The lost art of software modelling
Over the past decade, many teams have thrown away big design up front.
“We’re agile.”
“It’s not expected in agile.”
“Are we allowed to do up front design?”
“We don't do up front design because we do XP.”
Unfortunately, architectural thinking, documentation, diagramming, and modelling were also often discarded.
Simon Brown
Independent consultant specialising in software architecture, plus the creator of the C4 model and Structurizr
@simonbrown
Financial Risk System

1. Context
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 of losing money they are exposed to, 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 needs a new Risk System.

1.1. 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 to a network share at the close of business at 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

1.2. Reference Data System
The Reference Data System stores all of the reference data needed by the bank. This includes information about counterparties (other banks). A file-based XML export is also generated to a network share at 5pm in New York, and it includes some basic information about each counterparty. A new reference data system is due for completion in the next 3 months, and the current system will eventually be decommissioned. The current data export includes:

- Counterparty ID, Name, Address, etc...

2. Functional Requirements
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.

3. Non-functional Requirements
a. Performance
   - Risk reports must be generated before 9am the following business day in Singapore.
b. 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 slow but steady growth of 10 additional trades per day.
   - The Reference Data System 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.
c. Availability
   - Risk reports should be available to users 24/7, but a small amount of downtime (less than 30 minutes per day) can be tolerated.
d. Failover
   - Manual failover is sufficient; provided that the availability targets can be met.
e. 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.
f. Audit
   - The following events must be recorded in the system audit logs:
     - Report generation.
     - Modification of risk calculation parameters.
g. 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.
h. 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.
i. 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 the system.
     - When reports have not been generated before 9am Singapore time.
j. Data Retention and Archiving
   - Input files used in the risk calculation process must be retained for 1 year.
k. Interoperability
   - Interfaces with existing data systems should conform to and use existing data formats.
Design a software solution for the "Financial Risk System", and draw one or more architecture diagrams to describe your solution.
<table>
<thead>
<tr>
<th>File Retriever</th>
<th>Scheduler</th>
<th>Auditing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Archiver</td>
<td>Risk Assessment Processor</td>
<td>Risk Parameter Configuration</td>
</tr>
</tbody>
</table>
UML?
“I’m the only person on the team who knows it.”
In my experience, optimistically,

1 out of 10 people use UML
“You’ll be seen as old.”
“You’ll be seen as old-fashioned.”
“We don’t want to tell developers what to do.”
“It’s too detailed.”
Very elaborate waste of time
Just use a whiteboard!
“just use a whiteboard”
“The value is in the conversation.”
They are all excellent, as long as there is a conversation about their meaning and intent. It's the accompanying conversation that matters.
“the value is in the conversation” only works if you’re having the right conversations
What’s wrong these diagrams?
Swap and review your diagrams

1. Do the solutions satisfy the architectural drivers?
2. If you were the bank, would you buy this solution?
It’s impossible to answer those questions
If you can’t see and understand a solution, you can’t evaluate it.
If you’re going to use “boxes & lines”, at least do so in a **structured way**, using a **self-describing notation**
To describe a software architecture, we use a model composed of multiple views or perspectives.

Architectural Blueprints - The “4+1” View Model of Software Architecture
Philippe Kruchten
The description of an architecture—the decisions made—can be organized around these four views, and then illustrated by a few selected *use cases*, or *scenarios* which become a fifth view. The architecture is in fact partially evolved from these scenarios as we will see later.

**Figure 1 — The “4+1” view model**
Why is there a separation between the logical and development views?
Our architecture diagrams don’t match the code.
Model-code gap. Your architecture models and your source code will not show the same things. The difference between them is the model-code gap. Your architecture models include some abstract concepts, like components, that your programming language does not, but could. Beyond that, architecture models include intensional elements, like design decisions and constraints, that cannot be expressed in procedural source code at all.

Consequently, the relationship between the architecture model and source code is complicated. It is mostly a refinement relationship, where the extensional elements in the architecture model are refined into extensional elements in source code. This is shown in Figure 10.3. However, intensional elements are not refined into corresponding elements in source code.

Upon learning about the model-code gap, your first instinct may be to avoid it. But reflecting on the origins of the gap gives little hope of a general solution in the short term: architecture models help you reason about complexity and scale because they are abstract and intensional; source code executes on machines because it is concrete and extensional.
We lack a common vocabulary to describe software architecture
Figure 68. Diagram of a basic circuit.
Software System

Web Application

Logging Component

Relational Database

1 component  
noun  |  com-pö-nent  |  \kəm-ˈpän-t, ˈkäm-, kām-\n
Simple Definition of COMPONENT

: one of the parts of something (such as a system or mixture): an important piece of something

Source: Merriam-Webster's Learner's Dictionary
When drawing software architecture diagrams, think like a software developer.
If software developers created building architecture diagrams...
A common set of abstractions is more important than a common notation
A **software system** is made up of one or more **containers**, each of which contains one or more **components**, which in turn are implemented by one or more **code elements**.
The C4 model for visualising software architecture
c4model.com
Diagrams are maps that help software developers navigate a large and/or complex codebase.
The container diagram shows the containers that reside inside the software system boundary.
Container diagram for Internet Banking System

The container diagram for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)
The component diagram shows the components that reside inside an individual container.
The code level diagram shows the code elements that make up a component.
Plus some supplementary diagrams...
Abstractions first, notation second

Ensure that your team has a ubiquitous language to describe software architecture
The C4 model is notation independent
The lost art of software modelling?
How can we avoid copy-pasting elements across diagrams?
Stop using Visio!
Techniques

5. Continuous delivery for machine learning (CD4ML)
6. Data mesh
7. Declarative data pipeline definition
8. Diagrams as code

We're seeing more and more tools that enable you to create software architecture and other diagrams as code. There are benefits to using these tools over the heavier alternatives, including easy version control and the ability to generate the DSLs from many sources. Tools in this space that we like include Diagrams, Structurizr DSL, AsciiDoctor Diagram and stables such as WebSequenceDiagrams, PlantUML and the venerable Graphviz. It's also fairly simple to generate your own SVG these days, so don't rule out quickly writing your own tool either.
“Diagrams as code” is easy to author, diff, version control, collaborate on, integrate into CI/CD, etc
Domain language of diagramming
(no rules, no guidance)
“Diagrams as code 2.0” makes this model based, separating content from presentation.
workspace {

model {
    user = person "User"
    softwareSystem = softwareSystem "Software System"

    user -> softwareSystem "Uses"
}

views {
    systemContext softwareSystem {
        include *
        autoLayout
    }
}

Domain language of software architecture
(metamodel and rules)
workspace {

    model {
        user = person "User"
        softwareSystem = softwareSystem "Software System"

        user -> softwareSystem "Uses"
    }

    views {
        systemContext softwareSystem {
            include *
            autoLayout
        }
    }
}

workspace {

model {
    user = person "User"
    softwareSystem = softwareSystem "Software System" {
        webapp = container "Web Application"
        database = container "Database"
    }

    user -> webapp "Uses"
    webapp -> database "Reads from and writes to"
}

views {
    systemContext softwareSystem {
        include *
        autoLayout
    }

    container softwareSystem {
        include *
        autolayout
    }
}
user -> softwareSystem "Uses"

user -> webapp "Uses"
webapp -> database " Reads from and writes to"
container softwareSystem {
    include user -> service1 -> autolayout
}

Software System - Containers

Web Application

Service 1 Database

Service 1 API

Service 2 API

Service 1 [Group]

Service 2 [Group]
How can we avoid our diagrams becoming out of sync when we make changes to our code?
**Authoring tool**
Create diagrams as code (Java, .NET, TypeScript, Python, PHP, etc) or text (DSL, YAML) via a number of different authoring tools.

**Workspace**
A workspace is the wrapper for a software architecture model and views, described using the C4 model and an open JSON data format.

**Custom tool**
Your own tooling to parse the model and views; for integration into other rendering tools, dashboards, service catalogs, etc.

**Rendering tool**
Render views using multiple diagramming tools and formats (Structurizr cloud service/on-premises installation/Lite, PlantUML, Mermaid, WebSequenceDiagrams, Ilograph, etc.).
Diagramming tools are still the first choice for most teams, but some are starting to adopt modelling tools to improve consistency and enable diagram automatic generation.
Abstractions first, notation second
Thank you!