# Week 4: Designing and Documenting Software for Health

## Overview

This week we look at techniques and processes that help with
(a) the task of **designing software**
(b) the challenge of **communicating designs to other members of an extended multidisciplinary team**, ranging from developers to non-technical stakeholders.

## Learning outcomes

After completing week 4, you should have made progress towards learning objective LO7:
**Produce visual and textual documentation to communicate the design of a system in a formal and structured way**

## How this week relates to your assignment

The content and activities in this topic should help you to understand how to capture and **communicate the architectures of systems** at a variety of different levels and for a variety of audiences.

## Materials and activities

The role in software in health is important to helping to record and define the complex systems and tools that we can use to ensure that people are kept safe and well, both as they are being treated and when they are going about their normal lives. In this next video we will take a short look at health software.

### Activity: The Role and Nature of "Architecture" in Software

We typically encounter the concept of architecture, borrowed from building design, in two ways:

- When a new system is being devised, and part of the work to be done is to architect that system - to design it and maintain awareness of the "load-bearing" decisions in the process of its creation.
- When a system is in place, and we want to understand its design, and how it accomplishes its functions, when we talk about it's overall conceptual design as its "architecture".

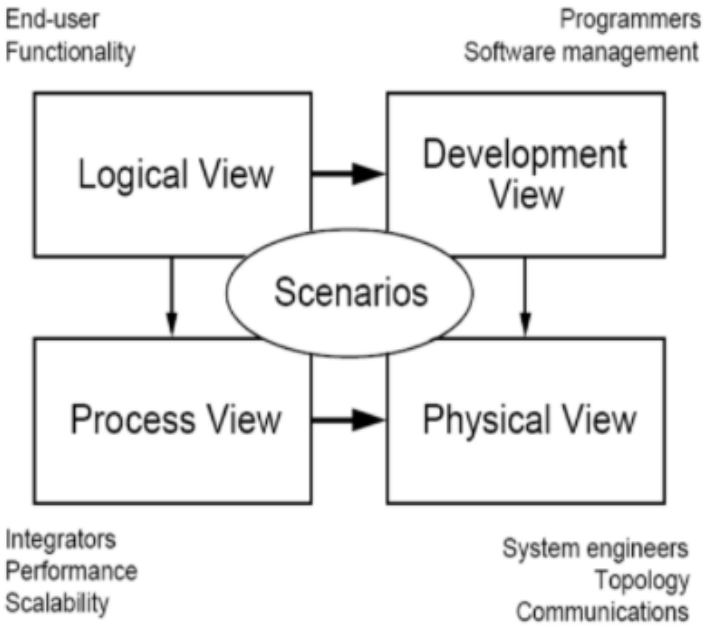
The text below gives a more detailed introduction to software architecture - as you read it, consider what kind of skills a software architect in the healthcare field might need, and what kinds of knowledge base that person might need to call on in the course of performing the role of system or project architect. Share your findings in the forum below.

- **Introduction to Software Architecture**

## Communicating Software Architecture

Whether you're developing something new, or documenting something that already exists, it's often easier to communicate something complex visually than it is to describe it in long-form text. Towards the end of the 20th century, the software industry experienced something of a convergence around the idea that an agreed visual documentation form would be a good thing, and so the **Unified Modelling Language** - **UML** - was born. You've already heard from Grady Booch, who was part of leading that process of standardisation.

UML consists of a variety of different visual formalisms - kinds of diagram - for describing the concepts that you saw in the architecture introduction in this figure:




**From:** *Philippe Kruchten* **Architectural Blueprints—The "4+1" View Model of Software Architecture** Paper published in IEEE Software 12 (6) November 1995, pp. 42-50 http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf

- Some parts of UML are designed to represent the **logical view** - how the software appears to and provides value to its users.
- Others describe how the software will be built - the **development view**.
- Others describe the ways in which the software will operate and interact with other software - the **process view**.
- The **physical view** describes the pieces that are actually built, and how they fit together.
- **Scenarios** are then used to link these views, to provide a common reference point across them all.

We'll look at some specific elements of UML in the sync days, but for now we're going to consider something that UML didn't really address, but which software architects have come to find increasingly useful.

Simon Brown, whose introduction to this is below, is leading a new movement to supplement technical UML with a model called **C4** - a model of **Systems in Context** that consist of

**Containers for Components that are made of Code**.

### [Visualising software architecture with the C4 model - Simon Brown]

As the lecture says, you can learn more about the C4 Model at c4model.com and he's also devised a tool for drawing C4 diagrams - structurizr.com - and if you sign up there using your UCL email address, you will get privileged access to an academic license for the tool.

### Activity: C4 and the APFT Wound care management system

- Sign up on Structurizr.com and try individually to capture (a) the current situation and (b) the future vision based on the information in the APFT dossier (see below) in high-level C4 diagrams.
- Then use the C4 diagram activity forum below to share your diagrams with the rest of the class. Review some others' postings. Is Simon Brown right about how groups work better with standardised diagrams?

## Introduction to Software Architecture

This essay is on software architecture. At the end of this you should understand:

- The role and meaning of “software architecture”
- The process of system design
- The role of communication
- Architecture at various different levels

Different types of software may be designed in different ways. For example, an application that runs on a laptop that is not connected to the internet will need to be designed to consider that it will need to store data on the laptop’s hard drive. If it needs to send information to a central system the application will need to be able to save those messages until it is connected, and also potentially validate that the messages have arrived safely and without error. Conversely, a web-based application will need a client computer that is always connected, consider the need to working with different browsers, and have to work from a web server that may be connected to a database server. These design considerations are known as the architecture of the software.

Architecture is “the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.” IEEE Computer Society, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems: IEEE Std 1472000. 2000.

The traditional job of an architect is someone who designs buildings. An architect has to consider the materials that are available for construction, how the new building is to be used by the people who inhabit it, the cost of construction, and the design of the features, amongst many other things. These can be summarised as:

- Architecture is the design of a building,

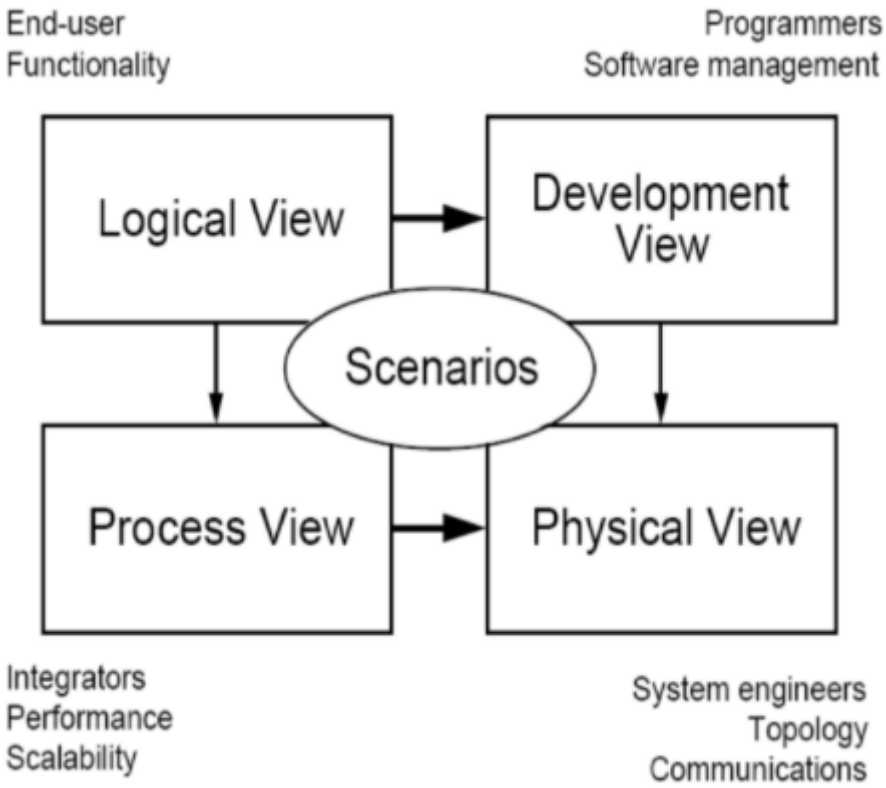
- It is done in response to a brief,
- It provides an analysis of the brief that verifies that design meets the brief without needing to build the system, to reduce risk and cost.
- And enables high-level design strategic design decisions in response to the needs of the future users of a building.

In order to communicate these assumptions and design requirements an architect will create drawings and models. These can then be used to communicate the “what” is to be build to others. These drawings:

- Define a vocabulary of concepts
- Consistently expressed
- Specify how a finished building should be including shapes, dimensions and materials
- Drawings don’t specify how to build the building

Software architecture fulfils much the same basic role. It creates an understandable and communicable abstraction of a complex system that will be developed as software. This then provides the developers, designers and stakeholders a basis to analyse the software solution’s behaviour before it has been built. As with traditional architecture this can massively reduce the risk of parts of the design being wrong, helps stakeholders to understand the consequences of design decisions on the whole system, allowing for changes to the design while it is still both relatively easy and cheap to do so.

Software architecture descriptions can also be organised into views in order to recognise the different viewpoints, requirements and models that different stakeholders have of a single system. This is similar to how different blueprints of a building may be drawn to describe different building needs: the sanitation and physical infrastructure, room design, etc. An example of this is the 4+1 model, as described in the figure below. This allows the architect to use different scenarios to explore the necessary application design from the perspectives of the end users, the developers, stakeholders interested in performance and future scalability as well as systems engineers and devops who need to understand the necessary hardware and physical infrastructure for the application.




As software development has grown, common features have been identified that are often re-used. These are known as **software design patterns**. These allow for standardised models of construction, following styles with recognised features to support particular functionality, such as performance.

An example of a commonly-used software pattern is **Model View Controller**, or **MVC**. This pattern allows for a separation of concerns that makes the software code easier to manage and the individual components reusable. In this pattern each component has the following features:

- **The Model**: This layer manages the data for the application. In many modern programming languages this is done using “object orientation” where the data model is designed around representations of the real-world objects that make up the system.
- **The View**: This is the view of the data presented to the user via some kind of user interface. For example in a web app this may be the part of an HTML page that displays the data. Separating the view from the other components means that the Model and Controller logic could be used - or re-used - in multiple places with different data. Similarly, different views in different places - such as in a web app and a mobile app - might be linked to the same model.
- **The Controller**: This manages altering data in the model, and is sometimes integrated with the View component (a touch screen widget that both shows the current data and allows a user to alter it by dragging something, for example). It can take input from the user and update the model accordingly.

In summary: software architecture has a key role in ensuring that complex software meets the needs of its users in a cost effective way. It allows for the requirements to be mapped to the design of the system and facilitates communication between stakeholders. It specifies this design in a common language that can be understood and shared. The development team can then use this design to order to build the software application in a way that meets the stakeholder requirements.

Back to summary

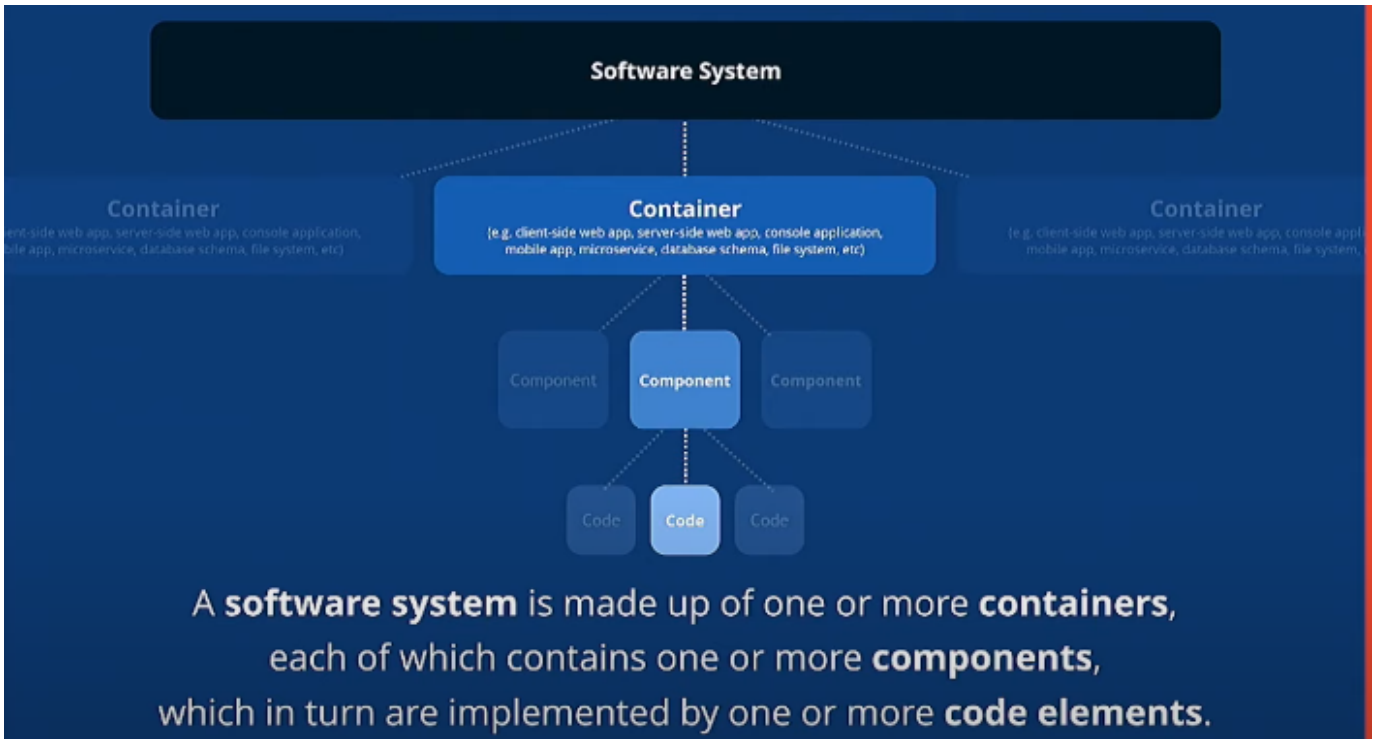
## Visualising software architecture with the C4 model - Simon Brown

https://www.youtube.com/watch?v=x2-rSnhpw0g

- Moving fast in the same direction as a team (being agile) requires **good communication**. Lack of communcation slows things down because teams do not have a common vision of what they are going to build/have built.
- There are many **different audiences** for diagrams and documentation, all with **different interests** (e.g. software architects, software developers, operations and support staff testers, Product Owners, project managers, Scrum Masters, management, business sponsors, potential customers, potential investors, ...)
- When drawing software architecture diagrams, think like a software developer.
- A **common set of abstractions** is more important than a common notation. UML gives you a set of elements and a standard notation to draw those elements in a diagram . However the abstractions should be much more important than the notation.

### What can we consider as abstractions?

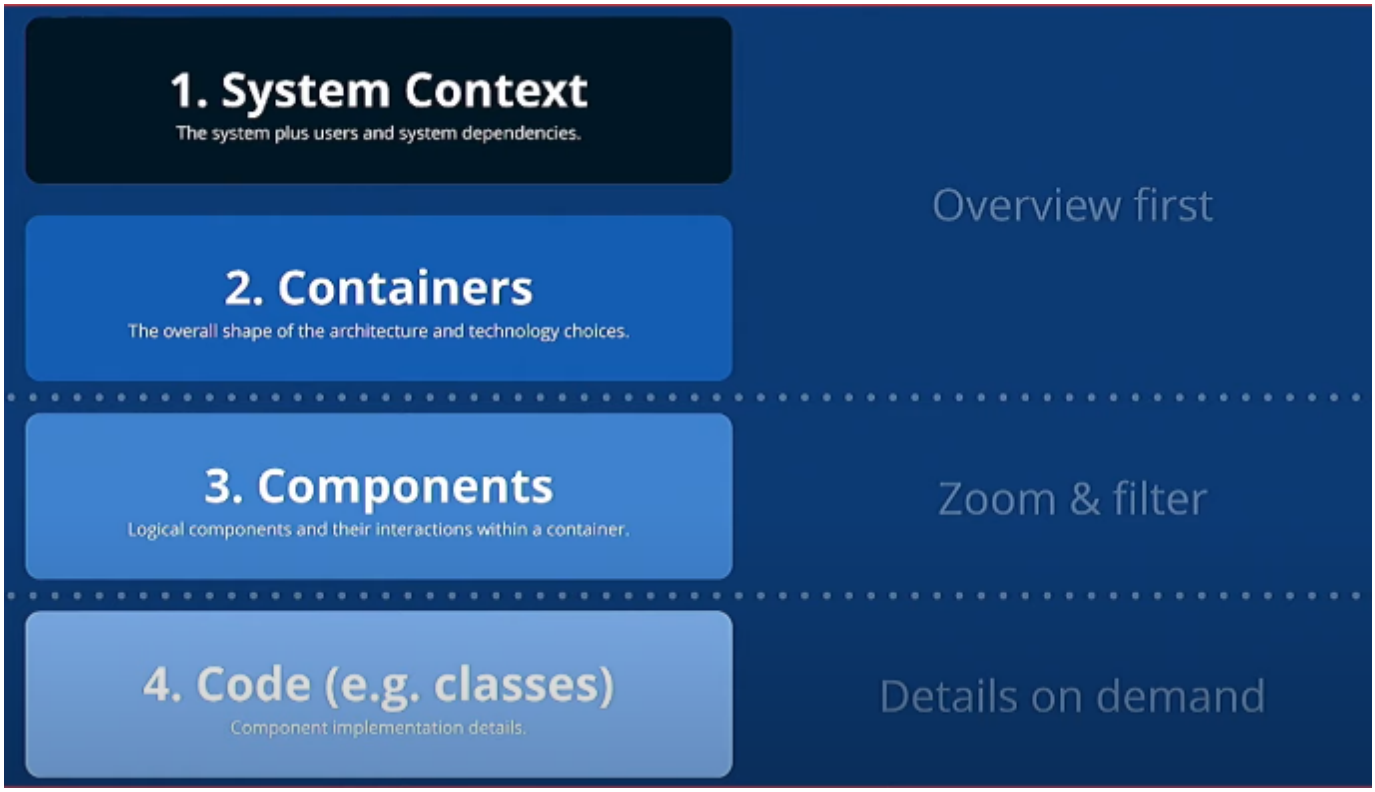
A **software system** is made up of one or more **containers** (i.e. applications or datastores, not dockers!), each of which contains one or more **components** (i.e. modular elements part of a container, e.g. an interface), which in turn are implemented by one or more **code elements**.



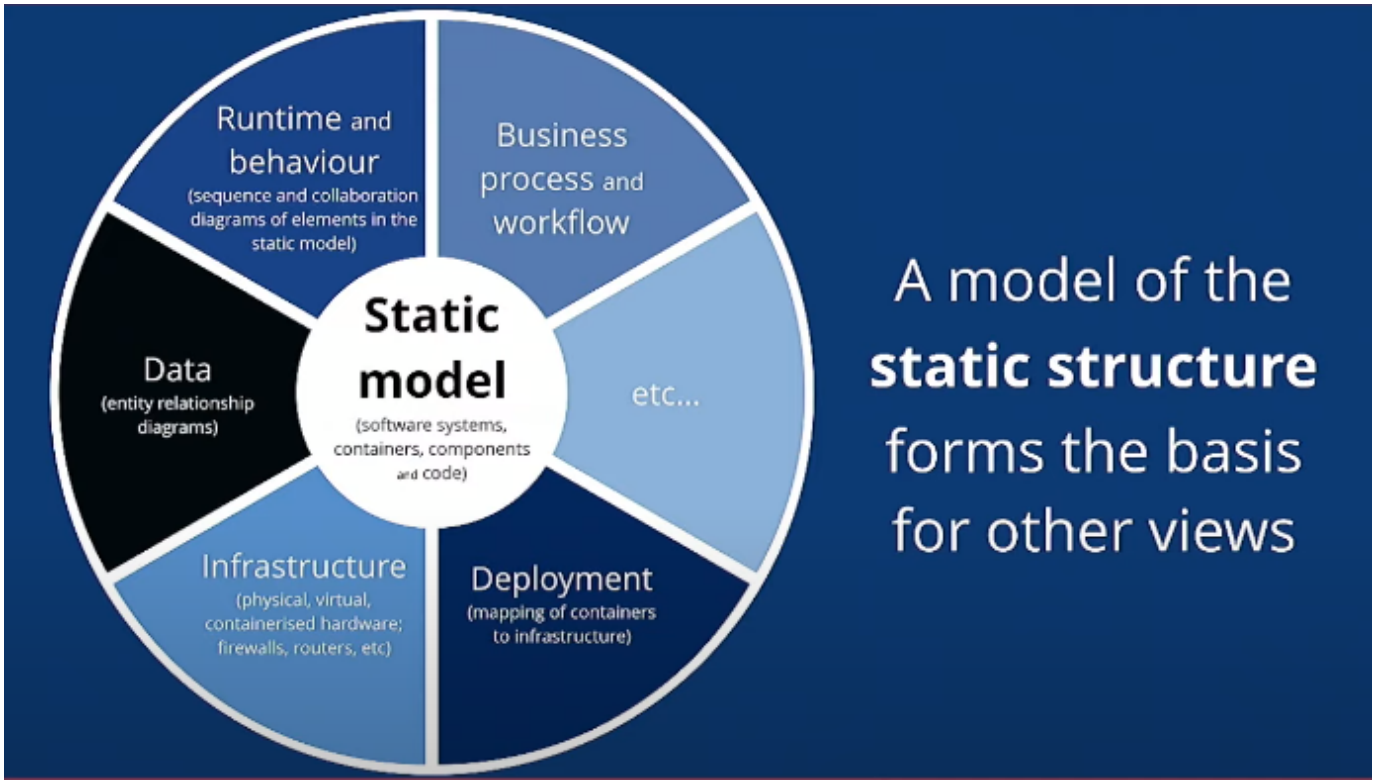
## C4 MODEL

C4:

- **Context**
- **Containers**
- **Components _ Code** (e.g. Classes)

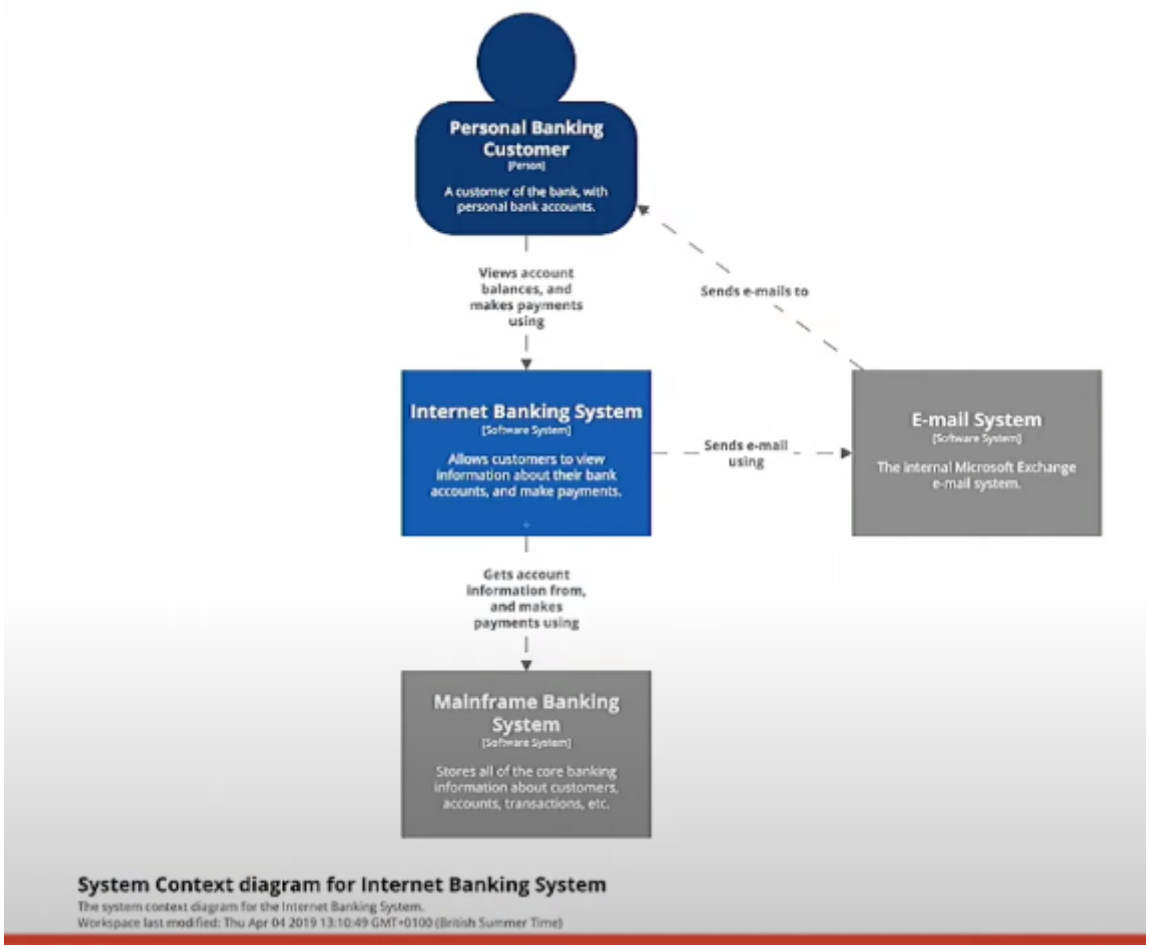



These 4 Cs are a collection of diagrams that can be drawn in any relevant order. **Diagrams** can be considerd as **maps**, i.e. tools that help software developers navigate a large and/or complex database.

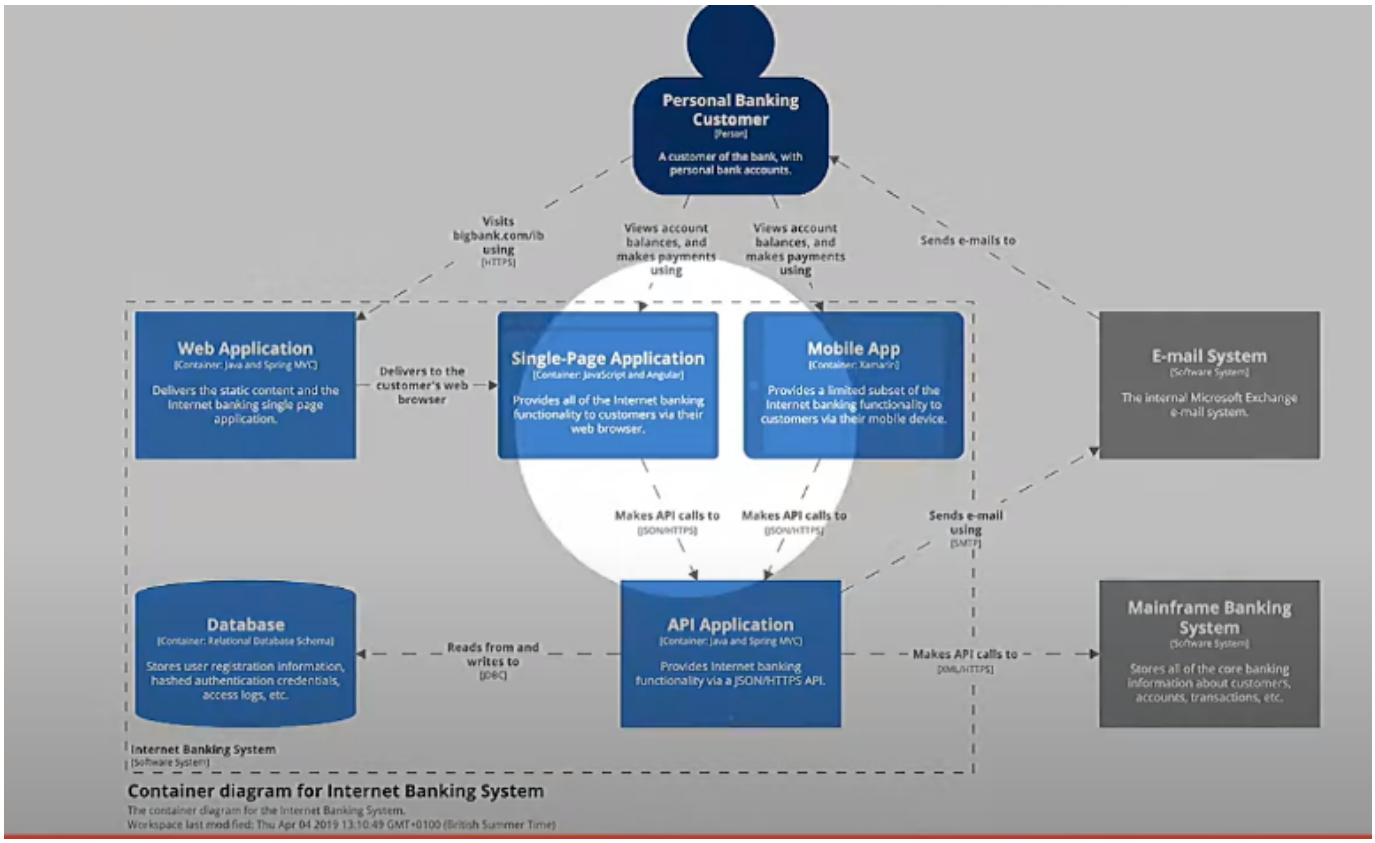



- **Example: Internet banking system**

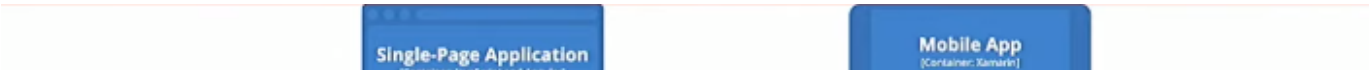
  - **Level 1: System context diagram**: it shows the system we are working on and the surrounding environment (e.g. agents, other systems, etc). Not too much detail is provided and it is good for a wide range of audiences



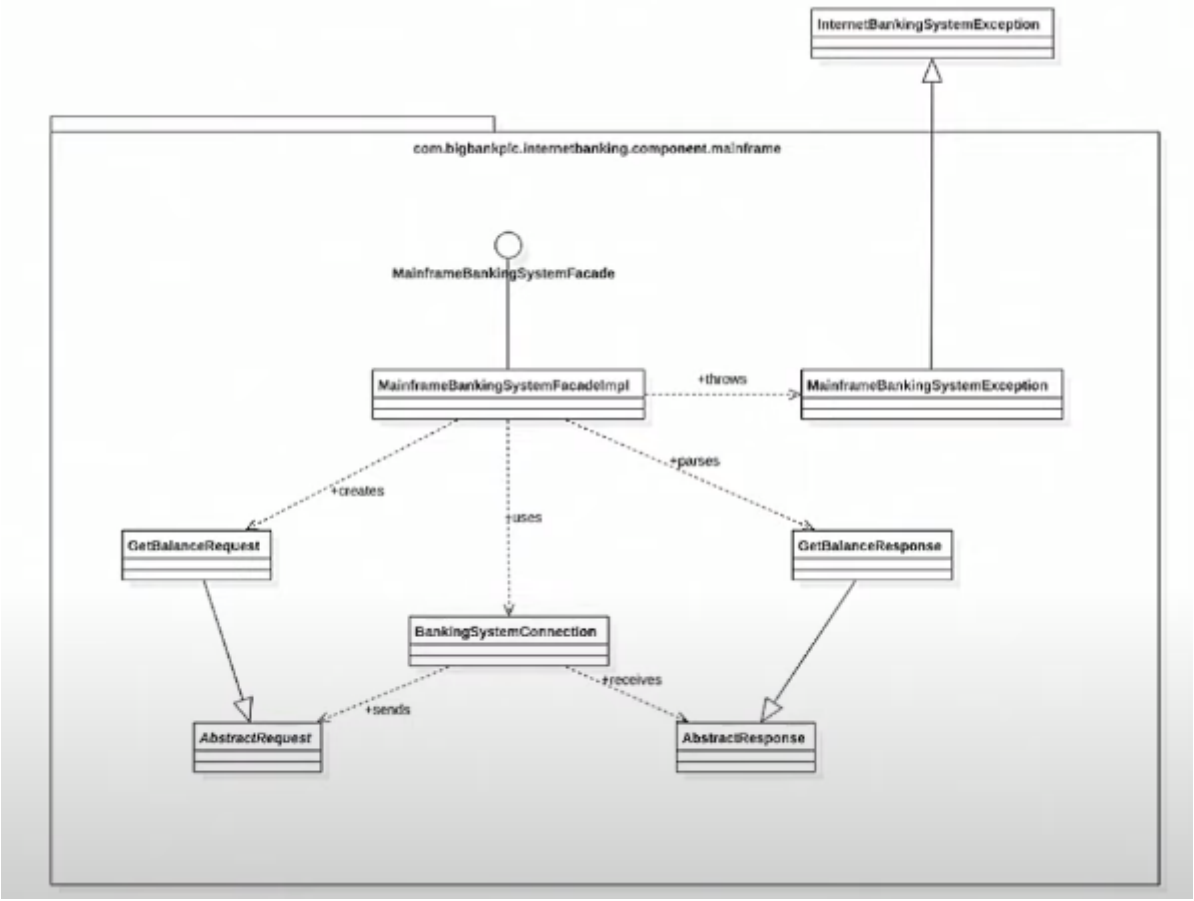
- **Level 2: Container Diagram**. It shows all the apps and data stores of the container and how they relate and connect at runtime.




- **Level 3: Component Diagram**. E.g. we can create one for the API application of the Internet Banking System.

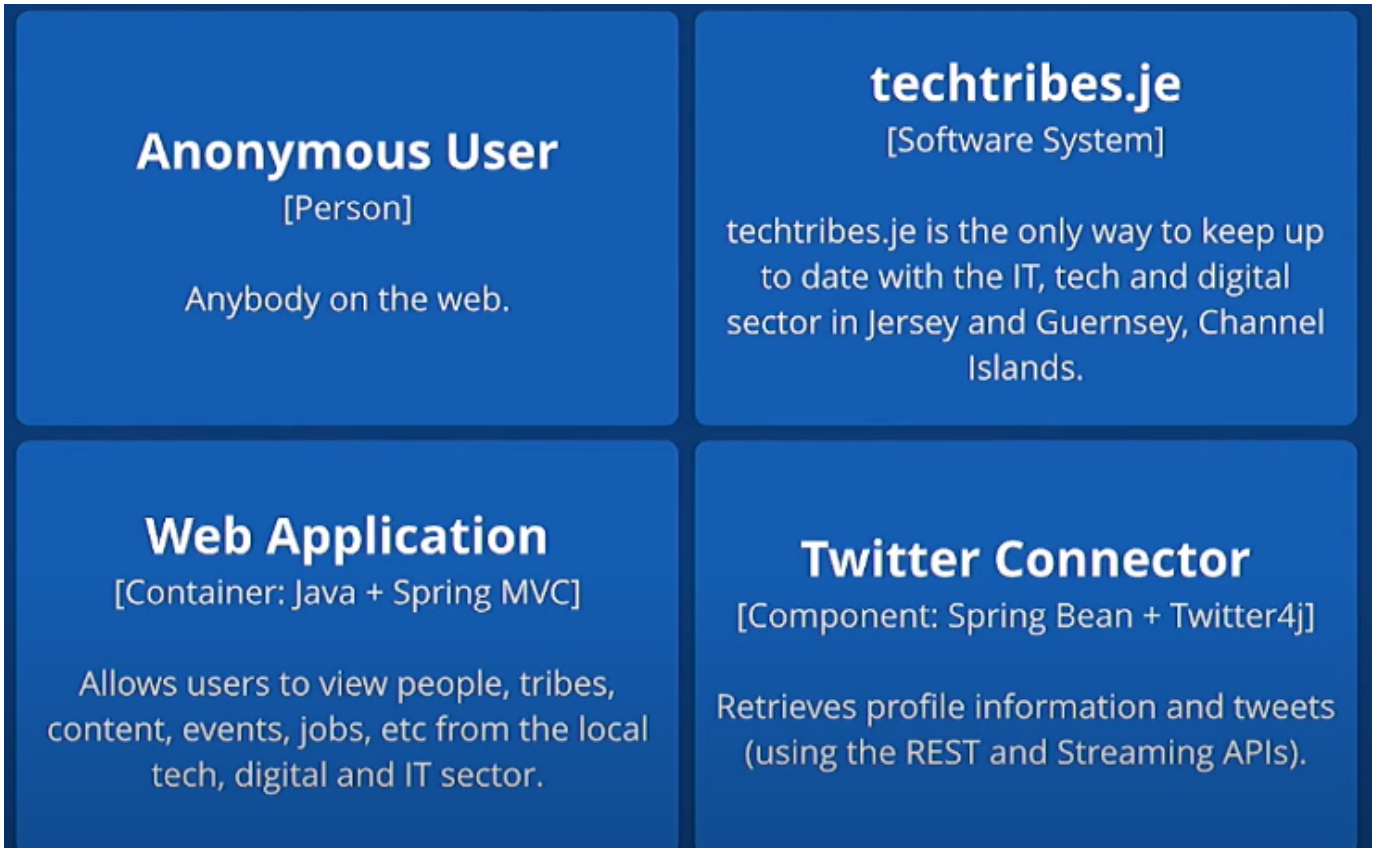


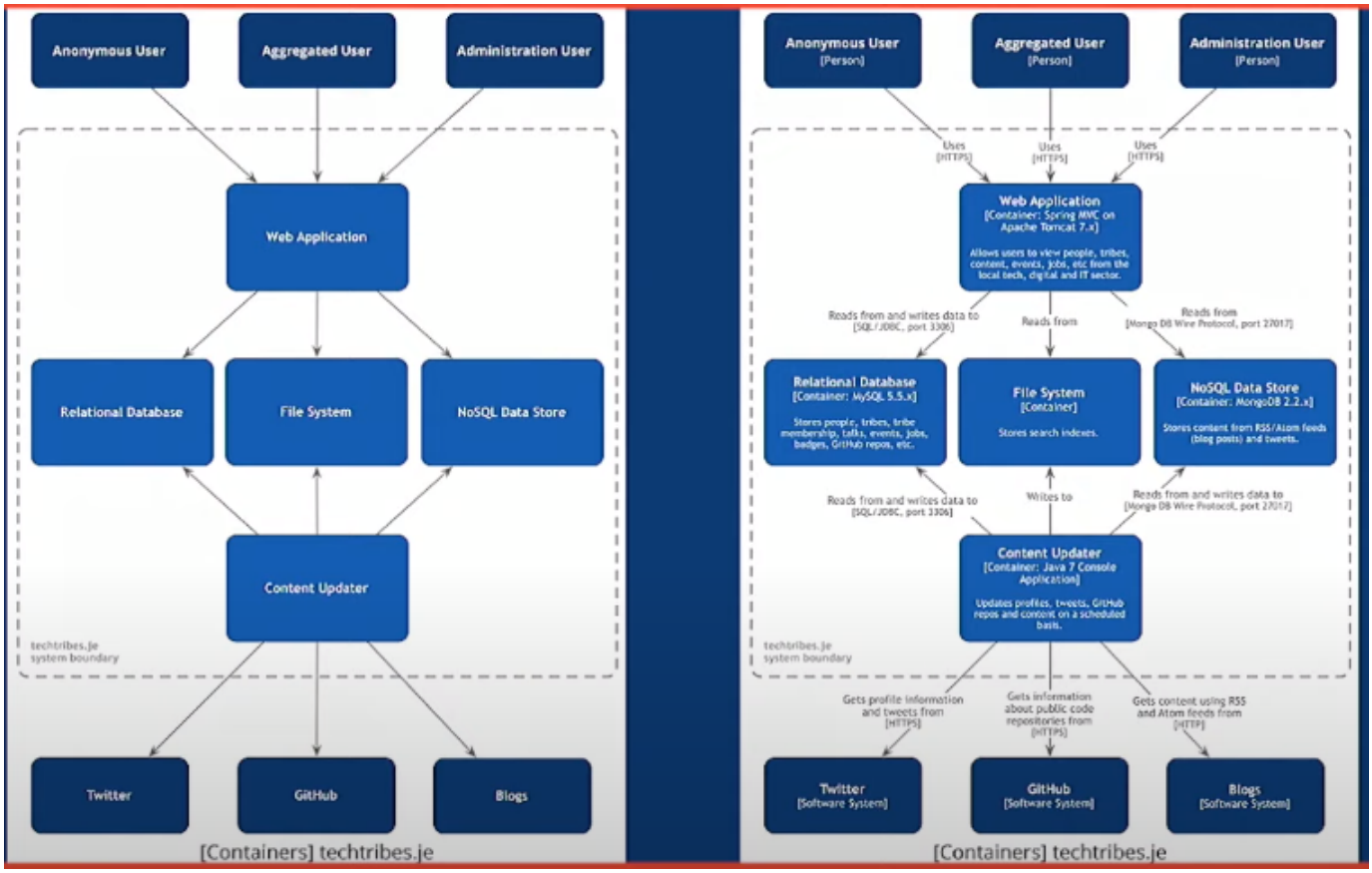

- **Level 4: Code Diagram**. It is not recommended to create this, it can be generated automatically. The only important thing is that Level 3 reflects the code in Level 4. UML however could be used to document a class diagram:




## Notation

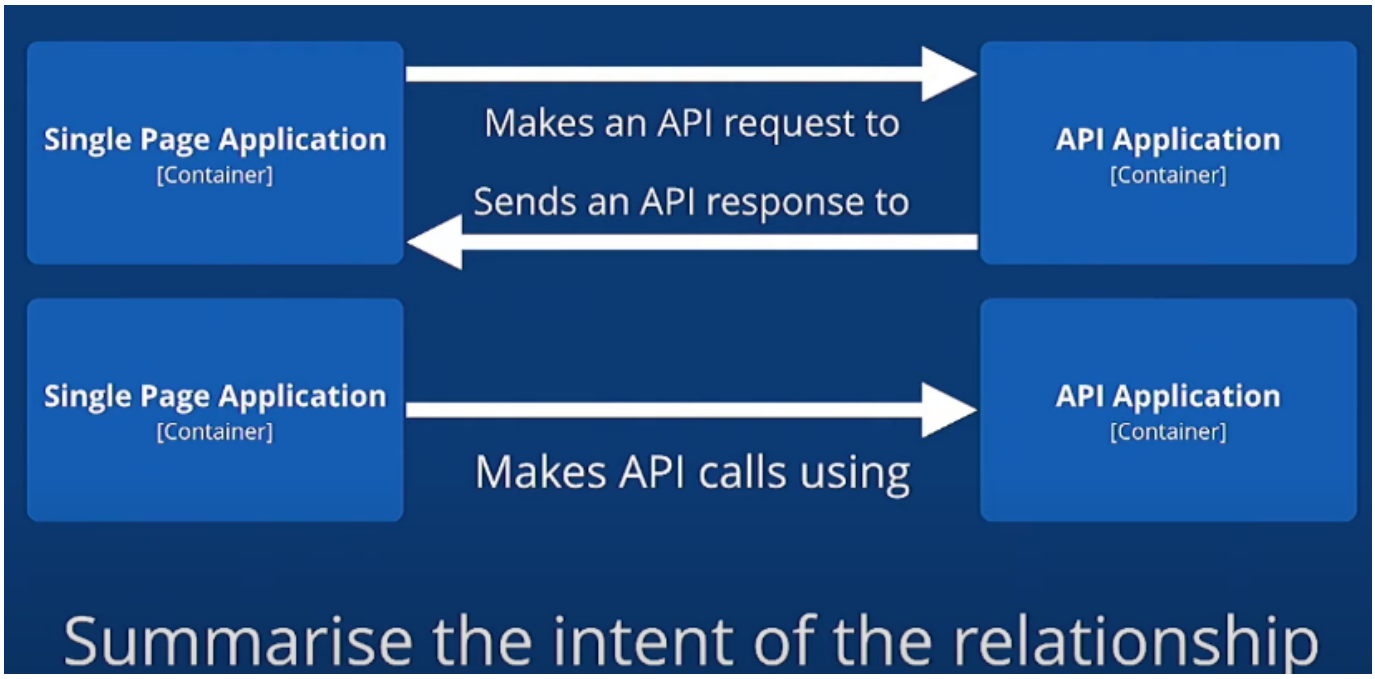
- **Put titles on pictures**: short and meaningful, including the diagram type, numbered if diagram order is important (e.g. System context diagram for Financial Risk System)
- **Layout**: sticky notes and index cards make a great substitute for hand-drawn boxes, especially if you don't have a whiteboard.
- **Visual consistency**: try to be consistent with notation and element positioning across diagrams.
- **Acronyms**: be wary of acronyms, especially those related to the usiness/domain that you work in
- **Elements/boxes**: start with simple boxes containing the element name, type, technology (if appropriate) and a description/responsibilities. Do not focus on shape and colours! Example:



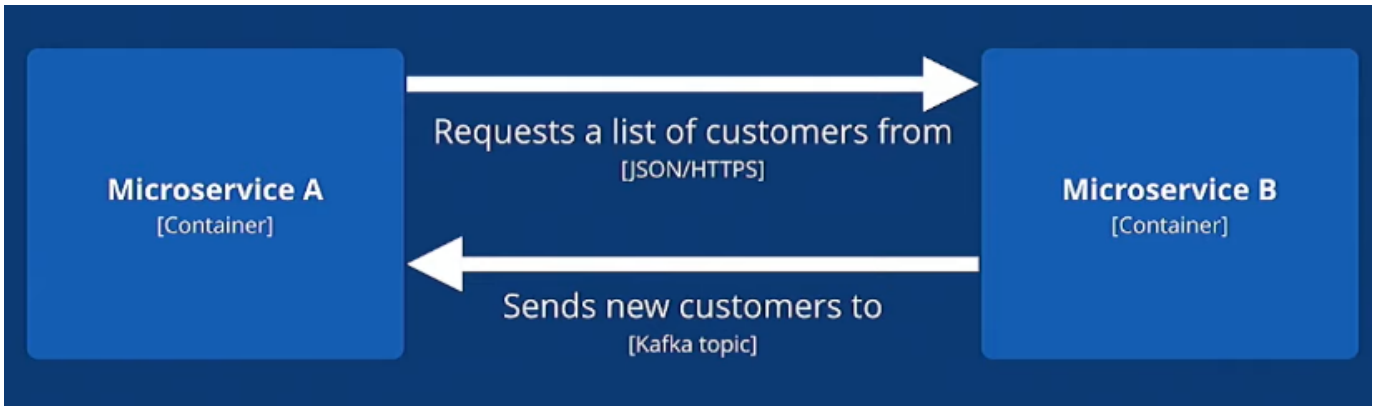




- **Lines**: favour **uni-directional lines** showing the **most important dependencies** or **data flow**, with an annotation to be explicit about the purpose of the line and direction. It is important

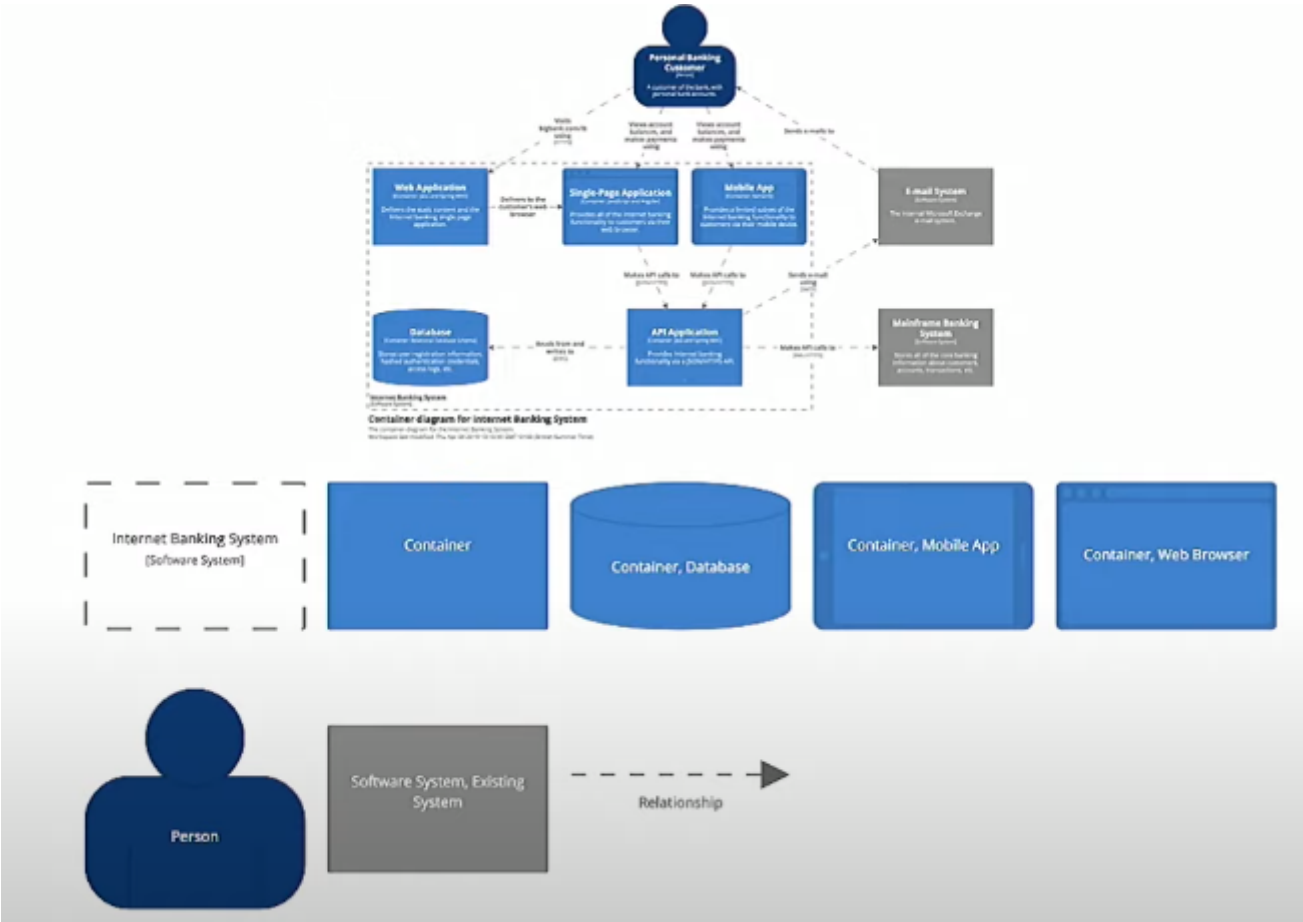
also to **summarise the intent of the relationship** (in this example, the double arrow has been summarised in one arrow only):



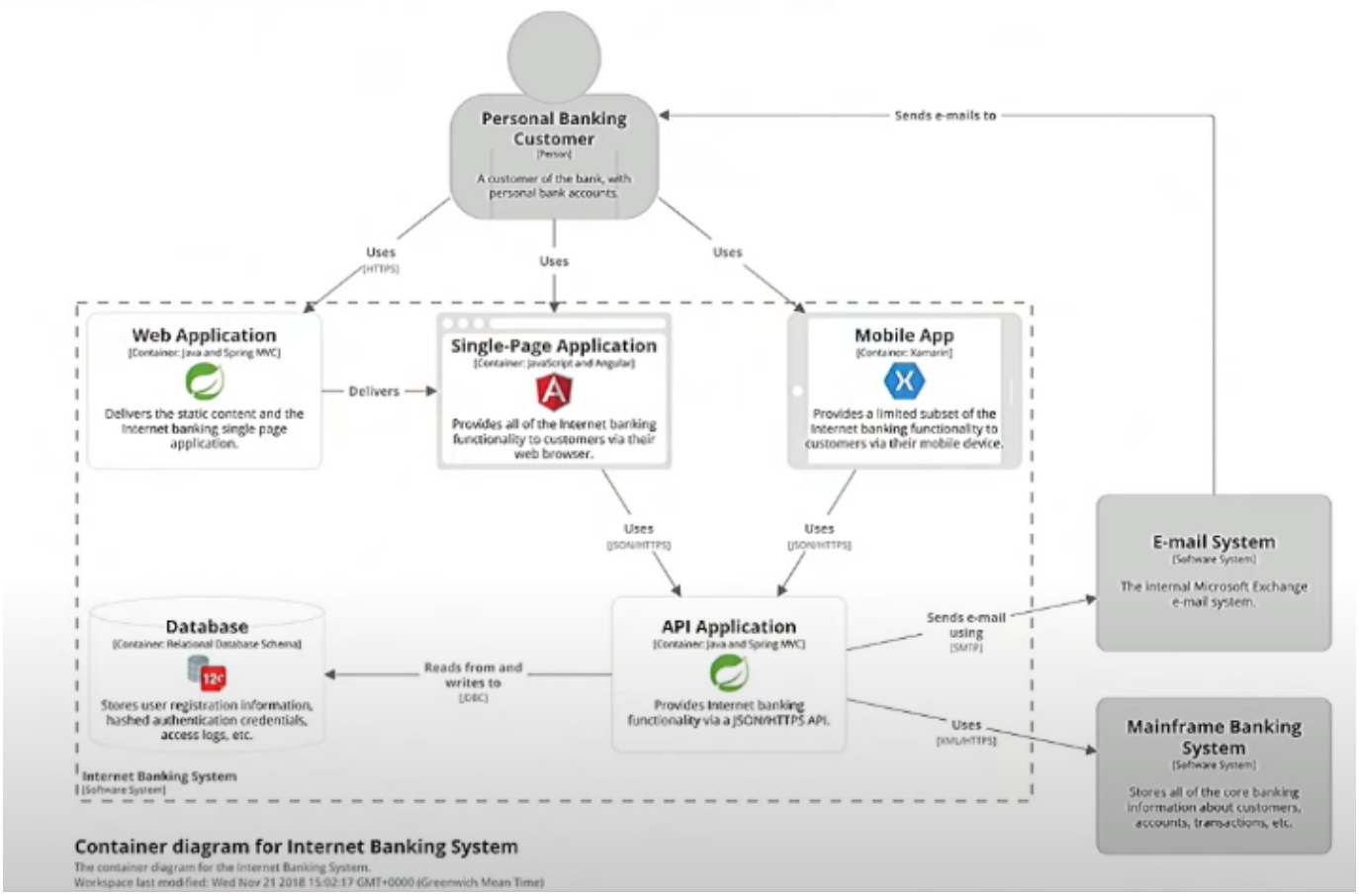

Show both directions, though, when the intents are very different:




- Define **key/legends** used in the diagram




- Use shape, colour and size to complement a diagram that already makes sense!
- Use icons to supplement text, not to replace it!



- Increase the **readibility** of **software architecture diagrams** so they can stand alone, without needing a verbal explanation.
- Any narrative should complement the diagram rather than explain it.

**What tools do you recommend?**

- C4-PlantUML
- Structurizr

Back to summary

## Activity: C4 and the APFT Wound care management system

- Sign up on structurizr.com and try individually to capture (a) the current situation and (b) the future vision based on the information in the APFT dossier (see below) in high-level C4 diagrams.
- Then use the C4 diagram activity forum below to share your diagrams with the rest of the class. Review some others' postings. Is Simon Brown right about how groups work better with standardised diagrams?

## Activity: Bridging between Health Service and System Views

- At F2F, we'll go beyond C4 and look at two parts of UML that are on Grady Booch's list of the most important - **use case diagrams** and **class diagrams**.
- However, before we can do that, we need to make a shift in how we think about the system. So far we've considered stakeholders and their requirements, and we've mapped the patient pathway through the health system. Now we need to consider how what we know about the process and the context translates into the second two Cs of C4 - **actual functional software components **made up of **actual working program code**.
- One way to do this is to start from the patient pathway and build a data flow diagram that looks not at the flow of patients through health services, but strictly at the **data and data flows that underpin each part of that process and what data has to move where in order to enable the right things to happen**. Each of the software components that we are specifying - and which a software engineering team will ultimately build - reflects some part of one of those **processes that capture data, enable it to flow, or persist it in a storage mechanism**.
- In some cases the software component will operate in the background; in others it will need a user's involvement, and thus some kind of user interface. In some cases it will provide access to another component or even another container, reading and or writing data,

- Part One of your task is to take your patient pathway for the APFT case study and see if you can see how a data flow diagram might look. What are the actors/agents? What is the data? What are the datastores? What processes capture data, and move it around the system?
- Part Two of your task is to reflect individually on which of these diagrams will be a useful part of your briefing to the CIO:
    - **Patient pathway**
    - **C4 high level**
    - **Data flow diagram**

  The diagram(s) you consider useful are those you might want to begin to work up ready for inclusion in your presentation submission and in your final briefing document. We'll look at more diagrams, including some **UML**, at F2F.