# Software Architecture

Davide Fucci

# Agenda

1. What is software architecture
2. Fundamental Structures
3. Architectural Patterns
   - *Hands on*: create your first architecture
4. The C4 architectural model
   - *Hands on*: improve your architecture using C4
5. Architecture diagram as code using Structurizr
   - *Hands on:* apply Structurizr
   - *Hands on (bonus):* apply pyStructurizr

# What is a Software Architecture?

# Definition

The architecture of a software system is the **set of structures** needed to reason about the system.

These structures comprise software elements, relations among them, and properties of both.

# Reasoning about a Software System

Who cares about the system's properties?

**Developer**

- Where to add a new feature?

**System Administrator**

- Can the system handle to 50 times the load before holiday season?

**Security Auditor**

- How is information security

**Software Architecture Lesson:**

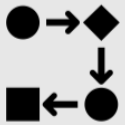
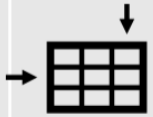
Everything all at once is not possible. There are always *trade-offs*!

- Want 100% service availability 24/7/365? Super-high operation costs!
- Guaranteed system response within 15ms? Likely need difficult shortcuts in code!

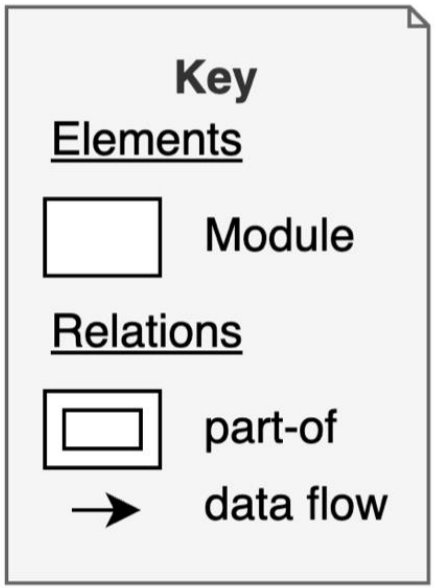
**Project Manager**

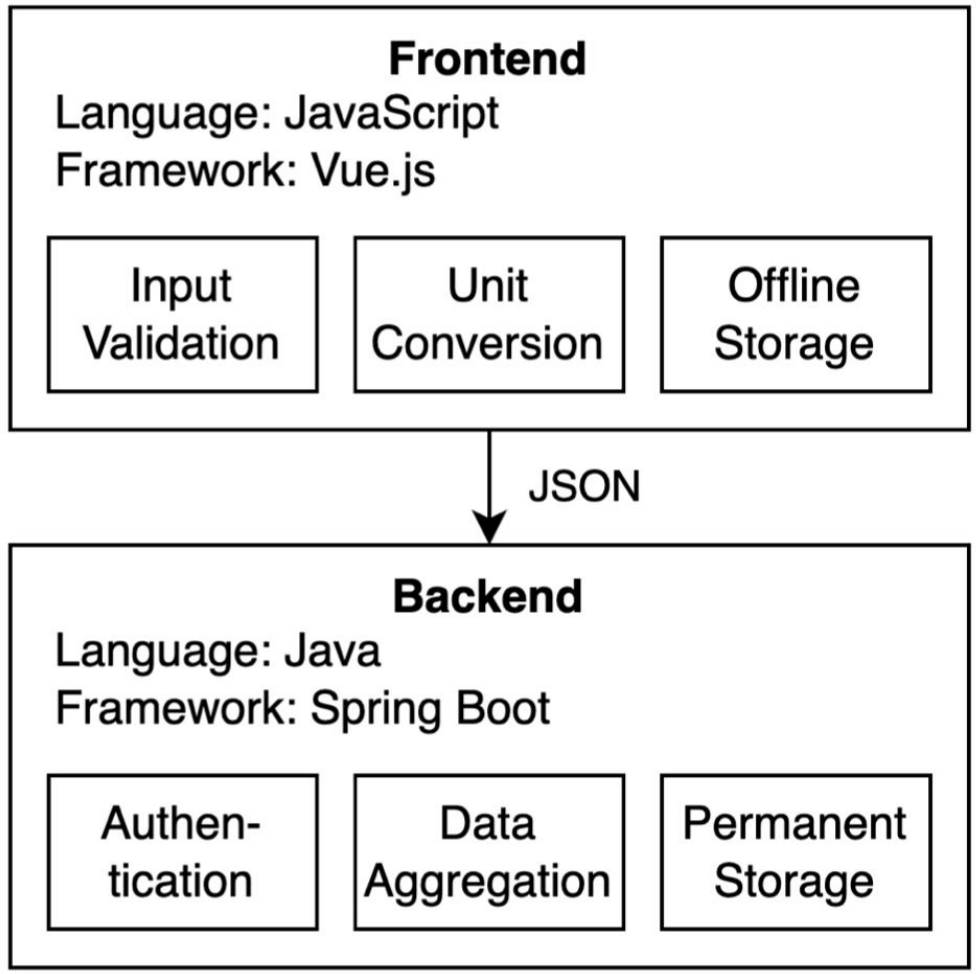
- How much to pay for our cloud?
- How many engineers do we need when?

# Common Structures in Architecture

| Module Structures | Component-and-Connector (C&C) Structures | Allocation Structures |
|---|---|---|
|  Decomposition of a system into smaller parts (typically static). |  Interaction between runtime elements (typically dynamic). |  Mapping of software structures to non-software structures or resources. |
|  **Developer:** Where to fix a bug? |  **Security Auditor:** How is information security guaranteed? |  **System Administrator:** Will the system handle to 50 times the load? |
| *hierarchical structures, layers, class and data structures, …* | *data flow, service structures, concurrency, …* | *deployment (components to machines), work assignment (tasks to people), …* |

# Example: Module Structure



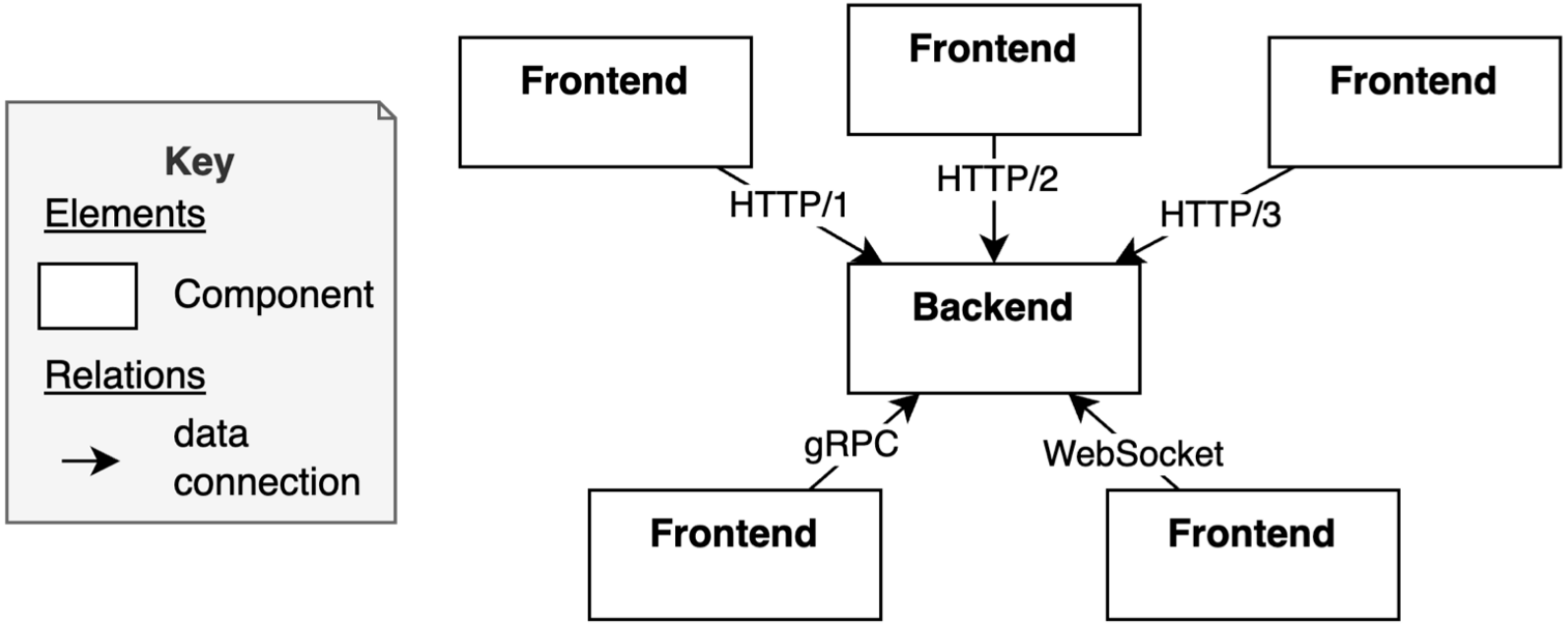

**Questions answered:**

- What parts exist during implementation?
- How are they related?

**Further Questions:**

- Other relations between existing parts?
- Other parts on the same level?
- Parts on the next level?

# Example: C&C Structure



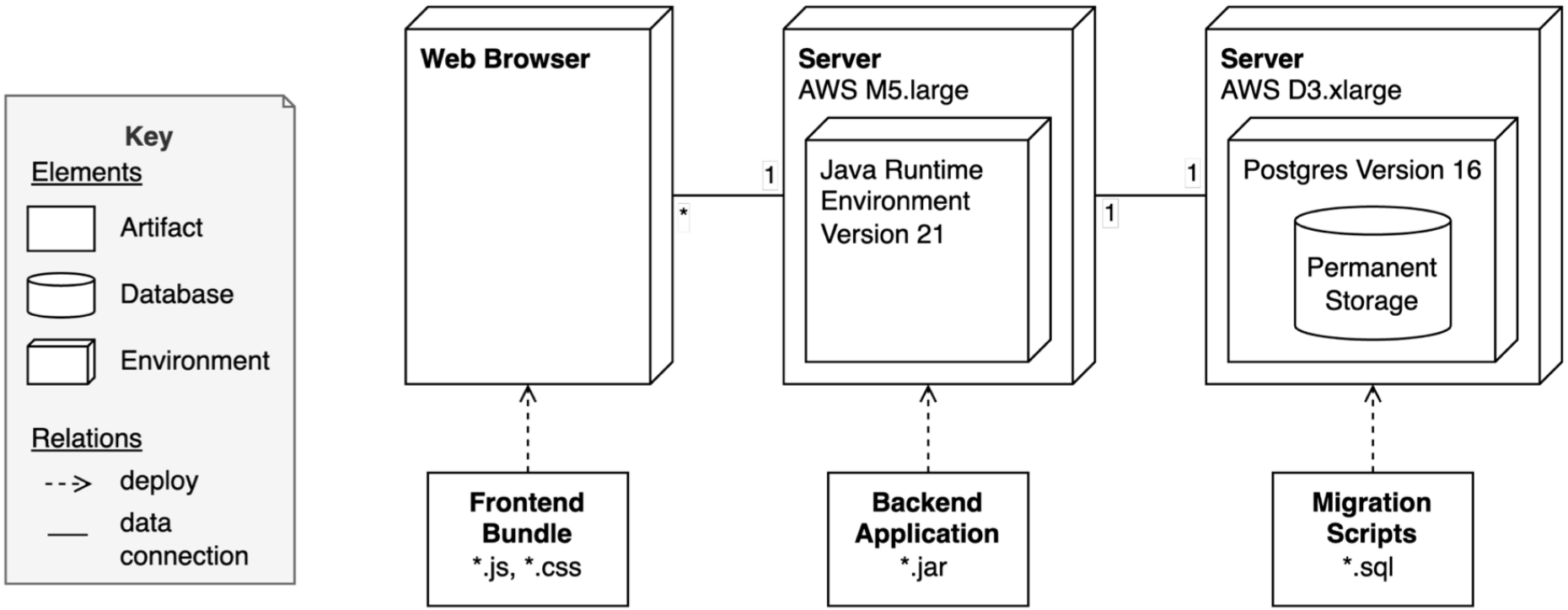

**Questions answered:**

- What parts exist during runtime?
- How are they connected?

**Further Questions:**

- Lifecycle of connections: How to establish, maintain, disconnect?
- Connections' properties (encryption, bandwidth, latency, fault-tolerance, …)?

# Example: Allocation Structure




**Questions answered:**

- How many instances of which parts are deployed where?

**Further Questions:**

- How to deploy? (Webserver to download the Frontend Bundle from? SSH for database? …)

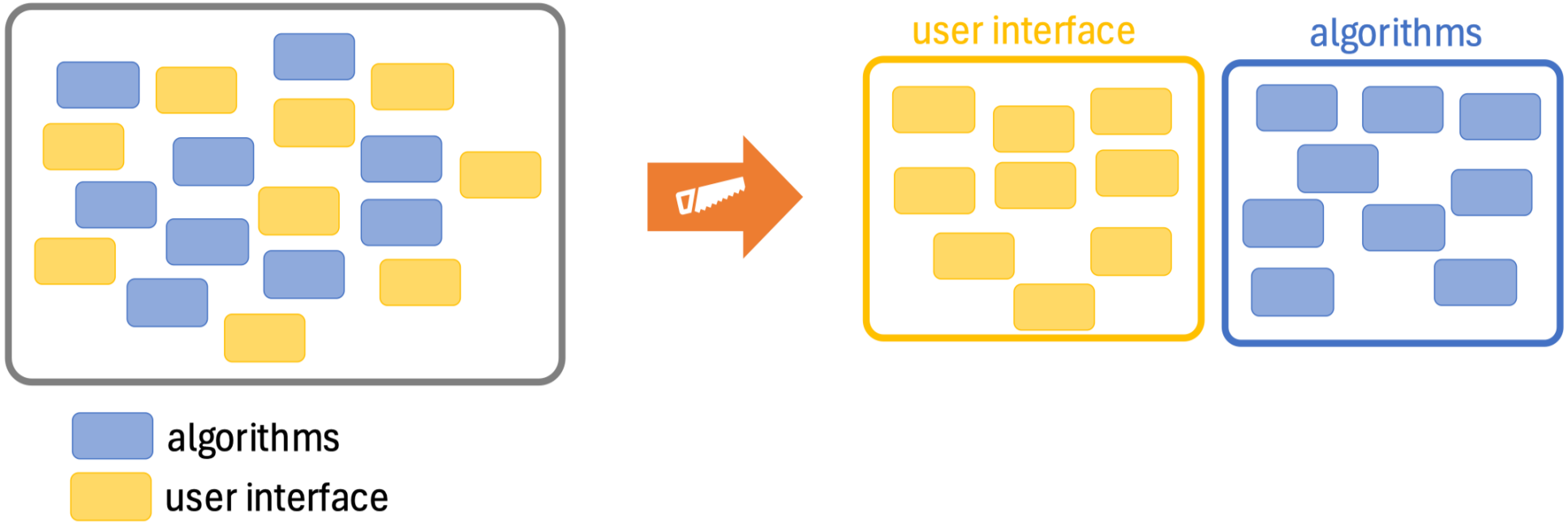
# What are solutions for solving the most common architectural concerns?

**Architectural pattern:** a high-level solution to recurring design problems in software architecture. It provides a general structure or organization.

**Architectural tactic:** a lower-level, specific technique or decision aimed at achieving a particular quality attribute defined in a pattern.

# Problem: we want the system to be easy to modify

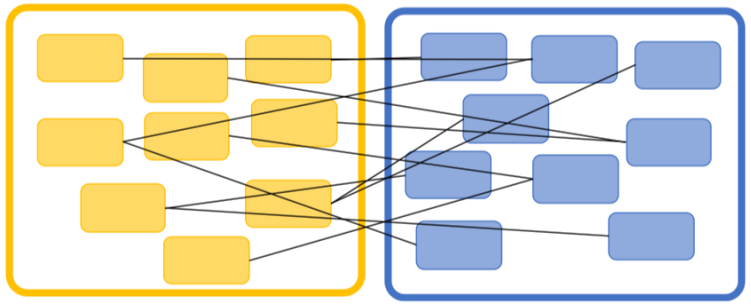
What can we do to improve the situation, to make the system easier to modify?



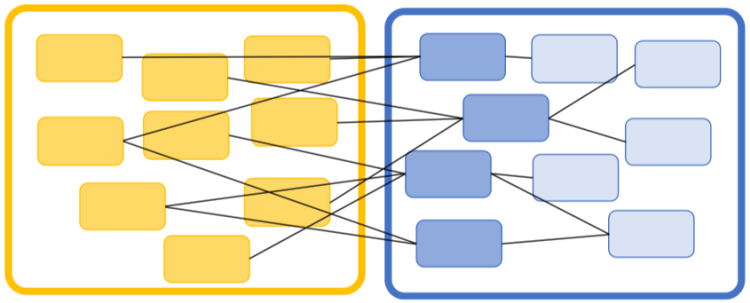

Idea: Split module by responsibility (increases coherence, keep changes local, lower risk of breaking other things while doing modifications) → such architectural ideas are called *"tactics"*

# Problem: we want the system to be easy to modify

## Tactic: **Encapsulation**

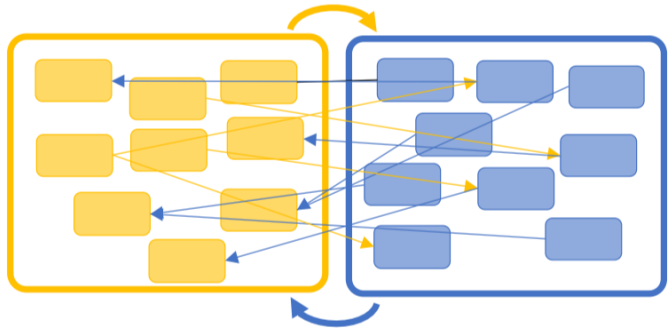


- hide details (private)

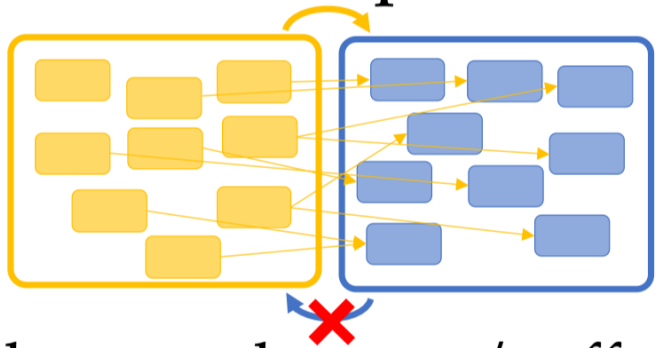


- Allows to change details without affecting dependents

## Tactic: **Restrict Dependencies**



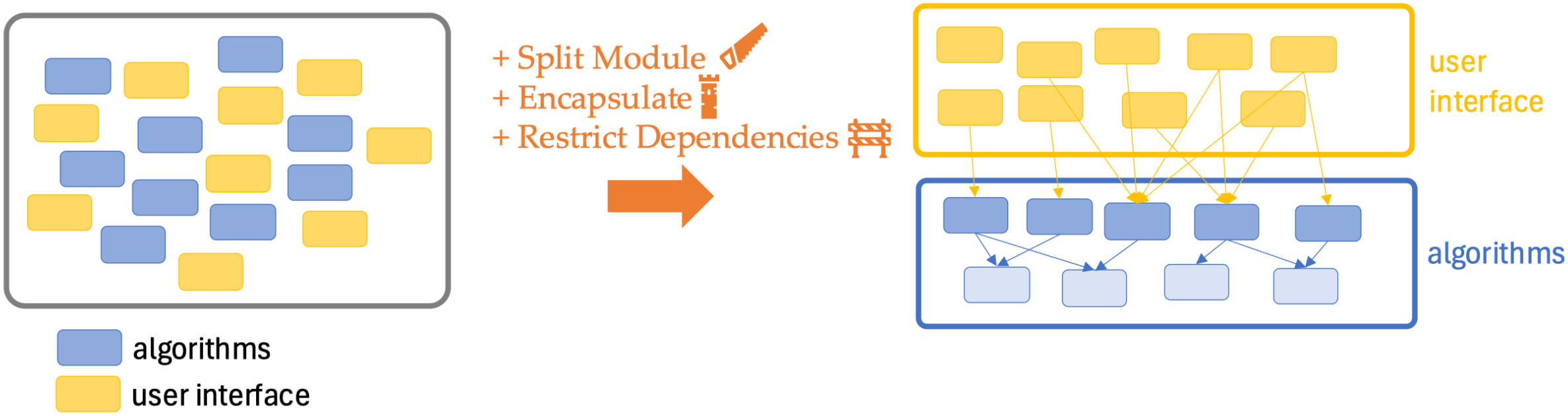
- limit what can depend on what



- Yellow can change w/o affecting blue
- (blue can be reused without yellow)

# Layer pattern

(One) combination these tactics → Layers Pattern



# Pattern: Layers

**Problem**: Achieve modifiability when different concerns should be developed and evolved separately

**Solution**:

- Partition system: *layers* (= elements)
- Relation: *allowed-to-use*, unidirectional

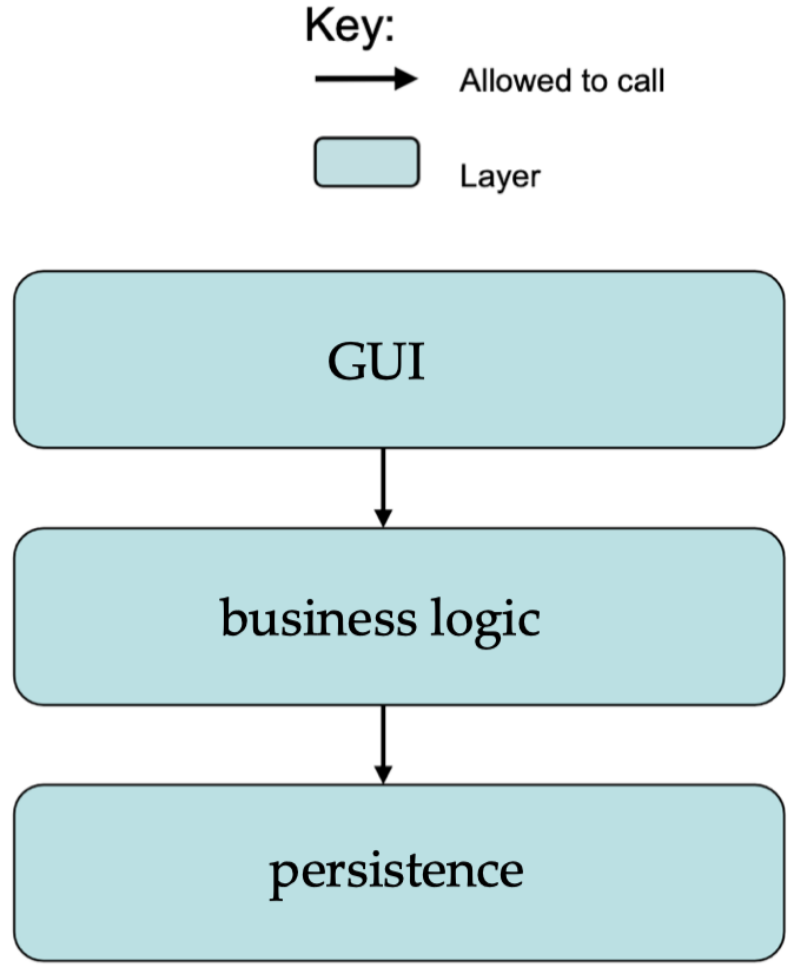
**Variants**: strict vs. relaxed layering (bypassing middle layers)

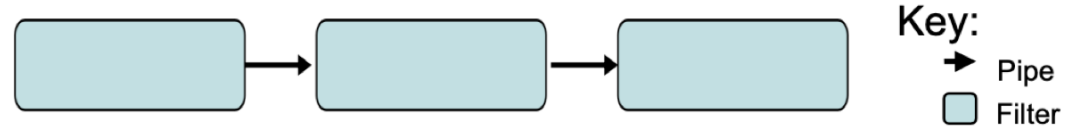
**Benefits**:

- Replacement of lower layers (e.g. different database vendor)
- Reuse of lower layers (e.g. one that abstracts OS-specifics)
- Limited number of interfaces each developer must understand
- Good *testability*

**Trade-Offs:**

- Interface design is not trivial, requires good understanding of concerns
- Possible *performance* penalty (with many layers and strict layering)
- Scaling (for *performance*) is not trivial






# Pattern: Pipes & Filters

**Problem & Context**: Transformation or processing of data/event stream, that has to be built by different developers, that naturally decomposes into stages, and requirements likely change.

**Solution**:

- Elements: *filters* which transform data
- Relation: *pipes,* unidirectional

**Benefits**:

- Can lead to a powerful collection of reusable filters:

**Trade-Offs:**

- *Performance* may suffer due to I/O between the filters
- Filters are stateless
- Filters may be easy to understand, but a pipeline is harder to reason about

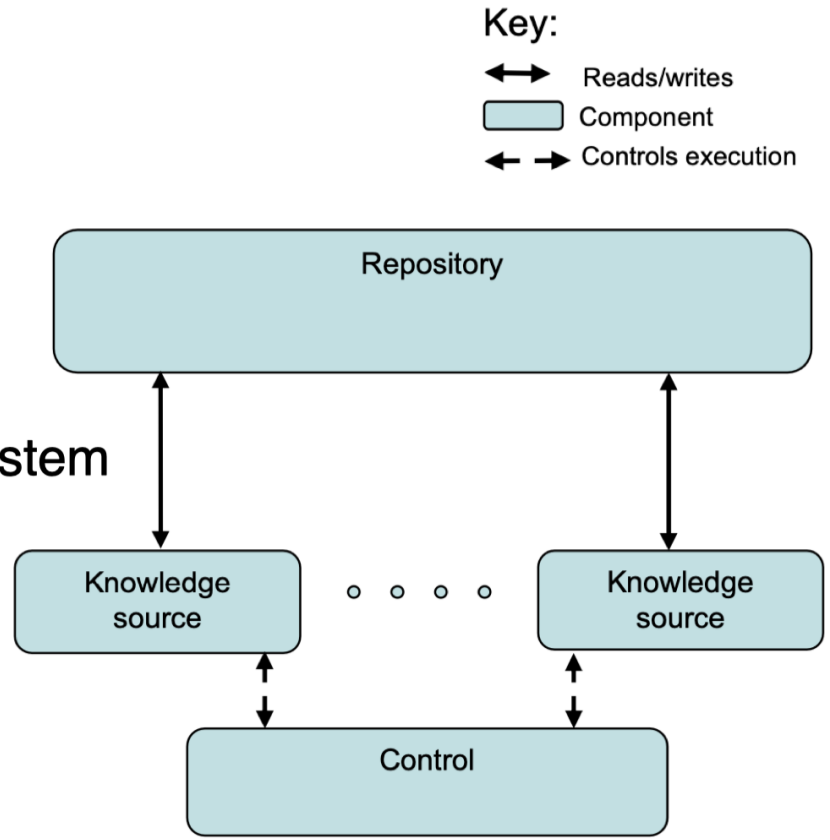
# Pattern: Blackboard

**Context:**

- No feasible deterministic solution for transforming raw data into high-level data structures, but decomposable problem
- Different algorithms for subtasks, no determined global order
- Example: self-driving car using sensors, localization, and control system

**Solution:**

- Intermediate solutions are stored in *repository*
- Independent programs *(knowledge sources)* evaluate their own applicability and improve the intermediate solution
- *Control* component *coordinates execution* of knowledge sources




**Benefits**: *maintainability* (add/remove KS) & *fault tolerance*

**Trade-Offs**: varying *performance*, can be non-deterministic (low *testability* and *reliability*), low *security* (access for all)

# Hands-on

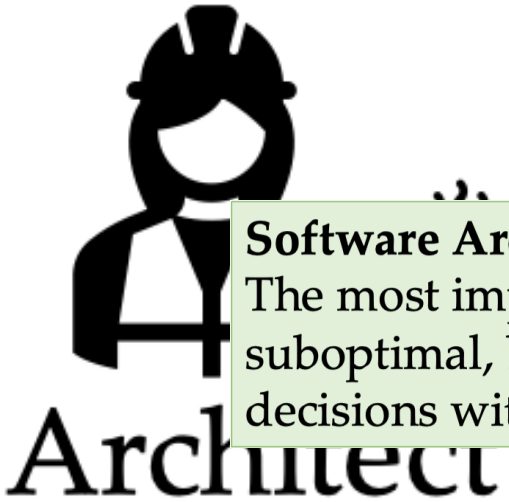
Based on the **requirements** you collected yesterday, start working on the **architecture** of your system.
Define (using pen&paper is fine for now):

- Module structure
- C&C structure
- Allocation structure

by answering the related questions.

Look out for opportunities to apply one of the design pattern!

Making **assumptions** is fine as long as you document them.




The architecture of a software system is the set of structures needed to reason about the system.

**Software Architecture Lesson:**
The most important part of architect's job is to make decisions. Often, a suboptimal, but *explicit* decision with known flaws is better than an *implicit* decisions with unknown consequences.

prise

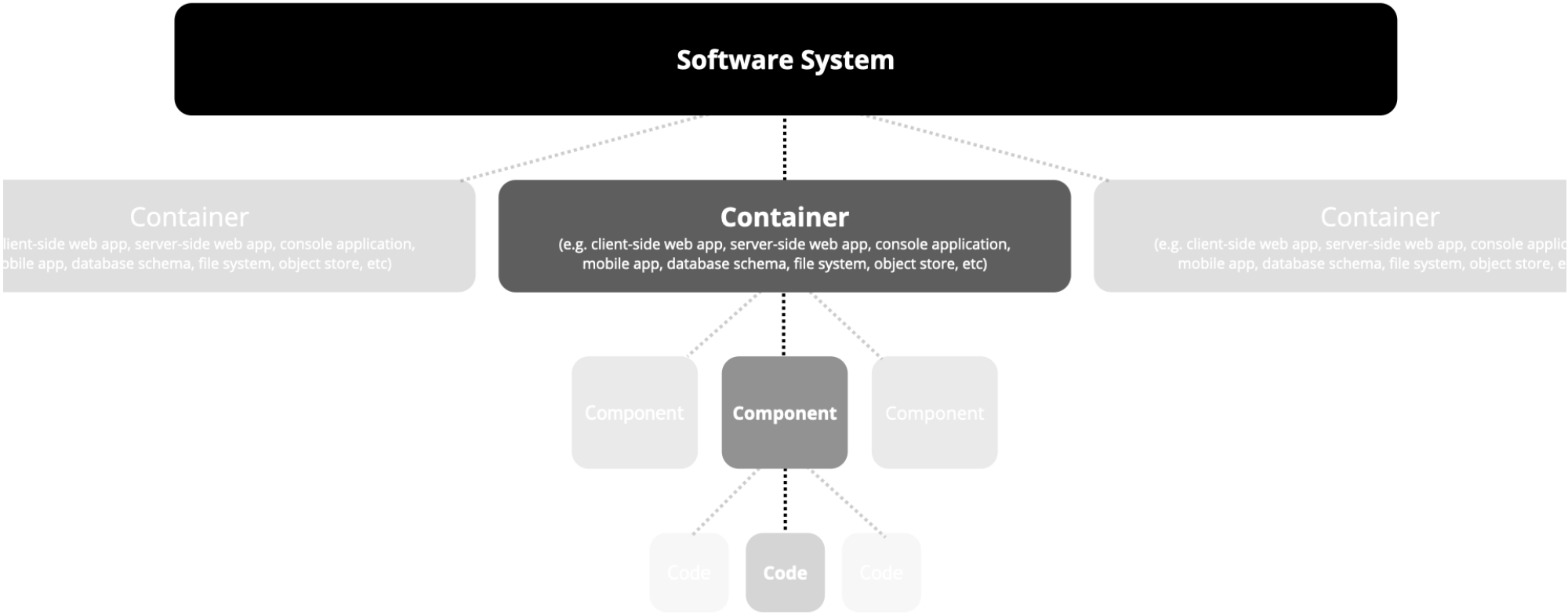
software elements, relations among them, and properties of both.

# What is a "Good" Architecture?

- An architecture is not inheritly *"good"* or *"bad"* – it's more or less *fit for purpose,* in the **context** of specific *goals*.
- An architect makes many technical & non-technical **decisions**
  - Identify necessary decisions to satisfy stakeholders' needs
  - … within organizational contstraints

> *"The life of a software architect is*
> *a long (and sometimes painful) succession of*
> *suboptimal decisions made partly in the dark."*
>
> – Philippe Kruchten

# The C4 model



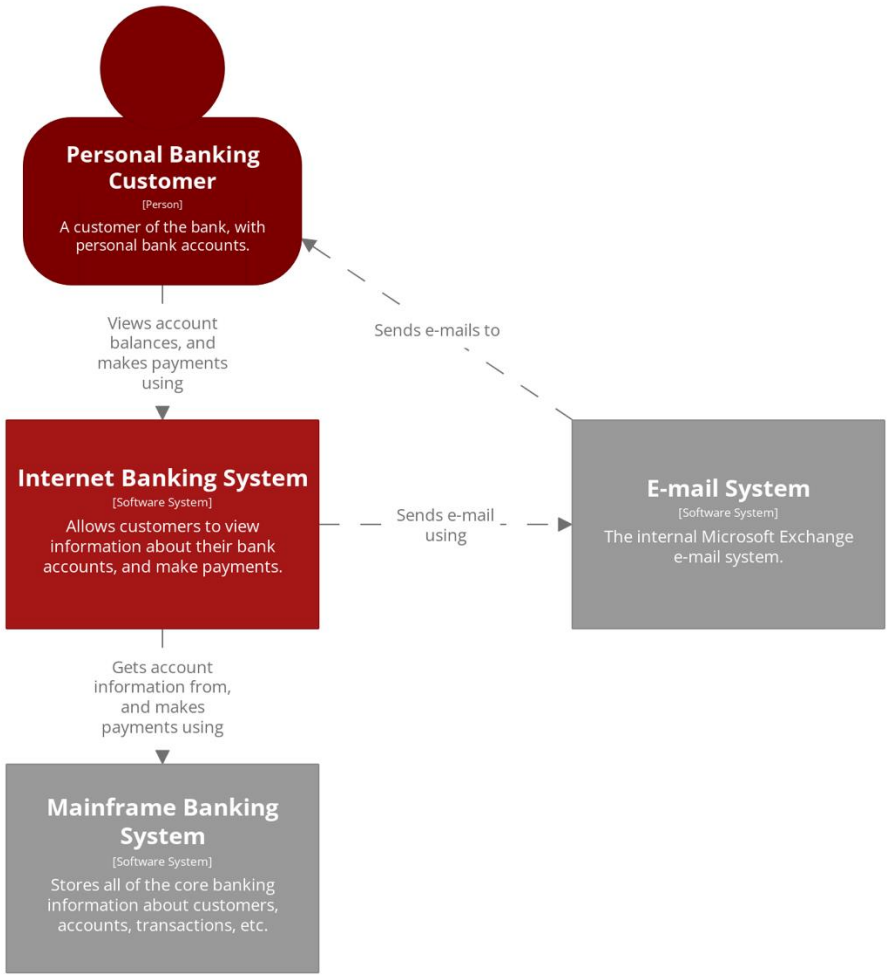

A **software system** is made up of one or more **containers** (applications and data stores), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (classes, interfaces, objects, functions, etc).

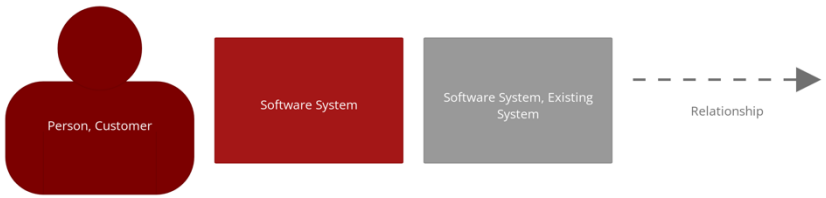
# C4 – Software Context

- A System Context diagram is a good starting point for diagramming and documenting a software system, allowing you to step back and see the big picture. Draw a diagram showing your system as a box in the centre, surrounded by its users and the other systems that it interacts with.
- Detail isn't important here as this is your zoomed out view showing a big picture of the system landscape. 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.

- Scope: A single software system.
- Primary elements: The software system in scope.
- Supporting elements:
  - People (e.g. users, actors, roles, or personas)
  - software systems (external dependencies) that are directly connected to the software system in scope.
- Intended audience: technical and non-technical people, inside and outside the software development team.

# C4 – Software Context (example)



# C4 – Software Context (exercise)

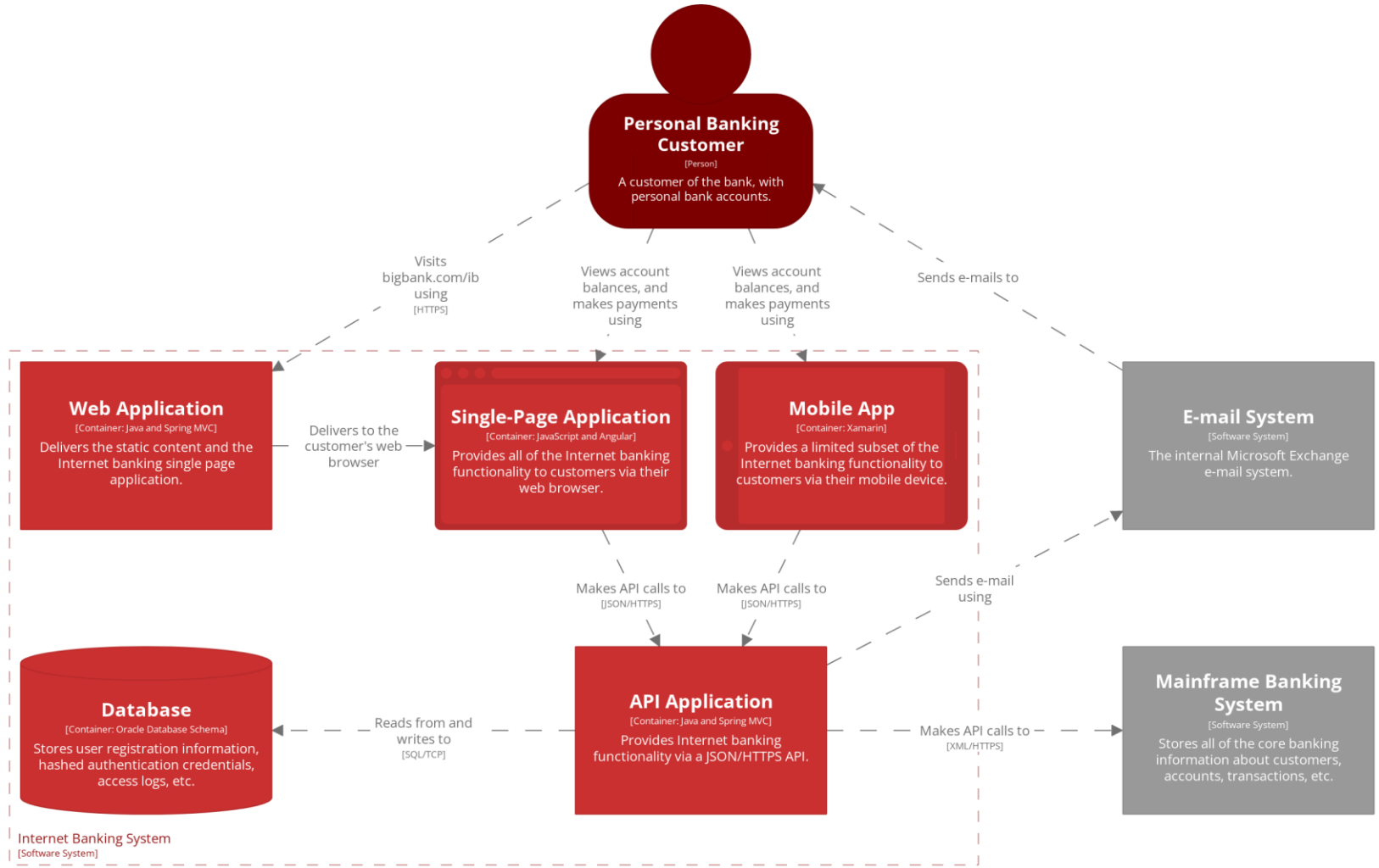
**Create a diagram of your Software System Context**

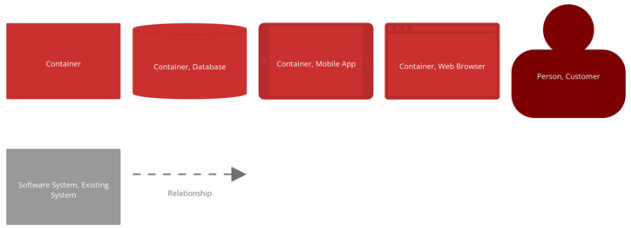
# C4 – Container

- A container is something that needs to be running for the overall software system to work:
  - *Server-side web application*
  - *Client-side web application*
  - *Client-side desktop application*
  - *Mobile app*
  - *Serverless function*
  - *Database*
  - *Content storage*
  - *File system*

- Scope: A single software system.
- Primary elements: Containers within the software system in scope.
- Supporting elements: People and software systems directly connected to the containers.
- Intended audience: Technical people inside and outside the software development team; including software architects, developers and operations/support staff.

# C4 – Container (example)




[Container] Internet Banking System

The container diagram for the Internet Banking System - diagram created with Structurizr.
Wednesday, March 22, 2023 at 8:16 AM Coordinated Universal Time

# C4 – Container (exercise)

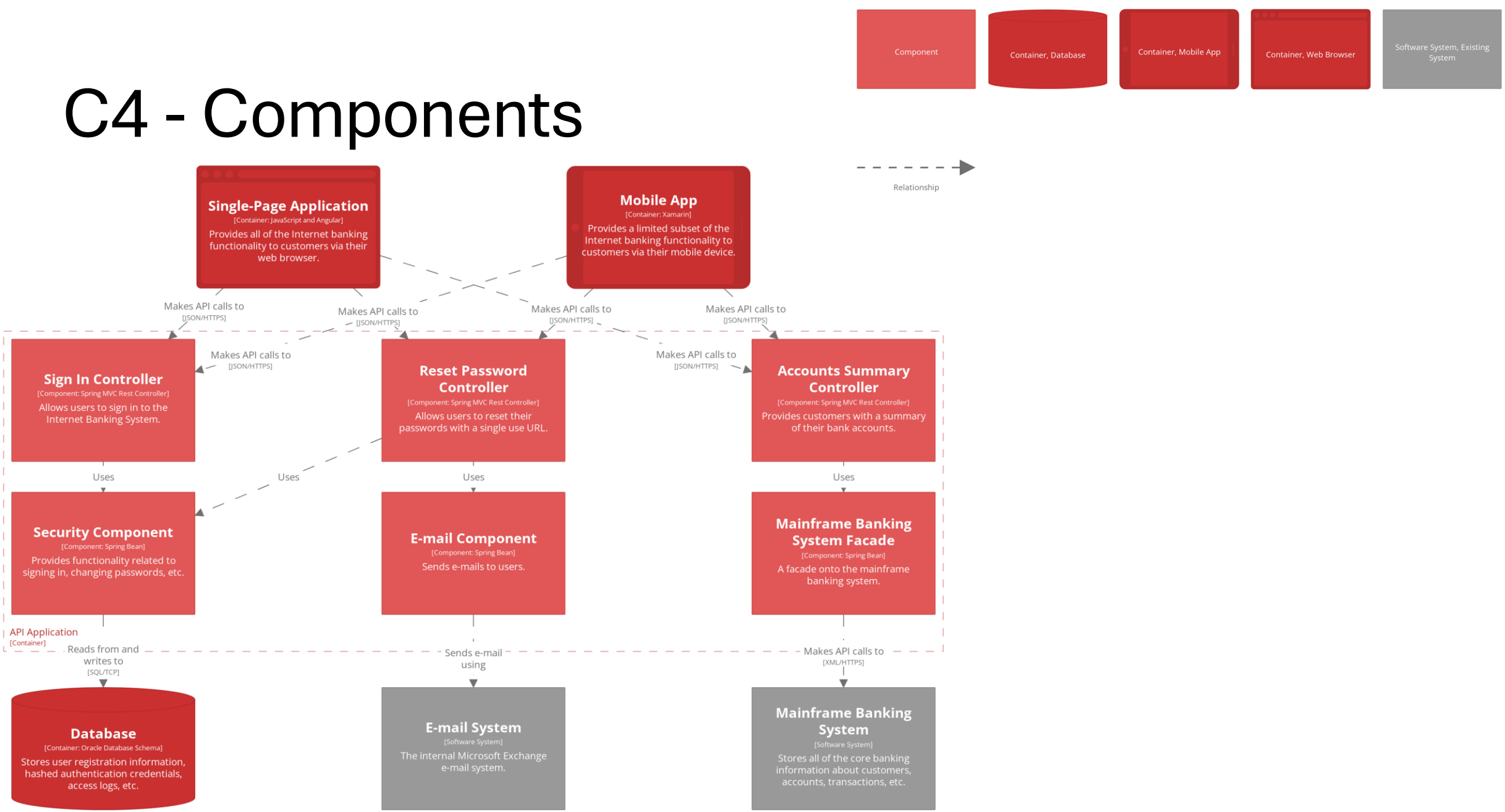
**Extend your diagram with the appropriate Containers**

# C4 - Component

- component is a grouping of related functionality encapsulated behind a well-defined interface
- components are *not* separately deployable units. Instead, it's the container that's the deployable unit.

- Scope: A single container.
- Primary elements: Components within the container in scope.
- Supporting elements: Containers (within the software system in scope) plus people and software systems directly connected to the components.
- Intended audience: Software architects and developers.

# C4 - Components




[Component] Internet Banking System - API Application

The component diagram for the API Application - diagram created with Structurizr.
Wednesday, March 22, 2023 at 8:16 AM Coordinated Universal Time

# C4 – Component (exercise)

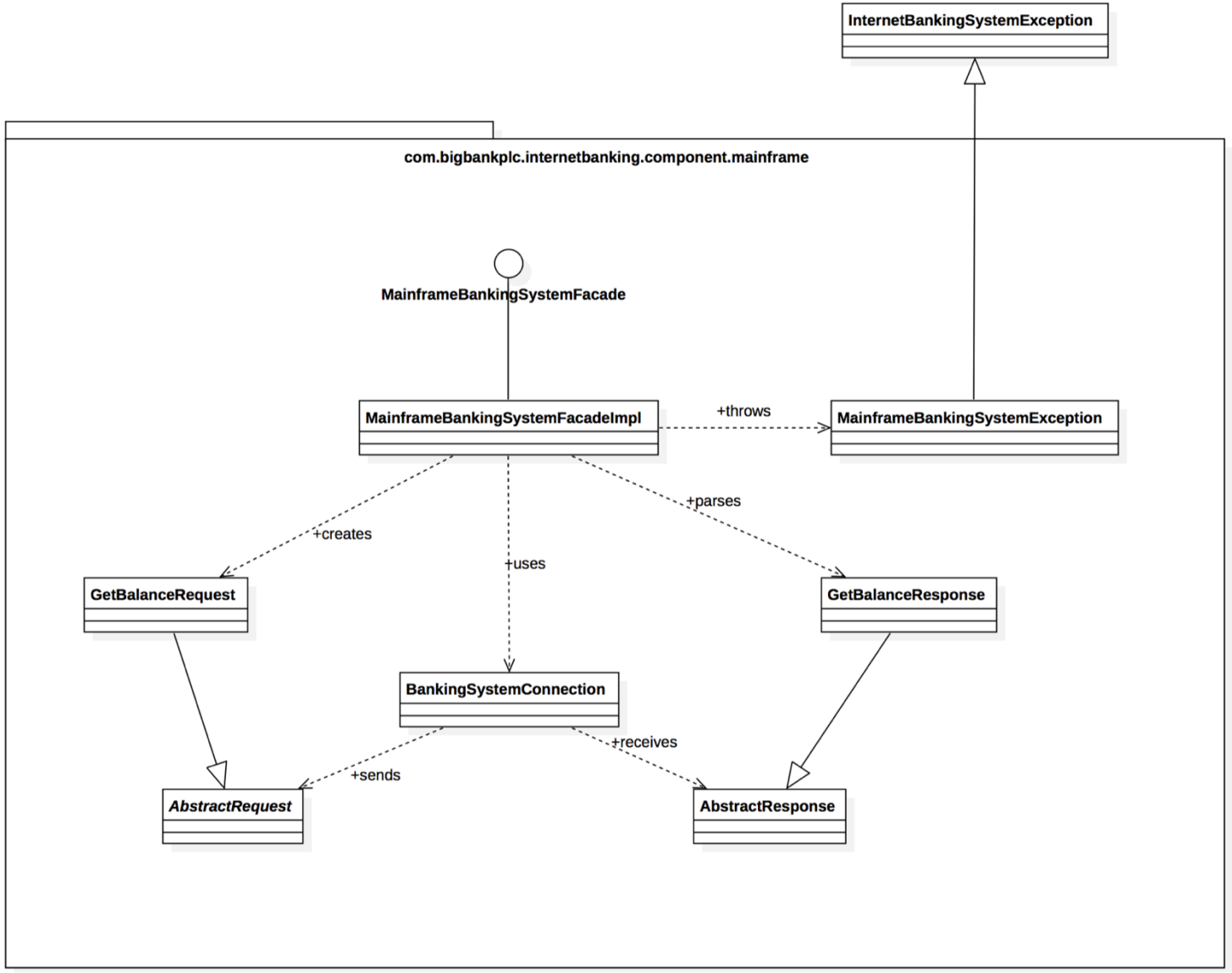
**Extend your diagram with the appropriate Components**

# C4 - Code

- components are made up of one or more code elements constructed with the basic building blocks of a programming language (classes, interfaces, enums, functions, objects)

- Scope: A single component.
- Primary elements: Code elements within the component in scope.
- Intended audience: Software architects and developers.

# C4 – Code (example)



# C4 - Notation

- Diagrams
  - Every diagram should have a title describing the diagram type and scope (audiences or “System Context diagram for My Software System”).
  - Every diagram should have a key/legend explaining the notation being used (shapes, colors, border styles, line types, arrow heads).
  - Acronyms and abbreviations (business/domain or technology) should be understandable by all audiences or explained in the diagram key/legend.
- Elements
  - The type of every element should be explicitly specified (e.g. Person, Software System, Container or Component).
  - Every element should have a short description, to provide an “at a glance” view of key responsibilities.
  - Every container and component should have a technology explicitly specified.
- Relationships
  - Every line should represent a unidirectional relationship.
  - Every line should be labelled, the label being consistent with the direction and intent of the relationship and specific.
  - Relationships between containers should have a technology/protocol explicitly labelled.

# Hands-on

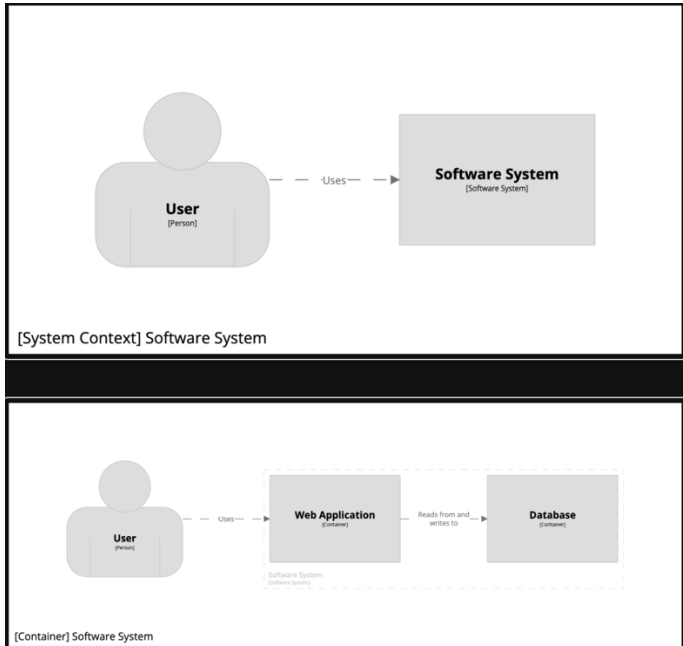
- Using the C4 model, improve the architecture of your project
  - Start with the System **Context**…
  - Create the appropriate **Containers** for it…
  - Define the **Components**

(better to use a diagramming tool at this stage)

# Diagrams as Code

- Creating diagrams through **textual** descriptions
- Why it is important?
  - Allows for version control
  - Easy to update
  - Integration in development workflow
  - Easier collaboration
  - Streamlined, consistent documentation

```
workspace "Name" "Description"

    !identifiers hierarchical

    model {
        u = person "User"
        ss = softwareSystem "Software System" {
            wa = container "Web Application"
            db = container "Database Schema" {
                tags "Database"
            }
        }

        u -> ss "Uses"
        u -> ss.wa "Uses"
        ss.wa -> ss.db "Reads from and writes to"
    }
```



# Structurizr tutorial

Structurizr builds upon “diagrams as code”, allowing you to create **multiple software architecture diagrams** using the C4 model

https://structurizr.com

# Structurizr – System context

- **Starting Point**: Define a Structurizr workspace
- A wrapper containing:
    - **Model**: Defines elements and their relationships
    - **Views**: Specifies the views rendered as diagrams

```
workspace "Name" "Description" {

}
```

# Structurizr – System context

**Model Definition**:

• Use the *model* keyword to define the software architecture.

Components:

• **Person**: Named “User” (identifier u)

• **Software System**: Named “Software System” (identifier ss)

Relationships:

• Defined using -> symbol.

• Example: User -> Software System with the description “Uses”

```
workspace "Name" "Description" {

    model {
        u = person "User"
        ss = softwareSystem "Software System"

        u -> ss "Uses"
    }

}
```

# Structurizr – System context

**System Context View**:

• Defines a single view with the **software system** (ss) as the scope.

- Diagram1: A unique key for referencing the diagram

**Include Statement**:

• include *: Includes the software system (ss), people, and other systems with direct relationships to/from it.

**Auto Layout**:

• autolayout lr: Enables automatic layout with **left-to-right** direction.

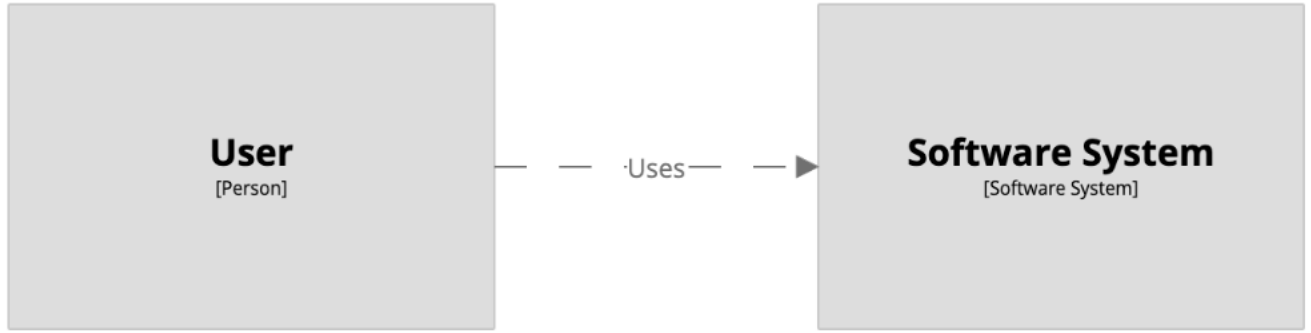
```
workspace "Name" "Description" {

    model {
        u = person "User"
        ss = softwareSystem "Software System"

        u -> ss "Uses"
    }

    views {
        systemContext ss "Diagram1" {
            include *
            autolayout lr
        }
    }

}
```

# Structurizr – System context

```
workspace "Name" "Description" {

    model {
        u = person "User"
        ss = softwareSystem "Software System"

        u -> ss "Uses"
    }

    views {
        systemContext ss "Diagram1" {
            include *
            autolayout lr
        }
    }

}
```

**Try it yourself!**

https://structurizr.com/dsl



# Structurizr – Containers

**Defining Containers**:

- **Containers**: Applications and data stores that make up the software system.
- **Structure**: Nested inside the software system definition (within curly braces).

**Relationships**:

- User → Web Application
- Web Application → Database

**Identifiers**:

- *!identifiers hierarchical*: Enables reference to containers

```
workspace "Name" "Description"

    !identifiers hierarchical

    model {
        u = person "User"
        ss = softwareSystem "Software System" {
            wa = container "Web Application"
            db = container "Database Schema" {
                tags "Database"
            }
        }

        u -> ss "Uses"
        u -> ss.wa "Uses"
        ss.wa -> ss.db "Reads from and writes to"
    }

    views {
        systemContext ss "Diagram1" {
            include *
            autolayout lr
        }
    }

}
```

# Structurizr – Containers

A container view must be defined to visualize the model.

- **Container View**:
  - Scope: The **software system** (ss).
  - include *: Includes containers inside the software system and any related people or systems.
  - autolayout lr: Maintains automatic layout with **left-to-right** direction.

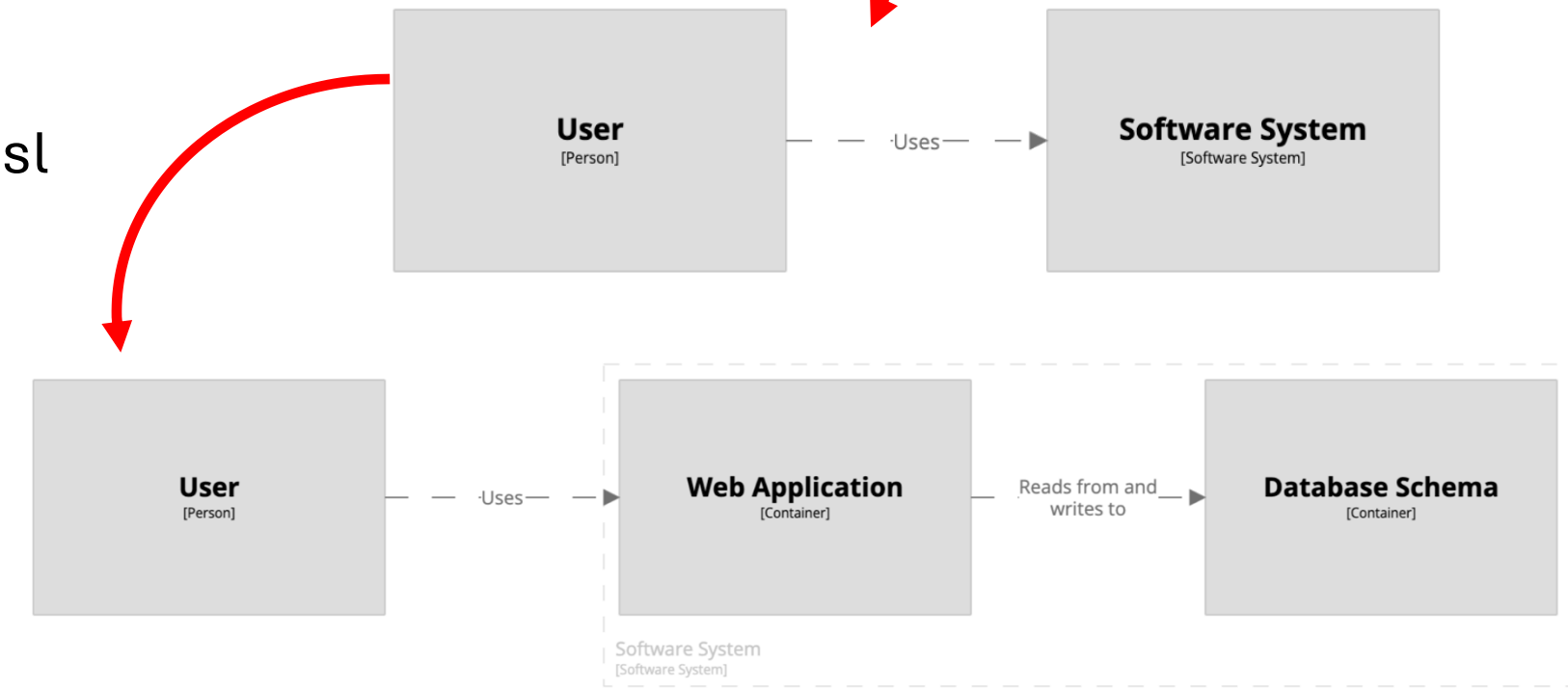
```
workspace "Name" "Description"

    !identifiers hierarchical

    model {
        u = person "User"
        ss = softwareSystem "Software System" {
            wa = container "Web Application"
            db = container "Database Schema" {
                tags "Database"
            }
        }

        u -> ss "Uses"
        u -> ss.wa "Uses"
        ss.wa -> ss.db "Reads from and writes to"
    }

    views {
        systemContext ss "Diagram1" {
            include *
            autolayout lr
        }

        container ss "Diagram2" {
            include *
            autolayout lr
        }
    }

}
```

# Structurizr – Containers

```
workspace "Name" "Description"

    !identifiers hierarchical

    model {
        u = person "User"
        ss = softwareSystem "Software System" {
            wa = container "Web Application"
            db = container "Database Schema" {
                tags "Database"
            }
        }

        u -> ss "Uses"
        u -> ss.wa "Uses"
        ss.wa -> ss.db "Reads from and writes to"
    }

    views {
        systemContext ss "Diagram1" {
            include *
            autolayout lr
        }

        container ss "Diagram2" {
            include *
            autolayout lr
        }
    }

}
```

**Try it yourself!**

https://structurizr.com/dsl



# Structurizr – Views

**Decide which elements to visualize**

- All elements: *include **
- Specific elements: *include u ss.wa ss.db*
- Specific dependencies: *include “->ss.wa->”*
- Specific types: *include “element.type==container”*
- Specific levels: *include “element.parent==ss”*

# Structurizr – Styling

## Adding Colors and Shapes

- Elements and relationships can have **text-based tags**, similar to HTML classes.
  - **Element**: Generic tag for all elements.
  - **Person**: Specific tag for people.
  - **Software System**: Tag for software systems.
  - **Container**: Tag for containers.

- **Styling Elements**:
  - Foreground color
  - Background color
  - Icon shape

```
views {
    ...

    styles {
        element "Element" {
            color white
        }
        element "Person" {
            background #116611
            shape person
        }
        element "Software System" {
            background #2D882D
        }
    }
}
```

# Structurizr – Styling

## Adding Colors and Shapes

- Elements and relationships can have **text-based tags**, similar to HTML classes.
  - **Element**: Generic tag for all elements.
  - **Person**: Specific tag for people.
  - **Software System**: Tag for software systems.
  - **Container**: Tag for containers.

- **Styling Elements**:
  - Foreground color
  - Background color
  - Icon shape

```
ss = softwareSystem "Software System" {
    wa = container "Web Application"
    db = container "Database Schema" {
        tags "Database"
    }
}
```

```
styles {
    ...
    element "Container" {
        background #55aa55
    }
    element "Database" {
        shape cylinder
    }
}
```

# Hands-on

**Use Structurizr to create a diagram-as-code of your C4 architecture**

# Bonus – pyStructurizr

```python
from pystructurizr.dsl import Workspace

# Create the model(s)
with Workspace() as workspace:
    with workspace.Model(name="model") as model:
        user = model.Person("User")
        with model.SoftwareSystem("Software System") as software_system:
            webapp = software_system.Container("Web Application")
            db = software_system.Container("Database")

    # Define the relationships
    user.uses(webapp, "Uses")
    webapp.uses(db, "Reads from and writes to")

# Create a view onto the model
workspace.ContainerView(
    software_system,
    "My Container View",
    "The container view of our simply software system."
)
```

https://github.com/nielsvanspauwen/pystructurizr