how the system works and it provides continuity between the two diagrams. In effect, a Container diagram is very similar to the System Context diagram, where you're zooming in to see what's inside the software system in scope.

Software system boundary

With this in mind, I recommend drawing a bounding box around the containers on your diagram to explicitly show the system boundary. This system boundary should correspond with the single box that appears on the System Context diagram.



The Container diagram shows a zoom in of the software system boundary

5.4 Interactions

Typically, communication between containers is *out-of-process* (or *inter-process*). It's very useful to explicitly identify this and summarise how these interactions will work. As with any diagram, I recommend annotating all interactions rather than having a diagram with a collection of boxes and ambiguous unlabelled lines connecting everything together. Useful information to annotate the interactions with includes:

- The purpose of the interaction (e.g. "reads/writes data from", "sends reports to", etc).
- The communication mechanism (e.g. Web Services, REST, Web API, Java Remote Method Invocation, Windows Communication Foundation, Java Message Service).
- The 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).

5.5 Motivation

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

- It makes the high-level technology choices explicit.
- It shows the relationships between containers, and how those containers communicate.

During my software architecture sketching workshops, I often see groups producing a high-level context diagram that shows the software system in question as a single black box, and a second diagram that shows *all* of the components that reside within the entirety of the software system. This second diagram is usually cluttered and confusing because it presents too much information. It's also usually not clear how the components are grouped together from a deployment perspective. The Container diagram provides a nice intermediate diagram between the two contrasting levels of abstraction. It also logically leads you to level 3 (a Component diagram), where you open up a single container to show only the components that reside within that container.

5.6 Audience

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

5.7 Required or optional?

All software systems should have a Container diagram.

6. Level 3: Component diagram

Following on from a Container diagram showing the high-level technology elements, the next step is to zoom in and decompose each container further to show the components inside it.

6.1 Intent

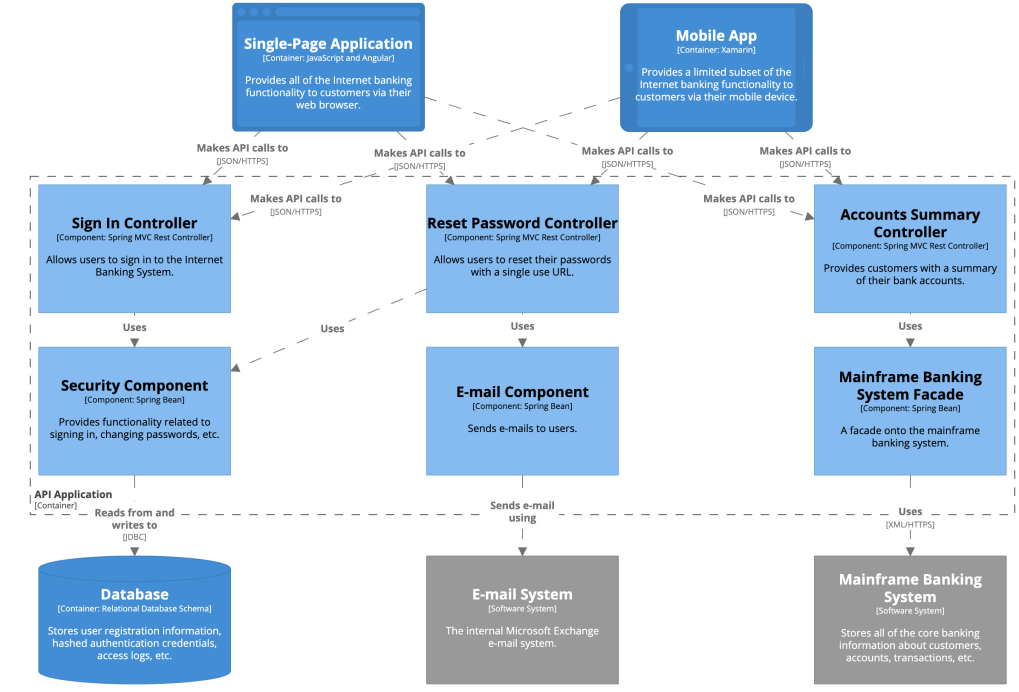
A Component diagram helps you answer the following questions.

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

6.2 Structure

Whenever people are asked to draw "architecture diagrams", they usually end up drawing diagrams showing the components that make up their software system. This is basically what a Component diagram shows, except we only want to see the components that reside within a *single* container at a time.

This is an example Component diagram for the fictional Internet Banking System, showing some (rather than all) of the components within the API Application.



Component diagram for Internet Banking System - API Application

The component diagram for the API Application.
Workspace last modified: Thu Apr 04 2019 13:09:10 GMT+0100 (British Summer Time)

Here, there are three Spring MVC Rest Controllers providing access points for the JSON/HTTPS API, with each controller subsequently using other components to access data from the Database and Mainframe Banking System, or send e-mails. In summary, this diagram shows how the API Application is divided into components, what each of those components are, their responsibilities and the technology/implementation details.

How much detail?

If you're drawing a Component diagram during an up-front design exercise, you might not have some of the technical details to hand. Once again, don't worry, add what you know. If, on the other hand, you're drawing a diagram to document an existing system, you'll have those finer details to hand; such as the frameworks you are using to help implement a component. As with many other aspects of the diagrams, the choice of how much detail to include is yours.

6.3 Elements

A Component diagram can include four types of elements; people, software systems, containers and components.

Components

As we saw when creating our shared vocabulary, components are the coarse-grained building blocks of your software system that live inside of a container. I'll capture the following information for each component:

- **Name**: The name of the component.
- **Technology**: The implementation technology for the component (e.g. Plain Old [Java|C#|Ruby|etc] Object, Enterprise JavaBean, Windows Communication Foundation service, etc).
- **Description**: A short description of the component, usually a brief sentence describing the component's responsibilities.

You can think about and identify components regardless of how the code is packaged and the architectural style in use. With this in mind, your Component diagram should reflect the architectural style in use; whether that's a layered architecture, hexagonal architecture or something else entirely.

Infrastructure components and cross-cutting concerns

Infrastructure components are important parts of most software systems, yet you may or may not want to include them on your Component diagram. For example, if you have a logging component, it's likely to be used by the majority of other components within the container. Drawing this component and all of the interactions can result in a very cluttered diagram. You have a number of options to deal with this:

1. Don't include the logging component if it doesn't add much value in helping you tell the story.
2. Write a note on the diagram that says something like, "All components shown here write logs via a Logging Component".
3. Include the logging component on the diagram, but don't show how it's connected to anything else. Instead, annotate elements that use the logging component (with a symbol/icon), or use a colour coding, which you can then describe in a diagram key/legend (e.g. "the asterisk denotes a relationship with the Logging Component").

Shared components and libraries

I'm often asked whether a Component diagram should include shared components (for example, from a shared or static library) and how such components should be represented. If including the shared component helps tell the story, then it should certainly be included. If you want to illustrate that a particular component is a shared component or sits within a specific library/module, again, you can use a notation (e.g. a symbol/icon or colour coding) to represent this fact.

Multiple component implementations

In many cases, there will only ever be a single implementation of a component. However, there may be times when you'll have a single component interface and a number of implementations. This is particularly true of software *products* (rather than bespoke software), where the collection of active components will be selected through configuration when the product is installed. Common examples include different implementations of data storage components (e.g. one for Microsoft SQL Server, one for MySQL, etc), different logging components (e.g. local disk or a message queue), or pluggable authentication components for integration with different identity providers.

Having multiple component implementations raises the question of how this should be illustrated on a Component diagram. If we take an example of a logging component with multiple implementations (e.g. local disk vs a message queue), one of which is chosen at deployment time via configuration, there are a few approaches to diagramming this:

1. The first approach is to omit the fact that there are multiple component implementations and draw a Component diagram as if there was only a single implementation. Here, I would draw the logging component and describe it's responsibilities, which in this case might be to "log errors and other system events".
2. If I wanted to include the fact that the component implementation can vary, I might add some additional text to the logging component box on the diagram to say something like "Log entries are stored using local disk or sent to a message queue, depending upon the implementation chosen by configuration at deployment time". You could also achieve the same result by highlighting the logging component using a symbol/icon or colour coding. The diagram key would then explain this.
3. The other approach is to have one Component diagram per implementation option. This works best if you only have a small number of component implementation combinations (e.g. you only have one or two components where the implementation can be swapped in at runtime). Having separate diagrams for specific component

implementations can also be useful if those component implementations themselves introduce other components, which wouldn't be seen on a diagram otherwise.

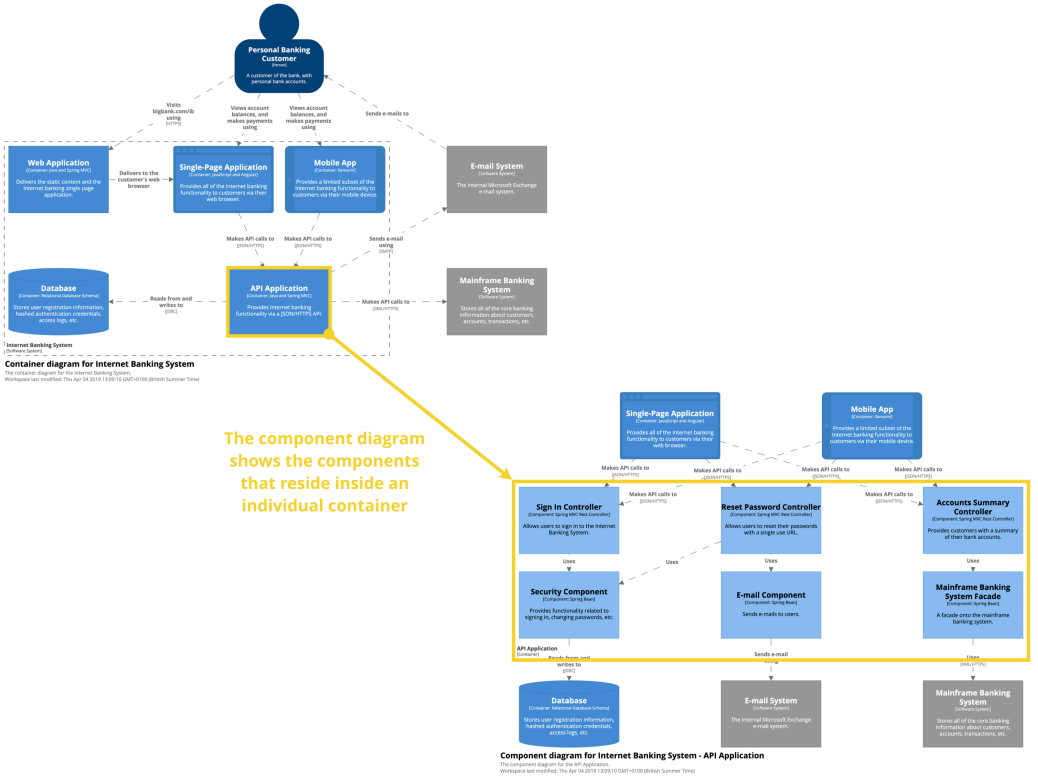
As with many of the things discussed in this book, there is no "right" answer, and it really depends on what story you need to tell.

People, software systems and containers

As with the Container diagram, it can be useful to include people, software systems and other containers to help put some context around the container you've zoomed in upon. The Internet Banking System Component diagram for the API Application includes the other containers and software systems that it interacts with.

Container boundary

If you're going to include people, software systems and containers on a Component diagram (and, again, you should), I recommend drawing a bounding box to explicitly show the boundary of the container.



The Component diagram shows a zoom in of a single container

6.4 Interactions

To reiterate the same advice given for other diagram types, it's useful to annotate the interactions between components rather than 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 data using", "delegates to", etc).
- Communication style (e.g. synchronous, asynchronous, etc).

6.5 Motivation

A Component diagram shows the components that reside inside an individual container. This is useful because:

- It shows the high-level decomposition of a container into components, each with distinct responsibilities.
- It shows where there are relationships and dependencies between components.
- It provides a high-level summary of the implementation details, including any frameworks or libraries being used.
- If the components shown on the diagram can be explicitly mapped to the code, you have a good way to really understand the structure of a codebase.

6.6 Audience

Technical people within the software development team.

6.7 Required or optional?

I don't typically draw a Component diagram for data storage containers (e.g. databases, file systems, content stores, etc), or for those containers that are very simple in nature (e.g. microservices). For monolithic applications, you might consider creating a Component diagram, especially during an up front design process. In real world use, many teams find this an optional level of detail for long-lived documentation, because of the potentially high degree of effort involved in creating the diagrams, and keeping them up to date. Also, once you have more than a handful of components, you'll find that the diagram starts to get very cluttered very quickly, which reduces the usefulness of the diagram. We'll look at some strategies for dealing with this later in the book.

7. Level 4: Code-level diagrams

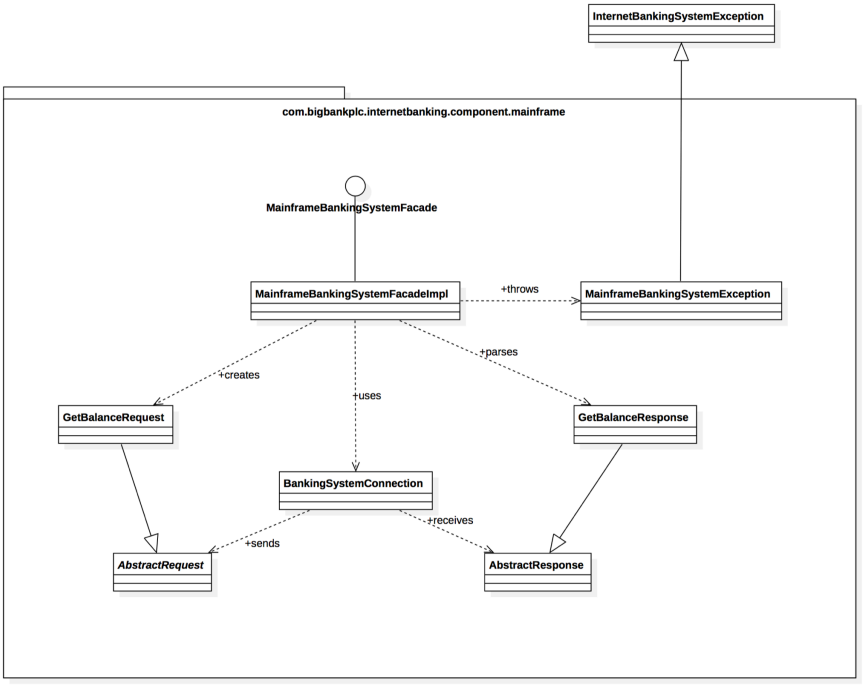
The final level of detail is one or more code-level diagrams showing the implementation details of an individual component.

7.1 Intent

The intent of a code-level diagram is to illustrate the structure of the code and, in this case, how a component is implemented.

7.2 Structure

If you're using an object-oriented programming language, probably the best way to do this is to use a UML class diagram, either by generating it automatically from the code or by drawing it freehand. Let's look at an example by zooming in on the "Mainframe Banking System Facade" component from the Internet Banking System "API Application" container.



A UML class diagram

How much detail?

The danger with UML class diagrams is that it's very easy to include a considerable amount of detail, and this is especially true if you are auto-generating diagrams from code. Although it's tempting to include every field/property/attribute and method, I would resist this temptation and only include as much information as you need to tell the story that you want to tell. Typically, I will only include the attributes and methods that are relevant to the narrative I want to create.

UML class diagrams have often been used to describe an entire application, but this just results in a huge mess of overlapping boxes and lines, regardless of how well-structured the code is. The key to using class diagrams is to limit their scope. In this case, scope is limited to the internals of a component.

7.3 Motivation

Having this final level of abstraction provides a way to map the high-level, coarse-grained components into real-world code elements. It helps to bridge what are sometimes seen as two very different worlds; the software architecture and the code.

7.4 Audience

Technical people within the software development team, specifically software developers.

7.5 Required or optional?

Since this level of detail lives in the code, software developers can get this detail *on demand* from the code itself. Therefore, this is definitely an optional level of detail, and generally not recommended. I don't typically draw class diagrams for anything but the most complex of components, or as a template when I want to describe a pattern that is used across a codebase.

8. Notation

Now that we've created a shared vocabulary and looked at how to create diagrams at a number of different levels of abstraction, let's look at notation.

8.1 Titles

The first thing that can really help people to understand a diagram is including a title. If you're using a notation like UML, the diagram elements will probably provide a clue as to what the context of the diagram is. That doesn't really help if you have a collection of diagrams that are all just boxes and lines though.

Include a short and meaningful title on every diagram, even if you are using UML. And if the diagrams should be read in a specific order, make sure this is clear in the title, perhaps by the use of a numbering scheme. To avoid any confusion, I also recommend including the *diagram type* in the title. For example, something like:

- **System Context diagram** for Financial Risk System
- [**System Context**] Financial Risk System

8.2 Keys and legends

One of the advantages of using a notation like UML is that it provides a standardised set of diagram elements for each type of diagram. In theory, if somebody is familiar with these elements, they should be able to understand your diagram. In the real world this isn't always the case, but this certainly *isn't* the case with "boxes and lines" diagrams where the people drawing the diagrams are inventing the notation as they go along.

There's nothing wrong with inventing your own notation, but make sure that you give everybody an equal chance of understanding it by including a key/legend somewhere on or nearby the diagram. Here are the things that you might want to include explanations of:

- Shapes

- Line styles
- Colours
- Borders
- Acronyms

You can often make assumptions and interpret the use of diagram elements without a key. For example, I've heard people say the following sort of thing during my software architecture sketching workshops:

> "the grey boxes seem to be the existing systems and the red boxes are the new systems"

Even if the notation seems obvious to you, I recommend playing it safe and adding a key/legend. Even the seemingly obvious can be misinterpreted by people with different backgrounds and experience.

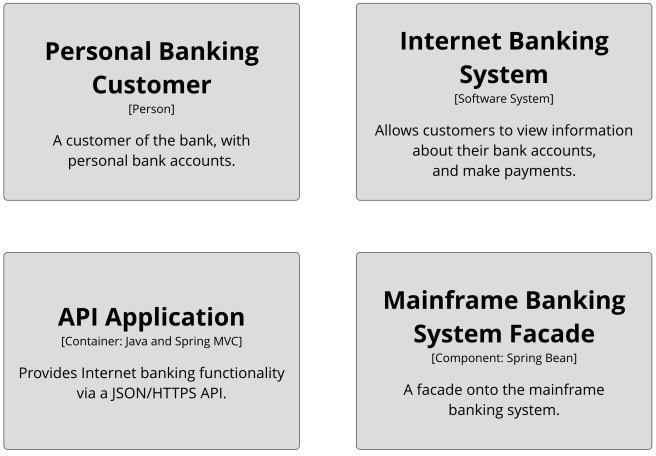
8.3 Elements

Most "boxes and lines" diagrams that I've seen aren't just boxes and lines, with teams using a variety of shapes to represent elements within their software architecture. For example, you'll often see cylinders on a diagram and many people will interpret them to be a database of some description. But this isn't always the case.

My recommendation is that you start with a pure "boxes and lines" diagram, using a very utilitarian notation and then add shapes, colour and borders to add additional information or make the diagram more aesthetically pleasing. In order to show some example software architecture diagrams in this book, I've needed to create my own notation, which includes the following information for each element:

- **Person**: Name and description.
- **Software system**: Name and description.
- **Container**: Name, technology and description.
- **Component**: Name, technology and description.

As you will have seen from the example diagrams, each of the elements is drawn as follows:



Examples of the elements I use on my diagrams

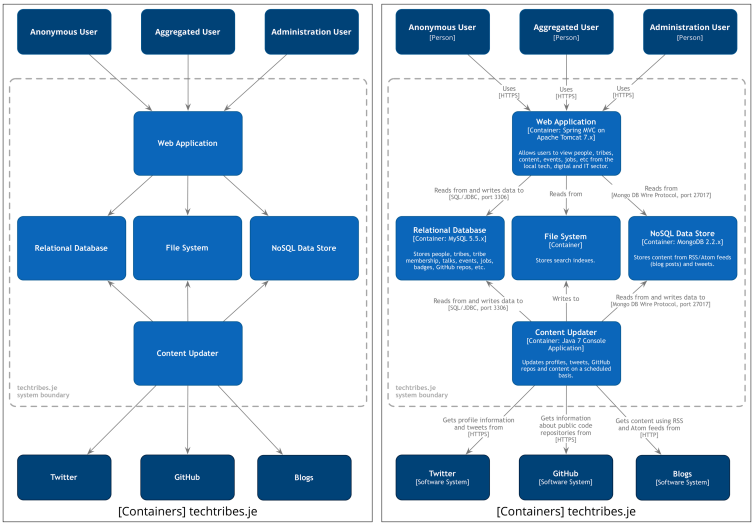
This is the notation that I've gradually settled on over the years. It's easy to draw on a whiteboard or in tooling, plus it works well on sticky notes and index cards. Do feel free to create your own notation though.

Description/responsibilities

If naming *is* one of the hardest things in software development, do resist the temptation to have a diagram full of boxes that only contain names. If you look at most software architecture diagrams, this is exactly what they are - a collection of named boxes. As with many other things, naming is always open to interpretation and ambiguity.

A really simple, yet effective, way to add an additional layer of information to, and remove ambiguity from, an architecture diagram, is to annotate diagram elements with a short descriptive statement of what their responsibilities are. A bulleted list (7 +/- 2 items) or a short sentence works well.

Provided it's kept short (and using a smaller font for this information can help too), *adding more text* onto diagrams can help provide a really useful "at a glance" view of what the software system does and how it's been structured. Take a look at the following diagrams - which do you prefer?



Adding additional descriptive text to diagram elements can remove ambiguity

Colour

Software architecture diagrams don't have to be black, white and various shades of grey. The use of colour is a great way to supplement a diagram that already makes sense. For example, colour can be used to provide differentiation between diagram elements or to ensure that emphasis is/isn't placed on them and you could colour-code elements according to:

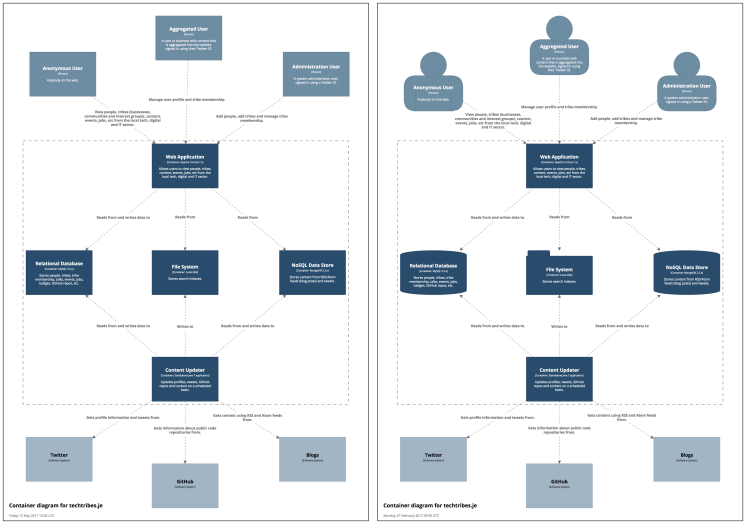
- Existing vs new.
- Off-the-shelf product vs custom build.
- Technology type or platform.
- Risk profile (e.g. risk to build; high-medium-low risk; red-amber-green).
- Size and/or complexity.
- Ownership (i.e. elements you own vs elements somebody else owns).
- Internal vs external (i.e. elements within your organisation vs those outside).
- Elements you're modifying or removing in the next release/sprint/phase vs those that will remain untouched.

If you're going to use colour, and I recommend that you should, make sure that it's obvious what your colour coding scheme is by including a reference to those colours in

the key/legend. Colour can make a world of difference. Just be aware of anybody on your team who suffers from colour blindness and make sure that your colour scheme works if the diagram will be printed on black and white printers.

Shapes

Using different shapes can be a great way to add an additional level of information, supplement, enhance or add emphasis to specific elements. Using shapes can also make a diagram look more aesthetically pleasing. Although this sounds shallow, people are more likely to look at diagrams that are easy on the eye. Consider the two versions of the techtribes.je container diagram that follow. They both show exactly the same information for every element (name, element type, description and technology, if applicable), but one uses only boxes whereas the other uses some shapes.



The diagram that uses shapes is certainly easier to read from a distance, with the shapes helping to provide a quicker "at a glance" view. But the shapes are simply enhancing the diagram; they don't really add any information that isn't already present in the text that resides inside the elements.

The Unified Modeling Language has numerous diagram types and an even higher number of element types that can appear on those diagrams. Anecdotally, interpreting the notation is one of the major reasons cited for not adopting UML, and many software developers have told me that there are simply too many diagram types and nuances in the notation.

In contrast, I like to use a very simple notation consisting of a small number of shapes. Over the numerous years that I've been running my software architecture sketching workshop, I've observed that most developers only typically use the following shapes on their diagrams:

- Boxes (squares, rectangles, rounded boxes, circles, ellipses and hexagons).
- People shapes (the "stick man"[1], "head and shoulders", etc).
- Cylinders (e.g. to represent databases).
- Folder shapes (e.g. to represent file systems).

My advice is to keep diagrams as simple as possible, but do feel free to use whatever shapes you like. Again, don't forget to include the shapes on the key/legend.

Borders

Like shapes, 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 on the key/legend.

Size

A quick note about the size of elements. Be careful about how you size elements on diagrams. I've witnessed a tendency for people to make assumptions about elements that are sized differently from others. Larger elements are often assumed to be larger, more complex or more significant; while smaller elements tend to take on the inverse characteristics. Unless you are specifically making a statement about size, complexity or significance, I recommend drawing all elements approximately the same size.

8.4 Lines

Lines are an important part of most architecture diagrams, acting as the glue that holds all of the boxes together. The big problem with lines is exactly that though - they tend to be thought of as the unimportant things that hold the other, more significant, elements of the diagram together. As a result, lines often don't receive much focus.

[1]You may have seen diagrams that have represented software systems as actors, using the traditional "stick man" icon. This comes from UML where "an actor specifies a role played by a user or any other system that interacts with the subject". I've done this myself in the past but I shy away from doing it now as it tends to cause too much confusion. After all, why would you want to visually represent a software system using a person shape?

Directionality

Even though most relationships between elements are bi-directional (e.g. a request followed by a response, or data flow in both directions), I usually choose the most significant direction and represent that as a uni-directional line. This raises the question, "which way do you point the arrows?".

In the techtribes.je example, on the context diagram, my users use techtribes.je, which in turn uses a number of other software systems. If we look at the line between techtribes.je and Twitter, I could have drawn it in a number of ways, based upon whether I wanted to show a dependency relationship or data flow:

- [techtribes.je] – gets profile information and tweets from -> [Twitter]
- [techtribes.je] – receives profile information and tweets from -> [Twitter]
- [Twitter] – sends profile information and tweets to -> [techtribes.je]

In this example, there is no "right answer". Personally, I tend to prefer showing dependency relationships and drawing a line from the initiator to the receiver. Other options are equally valid though and the style you adopt is your decision. My advice is to be consistent where possible and ensure that the descriptive text you use to annotate the line matches what you're trying to describe.

Description

As for the wording of the descriptions on lines, I will try to use wording that helps explain the direction of the arrow, often by using or ending the description with a *preposition*. A preposition is a type of word that expresses something about the relationship between elements of a sentence (e.g. to, from, with, on, in). For example, I will write:

- [techtribes.je] – gets profile information and tweets **from** -> [Twitter]
- [Web Application] – reads **from** and writes **to** -> [Database]

Rather than:

- [techtribes.je] – gets profile information and tweets -> [Twitter]
- [Web Application] – reads and writes -> [Database]

To check whether the description of a relationship makes sense, simply say the resulting sentence out loud. For example, "techtribes.je gets profile information and tweets **from** Twitter". If the sentence doesn't make sense, you likely need to tweak the wording.

Line style

As with elements, you can use different line styles and colour to add an additional level of information to your diagram. For example, perhaps synchronous interactions are illustrated using solid lines, whereas asynchronous interactions are illustrated using dashed lines. And perhaps HTTPS connections are coloured green, while HTTP connections are coloured amber.

Once again, ensure that any styling supplements the existing information wherever possible and that the styles you use are described on the key/legend.

One line vs many

It's common to have multiple relationships between diagram elements. In the techtribes.je example, techtribes.je gets both profile information and tweets from Twitter. This relationship is drawn as a single line, but there are really two APIs that are being used here. If you want to be more precise, feel free to use two separate lines to illustrate the two different relationships:

- [techtribes.je] – gets profile information from -> [Twitter]
- [techtribes.je] – gets tweets from -> [Twitter]

8.5 Layout

Using electronic drawing tools makes positioning 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 larger and 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 awkward if you're having to constantly draw, erase and redraw elements of your diagrams when you run out of space.

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

Examples of where sticky notes and index cards have been used instead of 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 and index cards *can* be a great way to get started with software architecture diagrams, but I find that the resulting diagrams can look cluttered. Oh, and sticky notes often don't stick well to whiteboards, so have some blu-tack handy!

8.6 Orientation

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

Most of the architecture diagrams that I see have the users and web-tier at the top, but this isn't always the case. Sometimes those same diagrams will be presented upside-down or back-to-front, perhaps illustrating the author's (potentially subconscious) view that the database is the centre of their universe. Although there is no "correct" orientation, drawing diagrams "upside-down" from what we might consider the norm can either be confusing or used to great effect. The choice is yours.

I will recommend that you try to put the most important thing in the centre of your diagram

and work around it. Additionally, try to keep the placement of elements consistent between diagrams. As an example, all of the people in my techtribes.je diagrams are placed at the top, and my system dependencies are placed at the bottom.

8.7 Acronyms

You're likely to have a number of labels on your diagrams; including names of software systems, domain concepts and terminology, etc. Where possible, avoid using acronyms and if you do need to use acronyms for brevity, ensure that they are documented in a glossary or on the key/legend. While the regular team members might have an intimate understanding of common domain acronyms, people outside or new to the team 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/term or add the acronym to the key/legend.

8.8 Quality attributes

In some cases, adding information about quality attributes provides an extra degree of narrative about what the software system does and how it has been designed. The simplest way to do this is to add text to the diagram. For example, the description of diagram elements could be extended to include a note about the number of potential users, the expected number of concurrent users, etc. The lines between elements can also include additional information about data volumes being transferred, target message latencies, target request/response times, etc. Other cross-cutting concerns such as security are harder to show on a diagram and are usually better left to being described in lightweight supplementary documentation.

When it comes to illustrating solutions that address quality attributes, you could add some text to indicate that particular parts of the software architecture are replicated or clustered to address scalability and/or availability quality attributes, for example. My context, container and component diagrams typically don't show this though, because I'll save that type of information for a separate deployment diagram that shows how technology is mapped onto infrastructure.

My simple advice is that you shouldn't try to force everything onto a diagram. Lightweight supplementary documentation is sometimes a better approach to create a narrative that explains how quality attributes are addressed.

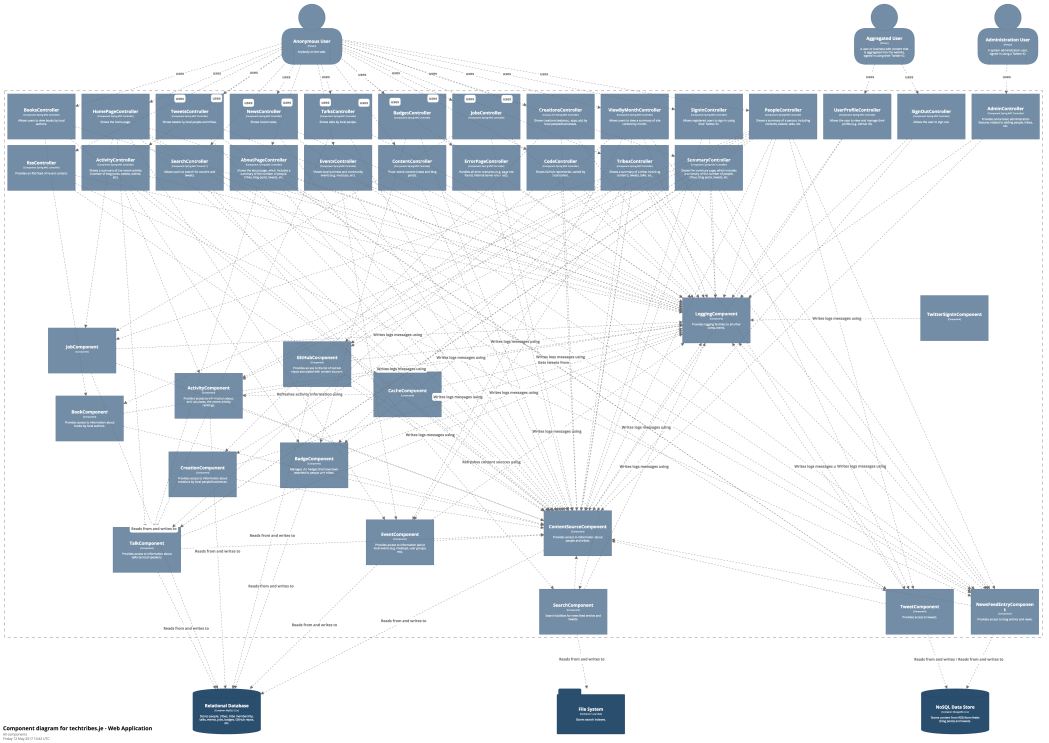
8.9 Diagram scope

The example diagrams I've presented in the book have all conveniently fitted on one page. Of course, in the real-world, you're likely to have larger and more complicated software systems. Simply by their very nature, component diagrams are prone to becoming large and complex. So what do you do when your diagram doesn't fit on one page?

A simple, yet naive, approach is to use a larger canvas. If you're working on A4 paper, grab some A3 paper or stick two sheets of A4 paper together. If you're working on A3 paper, try to find some flip chart paper. If you're working with an electronic drawing tool, simply increase the page size. The problem with increasing the page size is that it allows you to show more information, which leads to *more clutter*. I've seen gigantic diagrams on the walls of organisations that I've visited, some of which have been printed on special purpose plotter machines that accommodate very large paper sizes.

Even with a relatively small software system, it's tempting to try and include the entire story on a single diagram. For example, if you have a web application, it seems logical to create a single component diagram that shows all of the components that make up that web application. Unless your software system really is that small, you're likely to run out of room on the diagram canvas or find it difficult to discover a layout that isn't cluttered by a myriad of overlapping lines. Using a larger diagram canvas can sometimes help, but large diagrams are usually hard to interpret and comprehend because the cognitive load is too high. And if nobody understands the diagram, nobody is going to look at it.

As an example of this in action, let's look at what a component diagram for the techtribes.je web application would look like if you chose to include all of the components.



A component diagram for the techtribes.je web application

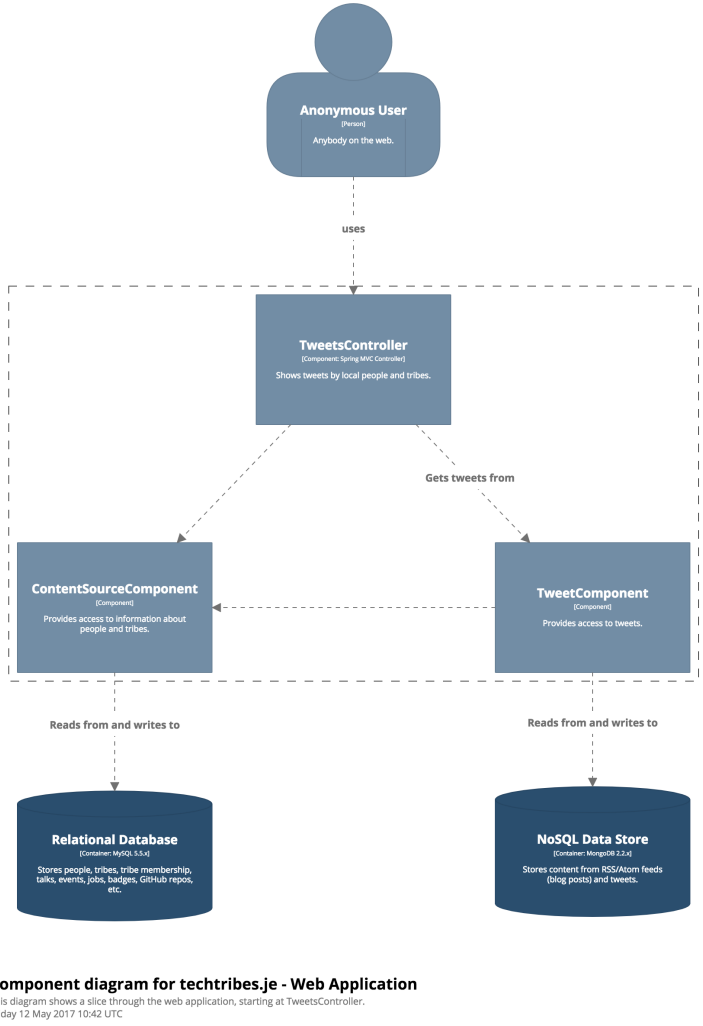
This component diagram has been automatically created using some tooling that identifies components and their dependencies from the code. It's comprehensive, but it's a mess! And it's difficult to determine whether this mess is caused by the architecture being a mess or the diagram showing too much information. If we look at this diagram a little more closely, we can see there are really three things that cause this diagram to be cluttered:

1. We're showing every web-MVC controller (the components at the top of the diagram), and therefore every dependency path through the web application.
2. The "ContentSourceComponent" is being used by a large number of other components (this may or may not be a good thing from an architectural perspective, of course).
3. The "LoggingComponent" is used by nearly every other component.

Removing the "LoggingComponent" will remove some of the clutter, but it doesn't really help that much. Increasing the page size won't help either. Instead, we need a different approach.

A better solution is to split that single complex diagram into a larger number of simpler diagrams, each with a specific focus around a business area, functional area, functional

grouping, bounded context, use case, user interaction, feature set, etc. For the techtribes.je web application component diagram, we could do this by creating a *single diagram per web-MVC controller*. For example, here's a component diagram that focusses on the "TweetsController" (the page on the website that shows recent tweets by local people and businesses).



A component diagram for the techtribes.je web application, focussed on the TweetController

The key with this approach is to ensure that each of the separate diagrams tells a different

part of the same overall story, at the *same level of abstraction*. In order to make it feasible to adopt this approach, you really need to use some tooling, but we'll cover that later.

8.10 Listen for questions

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

9. Diagrams must reflect reality

There seems to be a common misconception that "architecture diagrams" should present a high-level conceptual view of a software system, so it's not surprising that software developers often regard them as unnecessary.

9.1 The model-code gap

One of the biggest problems I see with software architecture diagrams is that they never quite match the code. Sometimes the diagrams are horribly out of date, perhaps because the overhead of maintaining them is too high, but at other times the abstractions being shown on the diagram don't actually reflect the code. And this is especially prevalent at the component level.

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

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

Instead, what we tend to do is look for related groups of classes and instead draw a diagram showing those. These related groups of classes are usually referred to as modules, components, services, layers, packages, namespaces, subsystems, etc. The same can be said when you're doing some up front design for a new software system. Although you could start by sketching out class diagrams, this is probably diving into the detail too quickly.

There are a number of benefits to thinking about a software system in terms of larger abstractions, but essentially it allows us to think and talk about the software as a small number of larger things rather than hundreds and thousands of small things. *Abstractions help us to reason about a big and/or complex software system.*

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

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

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

This is not a new problem. If you ask a software developer to draw a diagram to describe the software system they are working on, you'll likely get a high-level diagram with a small number of boxes. That diagram will be based on the developer's mental model of the software. If you reverse-engineer a diagram from the codebase though, you'll get a very different picture. It will be very low-level, precise and accurate because reverse-engineering tools typically show you a reflection of the structures in the code.

An architecturally-evident coding style

George's answer to the model-code gap is simple - we should use an "architecturally-evident coding style". In other words, we should make our code reflect our architectural ideas and intent. In concrete terms, this can be achieved by:

- **Naming conventions**: If you're implementing something that you think of as a component, ensure that this is apparent from the naming. For example, a class, interface, package, namespace, etc) could include the word "component".
- **Packaging conventions**: In addition, perhaps you group everything related to a single component into a single package, namespace, module, folder, etc.
- **Machine-readable metadata**: Alternatively, why not include machine-readable metadata in the code so that parts of it can be traced back to the architectural vision.

> In real terms, for example, you could use Java Annotations (e.g. `@Component`) or C# Attributes (e.g. `[Component]`) to signify classes as being architecturally important. These annotations or attributes could come from a framework that you're using, or you could create them yourself.

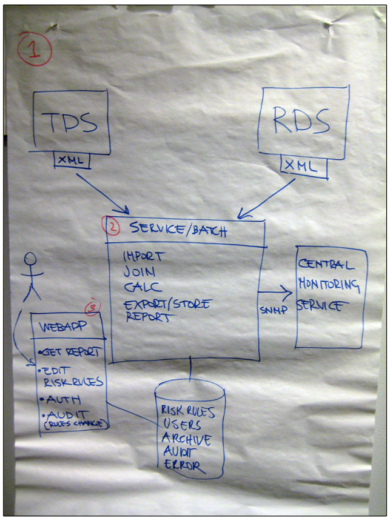
In simple terms, if you have a component diagram on the wall, each of the components should map to something real in the code. If there's a "Logging Component" box on the diagram, make sure there really is a "Logging Component" in the code. This is relatively easy to do but, in my experience, I rarely see teams doing this. Having a **simple and explicit mapping** from the architecture to the code can help tremendously when people are asked to comprehend a codebase, especially if it's new to them. In addition, there's clearly a relationship between the architecture of a software system and how that architecture is visualised as a collection of diagrams. The style of architecture you're using needs to be reflected on your software architecture diagrams; whether that's layers, ports and adapters, components, microservices or something else entirely.

9.2 Technology details on diagrams

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 software architecture diagrams omit any information about technology, instead focussing on illustrating the functional decomposition and major conceptual elements. Including technology choices (or options) on diagrams is another simple way to ensure that diagrams reflect reality, and prevents them looking like an ivory tower architecture where a bunch of conceptual components magically collaborate to form an end-to-end software system.

Drawing diagrams during the design process

Here's a photo of an architecture diagram produced during one of my software architecture workshops, and it's fairly typical of what I see because of the lack of any technology details.



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

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

Drawing diagrams retrospectively

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

- "the technology decisions will clutter the diagrams".
- "but everybody knows that we only use ASP.NET and Oracle databases".

Make technology choices explicit

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 software architecture

diagrams should not include technology. One of the reasons that software architecture has a bad reputation is because of the stereotype of ivory tower architects drawing very high-level pictures to describe their grandiose visions. I'm sure you've seen examples of diagrams with a big box labelled "Enterprise Service Bus" connected to a cloud, or perhaps diagrams showing a functional decomposition with absolutely no consideration as to whether the design is implementable.

I like to see software architecture diagrams have a grounding in reality and I don't consider technology choices to be "an implementation detail". One way to ensure that technology is considered is to simply show the technology choices by including them on software architecture diagrams. 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.

Of course, if you're retrospectively diagramming an existing codebase, the technology decisions have already been made and therefore you have the information to add to the diagrams. But what about when the diagrams are being drawn during an up front design exercise? Perhaps you don't know what technologies you're going to use. Or perhaps there are a number of options.

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 logical building blocks and ignoring technology? If the answer to these questions is "no", then my recommendation is to include as much information as possible. For example, if your container diagram shows a database but you're not sure whether it will be Microsoft SQL Server or MySQL, why not state this by annotating the database element with "Microsoft SQL Server or MySQL". Doing this at least makes it explicit that a decision needs to be made, and it shows the options that are under consideration too. Doing so provides a better starting point for conversations, particularly if you have a choice of technologies to use.

Encouraging people to include technology choices on their software architecture diagrams also tends to lead to much richer and deeper conversations that are grounded in the real-world. A fluffy conceptual diagram tends to make a lot of assumptions, but factoring in technology forces the following types of questions to be asked instead.

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

- etc

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

9.3 Would you code it that way?

As a final note related to creating diagrams that reflect reality, if you're doing up front design, a simple question to ask yourself is, "Does this diagram accurately portray what we are going to build? Would we code it that way?". And if you're drawing diagrams to describe an existing codebase, the question becomes, "Does this diagram accurately reflect how we have built the software? Did we code it that way?".

Let's illustrate this with a couple of scenarios that I regularly hear in my software architecture workshops.

Scenario 1: Shared components

Imagine that you're designing a software system that includes two web applications, perhaps one for human users and one that exposes an API. While thinking about the components that reside in each of these containers, it's not uncommon to hear a conversation like this:

- **Attendee**: "We would like to use a shared logging component in both of the web applications. Should we draw this logging component outside of the two web application containers since it's used by both of them?"
- **Me**: "Would you code it that way? Will the logging component be running outside of both web applications? For example, will it really be a separate container or process?"
- **Attendee**: "No, the logging component would be a shared component in a [JAR file|DLL|etc] that we would deploy as a part of both applications."
- **Me**: "In that case, draw it like that too. Include the logging component inside of each web application and simply label it as a shared component."

If you're going to implement something like a shared component that will be deployed inside a number of separate applications, 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.

Scenario 2: Layering strategies

Imagine you're designing a web application that is internally structured as a presentation layer, a services layer and a data access layer.

- **Attendee**: "Should we show that all communication to the database from the presentation layer goes through the services layer?"
- **Me**: "Is that how you're going to implement it? Or will the presentation layer access the database directly?"
- **Attendee**: "We were thinking of perhaps adopting the CQRS pattern so, for read requests, the presentation layer 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 presentation layer 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. If you can understand how you would code it, you can understand how to visualise it.

10. Other diagrams

The context, container and component diagrams that we've seen so far are often sufficient to describe a software system. However, sometimes it can be useful to draw some additional diagrams to highlight different aspects.

10.1 Architectural view models

There are a number of different ways to think about, describe and visualise a software system. Examples include IEEE 1471, ISO/IEC/IEEE 42010, Philippe Kruchten's 4+1 model, etc. What they have in common is that they all provide different "views" onto a software system to describe different aspects of it. For example, there's often a "logical view", a "physical view", a "development view" and so on.

Something you might be wondering is how the C4 model compares to some of the more popular architectural view models, such as Philippe Kruchten's 4+1 architectural view model and the collection offered in Software Systems Architecture by Eoin Woods and Nick Rozanski.

"4+1" View Model (Philippe Kruchten)

The "4+1" view model consists of five different views that can be used to describe a software system. The original definition can be found in Philippe's IEEE paper, Architectural Blueprints - The â€œ4+1â€ View Model of Software Architecture, although many people have refined the model since the original paper was published. Some of what you'll find written about "4+1" on the Internet doesn't necessarily match the content of the original paper too, and many people have subtly redefined the views of the model to better map onto the notation or methodology they were using at the time (e.g. UML). My summary of "4+1" is as follows:

- **Logical View**: This view describes the functionality delivered by the system. It's usually one or more high-level diagrams that show the major functional building blocks and how they are related.

- **Process View**: This view describes how the logical building blocks are combined together into physical processes (or execution units). It's used to capture concurrency, inter-process communication, etc.
- **Physical View**: This view describes the infrastructure and deployment topology of the software.
- **Development View**: This view describes the how the functional building blocks in the logical view are implemented by software developers using modules, components, classes, layers, etc.
- **Scenarios View**: A number of selected use cases, or scenarios, are used to drive and illustrate the content of the four previous views.

Software Systems Architecture (Eoin Woods and Nick Rozanski)

"Software Systems Architecture" by Eoin Woods and Nick Rozanski defines another model with which to describe a software system. This builds upon the "4+1" view model and presents a collection of seven viewpoints as follows:

- **Context Viewpoint**: This describes how the software system fits into the surrounding environment (i.e. people and other software systems).
- **Functional Viewpoint**: This is similar to the "Logical View" in the "4+1" model; it shows the functional building blocks that make up the software system. It was renamed to make the intent clearer (i.e. you're looking at the *functional* building blocks that make up the software system).
- **Information Viewpoint**: This describes how the software system stores and uses information, from a static structural perspective (e.g. entity relationship models, etc), how that information is used at runtime (e.g. the flow of information through system, information state models, etc), how it's archived, volumetrics and so on. This viewpoint allows information to be seen as a first-class citizen, rather than a by-product of the software system.
- **Concurrency Viewpoint**: This is similar to the "Process View" in the "4+1" model.
- **Development Viewpoint**: This is similar to the "Development View" in the "4+1" model.
- **Deployment Viewpoint**: This is similar to the "Physical View" in the "4+1" model.
- **Operational Viewpoint**: This viewpoint is used to describe the operational aspects of the software system; including installation, operation, upgrades, data migration, configuration management, administration, monitoring, support models, etc.

Common vocabulary

A big problem I've found in the real-world with many of these existing approaches is that it starts to get confusing very quickly if the whole team isn't versed in the terminology used. For example, I've heard people argue about what the difference between a "conceptual view" and a "logical view" is. And let's not even start asking questions about whether technology is permitted in a logical view. Perspective is important too. If I'm a software developer, is the "development view" about the code, or is that the "implementation view"? But what about the "physical view"? Code is the physical output, after all. But then "physical view" means something different to infrastructure architects. But what if the target deployment environment is virtual rather than physical?

A common theme throughout this book has been about creating a shared vocabulary, and the same applies when you're considering which other diagrams to draw. One way to resolve the terminology issue is to ensure that everybody on the team can point to a clear definition of what the various diagram types or architectural views are. Just be aware that different software architecture books often use different names to describe the same architectural view. This one included, of course.

Bridging the model-code gap

In my experience, a "Logical View" (or "Functional View") is typically what people think of when asked to draw "an architecture diagram". While that's a useful view to create, it's often confusing, especially for software developers, when that logical view of functional building blocks doesn't easily map onto or reflect the code. Also, this split between the "Logical View" and the "Development View" is often what I see in organisations where there is a clear separation between "the architects" and "the developers". This isn't necessarily a bad thing, but the transition between the "Logical" and "Development" views doesn't mandate a process hand-off between separate architecture and development teams. Process hand-offs tend to further exaggerate the distance between the "Logical" and "Development" views.

My personal preference is to minimise, and in fact *remove*, the gap between a logical view of what the system does and the real-world view of how that functionality is implemented in code. As we've already discussed in previous chapters, this is about minimising the model-code gap. In essence, the C4 model *spans and combines* what you might find in a "Logical View" and "Development View" into a single description of the static structure, across a number of different levels of abstraction:

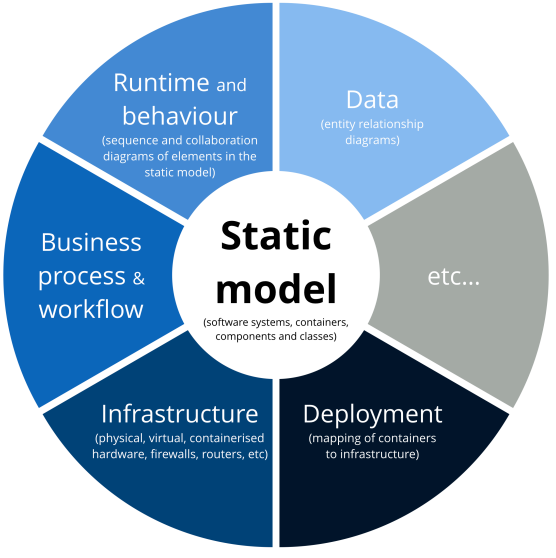
- **System Context**: This shows you what the system does and how it fits into the world around it (i.e. users and other software systems).

- **Containers**: This shows how the functionality delivered by the system is partitioned up across the high-level building blocks (i.e. containers).
- **Components**: This shows how the functionality delivered by a particular container is partitioned across components within that container.

You can say that the C4 model also includes some of what you would find in the "4+1 Process View", especially given that the container diagram shows execution units. That's certainly true, although there still may be occasions when it's worth creating a specialised version of a container diagram to highlight concurrency or synchronisation.

Understand the static structure first

The primary focus of this book is describing and communicating the *static structure* of a software system; from the big picture of how a software system fits into its environment down to its components and the classes that implement them. Once you have a shared vocabulary with which to describe the static structure of a software system (at varying levels of abstraction), it becomes easy to communicate other aspects of that system based upon the static structure.



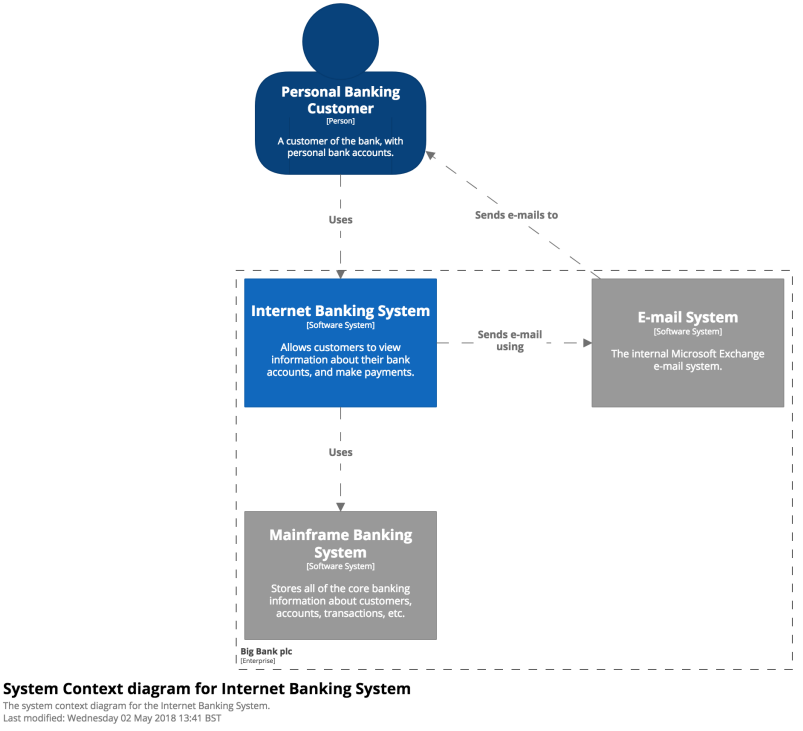
The static structure defines the core of the software architecture model

With this in mind, here are some other diagrams that you might want to consider creating.

10.2 System Landscape

The C4 model provides a static view of a *single software* system but, in the real-world, software systems never live in isolation. For this reason, and particularly if you are responsible for a collection of software systems, it's often useful to understand how all of these software systems fit together within the bounds of an enterprise. To do this, I'll add another diagram that sits "on top" of the C4 diagrams, to show the system landscape from an IT perspective. Like the System Context diagram, this diagram can show the organisational boundary, internal/external users and internal/external systems.

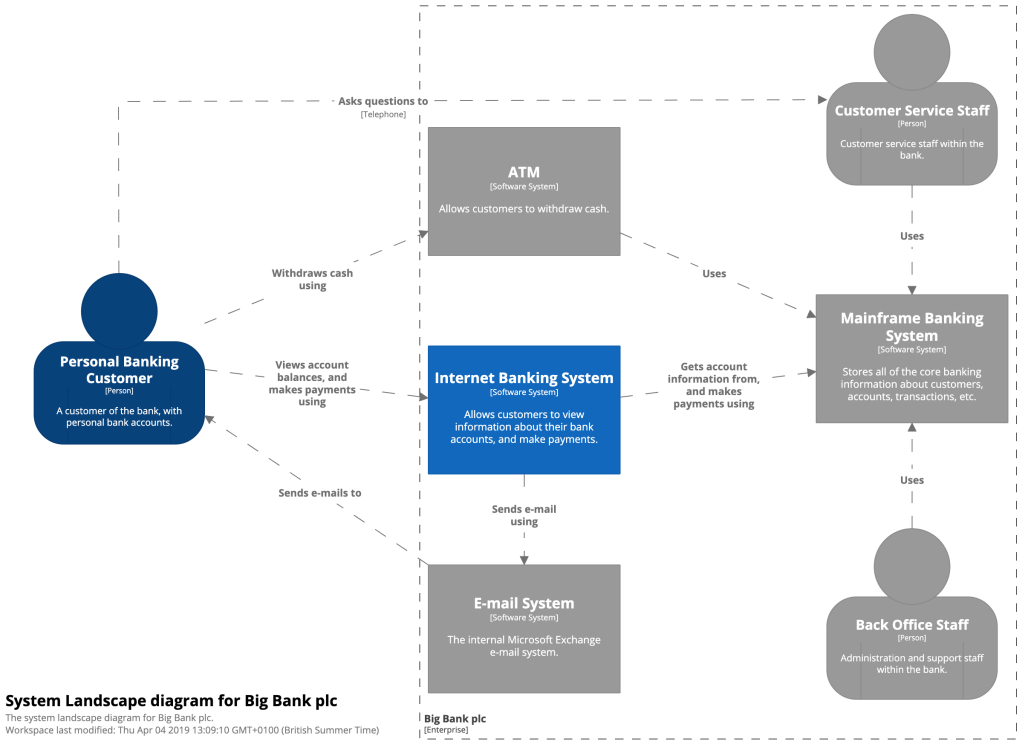
From a practical point of view, a System Landscape diagram is really just a System Context diagram without a specific focus on a particular software system. Here's the example System Context diagram for the Internet Banking System that we've seen before..



A System Context diagram for a fictional Internet Banking System

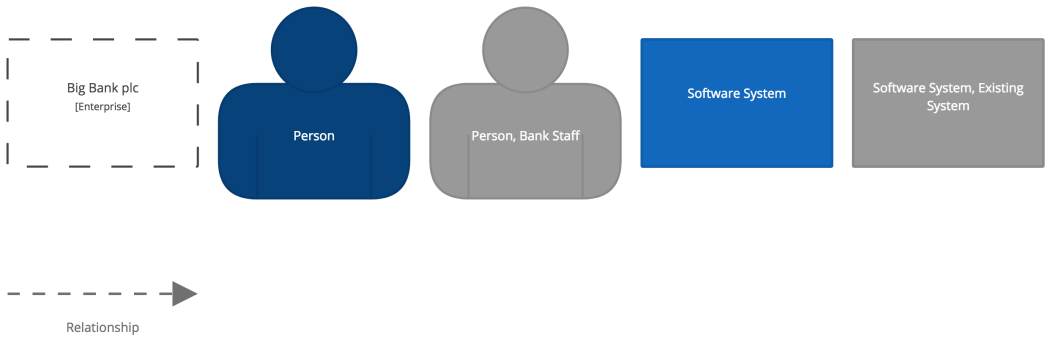
When drawing a System Context diagram, I usually only include the people and software systems that have a *direct relationship* with the software system in focus. In contrast, a

System Landscape diagram will typically show a larger portion of the IT landscape in order to tell a larger overall story.



A System Landscape diagram for the fictional "Big Bank plc"

Again, I've used a box with a dashed border to represent the boundary of the enterprise, alongside some additional element styles, so that we can see what is *internal* to the enterprise, and what is *external*.



The diagram key for the System Landscape diagram

As a caveat, I do appreciate that enterprise architecture isn't simply about technology but, in my experience, many organisations don't have an enterprise architecture view of their IT landscape. In fact, it shocks me how often I see organisations of all sizes that lack such a holistic view, especially when you consider that technology is usually a key part of the way they implement business processes and service customers. Diagramming the enterprise from a technology perspective at least provides a way to think outside of the typical silos that form around IT systems and the teams that are responsible for them.

10.3 User interface mockups and wireframes

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

10.4 Business process and workflow

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

10.5 Domain model

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

10.6 Runtime and behaviour

The majority of this book has concentrated on an approach to thinking about, describing and communicating software architecture that is focussed around static structure: software systems, containers, components and classes. Whenever I've needed to document a software system in the past, from a diagramming perspective anyway, most of what I've created echoes this sentiment, with the majority of my diagrams being descriptions of the static structure. My software architecture sketching workshops also confirm a similar trend. During the initial iteration (where I provide very little guidance and hopefully don't influence the outcome), I estimate that 80-90% of the diagrams I see reflect static structure.

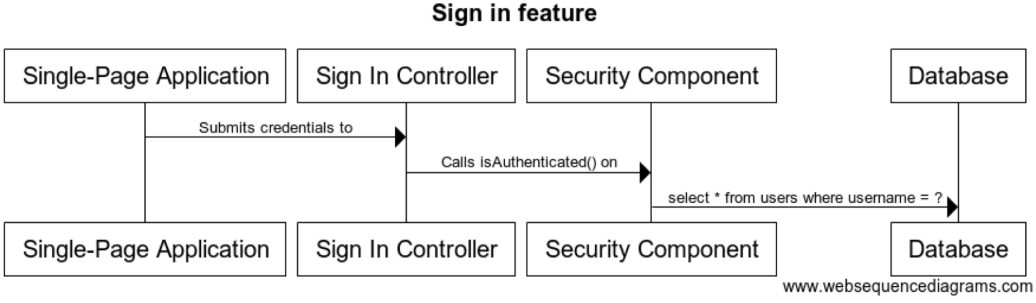
Software isn't static though, and needs to be executed in order to actually do something useful. For this reason, it can be useful to create diagrams that illustrate what happens at runtime (a "dynamic view") for important use cases, user stories or scenarios.

To do this, I simply take the concept of sequence and collaboration/communication diagrams from UML and apply them to the static elements in the C4 model. For example, you could illustrate how a use case is implemented by drawing a sequence diagram of how components interact at runtime. Or you could show the interaction between containers if you have more of a microservices style of architecture, where every service is deployed in a separate container.

UML sequence and collaboration diagrams are typically the way that most people do this, myself included, although I tend to use UML as a sketching notation rather than precisely following the specification. There are lots of UML introductions on the web, but essentially both diagram types show the same information, albeit from a slightly different perspective.

Sequence diagrams

A UML sequence diagram is typically made up of a number of items (left to right) and a timeline (top to bottom). The diagram illustrates how the various items collaborate (using horizontal arrows) by sending messages, making requests, etc. The vertical order of the arrows illustrates the sequencing. You can use sequence diagrams to illustrate any sequence of items collaborating. Commonly these items are code elements such as classes, but there's nothing preventing you showing people, software systems, containers or components. As example, here's a sequence diagram to illustrate how the sign in process works for the Internet Banking System.

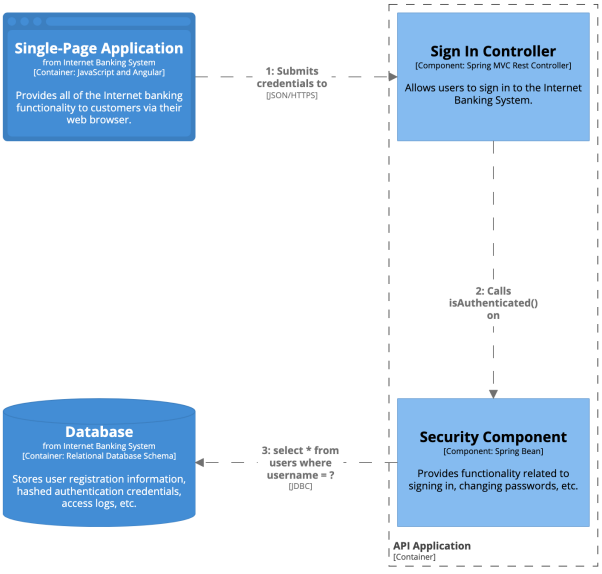


There are a number of tools and approaches to creating sequence diagrams, but they all create similar diagrams, with the official UML specification detailing ways to add more precision and semantics onto the diagrams.

Collaboration diagrams

The other approach is a collaboration diagram[1]. Essentially this shows the same information as a sequence diagram, although that information is presented in a different way. Typically, this is a simpler "boxes and lines" style diagram where the lines have been annotated and **numbered** to indicate the ordering of collaborations. Here's the same sign in scenario, this time illustrated using a collaboration diagram.

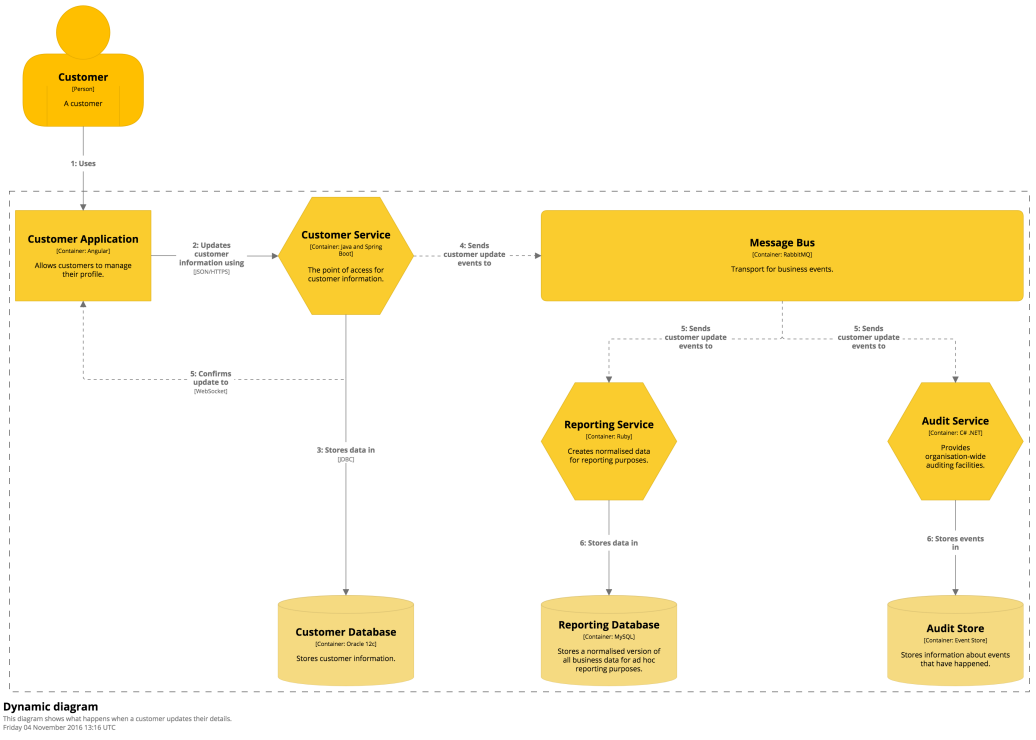
[1]In UML 2.x, this is called a communication diagram, but the purpose and content is essentially the same.



Dynamic diagram for API Application

Summarises how the sign in feature works in the single-page application.
Workspace last modified: Thu Apr 04 2019 13:09:10 GMT+0100 (British Summer Time)

You can also use the numbering of lines to illustrate parallelism, simply by giving multiple lines the same number.



In this example of a microservices architecture, the message bus is broadcasting the "customer update event" to the Reporting Service and Audit Service at the same time, so I've numbered these lines the same ("5"). Again, depending on the story you want to tell, you can choose how precise to be. Both the Reporting and Audit services are storing data in a data store, and I've chosen to number both these lines the same too ("6"), suggesting that this happens in parallel. But, of course, this may or may not be the case, depending on how long each service takes to process the event. In most cases, this is sufficient to tell the story. If you wanted to be more precise, you could use UML's concept of nested sequence numbering, using "5.1" instead of "6". I personally find the use of the nested sequence numbering confusing, especially when the nesting starts to get more than a couple of levels deep, so I tend to avoid it. But it's certainly another tool in the toolbox.

I prefer a collaboration diagram because it tends to be easier to draw, especially on a whiteboard, but either diagram type works provided you're not trying to show too many collaborations. And don't forget all of the tips about notation that you read about earlier because they are equally applicable here too.

When to create dynamic views

Given the number of execution paths through even a relatively small software system (user stories, success/failure scenarios, conditional logic, asynchronous processing, etc), it's obviously not practical to document everything. Especially not at the class or method level. In fact, if you want to figure out how something works, it's often easier to just dive into the code, run an automated test or use a debugger. This assumes that you have a good starting point and know where to look, of course.

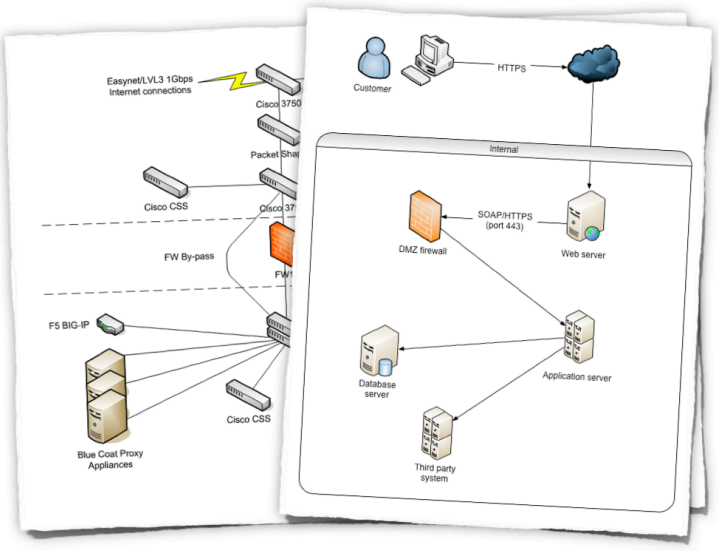
If I think back to the software systems that I've documented, and my documentation approach in general, I very rarely describe the dynamic aspects of a software system. A few examples where I have done this include:

- Explaining how a low-level design pattern works at the code level (the interactions between classes).
- Explaining the interactions between applications and services during user authentication when using a federated security provider (for example, the handshaking and interactions between an ASP.NET application running Windows Identity Foundation against Microsoft Active Directory Federation Services isn't straightforward).
- Explaining the typical flow of asynchronous messages/events that implement a business process.

While the dynamic aspects of a software system are certainly important, I don't typically find that documenting them adds much value. As I said, rather than documenting every execution path through a software system, I'll only do this in order to explain the significant or complex scenarios, especially where they are not evident by reading the code.

10.7 Infrastructure

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



Example infrastructure diagrams, created in Microsoft Visio

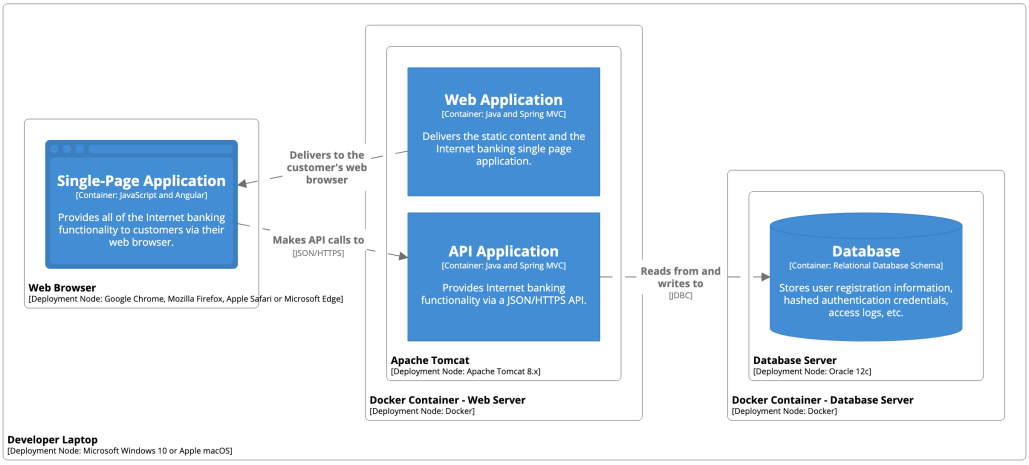
10.8 Deployment

It's often useful to describe the deployment mapping between containers and infrastructure. Even if your deployment is fully automated, it can still sometimes be useful to have a diagram summarising the deployment mapping. For example, a database-driven website could be deployed onto a single server or across a server farm made of up hundreds of servers, depending on the need to support scalability, resilience, security, etc.

The concept of a deployment diagram comes from UML, and it's used to describe the mapping of deployment artifacts (e.g. a deployable unit, such as a JAR file) to deployment nodes (i.e. devices or execution environments). I take a slightly simpler approach whereby my deployment diagrams show the mapping of *containers* (from the C4 model) to *deployment nodes*. A deployment node is something like:

- Physical infrastructure (e.g. a physical server or device).
- Virtualised infrastructure (e.g. IaaS, PaaS, a virtual machine).
- Containerised infrastructure (e.g. a Docker container).
- An execution environment (e.g. a database server, Java EE web/application server, Microsoft IIS, etc).

And deployment nodes can be nested. As an example, the following diagram illustrates what a development environment for the Internet Banking System might look like.

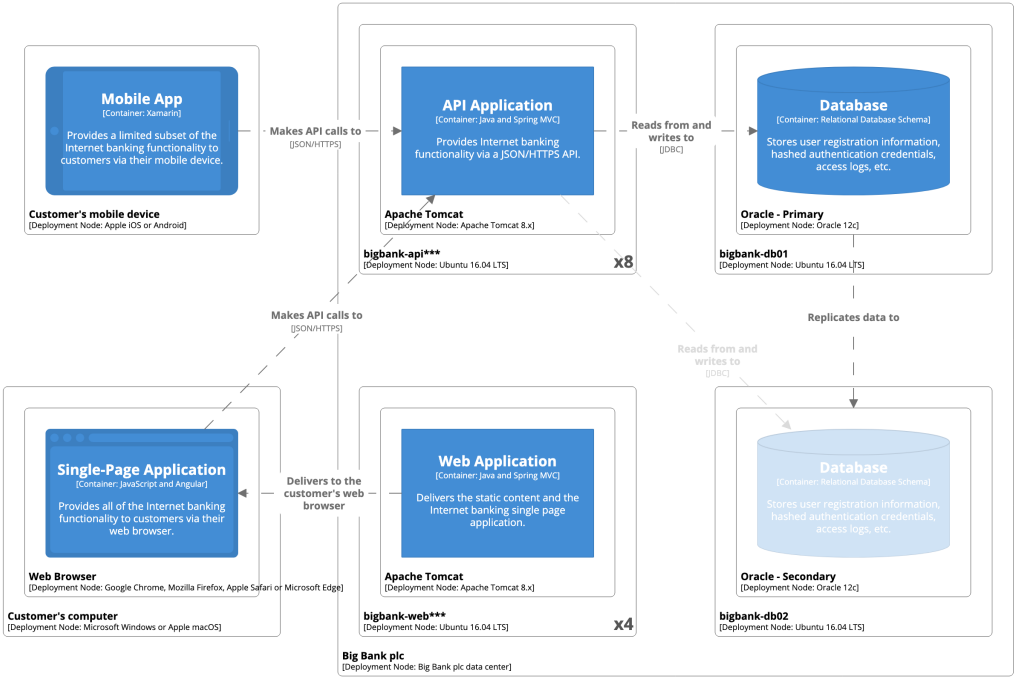


Deployment diagram for Internet Banking System - Development

An example development deployment scenario for the Internet Banking System.
Workspace last modified: Thu Apr 04 2019 13:09:10 GMT+0100 (British Summer Time)

A developer laptop has a native web browser installed, and that's where the Single Page Application will be running when we're doing development. There are then a couple of Docker containers. One runs a copy of Apache Tomcat, which is where we deploy development versions of the Web and API Applications. The other Docker container is running our development database.

For a slightly more complicated example, let's look at what the live deployment environment might look like.



Deployment diagram for Internet Banking System - Live

An example live deployment scenario for the Internet Banking System.
Workspace last modified: Thu Apr 04 2019 13:09:10 GMT+0100 (British Summer Time)

Customers may have computers or mobile devices, which is where the UIs end up running. Everything else is deployed in the Big Bank's data center, onto a number of separate Ubuntu servers. I've also shown a secondary database server, with data replication.

Feel free to include routers, load balancers, host names, IP addresses, or whatever other information you think helps you tell the story that you want to tell.

10.9 And more

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

II Document

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

11. Software documentation as a guidebook

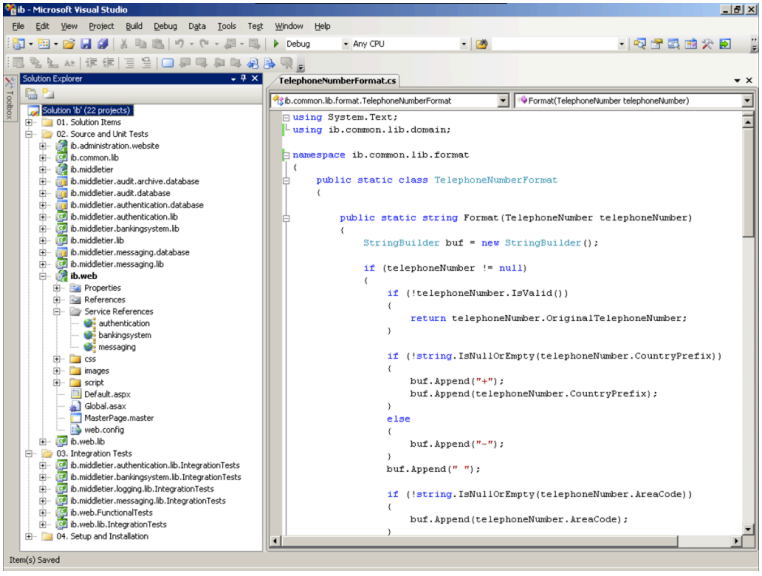
"Working software over comprehensive documentation" is what the Manifesto for Agile Software Development says and it's incredible to see how many software teams have interpreted those five words as "don't write *any* documentation". The underlying principle here is that real working software is much 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 not having a source of supplementary information about a complex software system can slow a team down as they struggle to navigate the codebase.

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



Where do you start?

If nobody has the time to walk you through the codebase, you can start to make your own assumptions based upon the limited knowledge you have about the project, the business domain, your expectations of how the team builds software and your knowledge of the technologies in use.

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

- The system has been written in C# on the Microsoft .NET platform.
- The overall .NET solution has been broken down into a number of Visual Studio projects and there's a .NET web application called "ib.web", which you'd expect since this is an Internet banking system ("ib" = "Internet Banking").
- The system appears to be made up of a number of architectural tiers. There's "ib.web" and "ib.middletier", but I don't know if these are physical or logical tiers.
- There looks to be a hierarchical naming convention for projects. For example, "ib.middletier.authenti "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.

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

- How the software system fits into the existing system landscape.
- Why the technologies in use were chosen.
- The overall structure of the software system.
- Where the various components are deployed at runtime and how they communicate.
- How the web-tier "knows" where to find the middle-tier.
- What approach to logging/configuration/error handling/etc has been adopted and whether it is consistent across the codebase.
- Whether any common patterns and principles are in use across the codebase.
- How and where to add new functionality.
- How security has been implemented across the stack.
- How scalability is achieved.
- How the interfaces with other systems work.
- etc

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

11.2 Our duty to deliver documentation

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

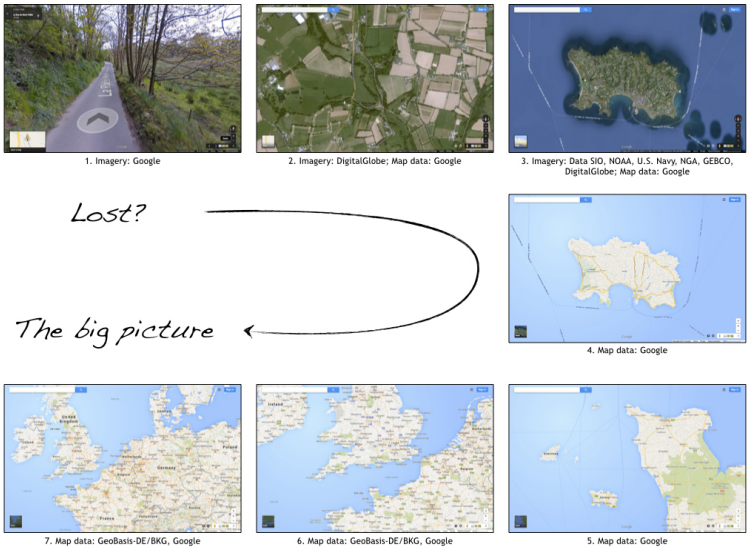
under an outsourcing and/or offshoring contract. I've seen IT consulting organisations deliver highly complex software systems to their customers without a single piece of supporting documentation, often because the team doesn't *have* any documentation. If the original software developers leave the 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?

11.3 Lightweight, supplementary documentation

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. I like to 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?

11.4 1. Maps

Let's imagine that I teleported you away from where you are now and dropped you in a quiet, leafy country lane somewhere in the world (picture 1). Where are you and how do you figure out the answer to this question? You could shout for help, but this will only work if there are other people in the vicinity. Or you could simply start walking until you recognised something or encountered some civilisation, who you could then ask for help. As geeks though, we would probably fire up the maps application on our smartphone and use the GPS to pinpoint our location (picture 2).



From the detail to the big picture

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

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

If I were to open up the codebase of a complex software system and highlight a random line of code, exploring is fun but it would take a while for you to understand where you were and how the code fitted into the software system as a whole. Most integrated development environments have a way to navigate the code by namespace, package or folder but often the physical structure of the codebase is different to the logical structure. For example, you may have many classes that make up a single component, and many of those components may make up a single deployable unit.

Diagrams can act as maps to help people navigate a complex codebase and this is one of the most important parts of supplementary software documentation. Ideally there should be a small number of simple diagrams, each showing a different part of the software system or level of abstraction. Visualisation of software architecture was the focus of the first part of this book.

11.5 2. Sights

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

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

11.6 3. History and culture

If you do ever visit Jersey, and you really should because it *is* beautiful, you may see some things that look out of kilter with their surroundings. For example, we have a lovely granite stone castle on the south coast of the island called Elizabeth Castle that was built in the 16th century. As you walk around admiring the architecture, eventually you'll reach the top where it looks like somebody has dumped a large concrete cylinder, which is not in keeping with the intricate granite stonework generally seen elsewhere around the castle.

A joining of two distinct architectural styles

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.

11.7 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 software developers, support staff, etc do their job effectively.

11.8 Describe what you can't get from the code

Exploring is great fun but ultimately it takes time, which we often don't have. Since the code doesn't tell the whole story, *some* supplementary documentation can be very useful, especially if you're handing over the software to somebody else or people are leaving and joining the team on a regular basis. My advice is to think about this supplementary documentation as a guidebook, which should give people enough information to get started and help them accelerate the exploration process.

Do resist the temptation to go into too much technical detail though because the technical people that will understand that level of detail will know how to find it in the codebase anyway. As with everything, there's a happy mid-point somewhere. The following headings describe what you might want to include in a software guidebook:

1. Context
2. Functional Overview
3. Quality Attributes
4. Constraints
5. Principles
6. Software Architecture
7. Code
8. Data
9. Infrastructure Architecture
10. Deployment
11. Operation and Support
12. Development Environment
13. Decision Log

There are, of course, a number of different documentation templates available, and this is my starting point for my own documentation. I would also recommend taking a look at arc42, which captures the same sort of information in a slightly different format, and the "Building Block View" complements the C4 model nicely.

11.9 Product vs project documentation

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

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

11.10 Keeping documentation up to date

We'll talk about tooling later but, from a process perspective anyway, my approach to documentation is "little and often". If you have a "definition of done" for your work items, simply add another line to the bottom that says something like, "Documentation and diagrams updated". From my own experience, doing a number of small doucmentation updates is much more palatable than sitting down for a few weeks with the sole purpose of writing doucmentation!

11.11 Documentation length

How long should documentation be? I get asked this question a lot. And rather than talk about numbers of pages, what I'm really looking for is something that I can read in a couple of hours, over a coffee or two, to get a really good *starting point for exploring the code.*

12. 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.

12.1 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)

12.2 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.

12.3 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.

12.4 Audience

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

12.5 Required

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

13. 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.

13.1 Intent

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

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

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

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

13.2 Structure

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

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

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

13.3 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.

13.4 Audience

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

13.5 Required

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

14. 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.

14.1 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.

14.2 Structure

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

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

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

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

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

14.3 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.

14.4 Audience

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

14.5 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.

15. 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.

15.1 Intent

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

15.2 Structure

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

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

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

15.3 Motivation

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

15.4 Audience

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

15.5 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.

16. Principles

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

16.1 Intent

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

16.2 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

16.3 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.

16.4 Audience

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

16.5 Required

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

17. 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.

17.1 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)

17.2 Structure

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

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

17.3 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.

17.4 Audience

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

17.5 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.

18. Code

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

18.1 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

18.2 Structure

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

18.3 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.

18.4 Audience

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

18.5 Required

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

19. Data

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

19.1 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?

19.2 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).

19.3 Motivation

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

19.4 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.

19.5 Required

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

20. 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.

20.1 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?

20.2 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. 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.

20.3 Motivation

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

20.4 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.

20.5 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.

21. Deployment

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

21.1 Intent

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

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

21.2 Structure

There are a few ways to structure this section:

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

2. Diagrams: UML or "boxes and lines" style deployment diagrams, showing the mapping of containers to infrastructure.

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).

21.3 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.

21.4 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.

21.5 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.

22. Operation and Support

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

22.1 Intent

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

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

22.2 Structure

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

22.3 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.

22.4 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.

22.5 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!

23. Development Environment

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

23.1 Intent

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

23.2 Structure

The type of things you might want to include are:

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

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

23.3 Motivation

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

23.4 Audience

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

23.5 Required

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

24. 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.

24.1 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

24.2 Structure

Again, keep it simple, with a short paragraph or architecture decision record 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.

24.3 Motivation

The motivation for recording the significant decisions is that this section can act as a point of reference in the future. All decisions are made given a specific context and usually have

trade-offs. There is usually never a perfect solution to a given problem. Articulating the decision making process after the event is often complex, particularly if you're explaining the decision to people who are joining the team or you're in an environment where the context changes on a regular basis.

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

24.4 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.

24.5 Required

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

III Tooling

This part of the book is about the use of tooling to visualise, document and explore your software architecture.

25. Sketches, diagrams, models and tooling

Now that we've created a shared vocabulary and seen how to draw some pictures at varying levels of abstraction, let's look at the life cycle of pictures and the various ways we can create them.

25.1 Sketches

Whether you're undertaking an up-front design exercise or retrospectively documenting an existing software system, most people will start with sketches on a piece of paper or whiteboard. Sketching software architecture diagrams, particularly on a whiteboard, is a great way to collaborate, exchange ideas and try things out. The tools are simple too; you simply need a canvas and some coloured marker pens.

To prevent the sketches from morphing into those ad hoc "boxes and lines" diagrams that we saw right back at the start of the book, I recommend that you take a few minutes to create your shared vocabulary and agree the diagram types that you want to produce. Be conscious of the notation too but don't worry about including all of the detail. In other words, do try to be precise, but don't worry too much about the fidelity of the diagrams, especially if the sketches will have a short lifespan.

25.2 Diagrams

At some point, you'll probably need to create something more formal than a collection of sketches on a whiteboard. Why? Perhaps you need to record the diagrams for historical purposes, or maybe the diagrams need to be included in technical specifications, work/bid proposals, etc.

Photos

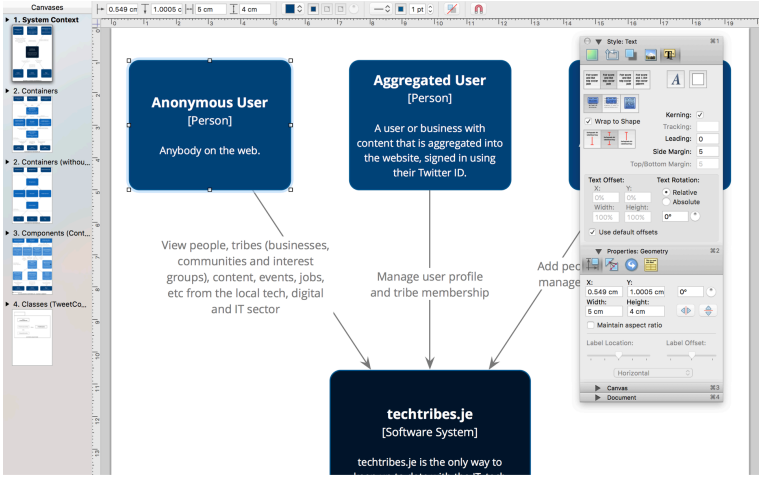
The simplest way to record sketches digitally is to take a photo. This is certainly an option if you're not worried about presenting the sketches, but often there's a need to create a version

of the sketches that looks a little more polished. The other disadvantage of photos is they are hard to update. Imagine you spot a missing diagram element after taking a photo and clearing the whiteboard.

Drawing tools

At this point, the default option for many people is to create an electronic version of the diagram using tooling, and there are many options. The most common is to use a desktop drawing tool such as Microsoft Visio, OmniGraffle, SimpleDiagrams, etc. There are some web-based solutions too; including draw.io, Gliffy, Lucidchart, etc. Alternatively you could use the diagram creation facilities in Microsoft Word, Microsoft PowerPoint, Apple Keynote, etc. Most of these drawing tools allow you to produce an image-based (e.g. PNG) export so the diagrams can be embedded in other documents or web pages. Some of the web-based tools provide direct integration with wikis such as Atlassian Confluence.

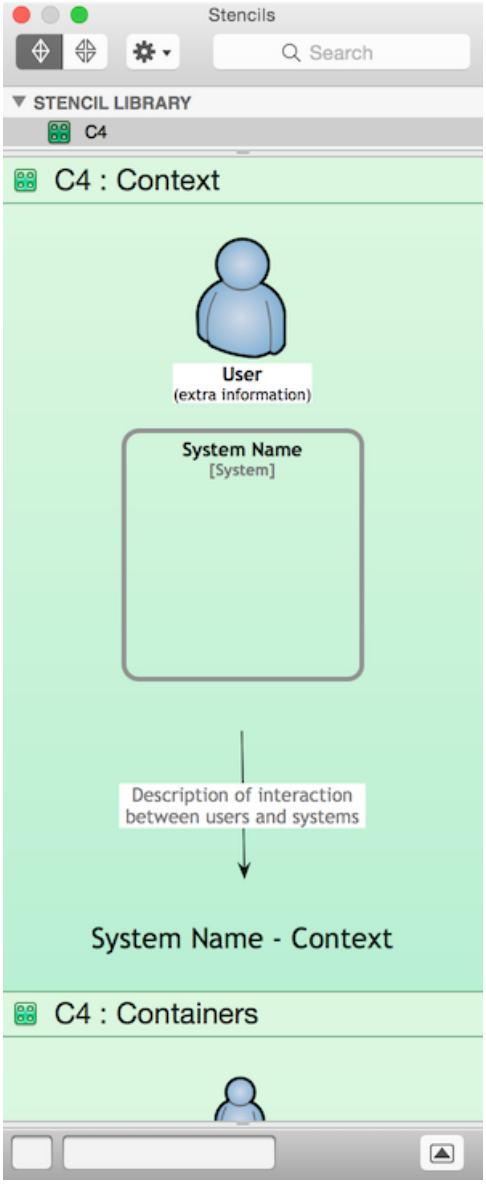
These drawing tools allow you to create any type of diagram you like, by dragging elements onto a canvas and customising the colours, text positioning, line styles, etc. There is a little up-front work required to recreate the notation you've used on your sketches, but after that it's a simple matter of copying and pasting elements to create your diagram. Beware though, it's very easy to spend a huge amount of time trying to make the diagrams look pretty!



A general purpose diagramming tool

Many of these tools allow you to create templates or stencils that you can use to make the diagramming process more efficient too. For example, Dennis Laumen has created an OmniGraffle stencil that will save you some time. After installing it, you can simply drag

the C4 elements (people, software systems, containers, components, etc) onto your diagram canvas and change the text (name and description) as needed.



A C4 stencil for OmniGraffle

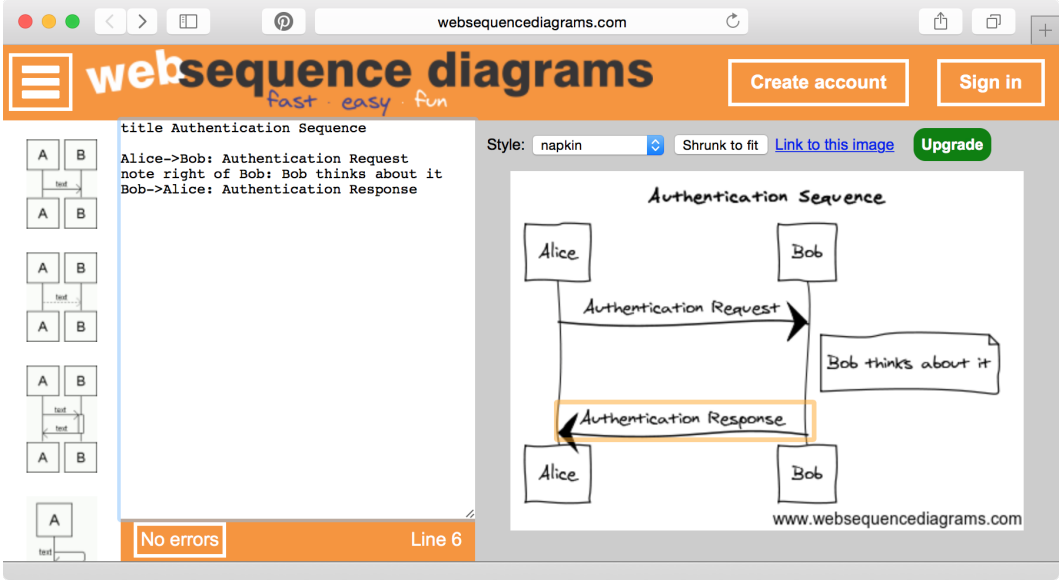
One problem with a general purpose diagramming tool is consistency, or rather, the lack of

it. Once you start creating multiple diagrams, you need to put some effort into ensuring that your diagram elements are named and styled consistently across those diagrams. This is easy to do when you only have a small number of simple diagrams, but the challenge increases with the number and complexity of your diagrams.

Another problem is that the files created by these drawing tools often aren't amenable to being version controlled. It's not that you can't add them to a version control system, it's more a case of it being tricky to understand the difference between diagram versions, especially if you have a binary file format.

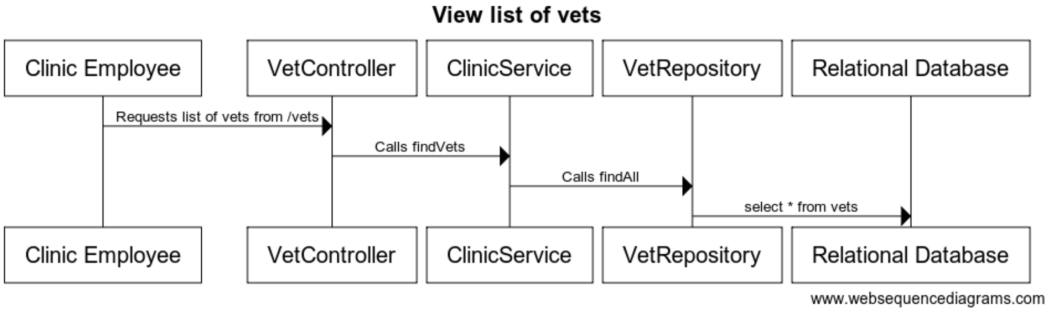
Text-based diagramming tools

A solution to the version control problem is to use a text-based diagramming tool such as WebSequenceDiagrams, yUML, nomnoml, PlantUML, etc. These tools allow you to write text that describes a set of elements and the interactions between them. The diagram is then created for you.



WebSequenceDiagrams allows you to create diagrams using text

Here's the sequence diagram that we saw before for the Spring PetClinic.



This particular example was created using WebSequenceDiagrams, using the following text:

```
title View list of vets

Clinic Employee->VetController: Requests list of vets from /vets
VetController->ClinicService: Calls findVets
ClinicService->VetRepository: Calls findAll
VetRepository->Relational Database: select * from vets
```

The majority of these tools are UML focussed, which is great if you want to use UML, not so much otherwise. You do also lose a degree of control over the graphical styling and layout of the resulting diagrams. The upside, of course, is that it's easy to create diagrams, at least simple diagrams anyway. Additionally, as software developers, we often find working with text much easier than messing around with boxes and lines in a drawing tool.

While text-based diagramming tools relieve some of the burden of manually creating, styling and moving boxes in a drawing tool, they still don't necessarily resolve the issue of consistency. For that you need to move onto modeling tools.

25.3 Models

The software architecture diagrams we've discussed so far are simply that - diagrams. Regardless of whether you're using pen and paper or a tool like Microsoft Visio, these diagrams are pictures created by drawing boxes and lines freehand on a diagram canvas. You have all of the control over what you draw, and with that control comes the responsibility to ensure the diagram is accurate and consistent. Diagrams are static and we can't ask them any questions. Diagrams are purely visual representations.

The other strategy is to create a model of our software system. In contrast to a collection of diagrams, a model is a non-visual representation or definition of the software system. You

can then create a number of visual representations (diagrams) based upon the content of that model. Models are also typically machine-readable, so they can be queried or transformed into other representations too.

Modeling tools

There are many tools that support this way of working; such as No Magic MagicDraw, Sparx Enterprise Architect, Archi, IBM Rational Software Architect, StarUML, ArgoUML, etc. There are also modeling tool plugins/extensions for many of the popular IDEs. Essentially they all follow the same principle, by providing you with a way to create and a populate a model. You then use this model to create a number of diagrams, where a diagram represents a specific view of the model.

As an example, let's imagine that we want to create a context diagram. With a diagramming tool, to draw a software system, we need to create a box and put some text inside it. And we need to do this for every software system we want to include on the diagram. With a modeling tool, we create a definition for each software system (e.g. by specifying its name and description) and then use that element on the diagram by dragging it onto a diagram canvas. If you need to use the same software system across two diagrams, you just drag the element onto the second diagram canvas.

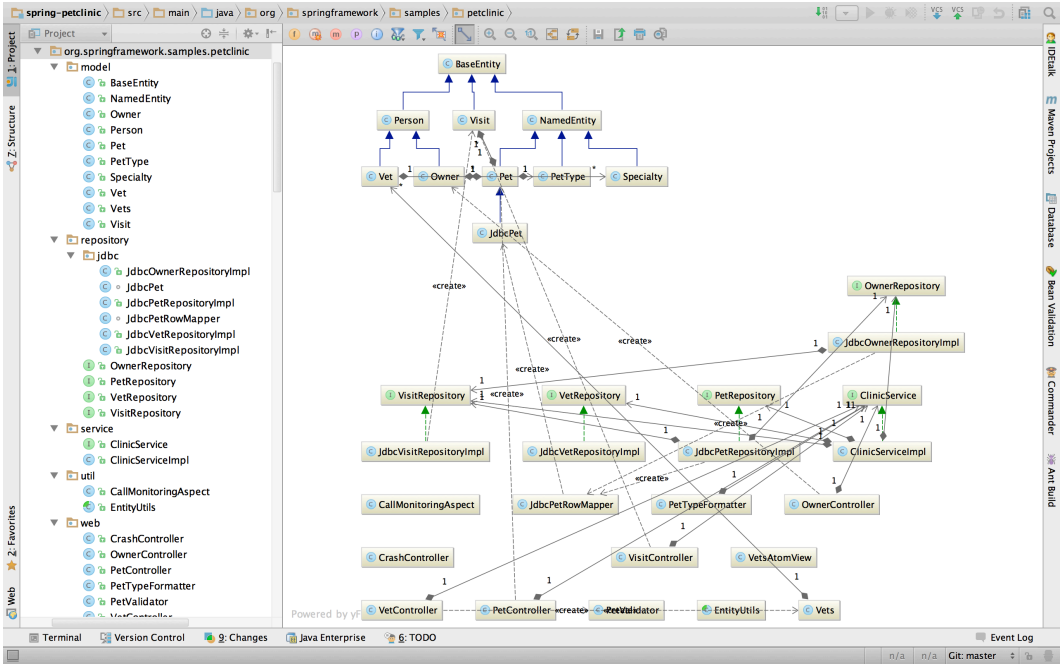
The power of having a model starts to come into play when you need to rename that software system. All you do is rename it in the model and all occurrences of the software system *across all diagrams* are renamed too. Compare this to a collection of diagrams where you need to check every diagram and rename any occurrences that you found. This is how a model introduces and improves consistency over a collection of static diagrams.

These tools typically support many different types of models and notations; including the Unified Modeling Language, SysML, ArchiMate and so on. This is great if you want to use these languages, otherwise you're out of luck. The other aspect you can't ignore is that you have to create the model, and often it can be a time-consuming task to populate the model with information. If you're modelling a software system as part of an up-front design exercise, the only real option you have is to use the modeling tool's user interface to populate the model. This can sometimes require lots of tedious data entry. If you have an existing codebase though, some modeling tools will provide the option to reverse-engineer the code and populate the model for you.

25.4 Reverse-engineering the software architecture model

There are many tools available that will help you reverse-engineer a codebase to create a model. Some of the modeling tools mentioned previously will do this, as will the popular IDEs. In addition, there's a category of static analysis tools that will do this too. Examples include Structure101, NDepend, Lattix, Sonargraph, etc. These tools work by scanning your codebase for elements and their dependencies. For an object-oriented programming language such as Java, this usually means creating a model of the code based upon the set of classes, interfaces and packages, along with the dependencies between all of these elements. Although the primary purpose of static analysis tools is to provide you information about the quality of your code, most of them also create some sort of architecture diagrams. This approach also resolves the thorny issue of how to keep diagrams up to date as a codebase evolves.

If you've ever tried to use a static analysis or modelling to automatically generate meaningful diagrams of your codebase via reverse-engineering though, you will have probably been left frustrated. The resulting diagrams tend to include too much information by default and they usually show you code-level elements rather than those you would expect to see on a software architecture diagram. We've seen a simple example of this already by trying to automatically create a UML class diagram from the Spring PetClinic codebase.



Most static analysis tools won't show you a single UML class diagram of a codebase, instead they'll start by showing you the top-level packages/namespaces and the dependencies between them. Double-click a package to expand it and you'll be shown the sub-packages and classes that reside within that package, along with the dependencies between them. Although some static analysis tools claim to generate "architecture diagrams", the diagrams they actually create are still very code focussed. Like us browsing a codebase, these tools see classes and interfaces in packages/namespaces when reverse-engineering code. Some tools can be given rules to recognise architectural constructs (e.g. layers or components) but this isn't typically the default out-of-the-box experience. In essence, these diagramming tools also suffer from the model-code gap. And furthermore, everything we need to understand the software system from an architectural perspective doesn't always exist in the code.

Why isn't the architecture in the code?

And this raises an interesting question. If the code is often cited as the "single point of truth", why isn't a description of the architecture in the code? To put it another way, if you give me your codebase, could I create some software architecture diagrams using only the information that exists in the source code? Let's look at this in the context of the C4 model.

Level 1: System Context

For a given software system, a system context diagram shows the key types of users (actors, roles, personas, etc) and system dependencies. Is it possible to get this information from the code?

- **Users**: If you're building a software system that has human users, I might be able to extract a list of user types from the code. For example, many software systems will have a security configuration file that describes the mapping between user types (e.g. roles, Active Directory groups, etc) and the parts of the system that such users have access too. Another possibility is to find an enumeration of the user types in the code itself, or perhaps in a database table. The implementation details will differ from codebase to codebase and technology to technology but, in theory, this information could be available somewhere.
- **System dependencies**: The list of system dependencies is a little harder to extract from a codebase. I could search the codebase for links to known libraries, APIs and service endpoints (e.g. URLs), and make the assumption that these are system dependencies. For example, if I see the twitter4j library on the classpath of a Java application, I can make an assumption that this application uses Twitter. But I don't know what the *intent* of the relationship is without exploring the code further. What about those system interactions that are done asynchronously, through the exchange of messages or documents via a message bus? And what about those system interactions that are done by copying a file to a network share? I know this sounds archaic, but it still happens. Understanding inbound system level dependencies is also tricky.

Level 2: Containers

Zooming in on the software system, a container diagram shows the various web applications, mobile apps, databases, file systems, console applications, etc and how they interact to form the overall software system. Again, some of this information will be present, in one form or another, in the codebase. For example, you could extract this information from:

- **IDE project files**: Information about executable artifacts (and therefore containers) could be extracted by parsing IntelliJ IDEA project files, Microsoft Visual Studio solution files, Eclipse workspaces, etc.
- **Build scripts**: Automated build scripts (e.g. Ant, Maven, Gradle, MSBuild, etc) typically generate executable artifacts or have module definitions that can again be used to identify containers.

- **Infrastructure provisioning and deployment scripts**: Infrastructure provisioning and deployment scripts (e.g. Puppet, Chef, Vagrant, Docker, etc) will probably result in deployable units, which can again be identified and this information used to create the containers model.

Extracting information from such sources is useful if you have a microservices architecture with hundreds of separate containers but, if you simply have a web application talking to a database, it may be easier to explicitly define this rather than going to the effort of scraping it from the code.

Level 3: Components

Zooming in further is the component diagram. Since even a relatively small software system may consist of a large number of components, this is a level that we certainly want to extract from the code. But it turns out that even this is tricky. Usually there's a lack of a consistent coding style, which makes it hard to identify components in the code. This is particularly true in older systems where the codebase lacks modularity and looks like a sea of thousands of classes interacting with one another. Assuming that there *is* some consistent structure to the code, "components" can be extracted using a number of different approaches, depending on the codebase and the degree to which an architecturally-evident coding style has been adopted:

- **Metadata**: The simplest approach, especially if you're starting a new project, is to annotate the architecturally significant elements in your codebase and extract them automatically. You could do this by adding a custom `@Component` annotation to Java classes that represent components. Similarly with a custom `[Component]` attribute in C#. If you have an existing codebase, another approach is to utilise the metadata already provided by a framework that you are using. For example, you could use the annotations provided by Spring (`@Controller`, `@Service`, `@Repository`, etc) or Java EE (`@EJB`, `@MessageDriven`, etc).
- **Naming conventions**: If no machine-readable metadata is present in the code, often a naming convention will have been consciously or unconsciously adopted that can assist with finding those architecturally significant code elements. For example, perhaps all of your web controllers can be found by searching the codebase for all types in a particular namespace that match the name `*Controller`.
- **Type hierarchies**: Something else you might have done is to make all of your components extend a specific type, such as an `AbstractComponent` base class or

interface. Again, you could search for concrete implementation types and identify these as components.
- **Packaging conventions**: Alternatively, perhaps each sub-package or sub-namespace (e.g. `com.mycompany.myapp.components.*`) represents a component.
- **Module systems**: If a module system is being used (e.g. OSGi, Java 9, RequireJS, etc), perhaps each of the modules represents a component.
- **Build scripts**: Similarly, build scripts often create separate modules/JARs/DLLs from a single codebase and perhaps each of these represents a component.

Information is missing from the code

Ideally I'd like to auto-generate the entire software architecture model from the code, but this isn't currently realistic because most codebases don't include enough information about the software architecture to be able to do this. One step in the right direction is to enrich the information that we *can* get from the code, with that which we *can't* get from the code.

25.5 Architecture description languages

There have been a number of attempts to create Architecture Definition Languages (ADLs) that can be used to formally define the architecture of a software system. The basic idea is that you create a textual description of a software system rather than drawing diagrams. As a software developer, this sounds appealing. After all, we have plenty of tooling to work with text, it's easy to diff two versions of text and we can version control text easily. Again, this is about creating a model of your system, which is then exposed through different views to create one or more diagrams.

The problem with ADLs is that, based upon my experience anyway, they are rarely used in real-world projects. There are a number of reasons for this, ranging from typical real-world time and budget pressures through to the lack of perceived benefit from creating a formal description of a software system that isn't necessarily reflective of, or connected to, the source code. Unless you're building software in a safety critical or regulated environment, the Agile Manifesto statement of valuing working software over comprehensive documentation holds true.

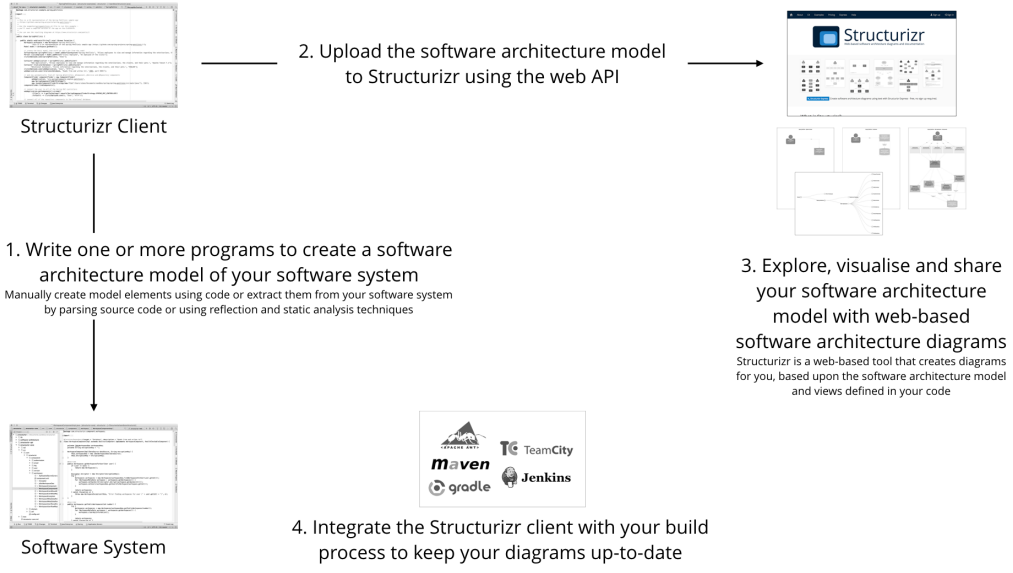
25.6 Structurizr

In my own attempt to solve this problem, I've created some tooling called Structurizr, which allows you to create a software architecture model with text, and keep it up to date

using static analysis and reflection techniques. Put simply, it's a way to create a software architecture model as code, and then have that model visualised by some simple tooling. There are two parts.

First is a client library that can be used to create a software architecture model using code. It's an **executable architecture description language** based upon the C4 model. There are currently two implementations, one for Java and one for .NET, both of which are open source. In summary, these client libraries provide a number of classes that implement the abstractions used in the C4 model: people, software systems, containers and components. To create a software architecture model, you create instances of these classes and connect them using a simple API.

The other half of the story is a web-based software as a service at structurizr.com, which includes free and paid services. Once you've created a software architecture model using the client library, you export it as a JSON document and upload it to structurizr.com, using a web API, which is able to visualise it.



Let's see how we might define a software architecture model for the Spring PetClinic.

Define the software architecture model

First is some boilerplate code to create a workspace, which itself contains a software architecture model.

```
Workspace workspace = new Workspace(
    "Spring PetClinic",
    "This is a C4 representation of the Spring PetClinic sample.");
Model model = workspace.getModel();
```

If I was going to draw a context diagram, it would simply consist of a single type of user (a clinic employee) using the Spring PetClinic system. With Structurizr, we can represent this in code as follows.

```
SoftwareSystem springPetClinic = model.addSoftwareSystem(
    "Spring PetClinic",
    "Allows employees to view and manage information regarding the
     veterinarians, the clients, and their pets.");

Person clinicEmployee = model.addPerson(
    "Clinic Employee", "An employee of the clinic");

clinicEmployee.uses(springPetClinic, "Uses");
```

Stepping down to containers, the Spring PetClinic system is made up of a Java web application that uses a database to store data. Again, we can represent this in code as follows (I've made some assumptions about the technology stack the system is deployed on).

```
Container webApplication = springPetClinic.addContainer(
    "Web Application",
    "Allows employees to view and manage information regarding the
     veterinarians, the clients, and their pets.", "Apache Tomcat 7.x");

Container relationalDatabase = springPetClinic.addContainer(
    "Relational Database",
    "Stores information regarding the veterinarians, the clients,
     and their pets.",
    "HSQLDB");

clinicEmployee.uses(webApplication, "Uses", "HTTP");
webApplication.uses(relationalDatabase,
    "Reads from and writes to", "JDBC, port 9001");
```

Stepping down again, we need to open up the web application to see the components inside it. Although we couldn't really get the two previous levels of abstraction from the codebase easily, we *can* get the components. All we need to do is understand what a "component" means in the context of this codebase. We can then use this information to help us find and extract them in order to populate the software architecture model.

The Spring MVC framework uses Java annotations (`@Controller`, `@Service` and `@Repository`) to signify classes as being web controllers, services and repositories respectively. Assuming that we consider these to be our architecturally significant code elements, it's a simple job of extracting these annotated classes (Spring Beans) from the codebase.

The client libraries include a **component finder**, which can be used to find components in a codebase. The Java version operates on the compiled byte code, using reflection, and can be customised by plugging in different component finder strategies. Pre-built strategies include those that look for types that follow specific naming conventions or inherit from specific base classes. There is also a Spring component finder strategy that understands how to find Spring components.

```
// and now automatically find all Spring @Controller, @Component, @Service and @Repository components
ComponentFinder componentFinder = new ComponentFinder(
    webApplication,
    "org.springframework.samples.petclinic",
    new SpringComponentFinderStrategy(
        new ReferencedTypesSupportingTypesStrategy()
    ),
    new SourceCodeComponentFinderStrategy(new File(sourceRoot, "/src/main/java/"), 150));

componentFinder.findComponents();
```

The `SpringComponentFinderStrategy` includes some rules that automatically collapse the interface and implementation of a Spring Bean, so the controllers, services and repositories are treated as "components" rather than a number of separate interfaces and classes. The dependencies between these components are also identified and extracted. In addition, the `SourceCodeComponentFinderStrategy` will parse the class-level Javadoc comment from the source file for inclusion in the model.

The final thing we need to do is connect the user to the web controllers, and the repositories to the database. This is easy to do since the software architecture model is represented in code.

```
// connect the user to all of the Spring MVC controllers
webApplication.getComponents().stream()
    .filter(c -> c.getTechnology().equals(SpringComponentFinderStrategy.SPRING_MVC_CONTROLLER))
    .forEach(c -> clinicEmployee.uses(c, "Uses", "HTTP"));

// connect all of the repository components to the relational database
webApplication.getComponents().stream()
    .filter(c -> c.getTechnology().equals(SpringComponentFinderStrategy.SPRING_REPOSITORY))
    .forEach(c -> c.uses(relationalDatabase, "Reads from and writes to", "JDBC"));
```

Define some views

With the software architecture model in place, we now need to create some views with which to visualise the model. The client library also includes a number of classes that correspond to the diagrams in the C4 model.

```
ViewSet viewSet = workspace.getViews();
```

First is the context diagram, which includes all people and all software systems.

```
SystemContextView contextView = viewSet.createSystemContextView(springPetClinic,
    "context", "The System Context diagram for the Spring PetClinic system.");

contextView.addAllSoftwareSystems();
contextView.addAllPeople();
```

Next is the container diagram.

```
ContainerView containerView = viewSet.createContainerView(springPetClinic,
    "containers", "The Containers diagram for the Spring PetClinic system.");

containerView.addAllPeople();
containerView.addAllSoftwareSystems();
containerView.addAllContainers();
```

And finally is the component diagram for the web application.

```
ComponentView componentView = viewSet.createComponentView(webApplication,
    "components", "The Components diagram for the Spring PetClinic web application.");

componentView.addAllComponents();
componentView.addAllPeople();
componentView.add(relationalDatabase);
```

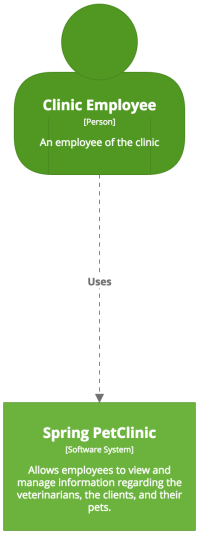
There are a few minor details omitted here for brevity (specifically related to styling the elements and relationships), but that's essentially all the code you need to create a software architecture model and views for this codebase. The full source code for this example can be found on the Structurizr for Java repository on GitHub.

You can then export the workspace as a JSON document and upload it to the Structurizr web API.

```
StructurizrClient structurizrClient = new StructurizrClient("key", "secret");
structurizrClient.putWorkspace(1, workspace);
```

Visualise the views

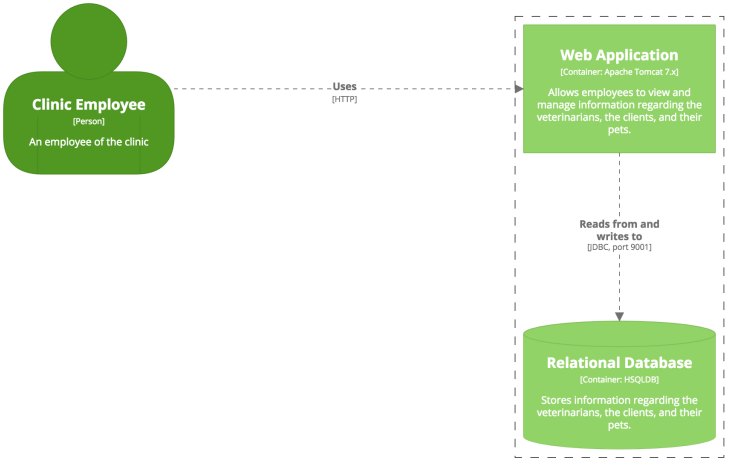
The result of visualising the Spring PetClinic model, after moving the boxes around (Structurizr doesn't provide any auto-layout facilities), is something like the following. Here's the context diagram.



System Context diagram for Spring PetClinic
The System Context diagram for the Spring PetClinic system.
Friday 18 November 2016 22:21 UTC

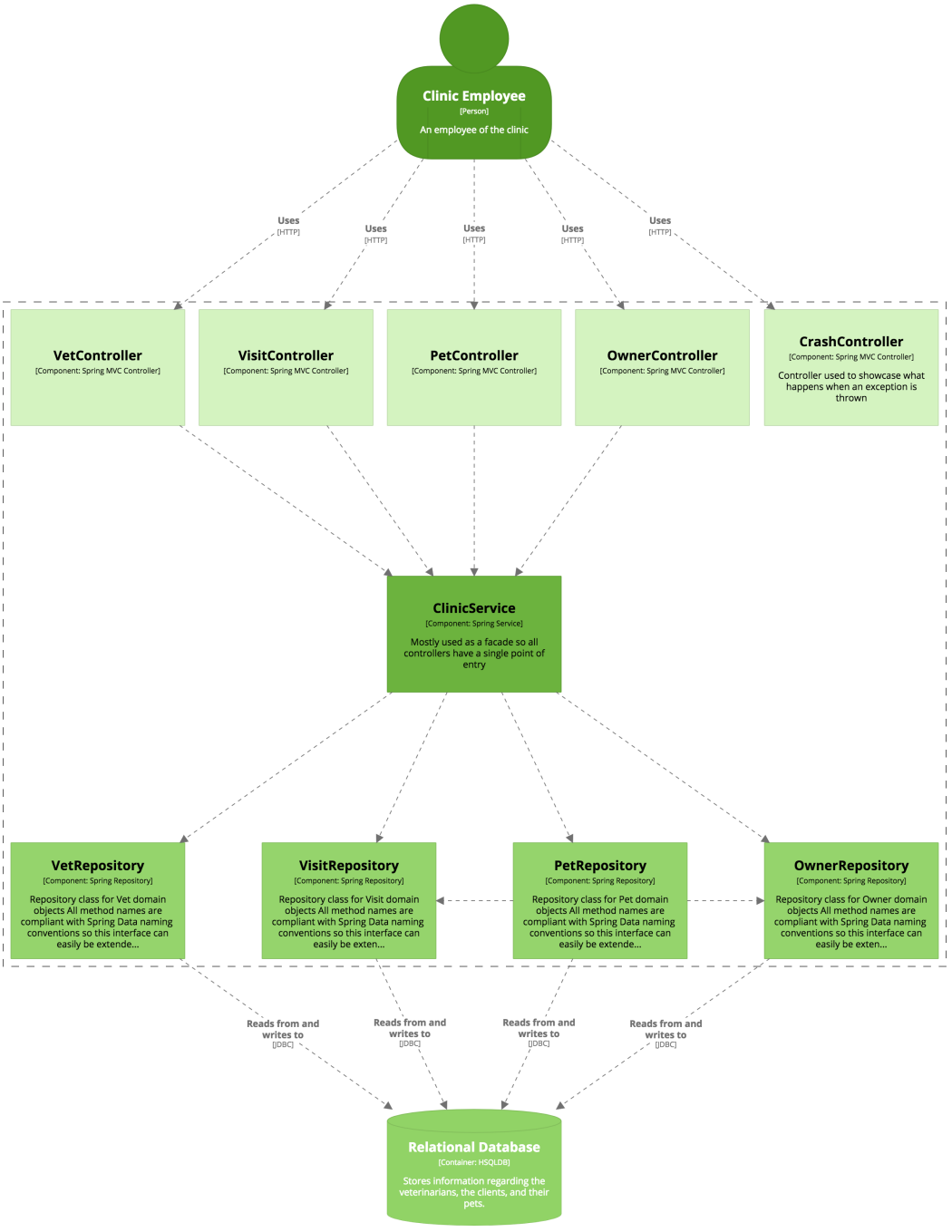
System context diagram for the Spring PetClinic system

Next is the container diagram.



Container diagram for the Spring PetClinic system

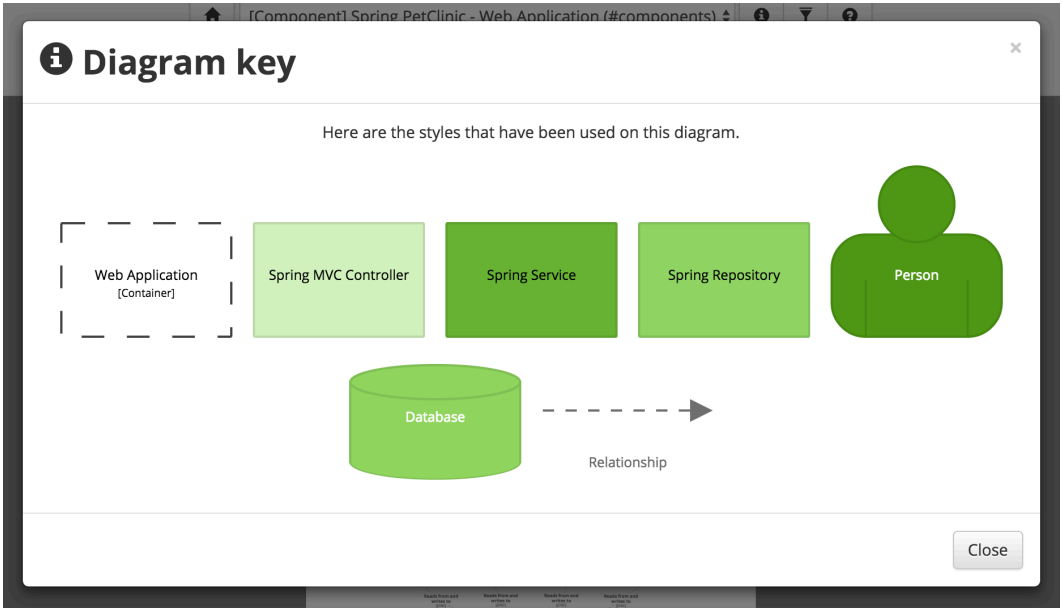
And finally we have the component diagram for the web application.



Component diagram for Spring PetClinic - Web Application
The Components diagram for the Spring PetClinic web application.
Friday 18 November 2016 22:21 UTC

Component diagram for the Spring PetClinic web application

Aside from changing the colour, size and position of the boxes, the graphical representation is relatively fixed. This in turn frees you up from messing around with creating static diagrams in drawing tools. In fact, Structurizr will automatically generate a diagram key too, based upon the definition of styles in your software architecture model.

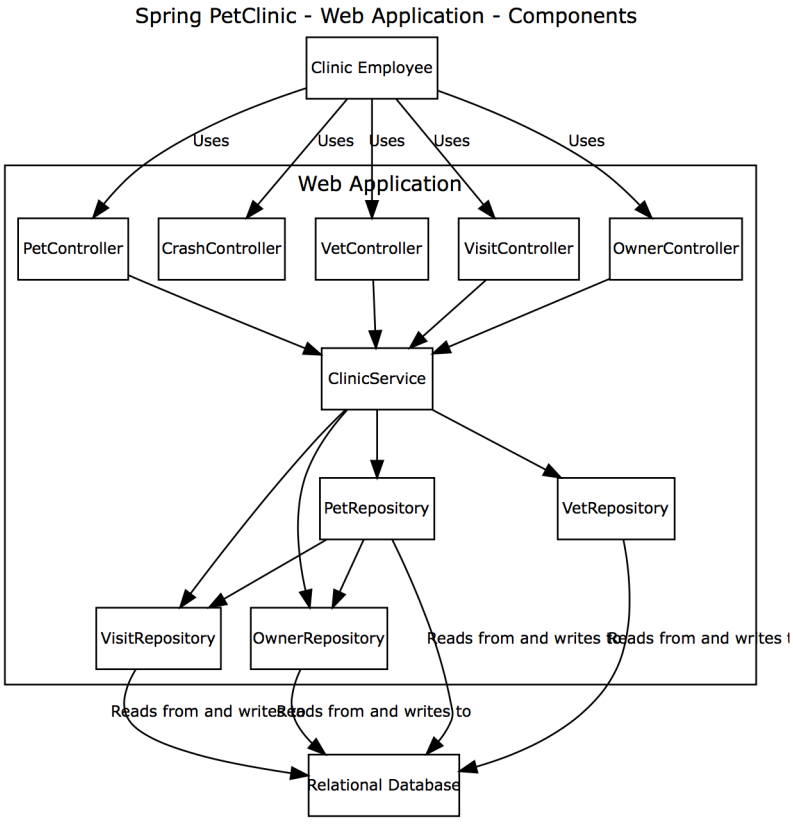


A diagram key for the component diagram

The live version of the diagrams can be found at structurizr.com and they allow you to double-click a component on the component diagram in order to navigate directly to the Spring PetClinic code that is hosted on GitHub. This in turn links the software architecture diagrams with the code. If the code we've just seen was integrated with a continuous build environment, your software architecture model remains to date when components are added, removed or modified.

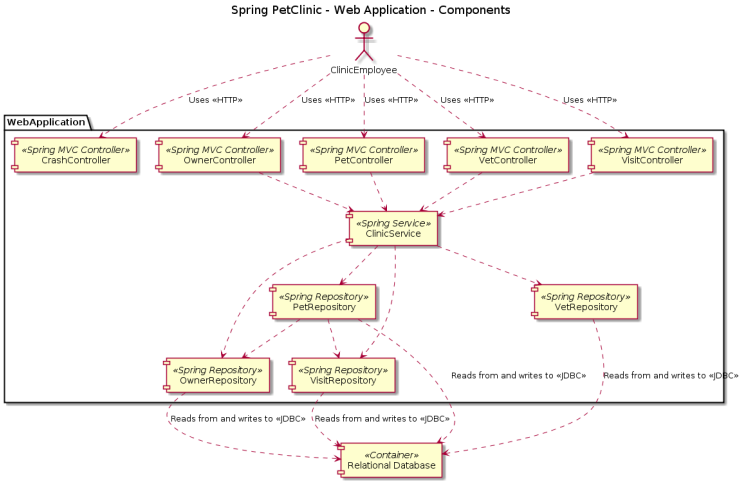
Alternative visualisations

It's worth pointing out that Structurizr is my vision of what I want from a simple software architecture diagramming tool, but you're free to take the output from the open source library and create your own tooling to visualise the model. For example, the Structurizr for Java client library includes a "DOT file writer" that uses Cyrille Martraire's dot-diagram library to output the views in a format that can be rendered by Graphviz.



Component diagram for the Spring PetClinic web application, rendered using Graphviz

There's also a writer that outputs the views to a format that can be rendered with PlantUML. The benefit of using PlantUML in conjunction with the open source Structurizr client library is that you are creating diagrams based upon a model, so you can be sure that elements remain consistently named across those diagrams.



Component diagram for the Spring PetClinic web application, rendered using PlantUML

Alternatively, you could export the Structurizr model to an XMI format (for importing into UML tools), a desktop app, IDE plugins, etc. The choice is yours.

Component findability

It's worth pointing out that the technique I've just described for automatically identifying components in a codebase only tends to work when the codebase is well structured and has a high degree of consistency. That might sound obvious, but let's take a look in more detail.

Given the codebase that you're working on now, how many rules would you need to use to correctly identify components in your codebase? For the Spring PetClinic example, the rule set is very simple:

- Find all classes that are annotated with the Spring MVC @Controller annotation.
- Find all classes that are annotated with the Spring @Service annotation.
- Find all classes that are annotated with the Spring @Repository annotation.

Using a framework like Spring has introduced its own set of rules, guidelines and conventions to the codebase, so we can simply use those as a basis for identifying components. If you weren't using a framework like Spring, perhaps you would need to take a different approach, and use rules like the following:

- Find all classes where the name of the class ends in the word "Controller".

- Find all classes where the name of the class ends in the word "Component".
- Find all classes where the name of the class ends in the word "Service".
- Find all classes that inherit from the AbstractComponent base class.
- Find all classes that implement the DataAccessObject interface.

Again, this rule set is still relatively small. On codebases where the development team hasn't been so disciplined about following principles and writing code in a consistent way, you might start to see rules like the following:

- Find all classes where the name of the class ends in the word "Component", excluding classes in package/namespace X because, despite their name, they are not really components.
- Find all classes annotated with Spring's @RestController annotation. These are all API components, with the exception of types X, Y and Z that actually return HTML rather than JSON.

Component identification rule sets containing many exceptional cases perhaps suggest a lack of consistency in your codebase, and that some refactoring is necessary.

"Software architecture as code" opens opportunities

Having the software architecture model as code, as an executable architecture description language, opens a number of opportunities. There are the obvious benefits of being able to create the model using static analysis and reflection, to extract components automatically from a codebase and keep this up to date when the code changes.

In addition, since the model is an object graph, you can slice and dice the model to produce a number of different views as necessary. For example, showing all components for even a small application will result in a very cluttered diagram. Instead, you can simply write some code to programmatically create a number of smaller, simpler diagrams; perhaps one per vertical slice based upon a web controller, user story, functional grouping, feature set, DDD aggregate, etc. You can also opt to include or exclude any elements as necessary. I typically exclude logging components because they tend to be used by every other component and serve no purpose other than to clutter the diagram. To do this, you simply remove the element from the views.

Since the models are code, they are also versionable alongside your codebase and can be integrated with your automated build system to keep your models up to date. This provides accurate, up-to-date, living software architecture diagrams that actually reflect the code.

25.7 Minimise the model-code gap

However you choose to visualise your software architecture, whether that's using whiteboards, Microsoft Visio, a modeling tool or something like Structurizr, you do need to consider how your software architecture reflects your code and vice versa. The model-code gap needs to be as small as possible in order to get meaningful diagrams that do reflect the code.

And it's here that we face two key challenges. First of all, we need to get people thinking about software architecture once again so that they are able to think about, describe and discuss the various structures needed to reason about a large and/or complex software system. And, secondly, we need to find a way to get these structures into the codebase using architecturally-evident coding styles, to ensure that the model-code gap is minimised. As an industry, I think we still have a long way to go but, in time, I hope that the thought of using a drawing tool like Microsoft Visio to manually create software architecture diagrams will seem ridiculous.

26. The C4 model with other notations and tools

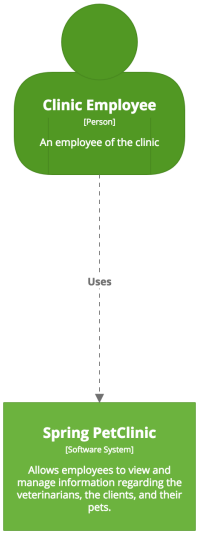
For completeness, let's look at some alternative ways to describe software architecture. For this purpose we'll use the Spring PetClinic.

26.1 Boxes and lines

First of all, let's recap how to describe the software architecture using the C4 model and the notation that I use, both on whiteboards and with Structurizr.

System context diagram

First of all is the system context diagram.



System context diagram for the Spring PetClinic system

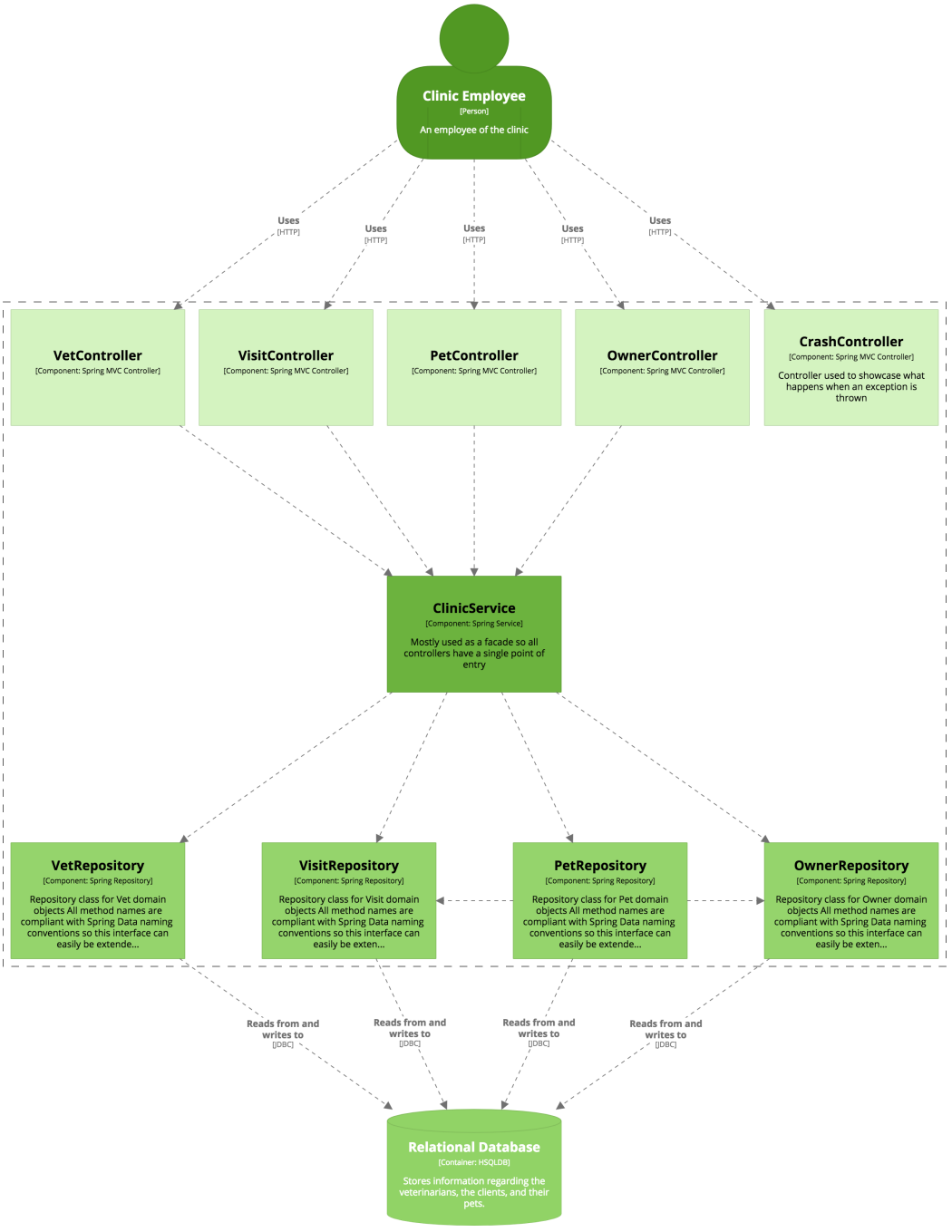
Container diagram

Next we have the container diagram.

Container diagram for the Spring PetClinic system

Component diagram for the web application

And finally we have a component diagram for the web application.



Component diagram for Spring PetClinic - Web Application
The Components diagram for the Spring PetClinic web application.
Friday 18 November 2016 22:21 UTC

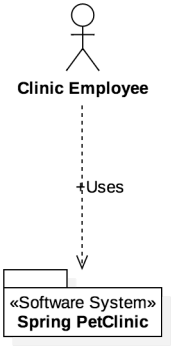
Component diagram for the Spring PetClinic web application

26.2 UML (with a modeling tool)

Many people find the system context and container diagrams a useful way to communicate software architecture, although the Unified Modeling Language doesn't really have diagram types that correspond exactly to these concepts. One common approach is to use a UML component diagram, with specific stereotypes added for clarity. Although this may not be considered "proper" usage of UML, the diagrams still have some value. Let's look at some examples.

System context diagram

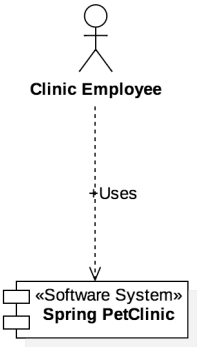
Here's what a system context diagram with UML might look like.



System context diagram for the Spring PetClinic system

The UML package symbol is being used to represent a software system and I've added my own `Software System` stereotype, although it seems common to use the standard `Subsystem` stereotype too. Adding a stereotype is useful to avoid having conversations explaining what the UML package symbol means and why you've used it to represent a software system. Since we're stereotyping the elements to indicate what they are, there's really no benefit to using the specific UML package notation (the folder shape). Some tools will allow you to modify the stereotypes and shapes using UML profiles, but that's out of the scope of this book.

An alternative is to use UML components to represent software systems, which can again be stereotyped to remove ambiguity.

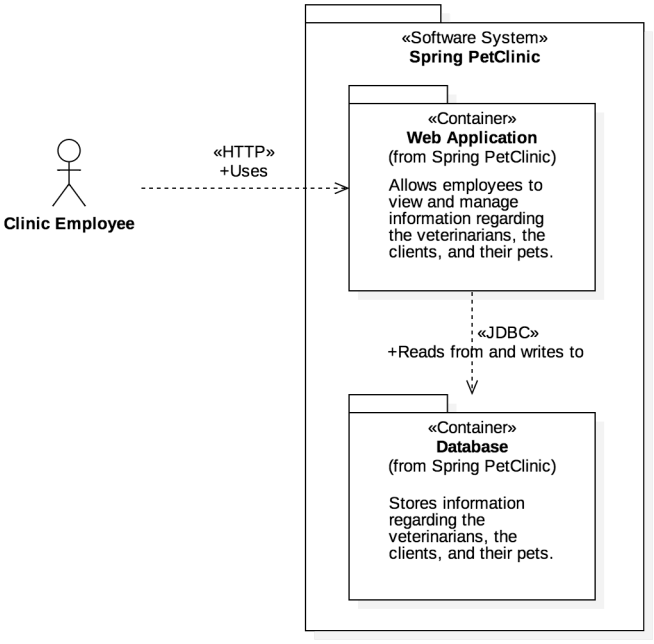


System context diagram for the Spring PetClinic system

Once again, there's really no benefit in having the UML component notation (the two boxes protruding from the left-side). Another alternative (not shown), is that you could use a UML use case diagram to summarise the system context, showing actors using the major use cases that the system implements. I don't think this works very well once you get more than a small number of use cases though.

Container diagram

A container diagram can be drawn in a similar way using a UML component diagram.



Container diagram for the Spring PetClinic system

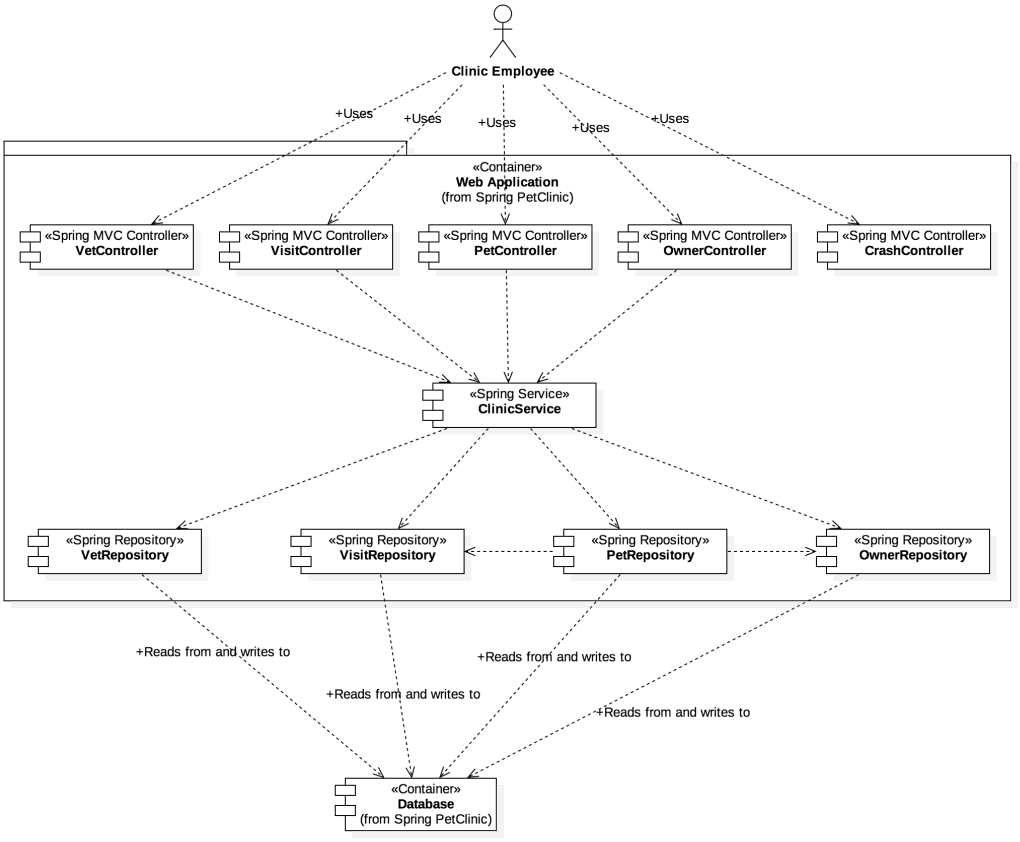
You'll notice that I've used a UML package as a way to group the containers and illustrate the software system boundary. Many tools will allow you to define packages in a hierarchical way. In this example, the `Web Application` package resides inside the `Spring PetClinic` package. Again, I've used some custom stereotypes on the elements and relationships.

The ability to add supplementary text to the diagram elements (e.g. to show container responsibilities) varies from tool to tool. If you're using UML components to represent containers, you can use operations as a way to add some documentation, although these are often rendered with a `()` suffix after the operation name. In other tools you can add notes, which are rendered as small rectangle with a corner "folded over". The use of many notes and connecting lines can quickly lead to cluttered diagrams though.

For this particular example, the tool that I'm using (StarUML) allows me to add arbitrary text to a diagram, and float it over elements on the diagram. It's basically just free-text, and not a part of the underlying model. It also isn't connected or grouped with the element on the diagram, which means that you need to make sure you move both the element and floating text if you want to reposition or resize it.

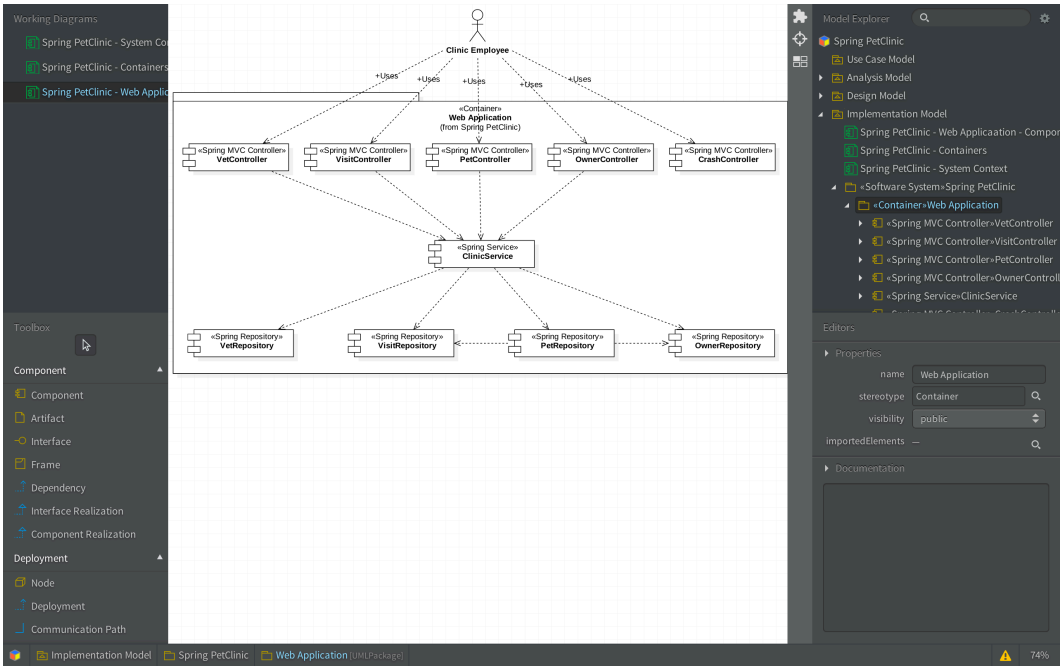
Component diagram for the web application

And last is the component diagram.



Component diagram for the Spring PetClinic web application

Again, I've used a package as the web application boundary and, from a model perspective, the components actually reside in the package. You can see this by looking at the top-right of the following screenshot.



StarUML

A summary of using UML with a modeling tool

The process of creating models and diagrams with UML tools can be quite exhausting because you usually need to click and right-click around much of the user interface in order to create elements in the model and set properties on those elements. It's a very manual process but there's really no other way to get data into these tools.

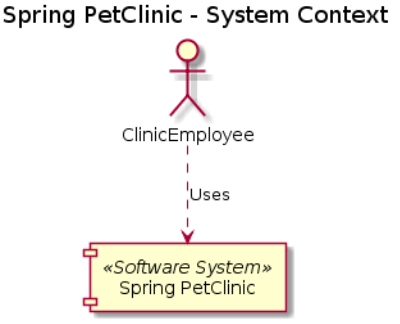
In addition, some UML tools are much more restrictive than others and you might find that, for example, you can't add packages to a component diagram. Alternatively, the tool will allow you to do this, but issue validation warnings or errors. If you find that you're needing to bend the UML notation or tooling in order to tell the story that you would like to tell, I suggest switching tools or notation.

26.3 UML (with PlantUML)

PlantUML is a textual implementation of the Unified Modeling Language and allows you to create single UML diagrams as text.

System context diagram

Here's what a system context diagram created with PlantUML looks like.



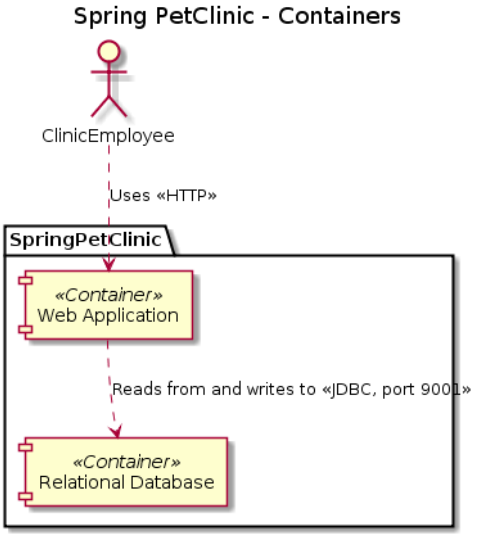
System context diagram for the Spring PetClinic system

This is created using the following text, which you can either paste into PlantUML online, or you can install the tools on your local computer.

```
@startuml
title Spring PetClinic - System Context
[Spring PetClinic] <<Software System>> as SpringPetClinic
actor ClinicEmployee
ClinicEmployee ..> SpringPetClinic : Uses
@enduml
```

Container diagram

A container diagram looks like this.



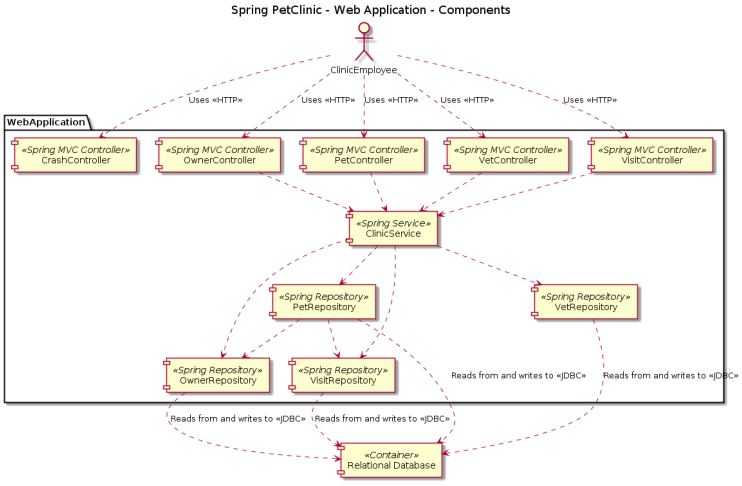
Container diagram for the Spring PetClinic system

Here's the textual definition.

```
@startuml
title Spring PetClinic - Containers
actor ClinicEmployee
package SpringPetClinic {
 [Web Application] <<Container>> as WebApplication
 [Relational Database] <<Container>> as RelationalDatabase
}
ClinicEmployee ..> WebApplication : Uses <<HTTP>>
WebApplication ..> RelationalDatabase : Reads from and writes to <<JDBC, port 9001>>
@enduml
```

Component diagram for the web application

And, finally, here's the component diagram for the web application.



Component diagram for the Spring PetClinic web application

The textual definition is starting to get long now.

```
@startuml
title Spring PetClinic - Web Application - Components
actor ClinicEmployee
[Relational Database] <<Container>> as RelationalDatabase
package WebApplication {
 [ClinicService] <<Spring Service>> as ClinicService
 [CrashController] <<Spring MVC Controller>> as CrashController
 [OwnerController] <<Spring MVC Controller>> as OwnerController
 [OwnerRepository] <<Spring Repository>> as OwnerRepository
 [PetController] <<Spring MVC Controller>> as PetController
 [PetRepository] <<Spring Repository>> as PetRepository
 [VetController] <<Spring MVC Controller>> as VetController
 [VetRepository] <<Spring Repository>> as VetRepository
 [VisitController] <<Spring MVC Controller>> as VisitController
 [VisitRepository] <<Spring Repository>> as VisitRepository
}
ClinicEmployee ..> CrashController : Uses <<HTTP>>
ClinicEmployee ..> OwnerController : Uses <<HTTP>>
ClinicEmployee ..> PetController : Uses <<HTTP>>
ClinicEmployee ..> VetController : Uses <<HTTP>>
ClinicEmployee ..> VisitController : Uses <<HTTP>>
ClinicService ..> OwnerRepository
ClinicService ..> PetRepository
ClinicService ..> VetRepository
ClinicService ..> VisitRepository
OwnerController ..> ClinicService
OwnerRepository ..> RelationalDatabase : Reads from and writes to <<JDBC>>
```

```
PetController ..> ClinicService
PetRepository ..> OwnerRepository
PetRepository ..> RelationalDatabase : Reads from and writes to <<JDBC>>
PetRepository ..> VisitRepository
VetController ..> ClinicService
VetRepository ..> RelationalDatabase : Reads from and writes to <<JDBC>>
VisitController ..> ClinicService
VisitRepository ..> RelationalDatabase : Reads from and writes to <<JDBC>>
@enduml
```

A summary of using UML with PlantUML

PlantUML is a useful tool and you can create diagrams very quickly once you understand the simple language. It does allow you a degree of control over the visual style of the diagrams, while the layout of elements is taken care of automatically by graphviz behind the scenes.

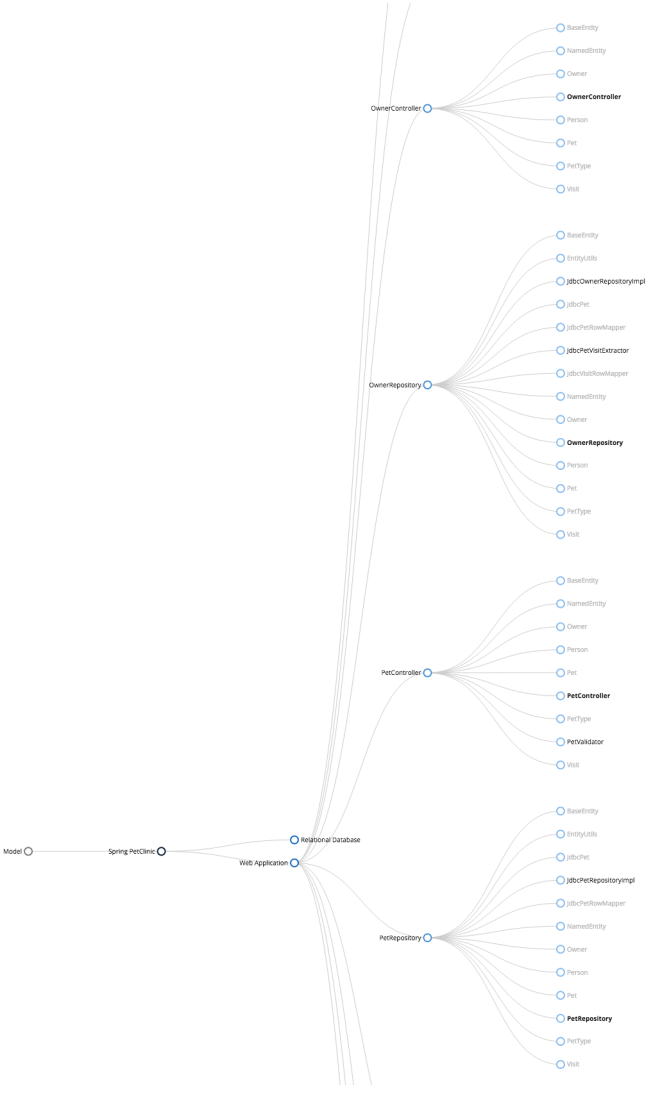
If you need to sketch a quick UML diagram to include in some documentation, PlantUML might be a good choice. It's not, however, a modeling tool where you create views on top of an underlying model.

27. Exploring your software architecture model

Once you do have a model of your software architecture, you can visualise and explore it in a number of ways.

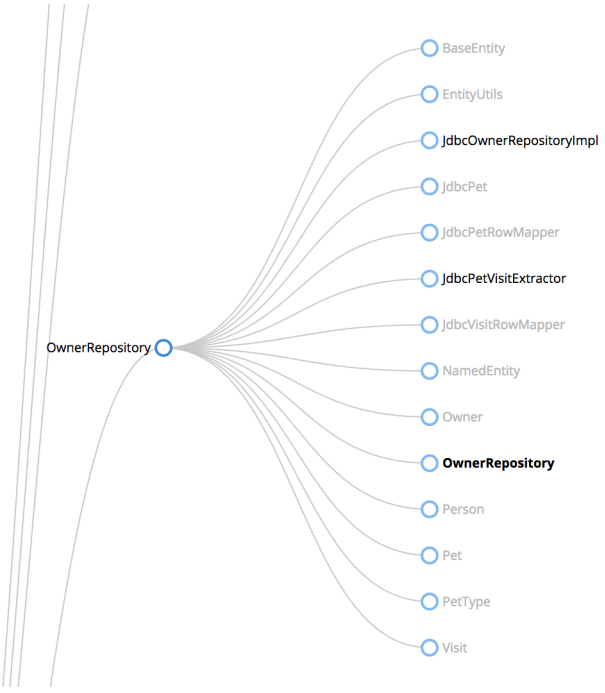
27.1 Static structure

The majority of this book has discussed the static structure of software systems, based upon a set of simple hierarchical abstractions of software systems, containers, components and classes. Since this is a tree structure, it's easy to visualise. Here's a visualisation showing the building blocks that make up the Spring PetClinic system.



A visualisation of the static structure

If we zoom in on the `OwnerRepository`, we can see the code level elements (interfaces and classes, in this case) that make up that component.

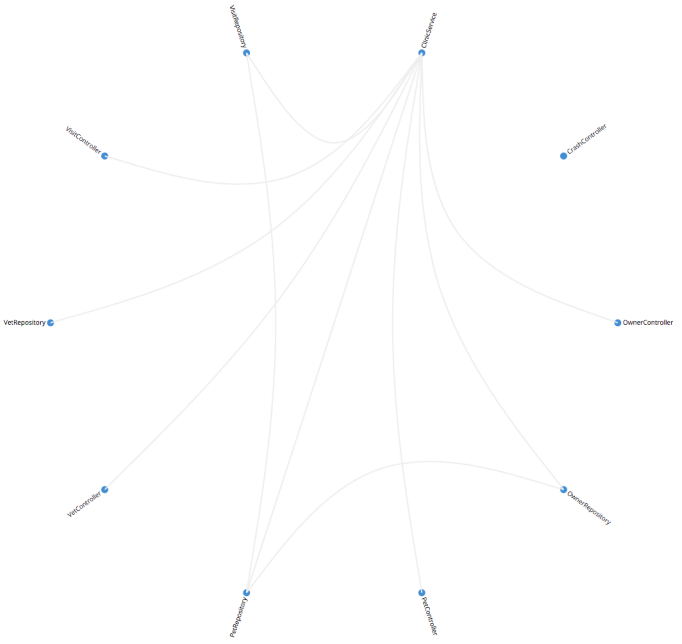


A zoom-in of the OwnerRepository component

Some styling has been added to provide more information about the code level elements. The bold `OwnerRepository` is the type that has been associated with the component, which in this case is a Java interface. The `JdbcOwnerRepositoryImpl` and `JdbcPetVisitExtractor` classes are unique to this particular component, and the faded names represent types that are shared between components.

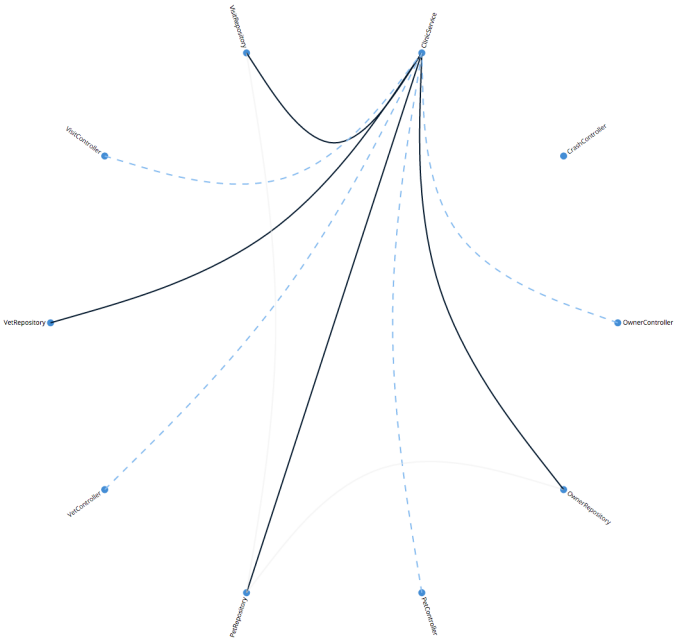
27.2 Dependency maps

Once you have a set of components in your software architecture model, especially if they are automatically extracted from the code, you can create a dependency map. Here's a dependency map of the components within the Spring PetClinic web application.



A dependency map of components

This diagram shows all of the components within the Spring PetClinic web application, and all of the relationships between them. Given that we have a model, we can also see the afferent (inbound) and efferent (outbound) dependencies for a given component.

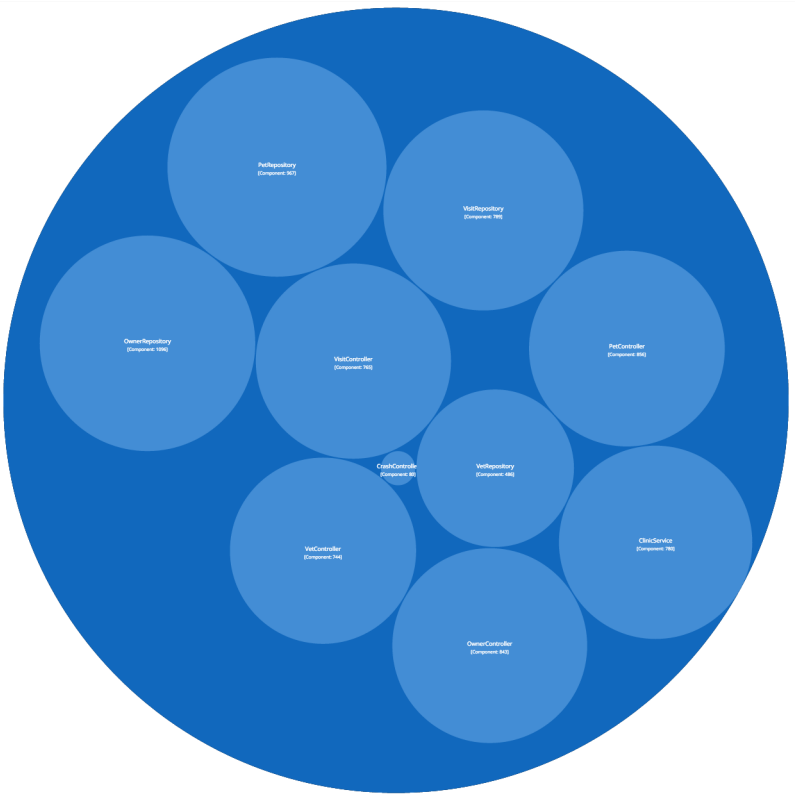


The inbound and outbound dependencies for the ClinicService component

This is the same diagram, but with the inbound (light blue, dashed) and outbound (dark blue, solid) dependencies highlighted for the `ClinicService` component. You can also additionally highlight direct cyclic dependencies between two components if they exist.

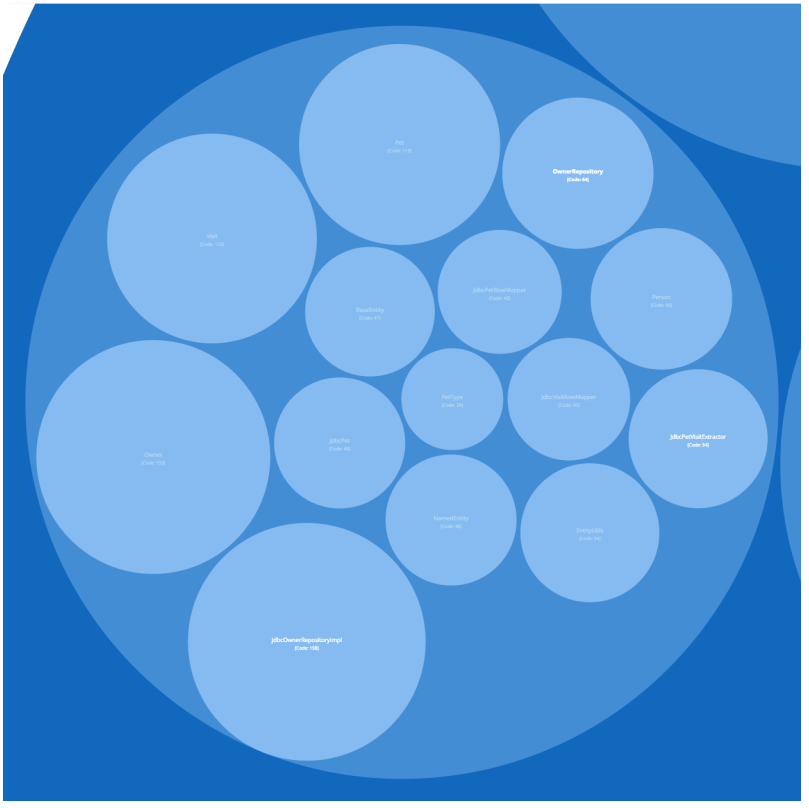
27.3 Component size or complexity

If your software architecture model includes information based upon the source code, such as a measure of the size and/or complexity of the various code level elements that make up each component, you can create a simple visualisation of this information too. Here are the components in the Spring PetClinic web application, where each component has been ranked based upon size (i.e. number of lines of source code).



Components ranked based upon size

If we zoom in further, we can see how the lines of source code are distributed across the code level elements that make up each component.



Code level elements that make up the OwnerRepository component, ranked based upon size

27.4 Other ways to explore

All of the visualisations we've just seen were created automatically using Structurizr's Free Plan, where the component model was automatically extracted from the code using static analysis and reflection. Frameworks like D3 make it relatively straightforward to create new visualisations from a model in a few lines of JavaScript, and there are a number of other tools that allow you to explore a codebase too.

- jQAssistant is a tool that performs static analysis of your codebase, and loads the resulting model into a neo4j graph database. You can then define rules based upon the static structure (the rules are expressed as queries in the Cypher language), and report violations of these rules as a part of your build process if you want to.

- CodeScene is a static analysis tool with a difference. In addition to performing static analysis of a codebase (with support for a number of different programming languages), CodeScene creates a model that also includes the human and social factors, based upon information from your source code control system. For example, it will allow you to see which parts of the codebase always change together, and which parts don't but perhaps should. In essence, it's an implementation of the ideas in Adam Tornhill's Your Code as a Crime Scene book.

A model of your software system is so much more powerful than a collection of static diagrams on a whiteboard or in Microsoft Visio. Once you have a model, you can explore it.

28. Appendix A: Financial Risk System

28.1 Background

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

Trade Data System

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

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

Reference Data System

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

28.2 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.

28.3 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.