The C4 model

for visualising software architecture

Simon Brown

The C4 model for visualising software architecture

Simon Brown

This book is available at https://leanpub.com/visualising-software-architecture

This version was published on 2025-11-19

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 numerous 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 ~40 countries later, I have gigabytes of anecdotal photo evidence - the majority of the diagrams use an ad hoc "boxes and arrows" notation with no clear notation or semantics. Designing software is where the complexity should be, not communicating it.

This book focuses 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 (decades?) 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 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 model for visualising software architecture and Structurizr - 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. You can find my website at simonbrown.je.

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.

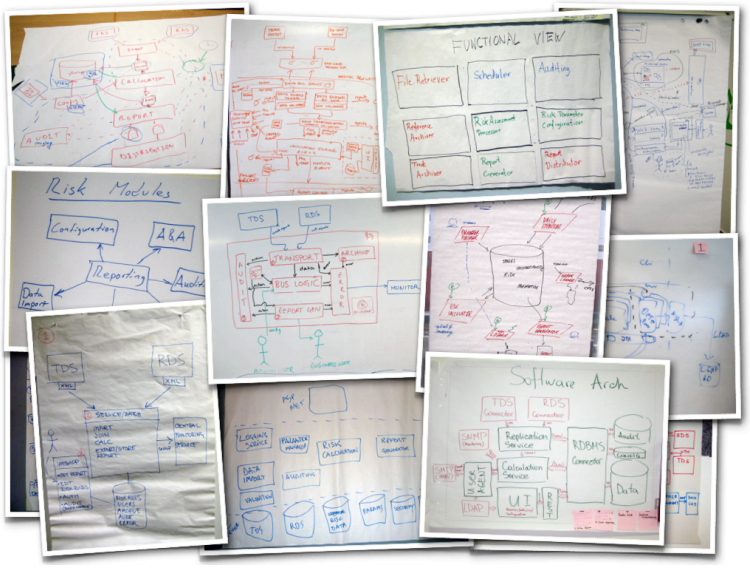
Through my training and workshops, I've asked thousands of software developers around the world to draw software architecture diagrams over the past decade, and continue to do so today. The results still surprise me, anecdotally suggesting that effective visual communication of software architecture 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 Modeling 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 arrows" 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 photos that follow 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, threat modelling, 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 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 workshop where I ask software developers to draw some diagrams 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 arrows" 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 four 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:

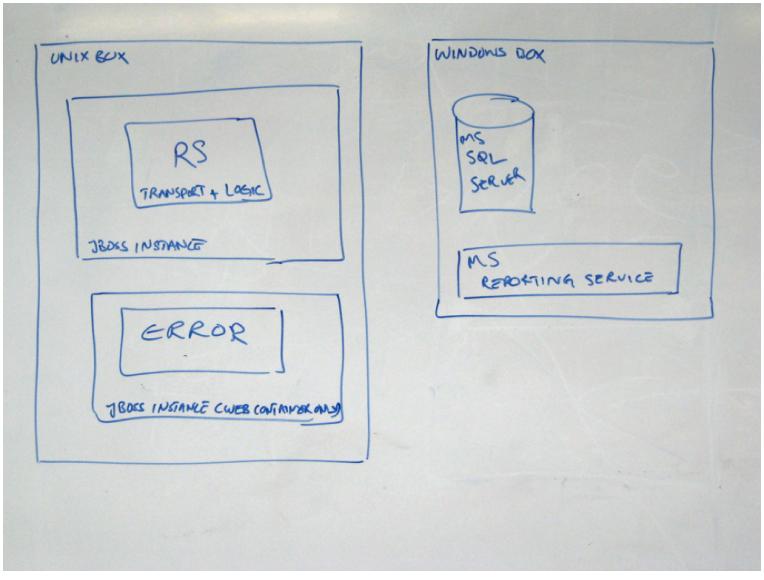
1. In their haste to adopt agile approaches in recent years, many software teams have "thrown out the baby with the bath water". Modelling and documentation have been thrown out alongside traditional plan-driven processes and methodologies. That may sound a little extreme, but many of the software teams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

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.

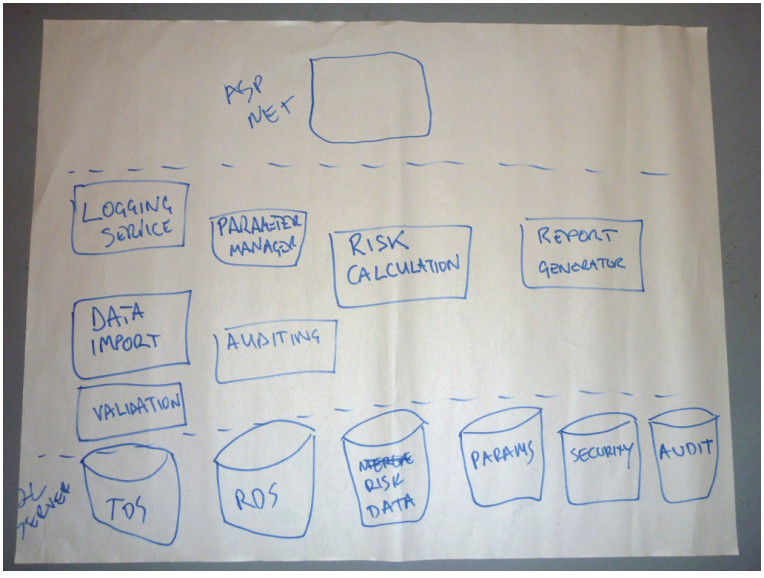


There's a Unix box and a Windows box, with some additional product selections that include JBoss (a Java EE application server) and Microsoft SQL Server. The problem is, I don't know what those products are doing, plus there seems to be a connection missing between the

Unix box and the Windows box. It's essentially a bulleted list of technologies 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.

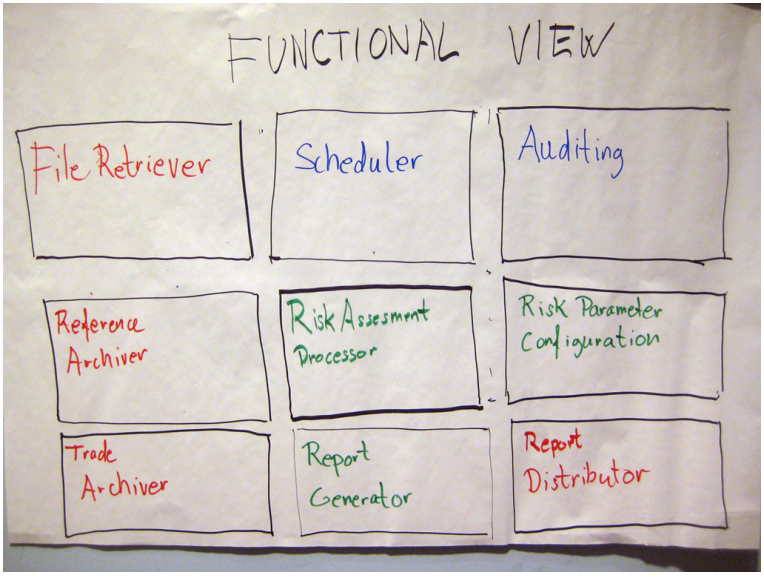


This is a three-tier solution (I think) that uses the Microsoft technology stack. There's an ASP.NET web application at the top, which I assume is being used for some sort of user interaction, although that's not shown on the diagram. The bottom section is labelled "SQL Server" and there are lots of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

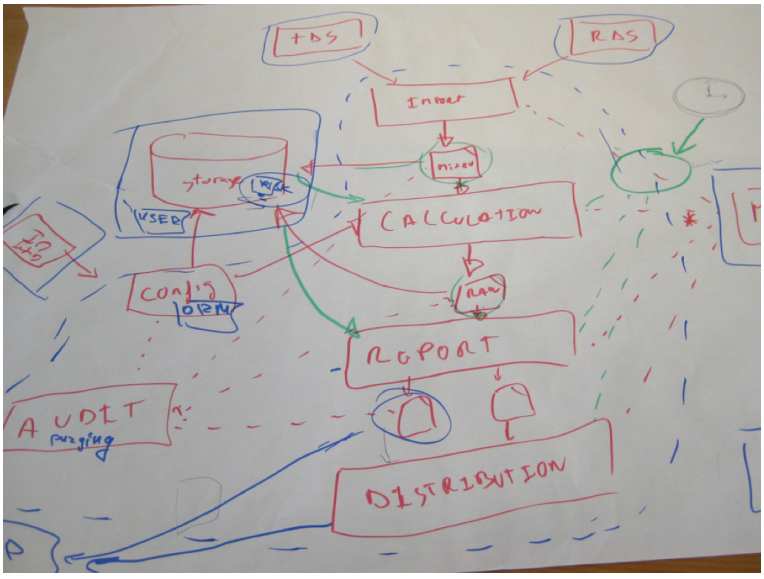


Essentially the group that produced this diagram has simply documented their functional decomposition of the solution into a number of smaller things. Imagine a building architect drawing you a diagram of your new house that simply had a collection of boxes labelled “Cooking”, “Eating”, “Sleeping”, “Relaxing”,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

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.



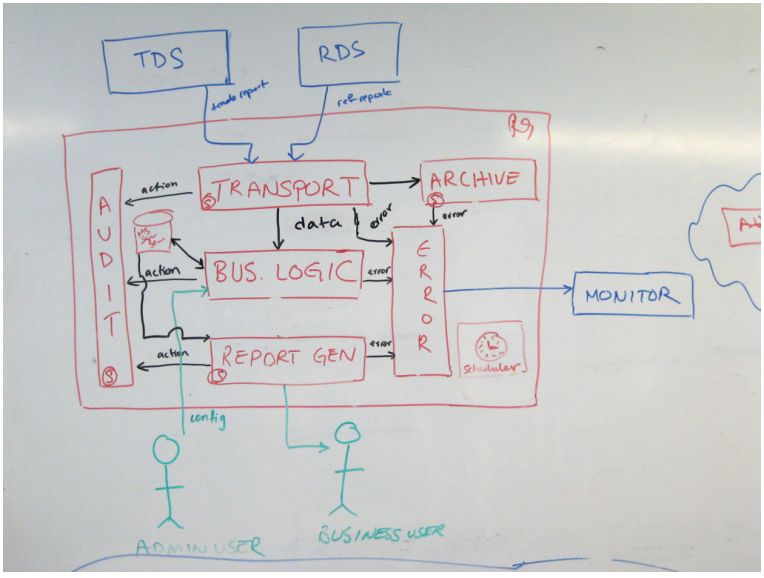
The central spine of this diagram is great because it shows how data comes in from the source data systems (TDS and RDS) and then flows through a series of steps to import the data, perform some calculations, generate reports, and finally distribute them. It's a simple activity diagram that provides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

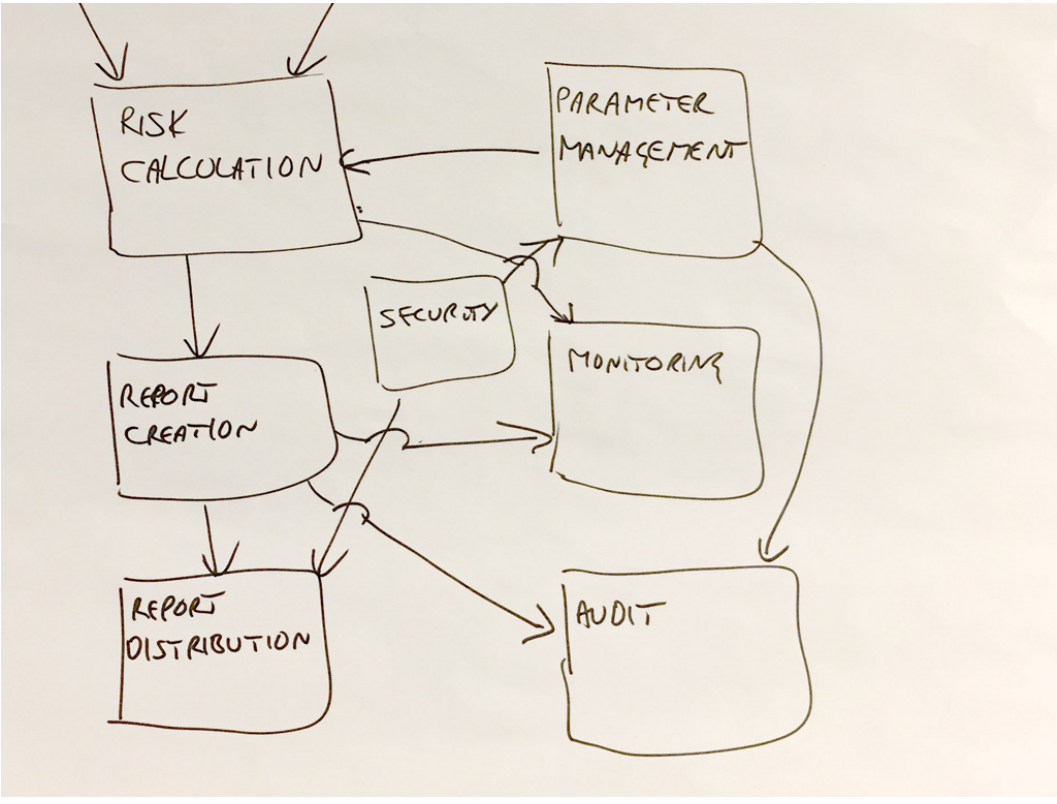
This is another very common style of diagram. Next time somebody asks you to produce a software architecture diagram of a system, present them this photo and you're done!



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! Most of the content is very generic. This diagram doesn’t tell you much about the business domain at all.

The “logical view”

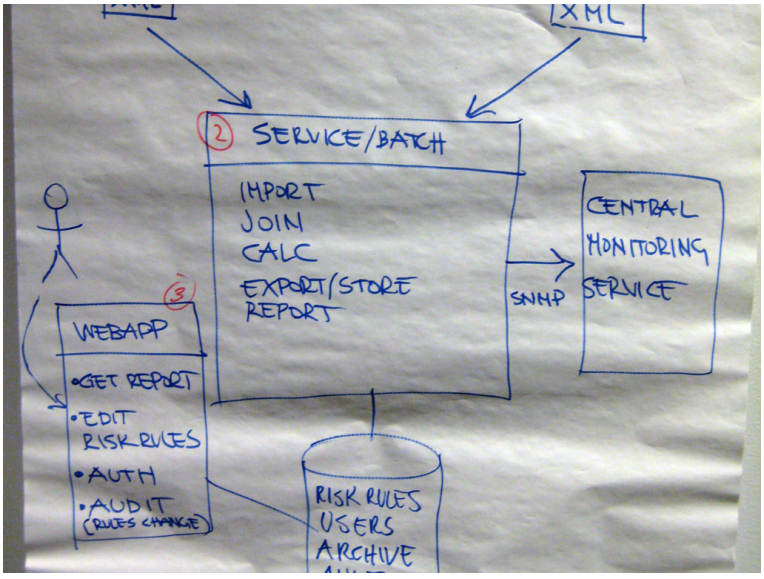
This diagram is also relatively common. It shows the “logical” building blocks that the software system is composed of, but offers very little information other than that.



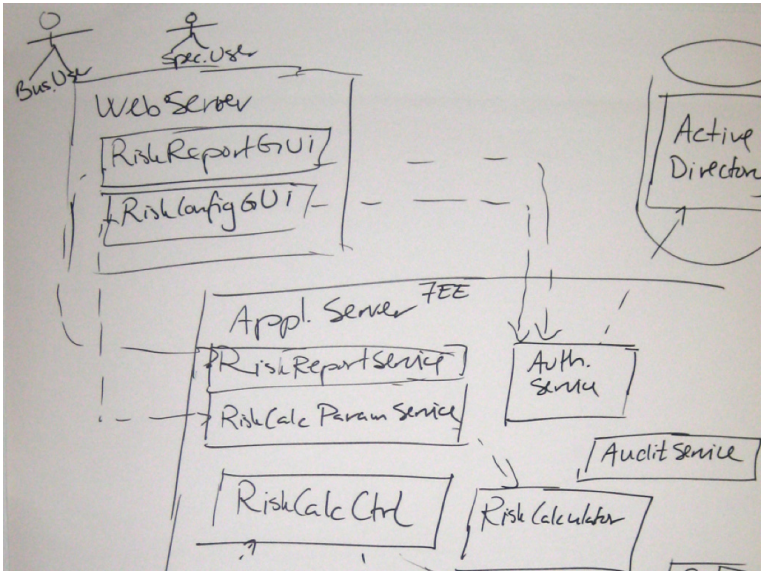
There's a common misconception that software architecture diagrams should be "logical" in nature rather than include any references to technology or implementation details, especially before any code is written. We'll look at this misconception later in the book.

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. 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, for reasons I will discuss later in the book.



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.

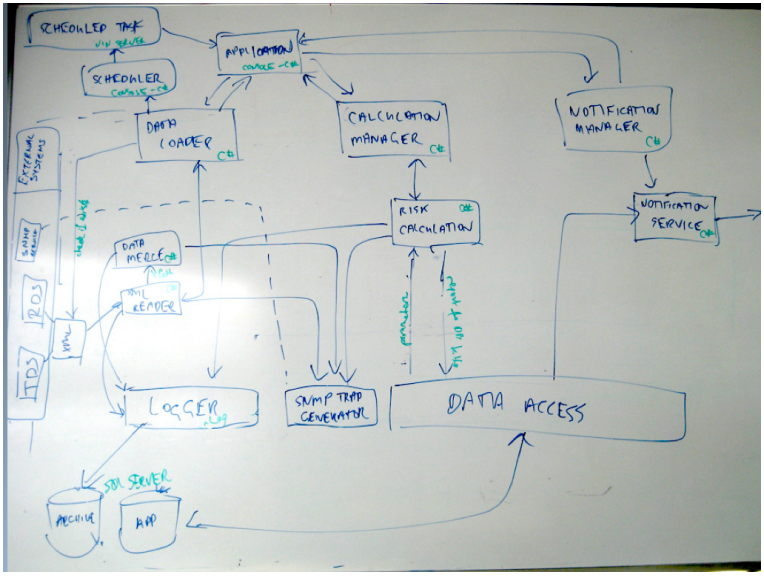


The lines between the web server and the application server have no information about how this communication occurs. Is it SOAP? A JSON web API? XML over HTTPS? gRPC? Remote

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

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.

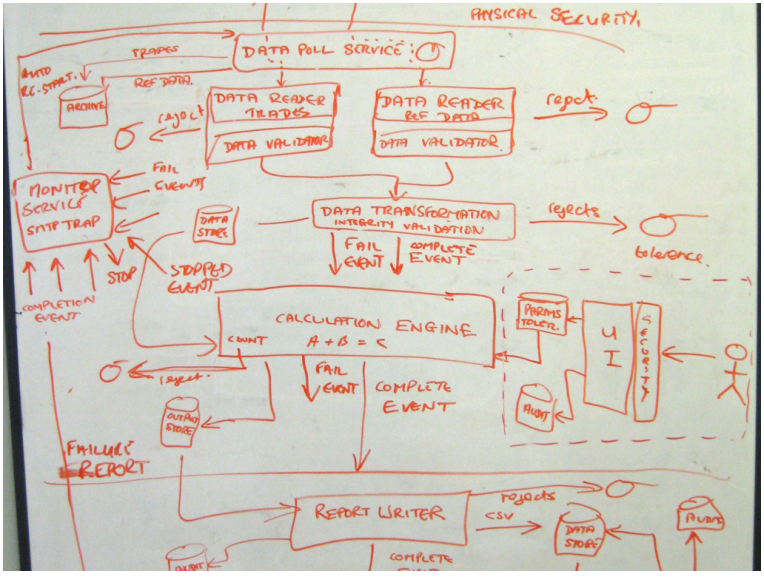


In the bottom left of the diagram is a SQL Server database, and at the top left of the diagram is a box labelled "Application". Notice how that same box is annotated (in green) "Console-C#". This system seems to be made up of a C# console application and a database. But what about the other boxes?

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 understand how they exist in the wider context of architectural building blocks.

Choose your own adventure

This is the middle part of a more complex diagram.



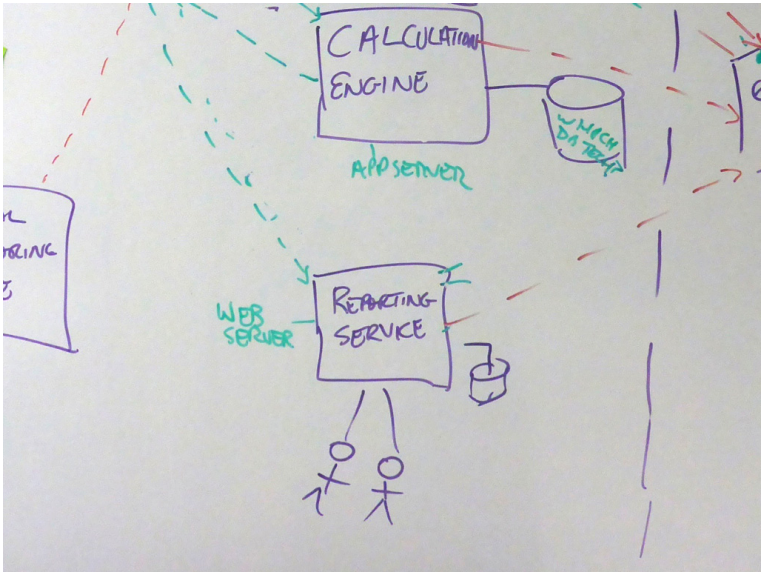
It's a little like those "choose your own adventure" books that I used to read as a kid. You would start reading at page 1 and eventually arrive at a fork in the story where you decide what should happen next. If you want to attack the big scary creature you've just encountered, you turn to page 47. 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 complicated, it's trying to show everything, and the single colour being used does not help. Removing some information and/or using colour coding to highlight the different paths through the architecture would help tremendously.

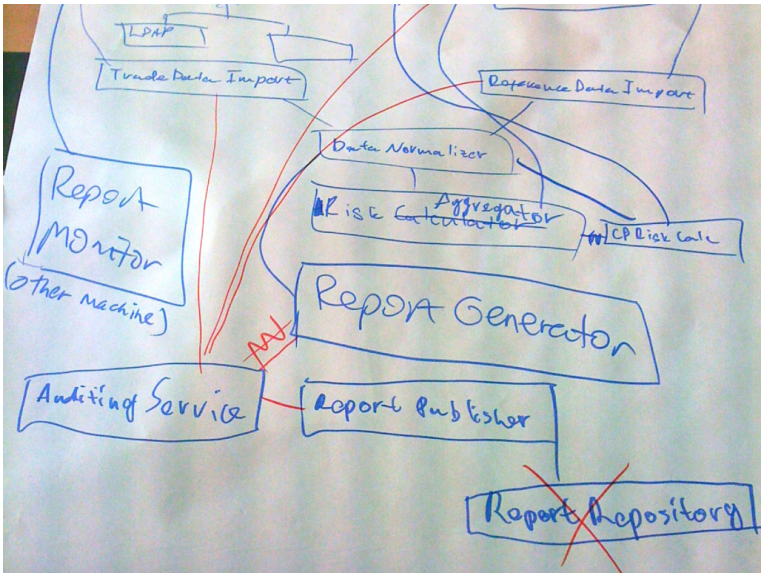
Anonymous clones

To pick up on something you may have noticed from previous diagrams, I regularly see diagrams that include unlabelled users/actors. Essentially they are anonymous 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, line styles, 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 are missing.
- 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, shapes, line styles, etc) is not consistent between diagrams.
- The naming of elements is not consistent between diagrams.
- The logical order 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 "boxes and arrows" 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.