The C4 model

for visualising software architecture

Simon Brown

The C4 model for visualising software architecture

Simon Brown

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

This version was published on 2023-02-24

Tweet This Book!

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

The suggested hashtag for this book is #sa4d.

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

#sa4d

For Kirstie, Matthew and Oliver

Contents

1. About the book

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 the present day, and creating software architecture diagrams 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.

2. About the author

I'm an independent software development consultant specialising in software architecture; specifically technical leadership, communication and lightweight, pragmatic approaches to software architecture. In addition to being the author of Software Architecture for Developers, I'm the creator of the C4 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.

3. 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.

Those architecture diagrams you have on your office wall or wiki - 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? If diagrams are to be useful, they need to reflect reality.

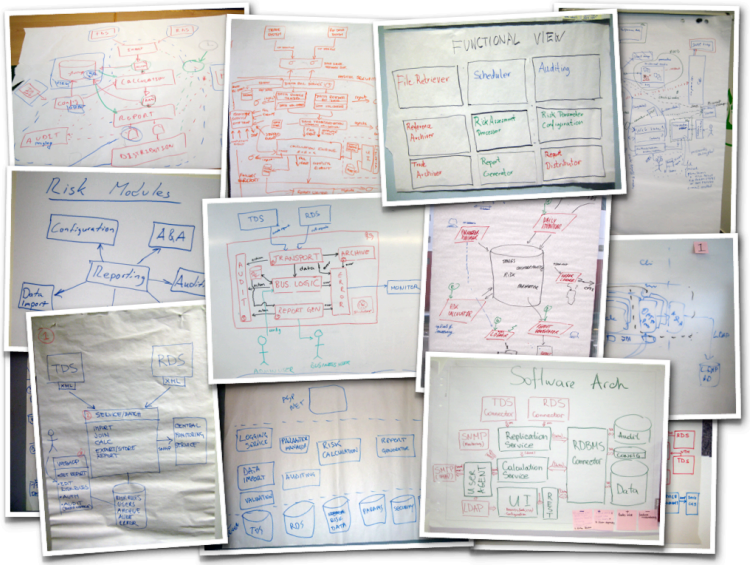
Through my training and workshops, I've asked thousands of software developers around the world to draw software architecture diagrams over the past decade, and continue to do so today. The results still surprise me, anecdotally suggesting that effective visual communication of software architecture is a skill that's sorely lacking in the software development industry. We've forgotten how to visualise software architecture; both during the software design/development process and for long-lived documentation.

3.1 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 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 related 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 architecture 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 the typical software architecture diagrams that I see on my travels and, as we'll see in the next chapter, they suffer from a number of problems.

3.2 A lightweight approach

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

way. 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" of the organisation/environment in which it exists.
- Create a shared vision for the development team.
- Provide a "map" that can be used by software developers to navigate the source code.
- Provide a point of focus for technical conversations about new features, technical debt, risk reviews, etc.
- Fast-track the on-boarding of new software developers into the team.
- Provide a way to explain what's being built to people outside of the development team, whether they are technical or non-technical.

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. Moving fast requires good communication.

3.3 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.

3.4 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:

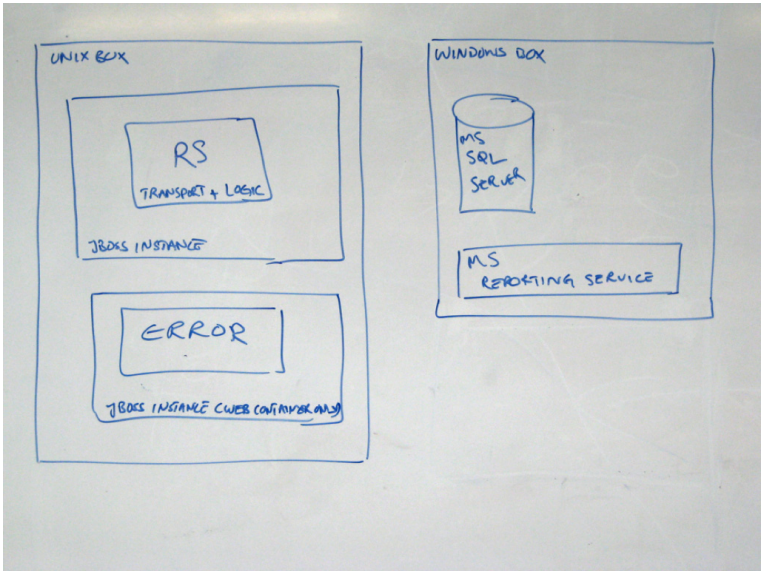
1. In their haste to adopt agile approaches in recent years, many software teams have "thrown out the baby with the bath water". Modeling and documentation have been thrown out alongside traditional plan-driven processes and methodologies. That may sound a little extreme, but many of the software teams 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.

3.5 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.

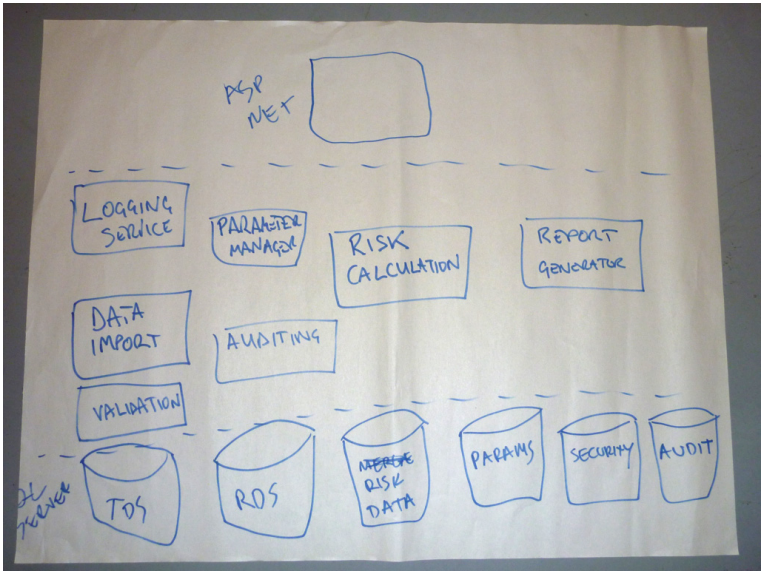


There's a Unix box and a Windows box, with some additional product selections that include JBoss (a Java EE application server) and Microsoft SQL Server. The problem is, I don't know

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.

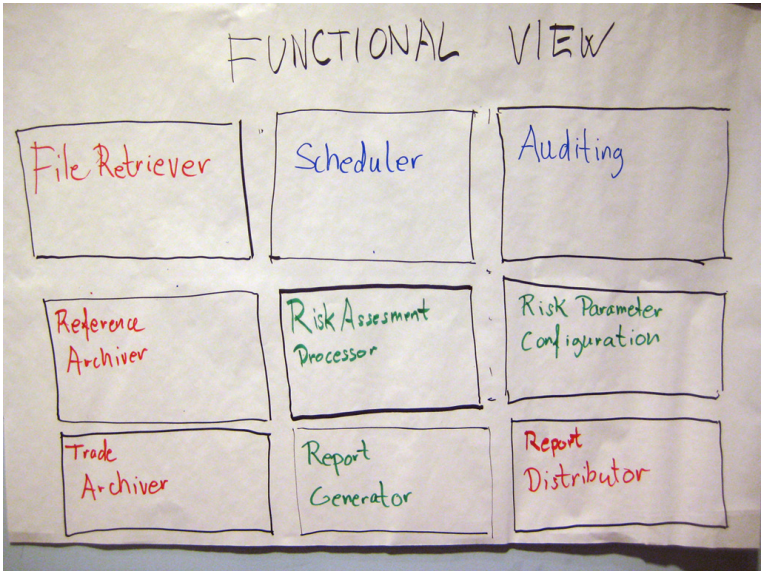


This is a three-tier solution (I think) that uses the Microsoft technology stack. There's an ASP.NET web application at the top, which I assume is being used for some sort of user interaction, although that's not shown on the diagram. The bottom section is labelled "SQL Server" and there are lots of 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.

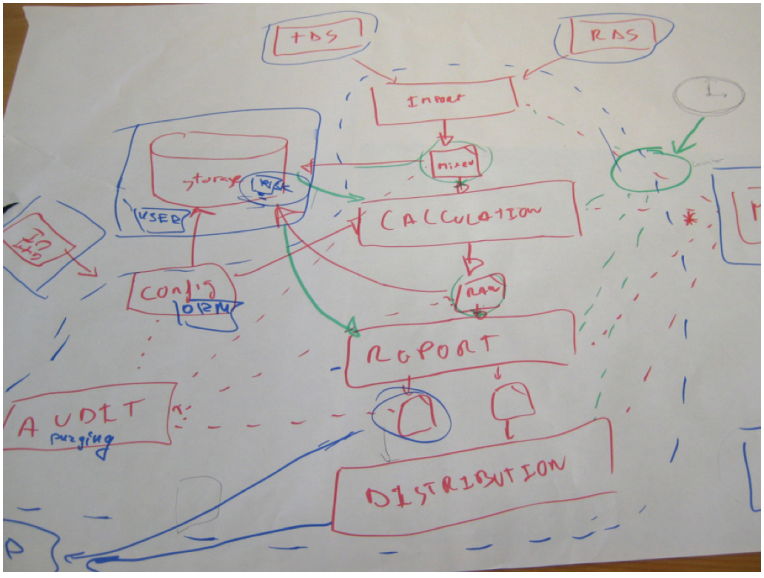


Essentially the group that produced this diagram has simply documented their functional decomposition of the solution into a number of smaller things. Imagine a building architect drawing you a diagram of your new house that simply had a collection of boxes labelled "Cooking", "Eating", "Sleeping", "Relaxing",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



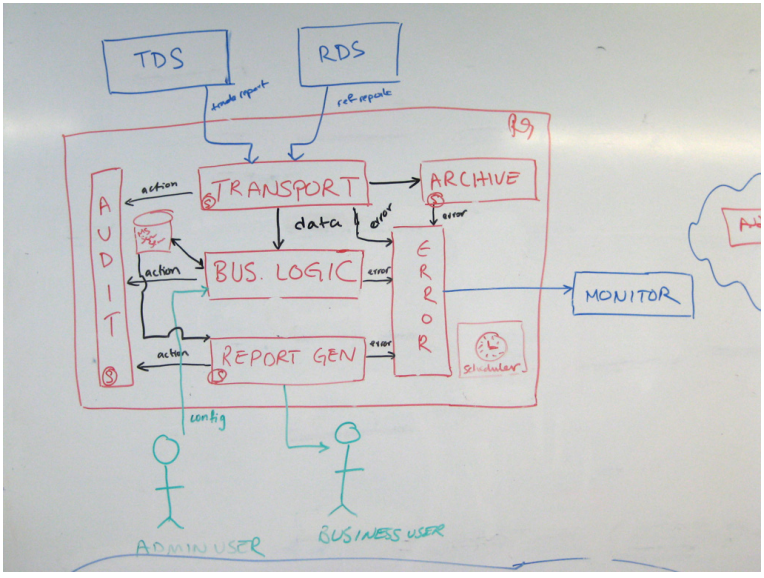
The central spine of this diagram is great because it shows how data comes in from the source data systems (TDS and RDS) and then flows through a series of steps to import the data, perform some calculations, generate reports and finally distribute them. It's a super-simple activity diagram that provides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

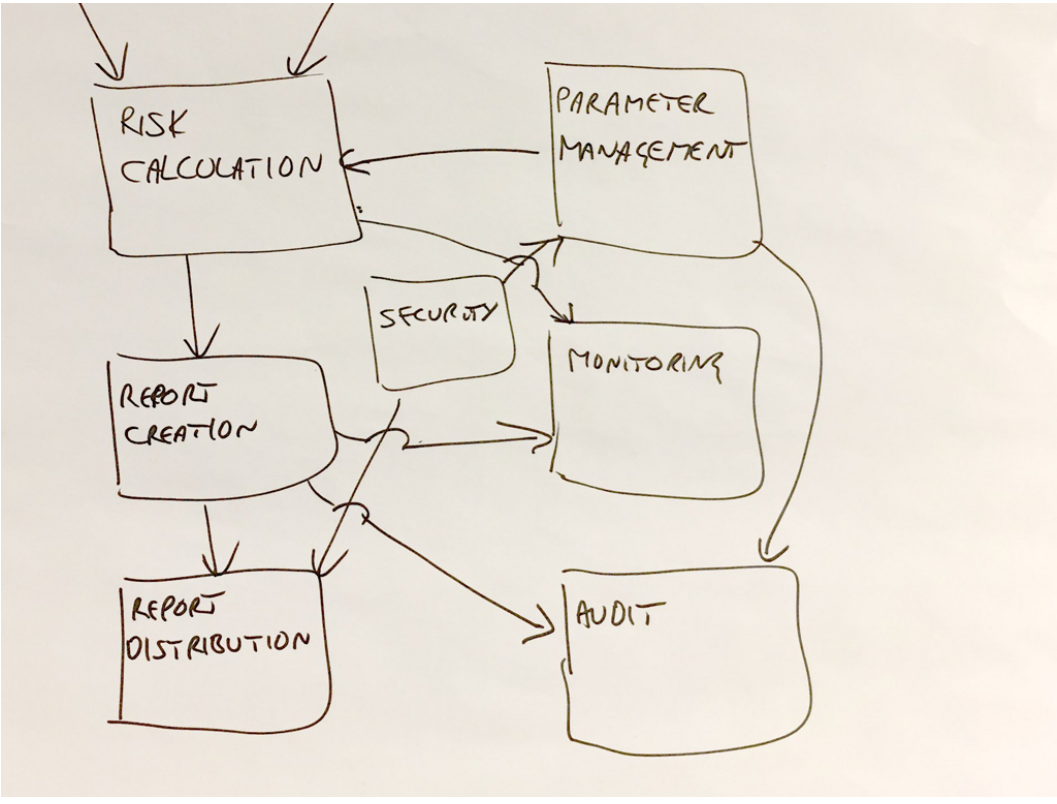
This is another very common style of diagram. Next time somebody asks you to produce a software architecture diagram of a system, present them this photo and you're done!



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"

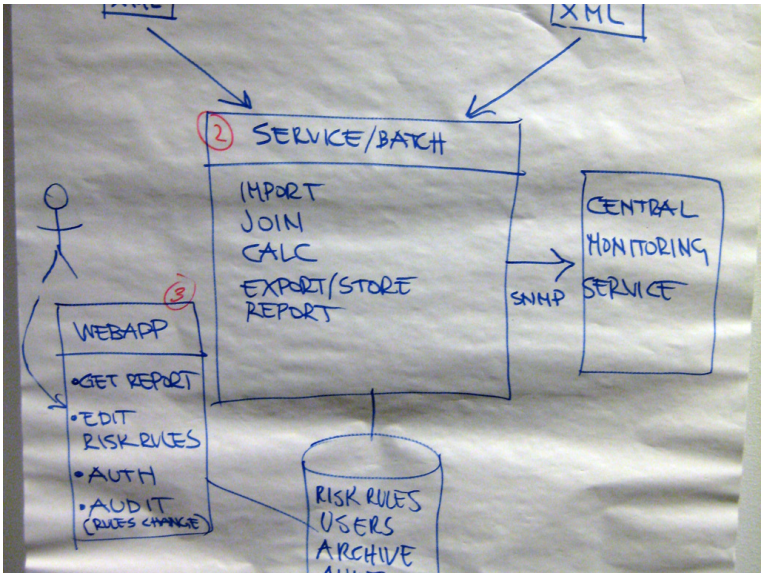
This diagram is also relatively common. It shows the logical (or conceptual, or functional) building blocks that the software system is comprised of, but offers very little information other than that.



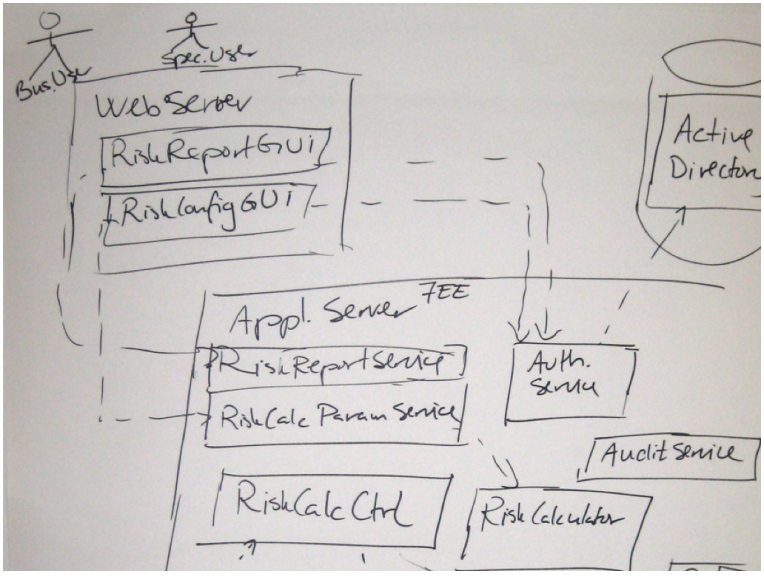
There's a common misconception that "software architecture" diagrams should be "logical" in nature rather than include any references to technology or implementation details, especially before any code is written. We'll look at this later in the book.

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.



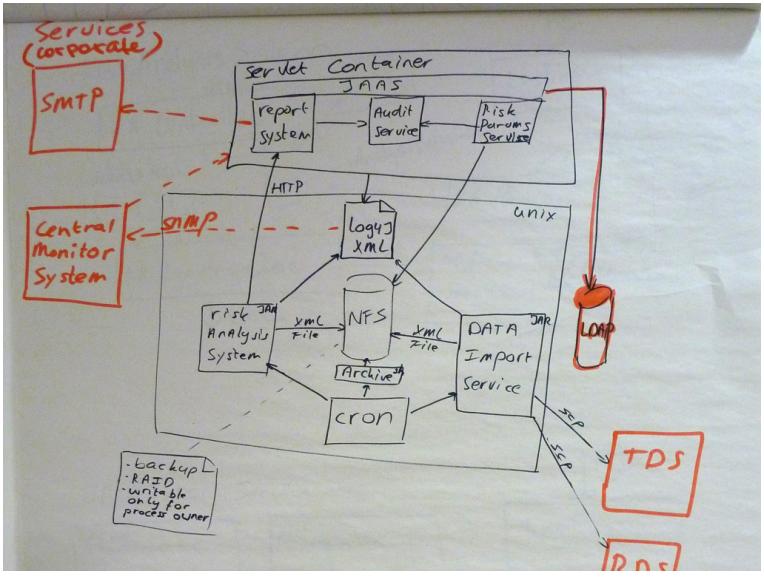
The lines between the web server and the application server have no information about how this communication occurs. Is it SOAP? A JSON web API? XML over HTTPS? Remote

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.



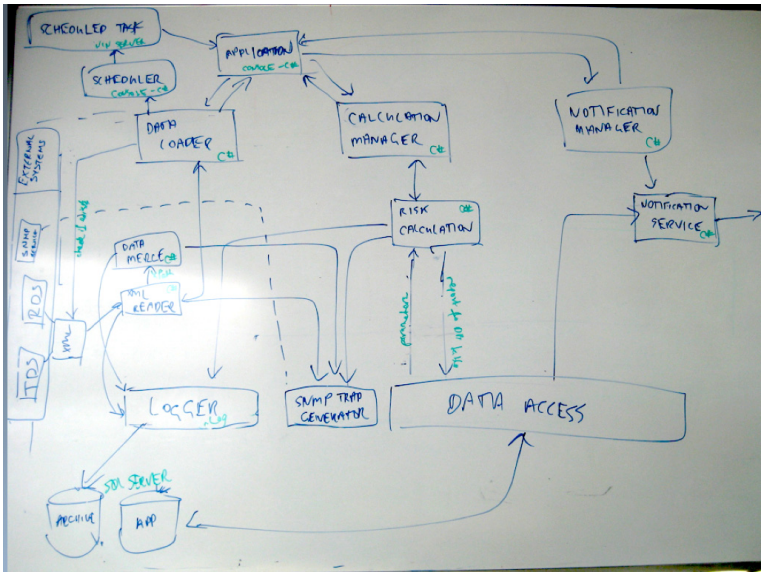
If you look at the Unix box in the centre of the diagram, you'll see two smaller boxes labelled "Risk Analysis System" and "Data Import Service". If you look closely, you'll see that both boxes are annotated "JAR", which is the deployment mechanism for Java code (Java ARchive). Basically this is a ZIP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

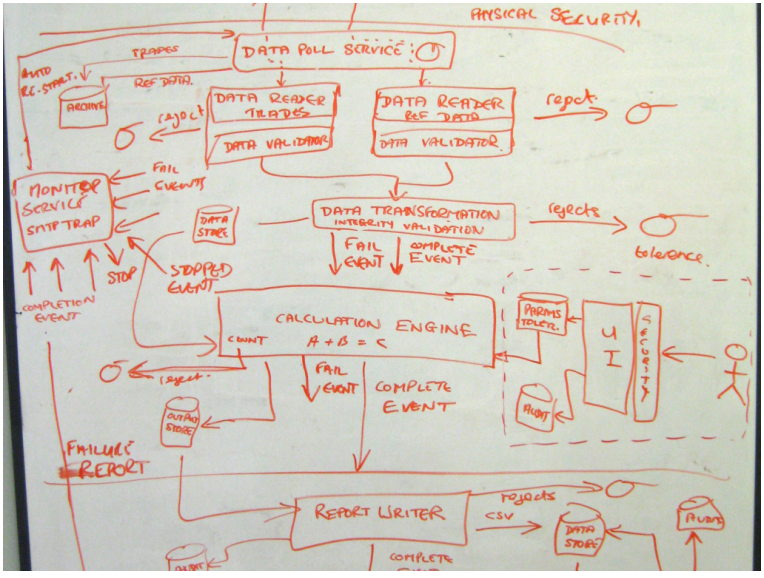


In the bottom left of the diagram is a SQL Server database, and at the top left of the diagram is a box labelled "Application". Notice how that same box is annotated (in green) "Console-C#". Basically, this system seems to be made up of a C# console application and a database. But what about the other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



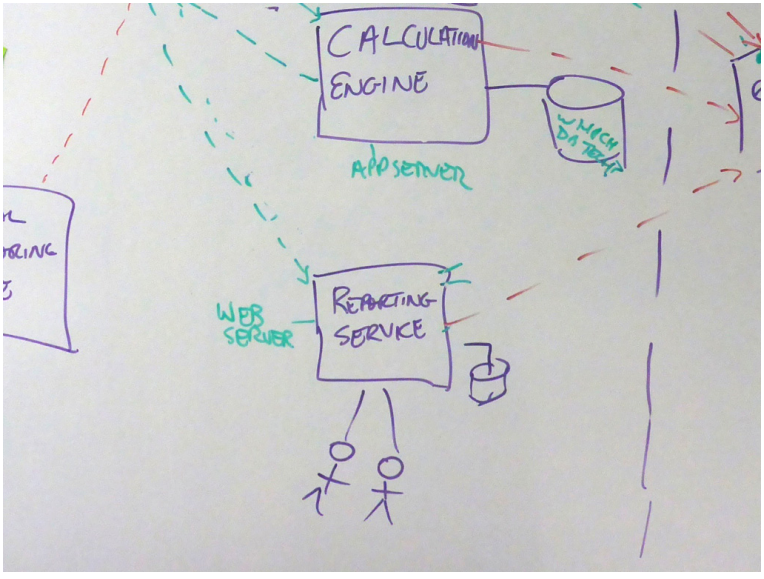
It's a little like those "choose your own adventure" books that I used to read as a kid. You would start reading at page 1 and eventually arrive at a fork in the story where you decide what should happen next. If you want to attack the big scary creature you've just encountered, you turn to page 47. 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.

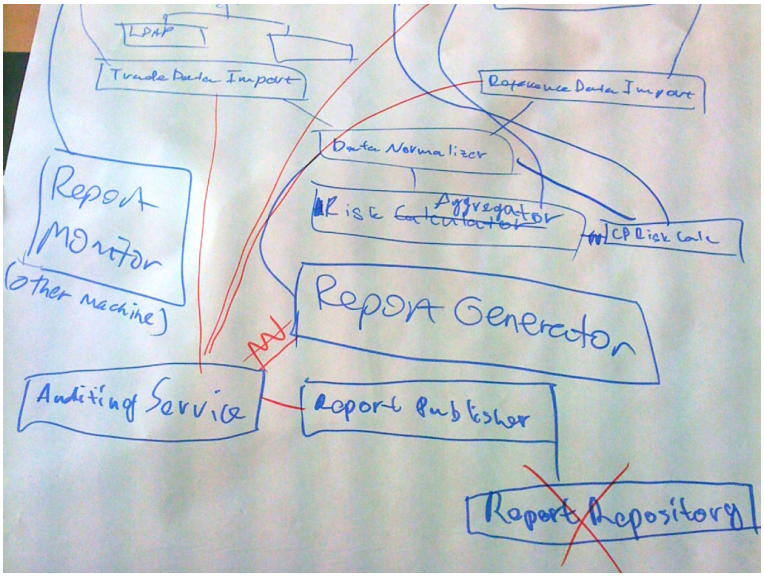
Stormtroopers

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!



3.6 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.

3.7 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.

4. 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.*

4.1 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.

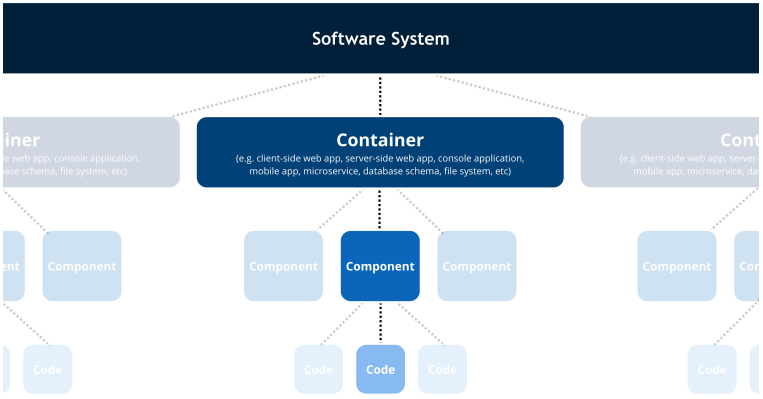
4.2 Static structure

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.

4.3 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:

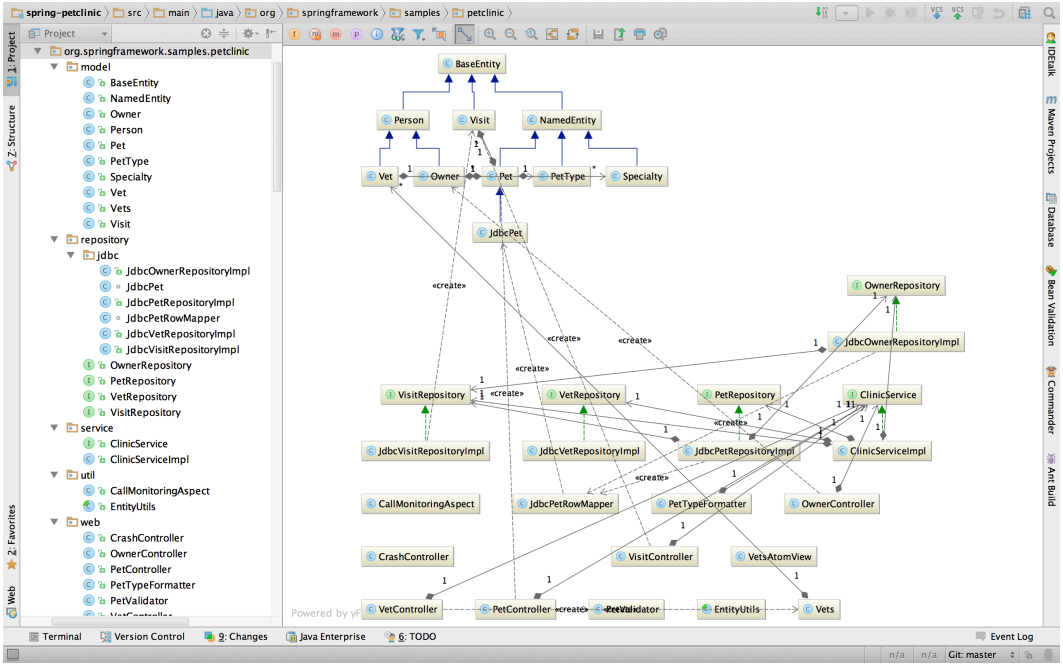
- **Object-oriented programming languages (e.g. Java, C#, C++, etc)**: A component is made up of classes and interfaces.
- **Procedural programming languages (e.g. C)**: A component could be made up of a number of C files in a particular directory.
- **JavaScript**: A component could be a JavaScript 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.

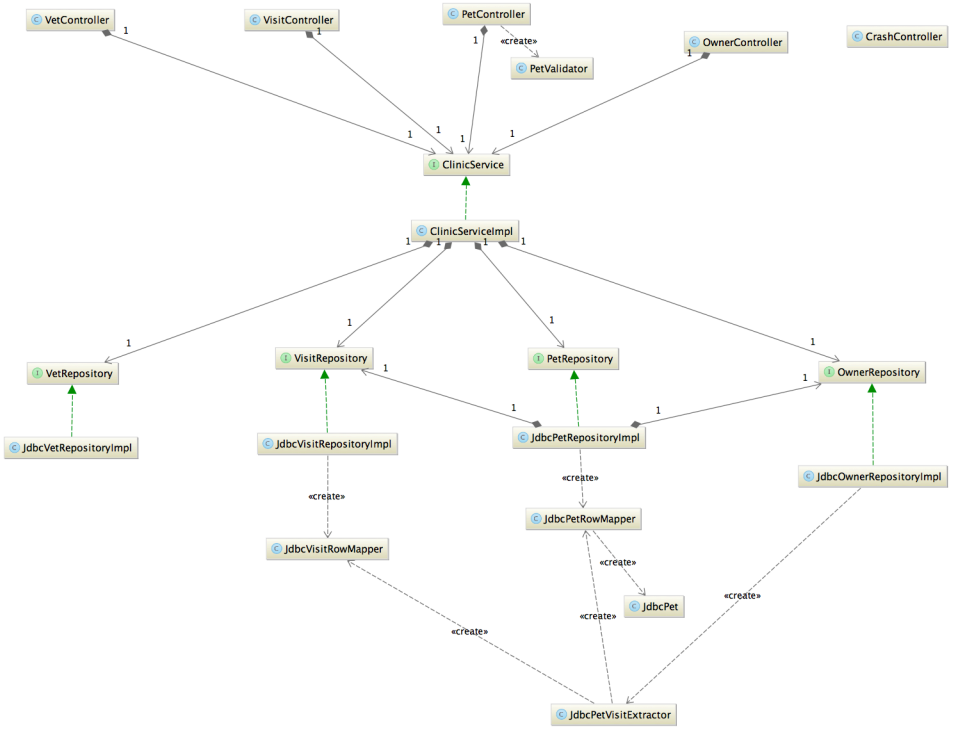
The version of the web application we'll look at here is a typical layered architecture consisting of a number of web MVC controllers, a service containing "business logic" and some repositories for data access. There are also some domain and util classes too. If you

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.

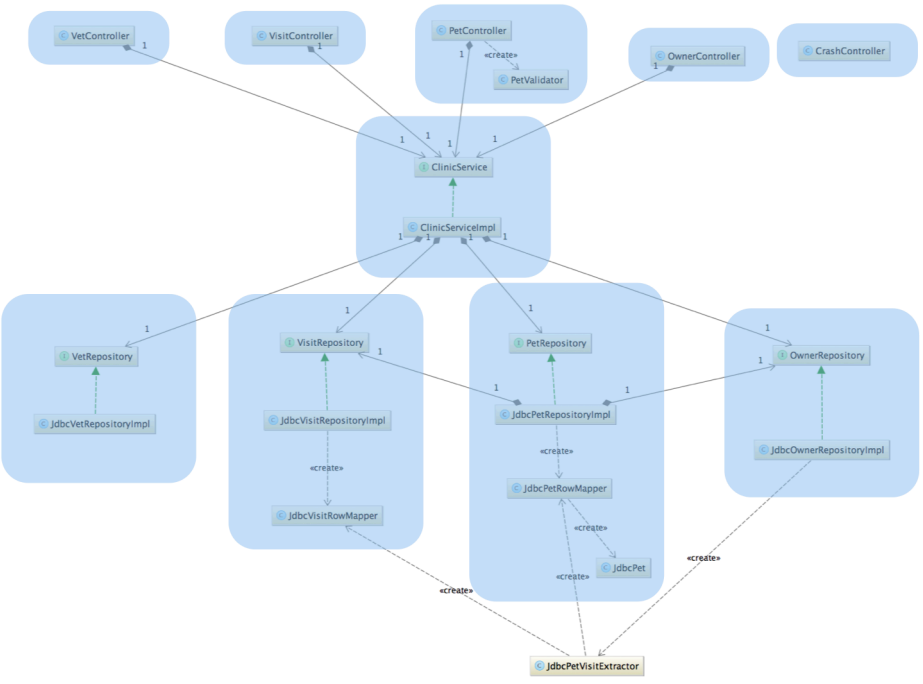


As you would expect, this diagram is showing you all of the Java classes and interfaces that make up the Spring PetClinic web application, plus all of the relationships between them. The properties and methods are hidden on the diagram because they add too much noise to the picture. This isn't a complex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

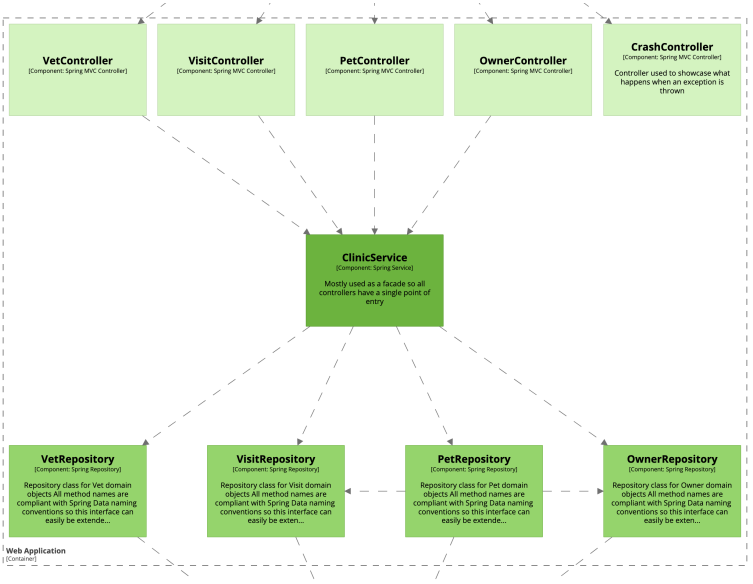
Let's remove those classes that aren't useful to having an "architecture" discussion about the system. In other words, let's only show those classes/interfaces that have some significance from a static structure perspective. In concrete terms, for this specific codebase, it means excluding the model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

4.4 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.

4.5 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.

4.6 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.

4.7 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.

4.8 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.

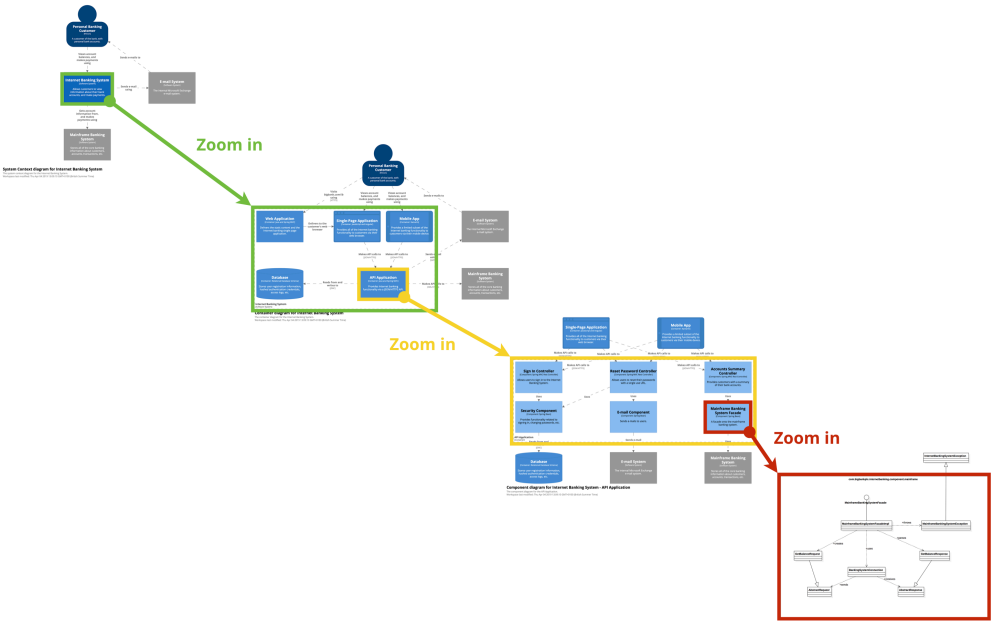
As an example, one organisation I worked with builds software in C to run on mobile devices. Instead of "container" and "component", they use the terms "component" and "module" respectively. In this case, a "component" refers to an executable built in C, which in turn is made up of a number of modules. 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.

5. 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.

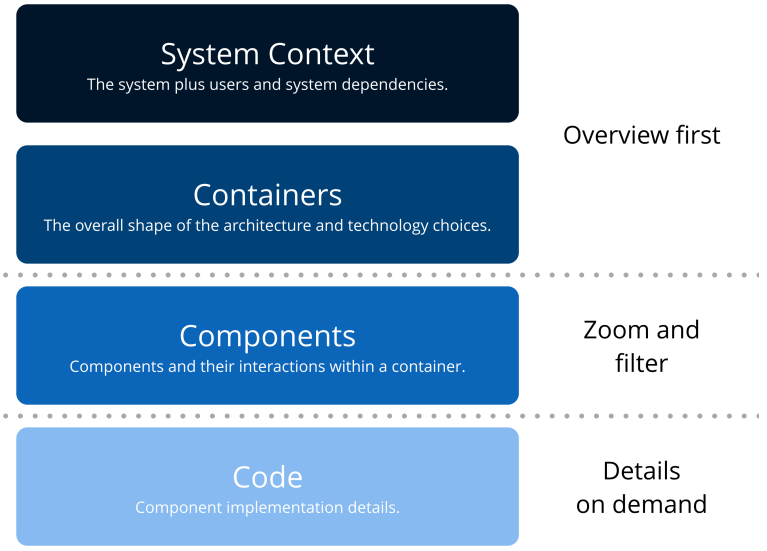


A summary of the C4 model

5.1 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.

6. 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.

6.1 Intent

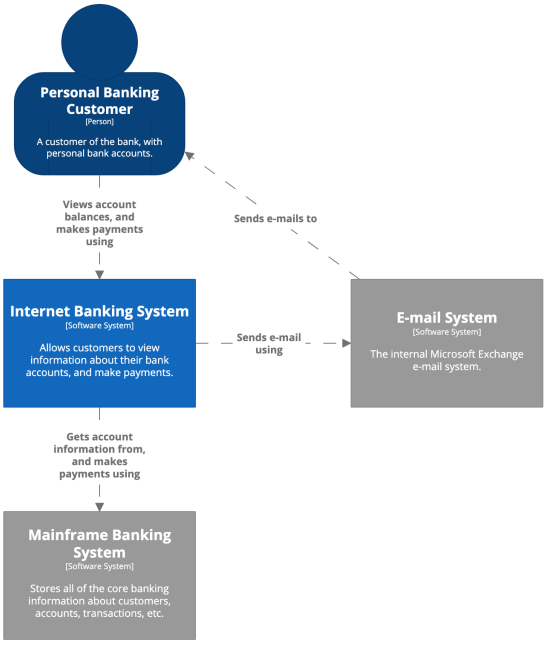
A System Context diagram helps you to answer the following questions.

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?

6.2 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.

6.3 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

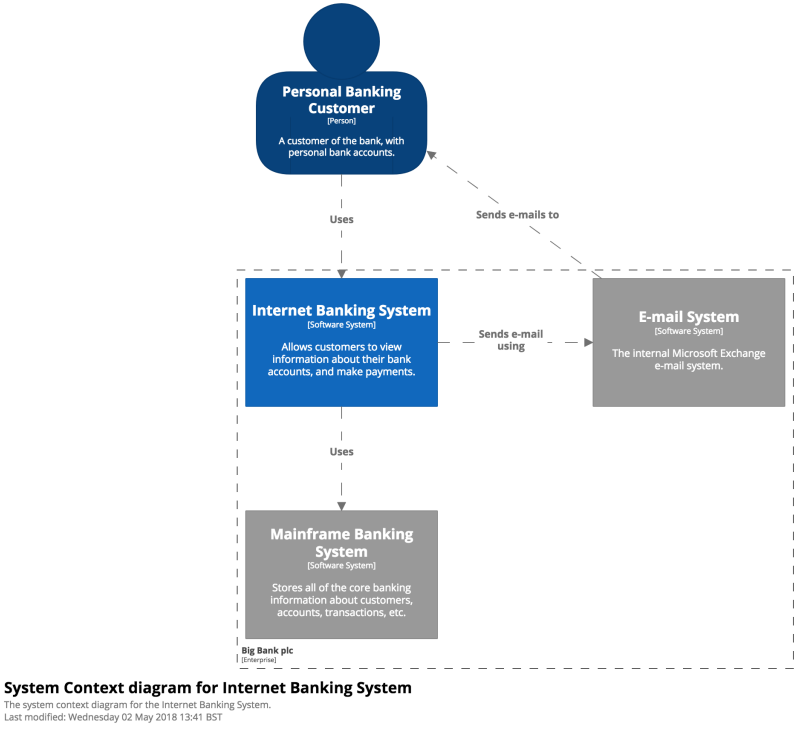
These are the other software systems that your software system interacts with. Typically these software systems sit outside the scope or boundary of your own software system, and you don't have responsibility or ownership of them. The Internet Banking example is very clear in this respect. As a team 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

6.4 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).

6.5 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.

6.6 Audience

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

6.7 Required or optional?

All software systems should have a System Context diagram.

7. 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.

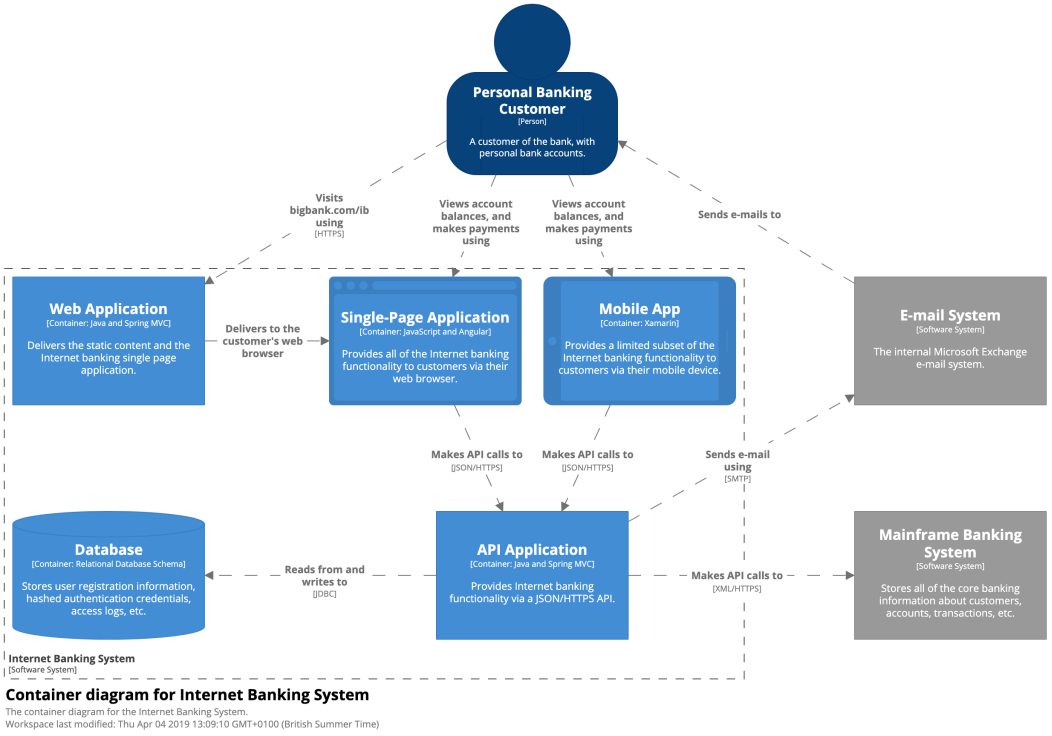
7.1 Intent

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?

7.2 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.

7.3 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. Java/Spring MVC application, ASP.NET web application, C# Windows Service, Relational Database Schema, 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.

If you're undertaking an upfront design exercise and you don't quite know which technology you're going to use to build a particular container, then perhaps instead mention the short-list of technology choices or the general type of technology (e.g. "Relational database schema"). On that note, if a container is a standard RDBMS database/schema, and doesn't make use of any proprietary features, you don't necessarily need to specify the technology. Instead you could mention the technology on a deployment diagram. But if your database is using

specific features of the RDBMS (e.g. Oracle stored procedures), and can't be deployed onto a different RDBMS without changes, then you should specify the technology to make that clear.

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.

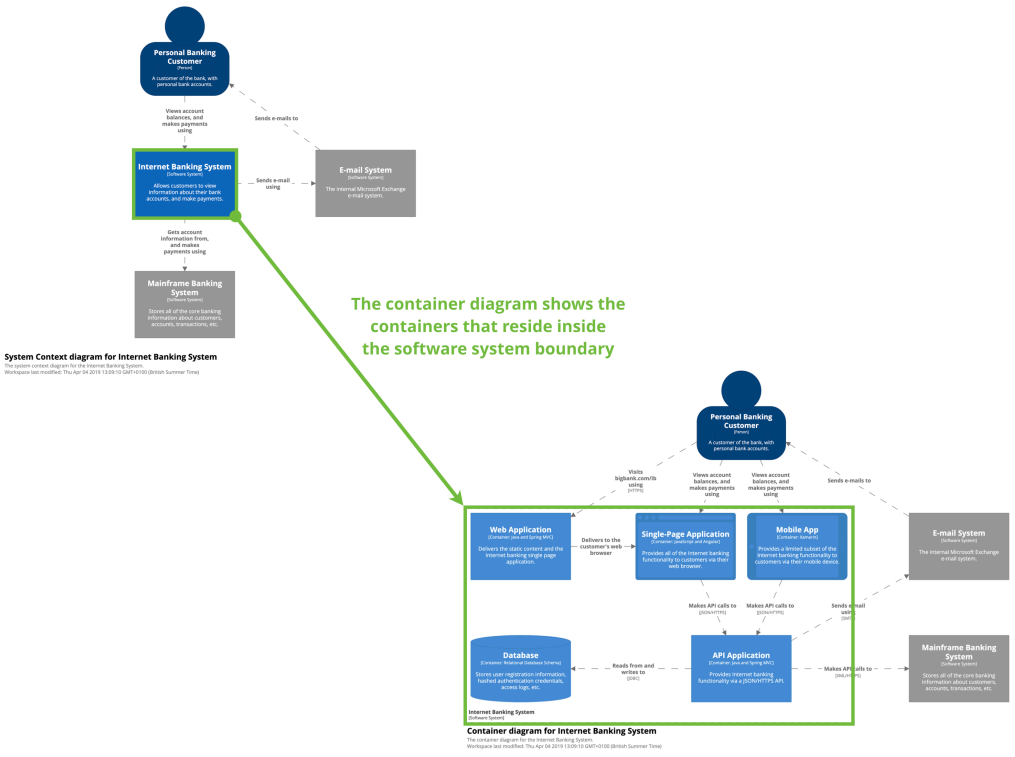
If you're building a software system that is using Amazon S3 for storing data, it's true that you don't run S3 yourself, but you do have ownership and responsbility for the buckets you are using. Similarly with Amazon RDS, you have (more or less) complete control over any database schemas that you create. 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

7.4 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).

7.5 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.

7.6 Audience

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

7.7 Required or optional?

All software systems should have a Container diagram.

8. 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.

8.1 Intent

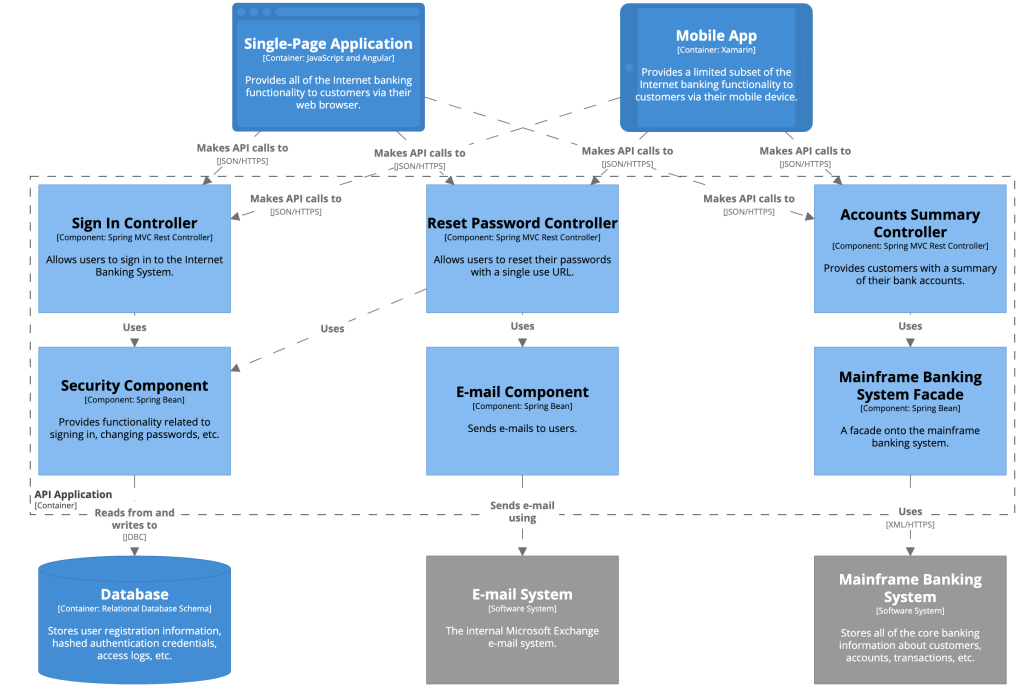
A Component diagram helps you answer the following questions.

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?

8.2 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 ideally 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.

8.3 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, Spring Component, Enterprise JavaBean, Windows Communication Foundation service, etc).
- **Description**: A short description of the component, usually a brief sentence describing the component's responsibilities.

Your component diagram should reflect the components that reside within your container and the architectural style in use. If you consider your container to consist of components organised in layers, your component diagram should show components organised in layers. If your container makes use of components organised in a ports and adapters, hexagonal, or clean architecture style, the diagram should reflect this too.

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

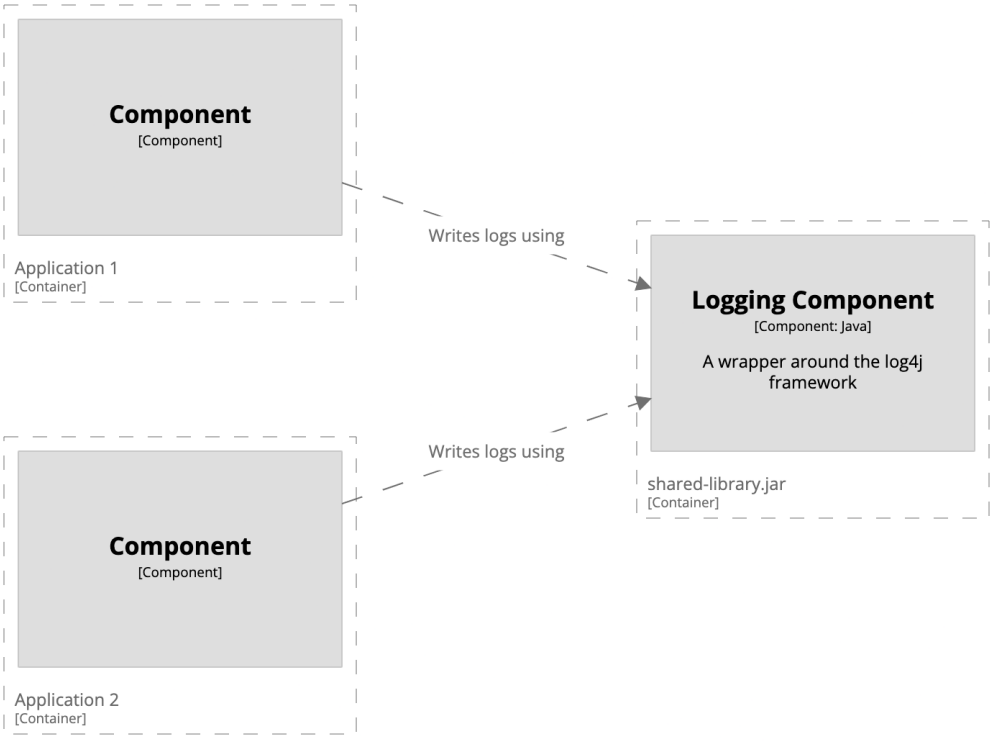
I'm often asked whether a component diagram should include libraries or the shared components from those libraries. And the follow-up question is about how such things should be represented.

The first part of the question is easy to answer. If including these things helps tell the story, then, yes, you should consider including it. The second part of the question requires a little more thought.

Let's imagine that we're building a Java application, and we've included the log4j library for logging. I'd categorise this as a "utility library", it's code that we're using for its utility, and I generally don't show such things on my component diagrams.

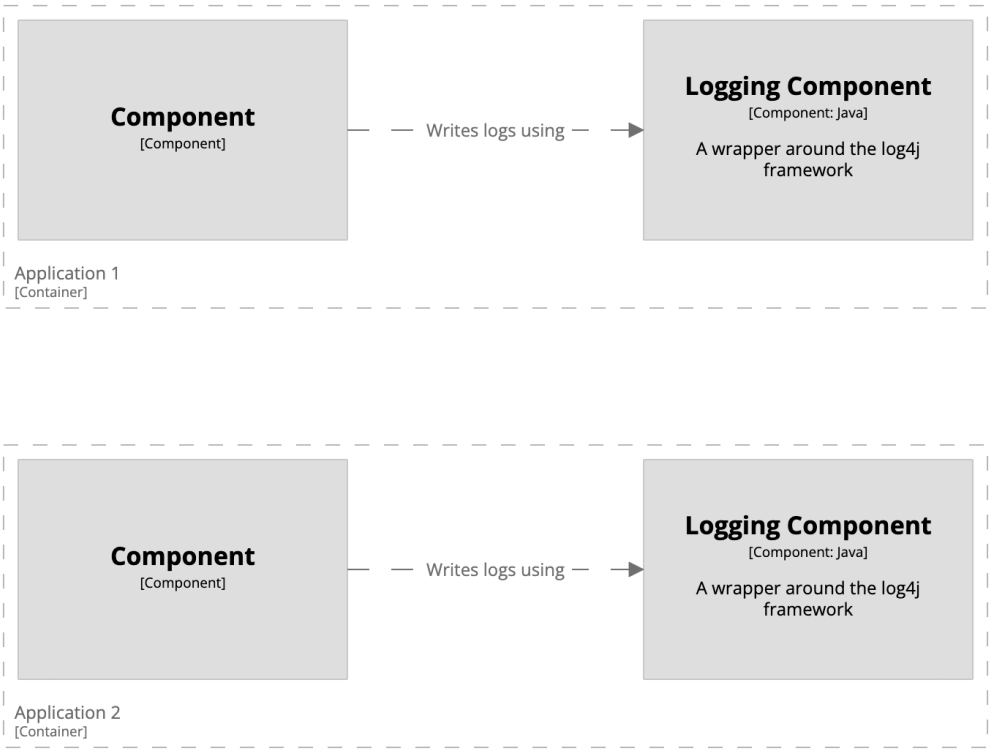
Now let's change the scenario, and imagine that we have a couple of Java applications that make use of a shared library - perhaps a `.jar` file that is included in both applications at build time. This could be an open source library, or some shared code that we've built ourselves. Let's make this more specific, and say that we've built a "Logging Component" that provides a nice wrapper around the functionality provided by the log4j library. This shared library is named `shared-library.jar` and it's included in both of our Java applications. In essence, we're sharing code between two applications, via a shared library that is built into both of those applications.

The way that many people will instinctively (and incorrectly) try to diagram this is to model the shared library as a container, as follows:

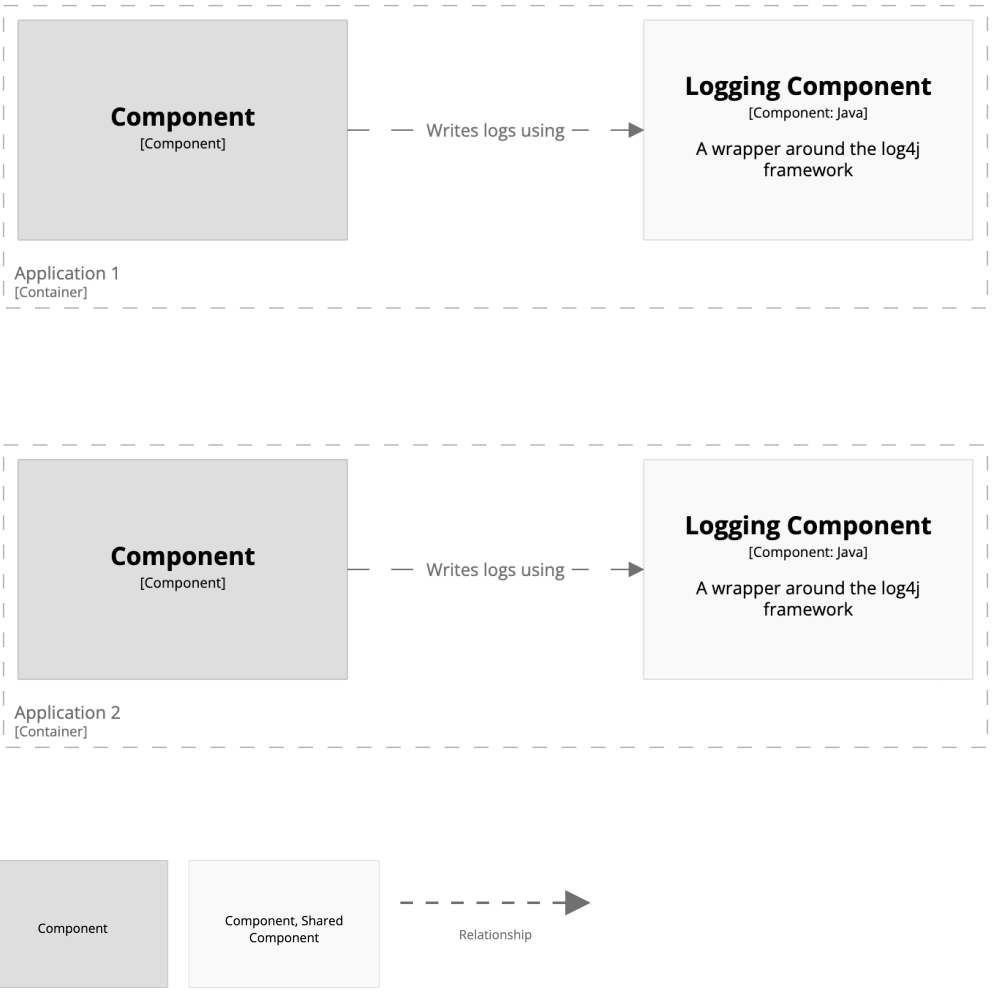


If the shared library is a separately deployable/runnable application (i.e. it *is* a C4 model "container"), and offers a remotable interface for clients, then this diagram might be accurate. In most cases it's not, so please don't do this!

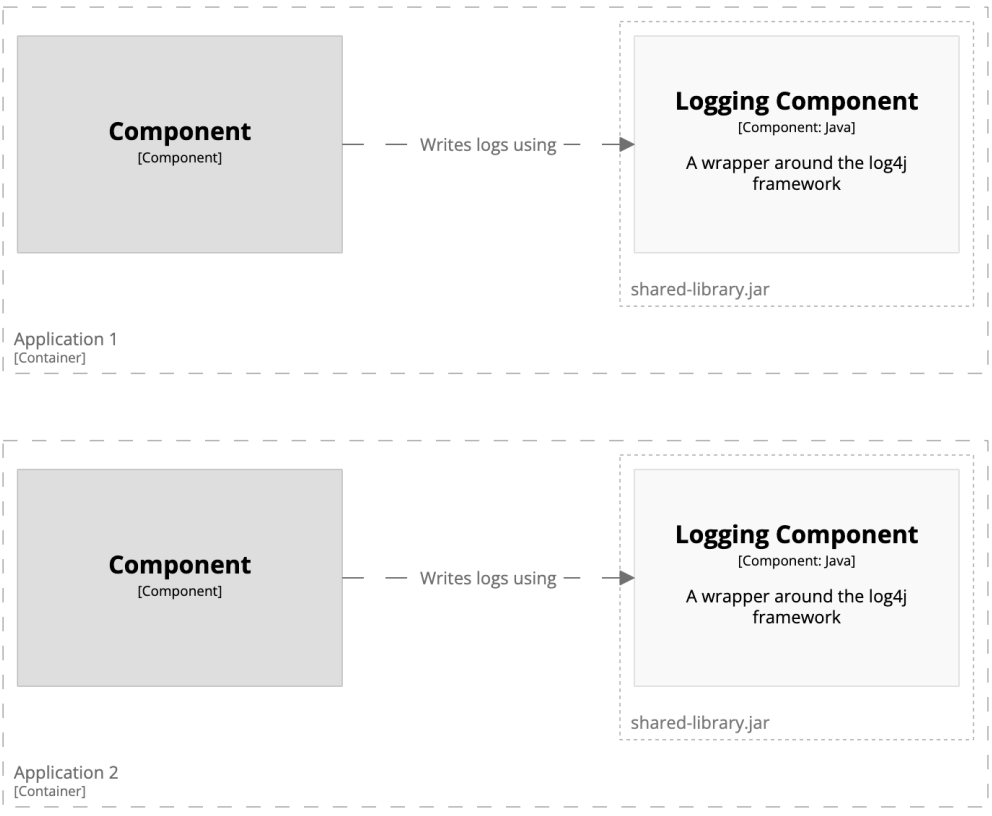
With this in mind, the first correct way to diagram shared components is to include them everywhere they are used. In this exampe, we can include the "Logging Component" in both of the component diagrams - one component diagram for each application. It might look like this:



At this point, other than being named the same, it's not clear that the "Logging Component" is shared between the two applications. One option to highlight this is to use a specific notation to represent shared components - perhaps a symbol/icon or a colour coding, as shown in the next example.



If you wanted to be more explicit, you could add a bounding box around the shared components, perhaps showing the name of the library itself.



The key takeaways are (1) the focus of a component diagram should be components rather than libraries, and (2) I generally consider libraries to be the packaging/distribution mechanism for components rather than components themselves.

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.

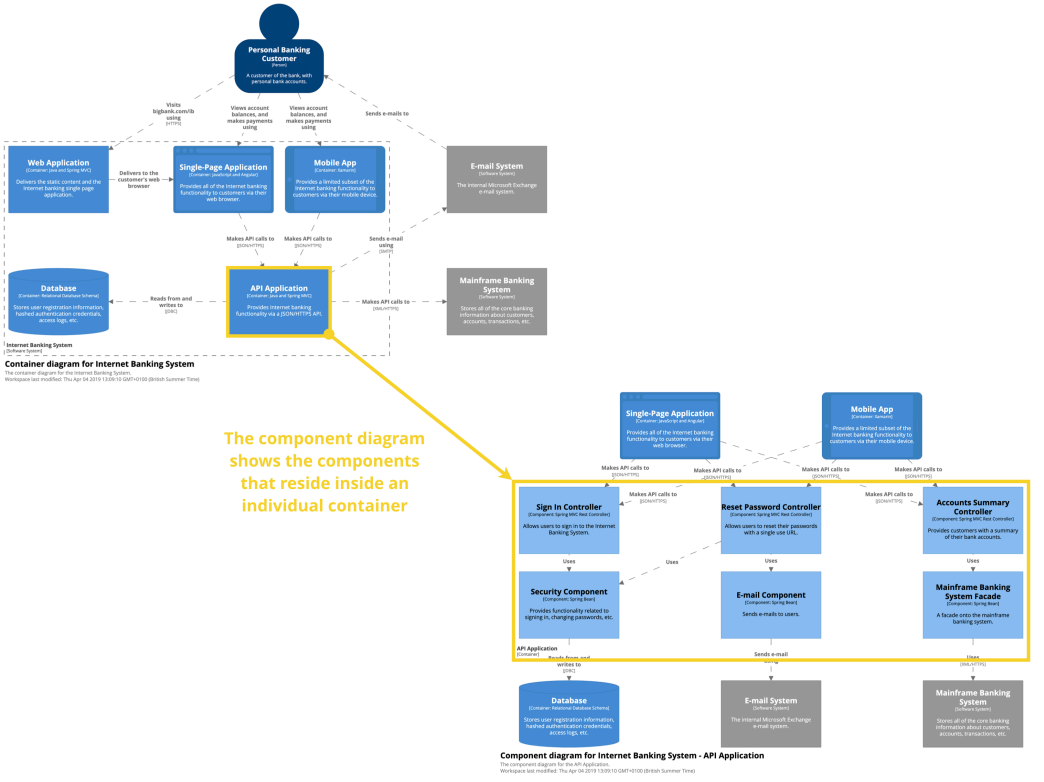
As with many of the things discussed in this book, there is no "right" answer, and it really depends on what story you need to tell.

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

8.4 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).

8.5 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.

8.6 Audience

Technical people within the software development team.

8.7 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.

9. Level 4: Code-level diagrams

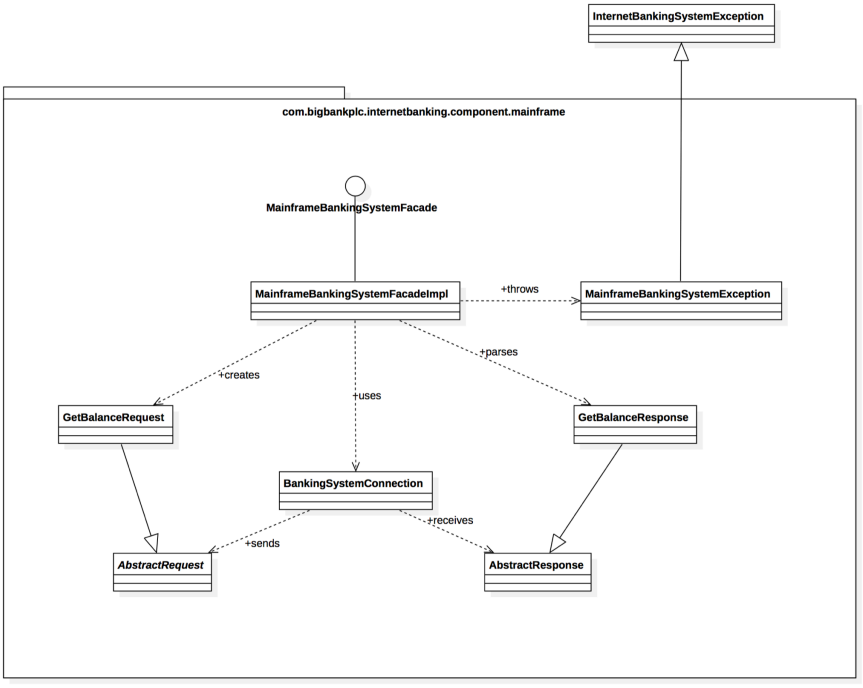
The final level of detail is one or more code-level diagrams showing the implementation details of an individual component.

9.1 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.

9.2 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.

9.3 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.

9.4 Audience

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

9.5 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.

10. 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.

10.1 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

10.2 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.

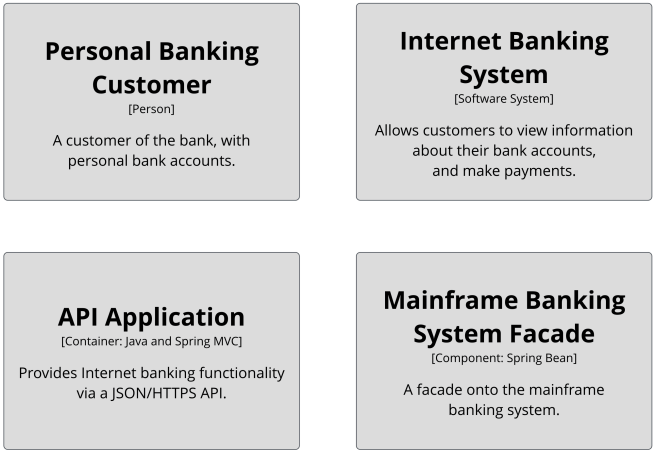
10.3 Elements

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

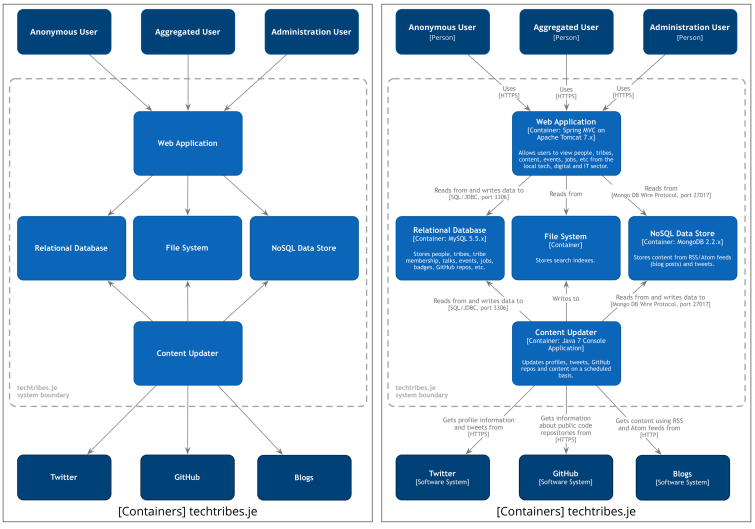
This is the notation that I've gradually settled on over the years. It's easy to draw on a whiteboard or in tooling, plus it works well on sticky notes and index cards. Do feel free to create your own notation though.

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:

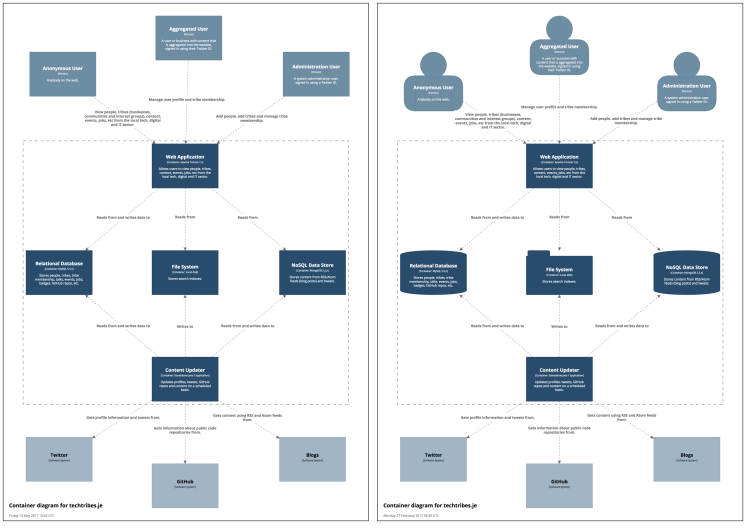
- Existing vs new.
- Off-the-shelf product vs custom build.
- Technology type or platform.
- Risk profile (e.g. risk to build; high-medium-low risk; red-amber-green).
- Size and/or complexity.
- Ownership (i.e. elements you own vs elements somebody else owns).
- Internal vs external (i.e. elements within 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.

10.4 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]

10.5 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.

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!

10.6 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.

10.7 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.

10.8 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.

10.9 Diagram scope

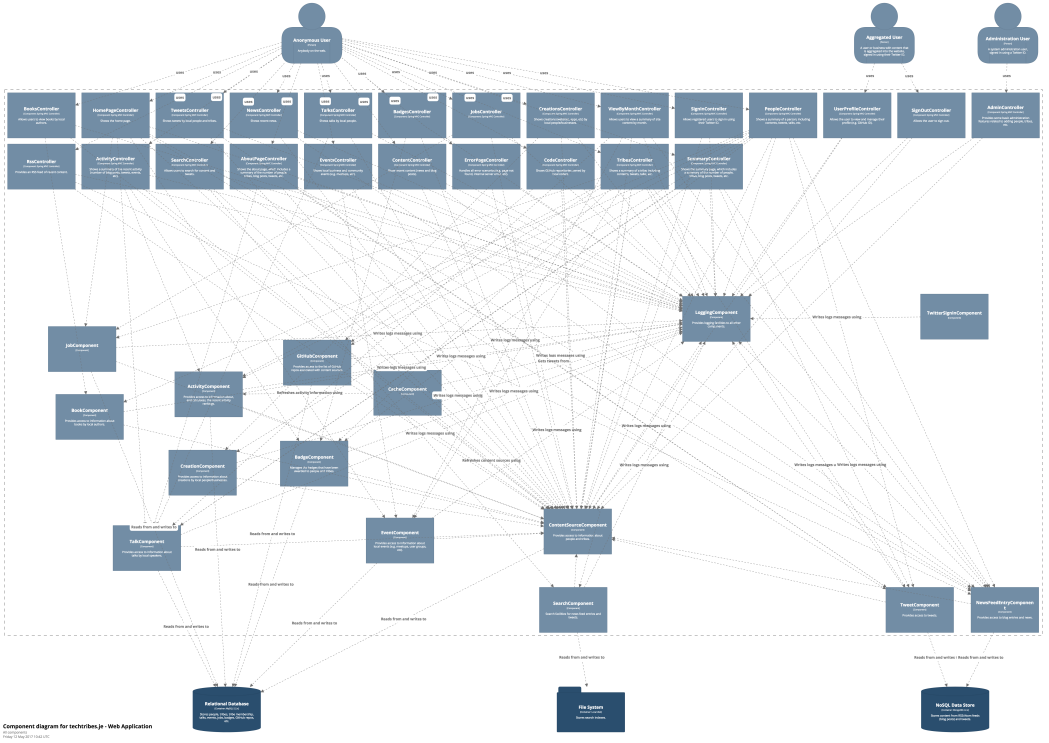
The example diagrams I've presented in the book have all conveniently fitted on one page. Of course, in the real-world, you're likely to have larger and more complicated software systems. Simply by their very nature, component diagrams are prone to becoming large and complex. So what do you do when your 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

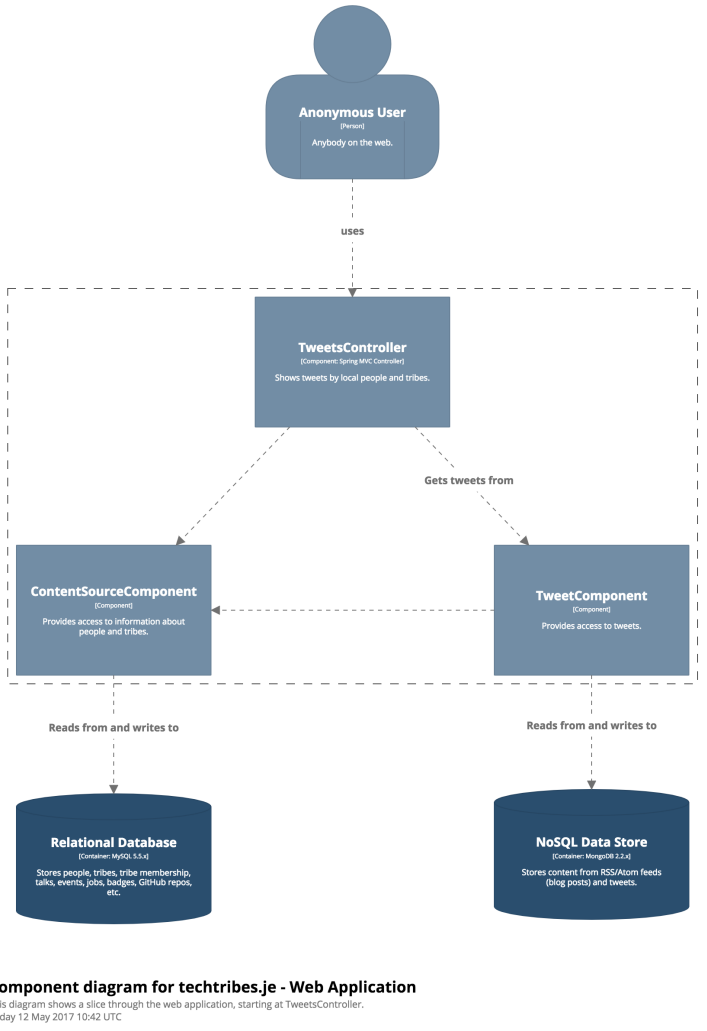
This component diagram has been automatically created using some tooling that identifies components and their dependencies from the code. It's comprehensive, but it's a mess! And it's difficult to determine whether this mess is caused by the architecture being a mess or the diagram showing too much information. 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.

10.10 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.

11. 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.

11.1 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.

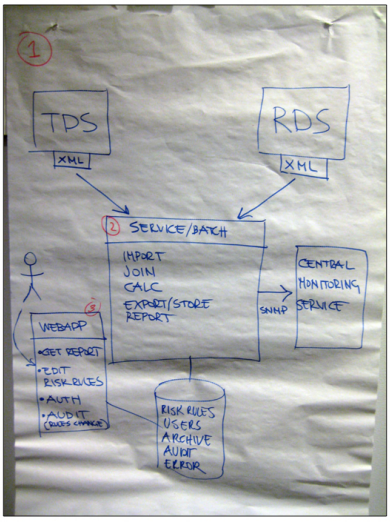
In simple terms, if you have a component diagram on the wall, each of the components should map to something real in the code. If there's a "Logging Component" box on the diagram, make sure there really is a "Logging Component" in the code. This is relatively easy to do but, in my experience, I rarely 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.

11.2 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.

11.3 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.

12. Deployment diagrams

The C4 model diagrams are designed to describe the static structure of a software system, at different levels of abstraction. But in order to deliver value, that software needs to be executed somewhere, on some infrastructure. Previously that infrastructure would be physical servers, but today it could be anything from virtualised servers and "Infrastructure as a Service" (IaaS) through to Docker containers and "Platform as a Service" (PaaS) environments.

12.1 Network and infrastructure diagrams

Back when physical servers were the primary way that organisations deployed their software, you would often see teams creating "network diagrams" or "infrastructure diagrams". These were essentially a map of the network infrastructure; showing servers, routers, firewalls, load balancers, network switches, VLANs, etc.

These diagrams were typically hand-crafted by infrastructure/network/operational engineers, and often not understood well by software engineers. These diagrams definitely have their use, but usually didn't show much detail about where and how the software would run. It was also common for the infrastructure team to have their set of infrastructure diagrams, and the developmen team to have a separate set of software architecture diagrams, leading to a gap between these two views.

12.2 Deployment diagrams

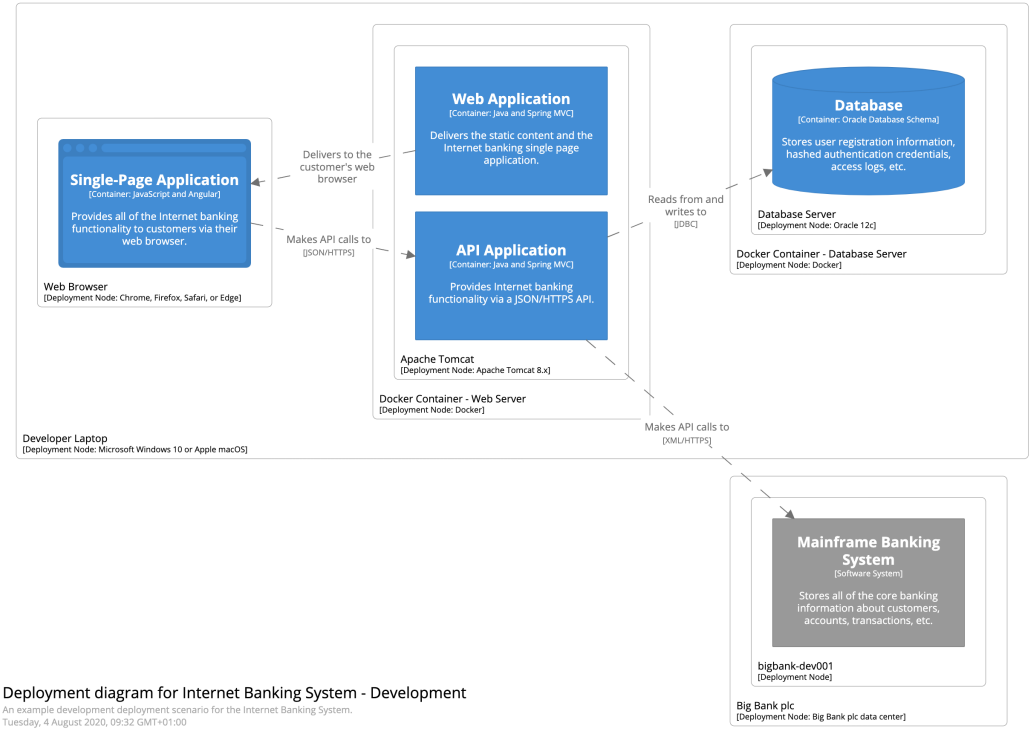
The DevOps movement has sought to bring these two separate disciplines together. From a diagramming perspective, one way to also do this is to describe how the software is deployed onto infrastructure in a "deployment diagram". Even if your deployment is fully automated, it can still be useful to have 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) onto deployment nodes (i.e. devices or execution environments). UML deployment diagrams do work, but I often find them seeming "disconnected" from other views of the software. For example, a UML deployment diagram might show a specific artifact (e.g. a JAR file or DLL) deployed onto a server, but there's often no other diagram defining what is contained within that deployable artifact.

With the C4 model as a basis, I take a slightly simpler approach whereby my deployment diagrams show the mapping of software systems or containers (from the C4 model) to deployment nodes. A deployment node is something like:

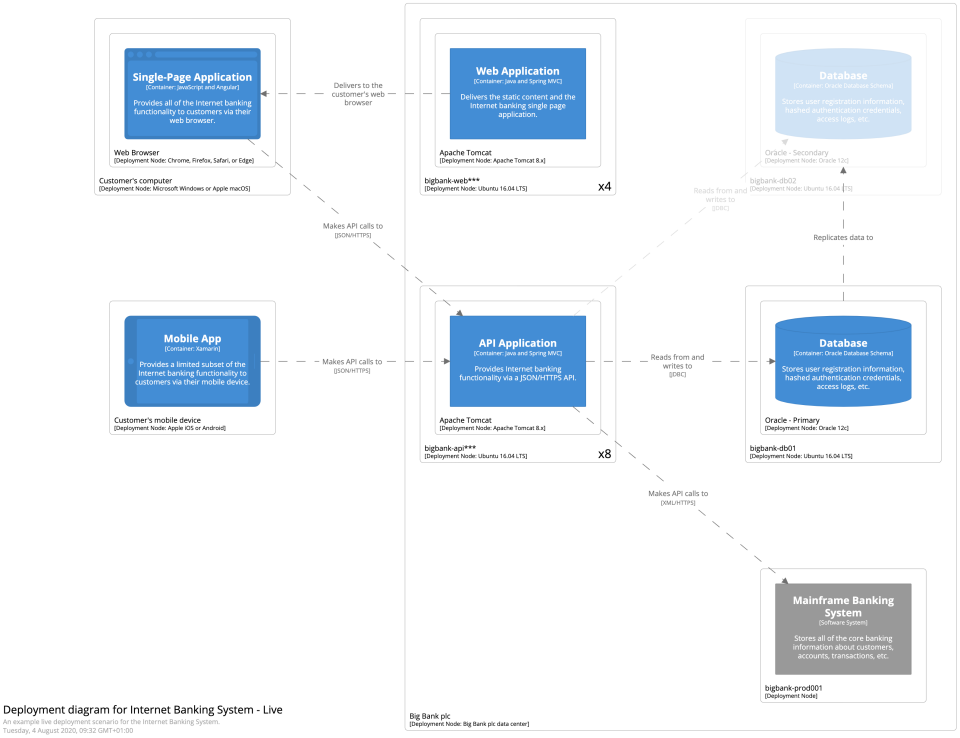
- Physical infrastructure (e.g. a physical server or device).
- Virtualised infrastructure (e.g. IaaS, PaaS, a virtual machine).
- Containerised infrastructure (e.g. a Docker container).
- An execution environment (e.g. a database server, Java EE web/application server, Microsoft IIS, etc).

And deployment nodes can be nested. In essence, you're showing *instances* of software systems/containers, and how they're deployed onto infrastructure. As an example, the following diagram illustrates what a development environment for the Internet Banking System might look like.



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. We're also making use of a development version of the Mainframe Banking System, deployed on a development server somewhere in the bank.

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

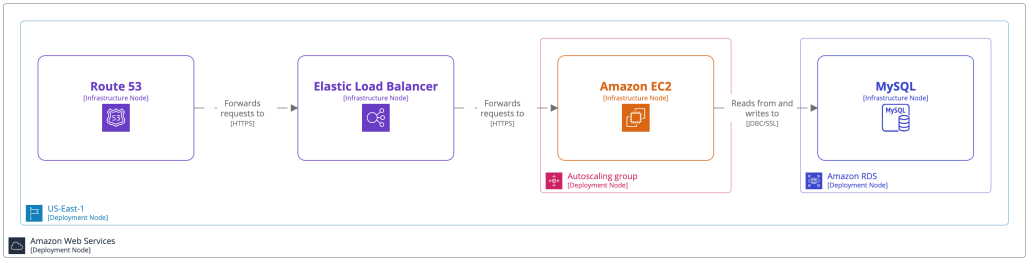


Customers may have computers or mobile devices, which is where the UIs end up running. Everything else is deployed in the Big Bank's data center, onto a number of separate Ubuntu servers. I've also shown a secondary database server, with data replication.

You may also want to include "infrastructure nodes" such as DNS services, routers, load balancers, firewalls, etc. Feel free to also include hostnames, IP addresses, or whatever other information you think helps you tell the story that you want to tell.

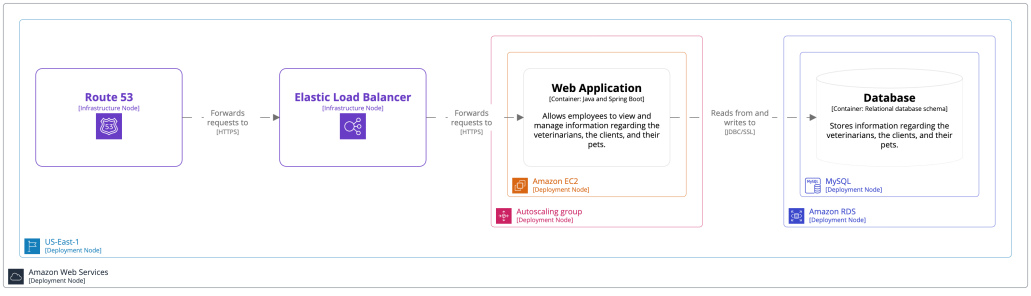
12.3 Cloud architecture diagrams

I see a lot of discussion about "cloud architecture diagrams", with people specifically asking how to create them, which tools to use, and where to get the icon sets. Many of the example cloud architecture diagrams that I see are relatively generic, and look something like this.



This diagram shows that a number of AWS services are being used; including Route 53, Elastic Load Balancer, EC2 (with auto-scaling), and MySQL on RDS. It looks good, it's colourful, and it uses the appropriate icons from the AWS icon set. It's a decent first attempt because it helps to highlight which AWS services are in use. But that's also the biggest problem - it's really an infrastructure diagram, and is only telling you half of the story. Understanding which AWS services are being used is great, but I'd really like to also know how you are using them.

Most software development teams would be better creating a deployment diagram, which additionally shows the mapping of containers (applications and data stores) to that infrastructure. Here's a better example.



This diagram again shows the AWS services in use, but it also shows what they are being used for, through the inclusion of the web application and database. You can, of course, add more detail as needed; including VPCs, VLANs, firewalls, IP addresses, instance counts, scaling limits, etc. It's worth noting that the same advice applies to diagramming Docker and Kubernetes environments too - don't just focus on the infrastructure, remember to show your own applications and data stores.

13. 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.

13.1 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 "4+1" 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:

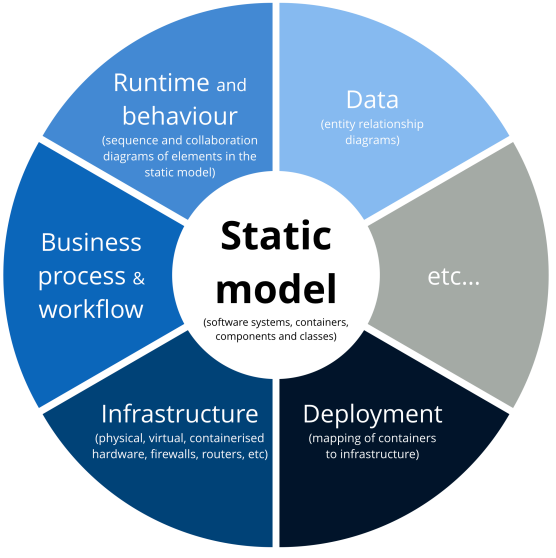
- **System Context**: This shows you what the system does and how it fits into the world around it (i.e. users and other software systems).

- **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.



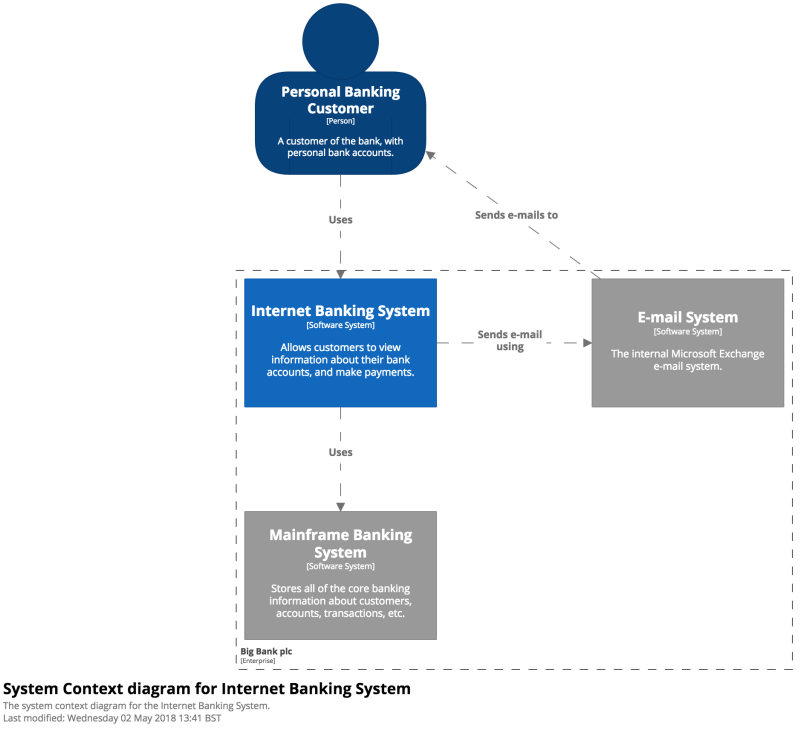
The static structure defines the core of the software architecture model

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

13.2 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.

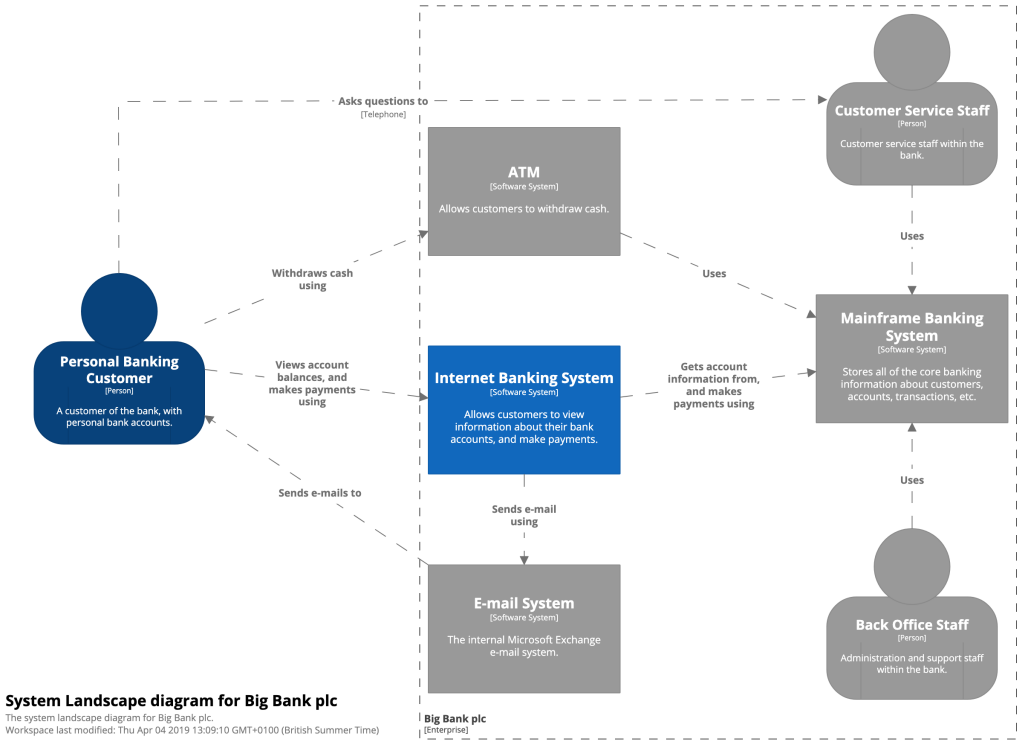
From a practical point of view, a System Landscape diagram is really just a System Context diagram without a specific focus on a particular software system. Here's the example System Context diagram for the Internet Banking System that we've seen before.



A System Context diagram for a fictional Internet Banking System

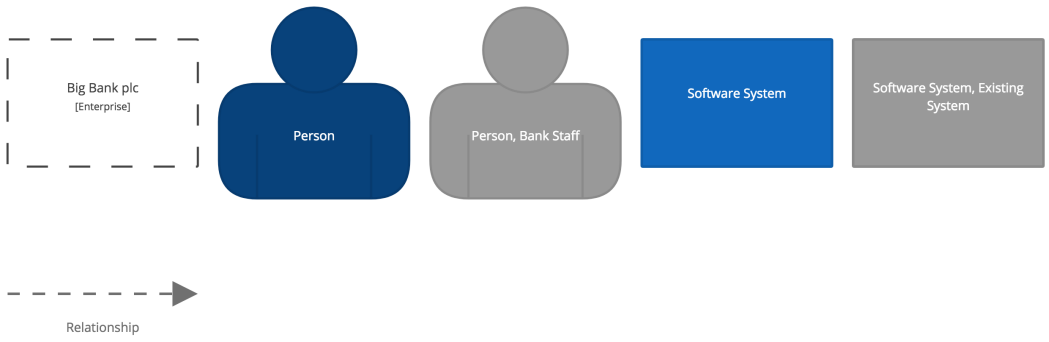
When drawing a System Context diagram, I usually only include the people and software systems that have a *direct relationship* with the software system in focus. In contrast, a

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.

13.3 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.

13.4 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.

13.5 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.

13.6 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.

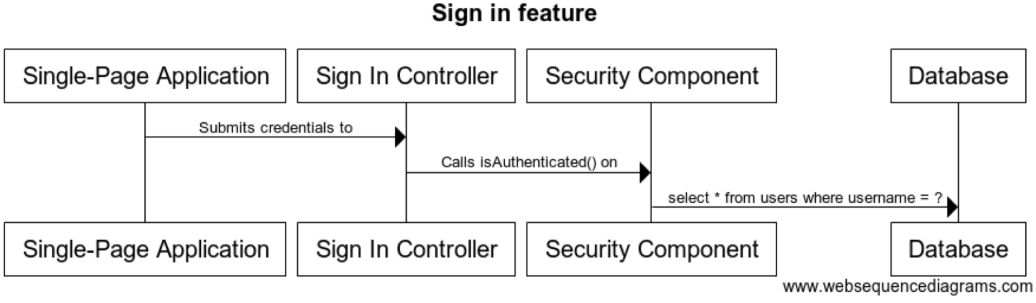
Software isn't static though, and needs to be executed in order to actually do something useful. For this reason, it can be useful to create diagrams that illustrate what happens at runtime (a "dynamic view") for important use cases, user stories or scenarios.

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.

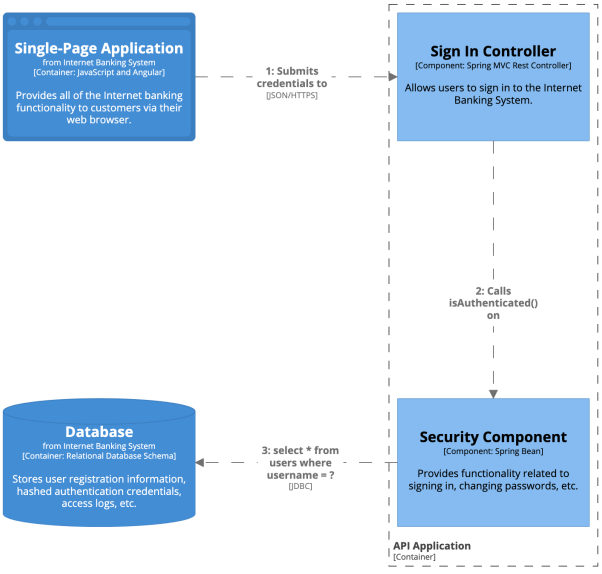


There are a number of tools and approaches to creating sequence diagrams, but they all create similar diagrams, with the official UML specification detailing ways to add more precision and semantics onto the diagrams.

Collaboration

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.

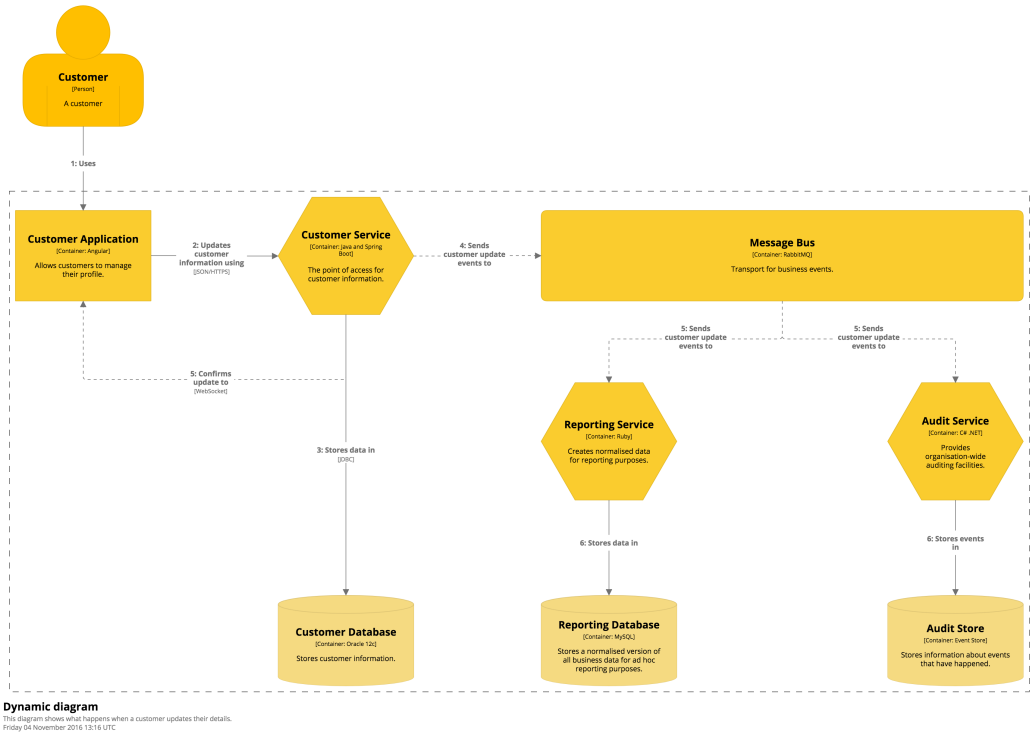
[1]In UML 2.x, this is called a communication diagram, but the purpose and content is essentially the same.



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.

13.7 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.

14. Appendix A: Financial Risk System

14.1 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.

14.2 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.

14.3 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.