

# System architecture and design

---

Paolo Burgio  
paolo.burgio@unimore.it



**UNIMORE**  
UNIVERSITÀ DEGLI STUDI DI  
MODENA E REGGIO EMILIA

High Performance  
Real Time **Lab**



# Pizza as a Service 2.0

<http://www.paulkerrison.co.uk>

Tradition  
On-Premises  
(legacy)

Conversation

Friends

Beer

Pizza

Fire

Oven

Electric / Gas

Homemade

Infrastructure as a  
Service  
(IaaS)

Conversation

Friends

Beer

Pizza

Fire

Oven

Electric / Gas

Communal  
Kitchen

Containers as a  
Service  
(CaaS)

Conversation

Friends

Beer

Pizza

Fire

Oven

Electric / Gas

Bring Your Own

Platform as a  
Service  
(PaaS)

Conversation

Friends

Beer

Pizza

Fire

Oven

Electric / Gas

Takeaway

Function as a  
Service  
(FaaS)

Conversation

Friends

Beer

Pizza

Fire

Oven

Electric / Gas

Restaurant

Software as a  
Service  
(SaaS)

Conversation

Friends

Beer

Pizza

Fire

Oven

Electric / Gas

Party



You Manage



Vendor Manages

Configuration

Functions

Scaling...

Runtime

OS

Virtualisation

Hardware



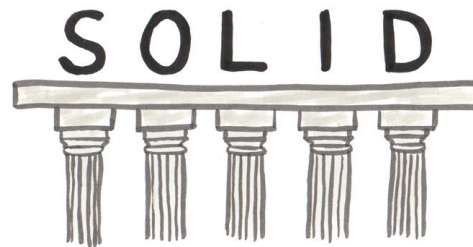
# System design

---

*We (finally!!) are going to translate customer specifications into a set of technological specifications that developers can understand*

Output of this step: the a system architecture

- › Identify a set of modules
- › Each module has a single specific functionality (or sub-functionality)
- › And we need to describe their interaction with other modules (i.e., their contracts\*)



*\* aka: prototypes, OOP-like interfaces, Web endpoints, C/C++ headers...*



# Ingredients

---

## Decomposition

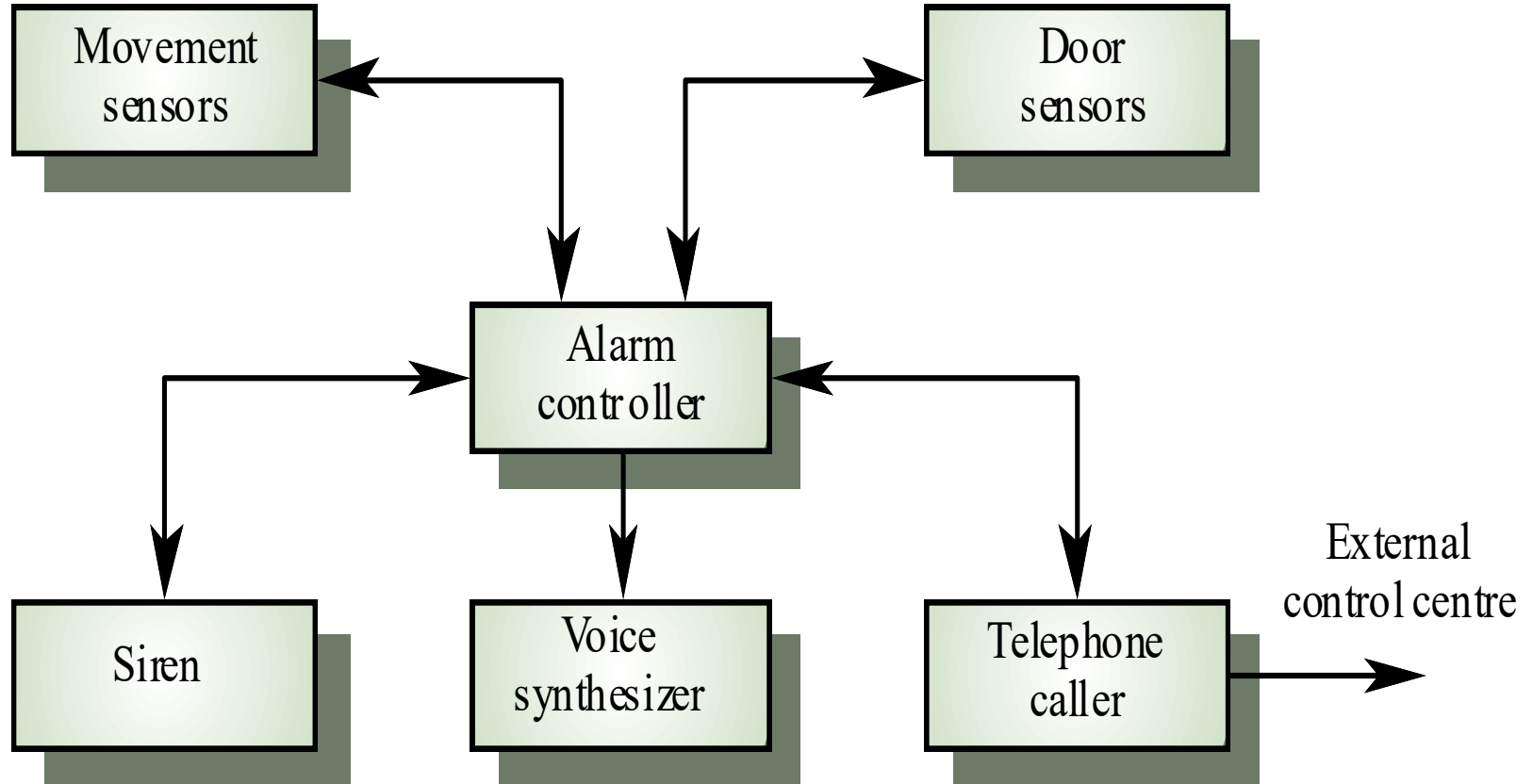
- › First into subsystems, that interact among themselves, but that **do not depend among themselves**
- › Then into modules and sub-modules, each **providing a specific service** to other modules
- › Then into components, the basic unit of implementation (e.g., Java libraries)

## Identification and assignment of **Control**

- › Who does what?
- › Identifying active components and passive components
- › Where are the threads/processes? ITA: *"Chi ha il pallino"*?



# Example: smart home alarm





# Exercise

---

Let's do this for our amazing project!

Modules

> ...

> ....



# Modules

---

Group functionalities that are in tight relation

- › Es: everything that relates to user accounts (CRUD)
- › At system design level, we need to clearly identify interfaces toward other modules/the external world
- › The, identify the sub-functionalities that each module shall produce

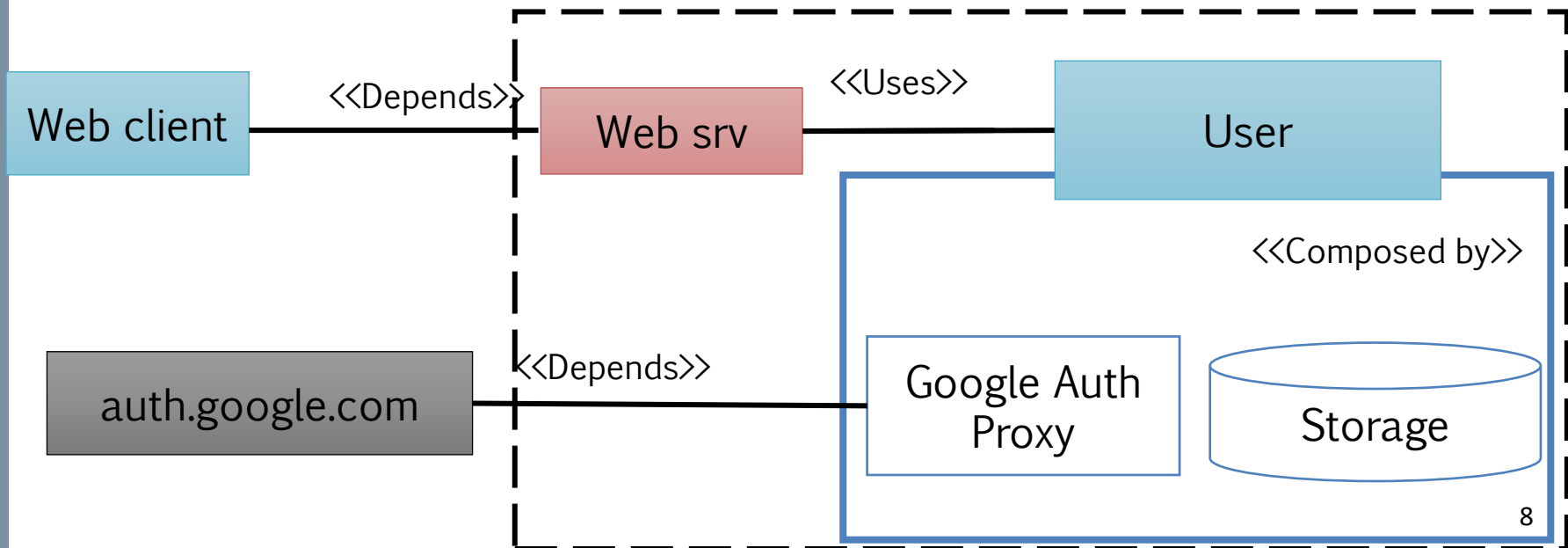




# Inter-module relations

Typically

- › Modules expose services that are **used** by other modules to complete theirs
- › Modules are **composed** by sub-modules (so, we can work at different level of details)
  - *Divide et impera!*
- › Modules **depend** on other modules (typically, to adhere/follow a sequence diagram for a specific use-case)







# Partitioning strategy

---

## Top-down

- › From specs, to services, to modules, to components, etc...
- › Streamlined from documentation!

## Bottom up

- › Data-structure/functionality centric
- › Typical if we already have a framework/codebase

Going on with the project, you realize that we mix the two...



# Architectural patterns



# Client-server architecture

---

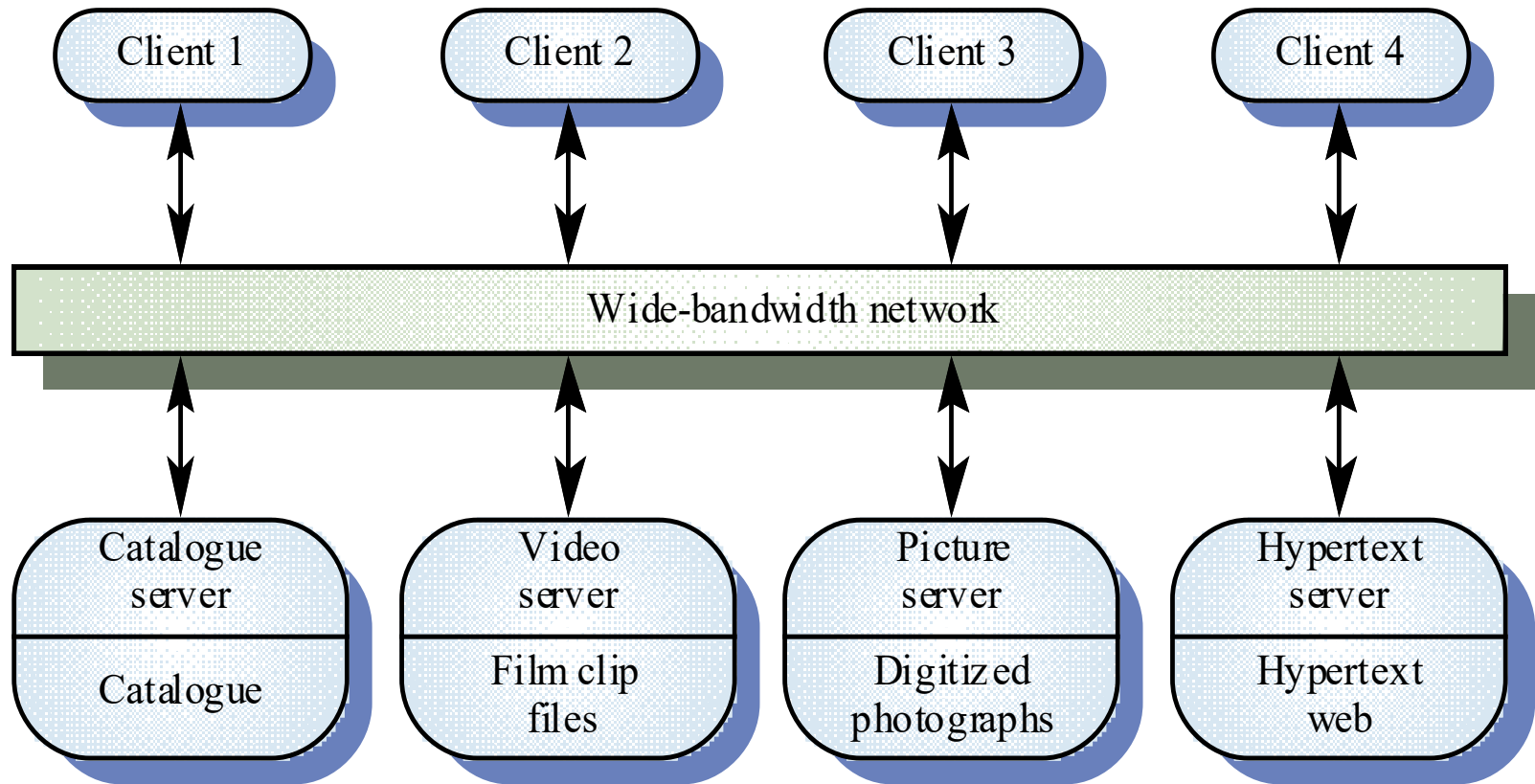
Typical of distributed systems, it is composed by

- › One or more servers, offering generic services
  - Accounting, storage, customer-specific logics..
- › Clients that use those services
  - Web apps, mobile apps...
- › A communication network, here assumed as “first class citizen”
  - On 24/7, e.g., such as power provisioning
- › Communication is asymmetric, hence based on requests and responses
- › Quality-of-Service (QoS), Service-Level Agreement (SLA) shall be agreed





# Example: web services





# Why client-server? (and why not?)

---

## Pros

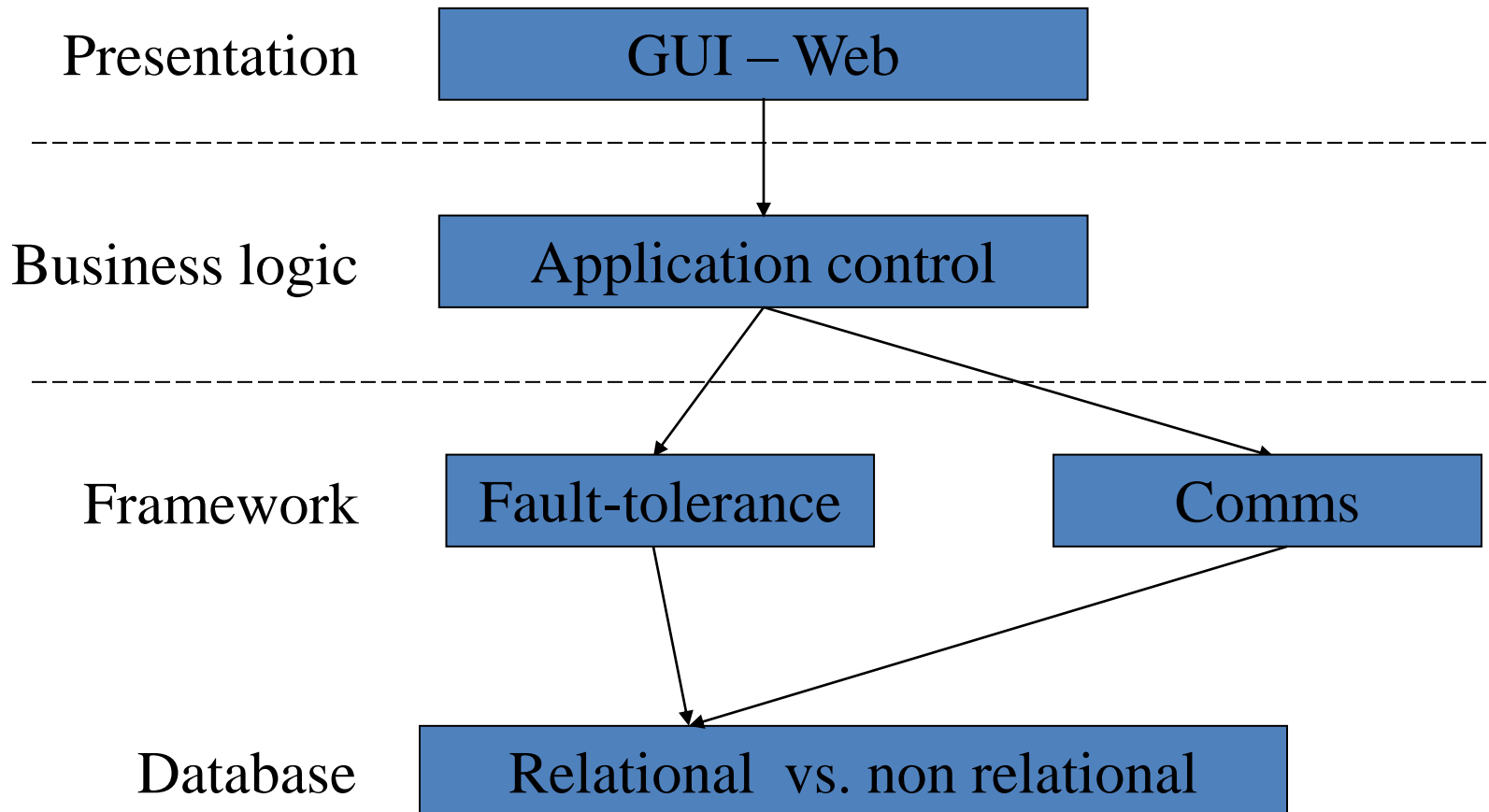
- › It is easy to perform data distribution and responsibilities
- › Can scale the number of clients
  - “It is easy to add new clients”
- › Can scale the number of servers
  - “It is easy to add new servers”
  - Both horizontally (increasing the nr of machines for a single services) and vertically (increasing their resources)

## Cons

- › Typically requires a number of resources, and can be redundant
- › Servers must be known by clients
  - We need a naming service
- › We create a dependency!
  - What if we change URLs?



# Multi-tier architecture





# Design of control

---

What do we mean by “control”?



# Design of control

---

What do we mean by “control”?

- › “Who does what?”
- › “Who *runs* the use cases?” vs “Who has the *logic* that implements use-cases”?
- › Follow the vertical bar of sequence diagrams

Can be centralized, or de-centralized

- › Has strong implications on the system architecture





# 1. Centralized control (synchronous)

---

A single system serves all requests (e.g., a web server)

- › He depends on other sub-systems
- › Typical, when we design the frontend of a web-app (also call *service*)
- › Based on synchronous communication (e.g., function calls)

## Pros and cons

- › Single point of access (easy to implement)
- › Single point of failure (require thorough design of SLAs)

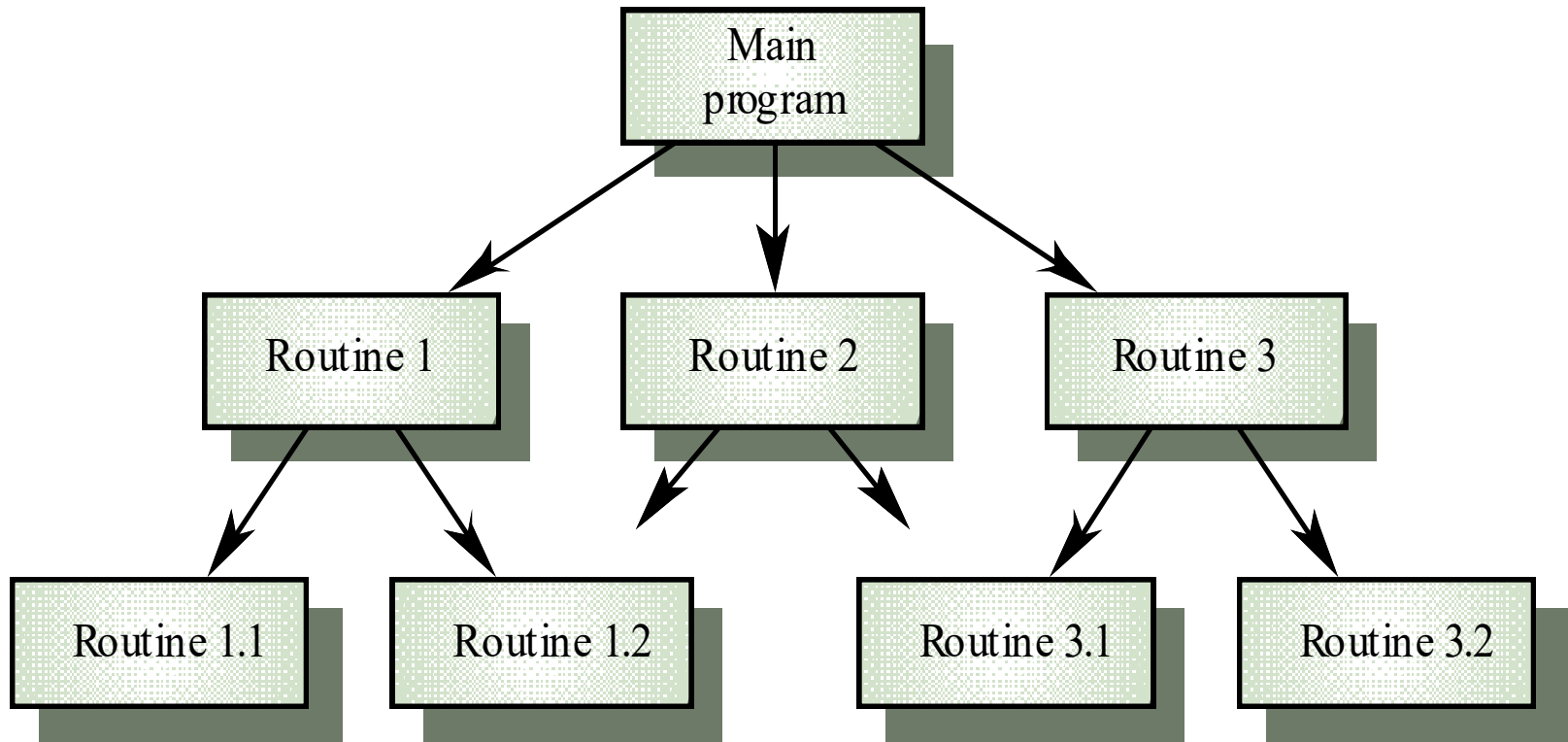
Noticeable examples

- › Request-response in sequential systems
- › Master-slave in parallel systems



# Request-response

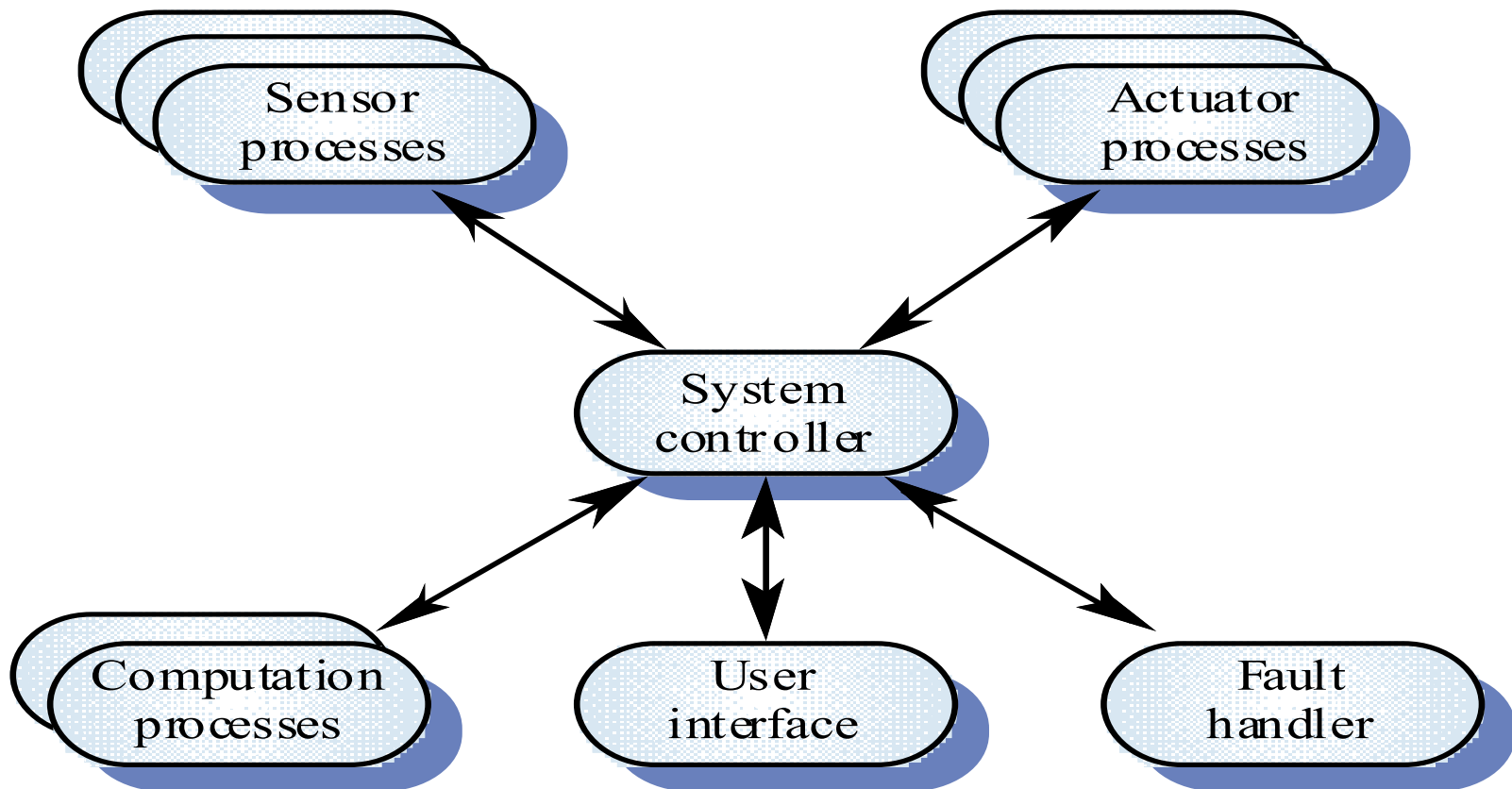
- › Based on function calls
- › Do I need to say more? ☺



# Master-slave: multi-process

---

- › Requires inter-process communication: Sockets, ROS, etc





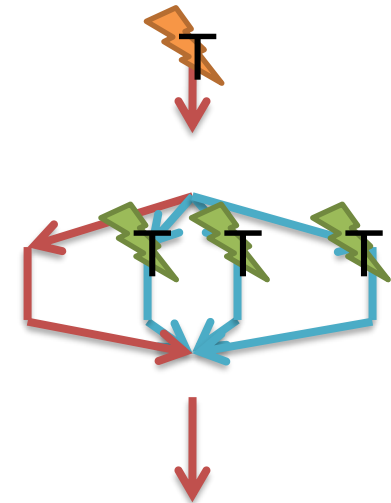
# Master-slave: multi-thread

- › Inter-process model, based on shared memory
- › Es: PThreads, GPUs, etc...

```
int main()
{
    int err;
    pthread_t mythreads[NTHREADS];
    for (int i=0; i<NTHREADS; i++)
        err = pthread_create (&mythreads[i], // ==> FORK
                             &myattr,
                             my_pthread_fn,
                             NULL);

    // Here, the main thread can do other stuff!
    other_fn();

    for (int i=0; i<NTHREADS; i++)
        pthread_join(mythreads[i], &returnvalue); // <== JOIN
}
```





## 2. Event-based (asynchronous)

---

Every sub-system module works independently, without knowing the others

- › Based on asynchronous communication

### Pros and cons

- › Distributed system (more complex to implement)
- › Loosely-coupled interaction between modules (more robust, removes dependencies)
- › *(You might start realizing: in informatics, dependencies **are a big problem**)*

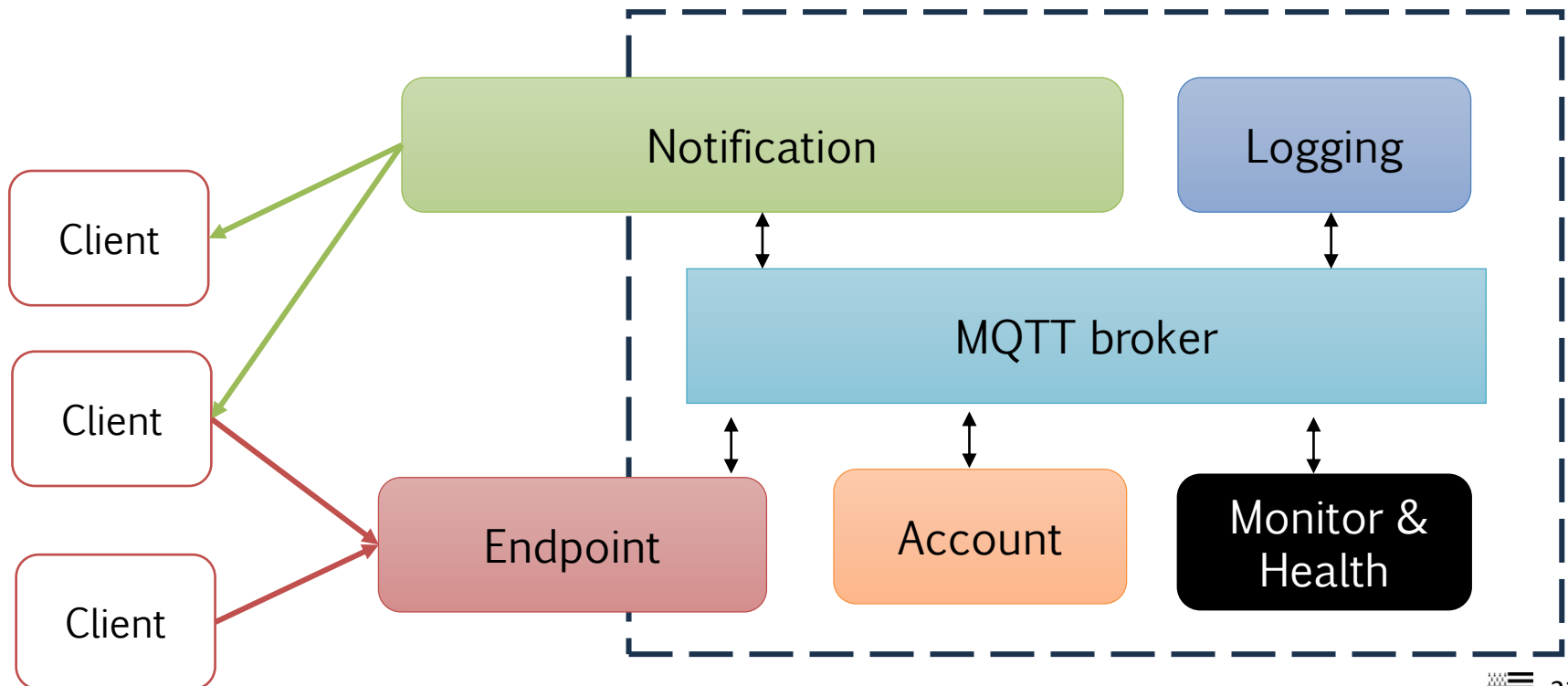
### Noticeable examples

- › Broadcast models, in highly parallel systems
- › Interrupt-based model, inside computers



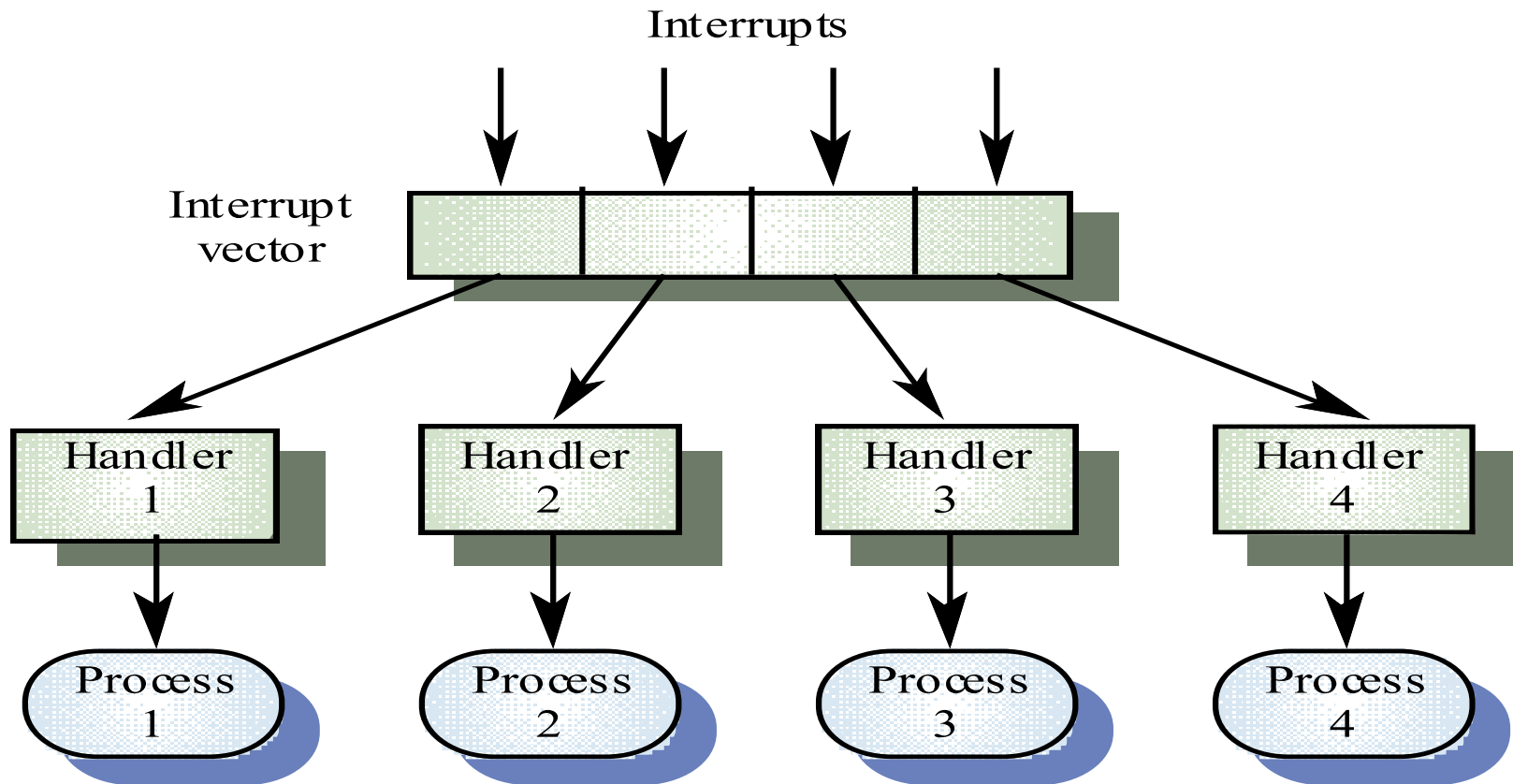
# Event-based

- › Message broker as first class citizen (MQTT; COAp)
- › Typical for modern micro-service infrastructures
- › DMZ with no (major) security issues



# Interrupts

- › What happens inside our computers





# Bonus pills of computers

---

- › Interrupts: want to know more?







# Specifications for the single module

---

Once we defined the architectural model, let's write specifications for the single module

## Basic principles

- › Every module shall be as much as possible independant on other modules (*low coupling, loosely-coupled*)
- › Minimal inter-module knowledge between developers
- › Services that are highly dependant shall belong to the same module (*high cohesion*)
  - E.g., “update age” functionality, and storage of user data

First of all, defining the contract/interface towards other modules!



# Module contract

---

- › I don't use the word "interface" because it might be misleading..but it's actually an interface!

We must clearly define (possibly in UML)

- › Which functionalities are we exposing?
  - "Update age", "Delete user"
- › How do we expose them?
  - Functions to invoke? Services to call?
- › Input-output parameters
  - Number of parameters, their types..

UML provides the same notations both for the analysis phase and for the design & implementation phase (remember the "different level of abstraction" thing?)


**...SO...**



# Let's talk about coding!

---





# Model View Control

*Aka: il papà di tutti...*





# MVC – Model View Control

---

Partitioning strategy for software components/modules

- › Model – represents the status of the application
  - How we represent the world, how we store it, how we communicate it (Data Transfer Objects)
- › View – how we show the Model
  - Basically, the user interfaces
- › Control – application logics, how we modify the model
  - Directly inherited by behavioral diagrams

As a general rule, Model, View and Control **must** be (at least) in separate files!

- › Often, in separate packages/components/libraries



# Example: MVC in Java EE

---

## Model

- › We store the status of our model in components implemented in **JavaBeans**
- › Eases deployment/mapping of these data into Databases, files, session objects, DTOs, ...

Three simple rules. JavaBeans classes:

- › Must implement `java.io.Serializable`
- › Should have a public constructor with no-args
- › Properties/fields must be private, and have public getters and setters methods



# Example: MVC in Java EE

---

```
public class Person implements java.io.Serializable {  
    private int id;  
    private String name;  
  
    // Ctor  
    public Student() {}  
  
    // Setter for Id  
    public void setId(int id) { this.id = id; }  
  
    // Getter for Id  
    public int getId() { return id; }  
  
    // Setter for Name  
    public void setName(String name) { this.name = name; }  
  
    // Getter for Name  
    public String getName() { return name; }  
}
```



# Example: MVC in Java EE

## View

- › In JEE, we use Java Server Pages (JSPs), which directly access our model
- › Here, we use the `oracle.jsp.dbutil.ConnBean` to access to a DB

```
<%@ page import="java.sql.*, oracle.jsp.dbutil.*" %>

<jsp:useBean id="cbean" class="oracle.jsp.dbutil.ConnBean"
scope="session">
    <jsp:setProperty name="cbean" property="dataSource"
        value="<%=request.getParameter("myRecord") %>" />
</jsp:useBean>

<% try {
    cbean.connect();
    String sql="SELECT ename, sal FROM scott.emp ORDER BY ename"
    CursorBean cb = cbean.getCursorBean (CursorBean.PREP_STMT, sql);
    System.out.println(cb.getResultAsHTMLTable());
    cb.close(); cbean.close();
} catch (SQLException e) {
    //...
}
%>
```





# Example: MVC in Java EE

---

## Controller

- › JSPs or Servlets as JSP backends (aka: Code Behind)

```
@WebServlet(name = "MyServlet", urlPatterns = "/my-record")
public class MyServlet extends HttpServlet {
    // MyWervice holds the model
    private MyService myService = new MyService();

    @Override
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException {

        String myID = request.getParameter("id");
        // Get the object by underlying logics...
        MyService.get(Integer.parseInt(myID))
            .ifPresent(s -> request.setAttribute("myRecord", s));

        // .. and forward it to the JSP
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("/WEB-INF/jsp/my-record.jsp");
        dispatcher.forward(request, response);
    }
}
```



# Integrating MVC parts

---

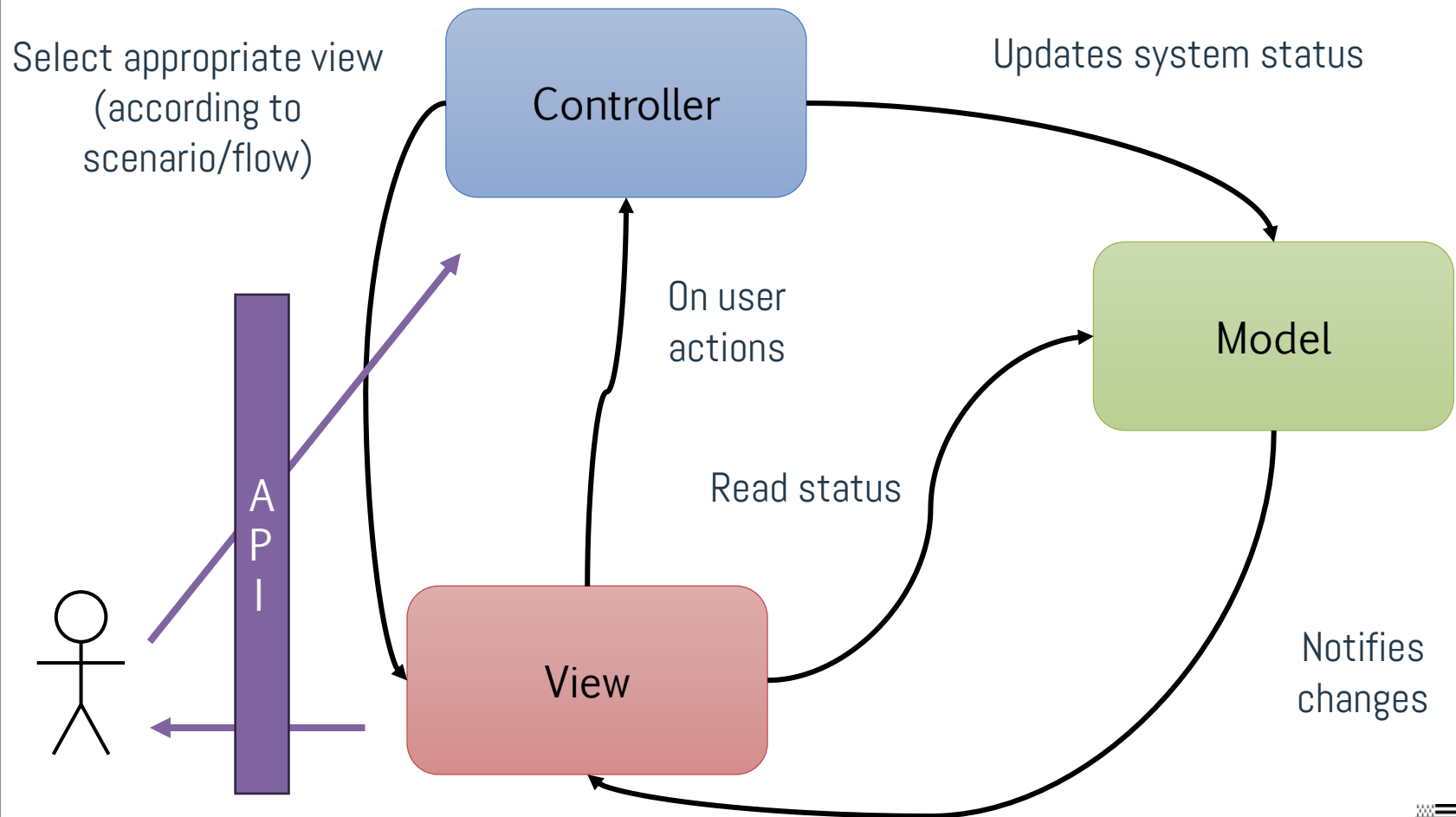
View accesses to Model with getters

Control modifies Model with setters, and accesses it with getters

View and Control are decoupled

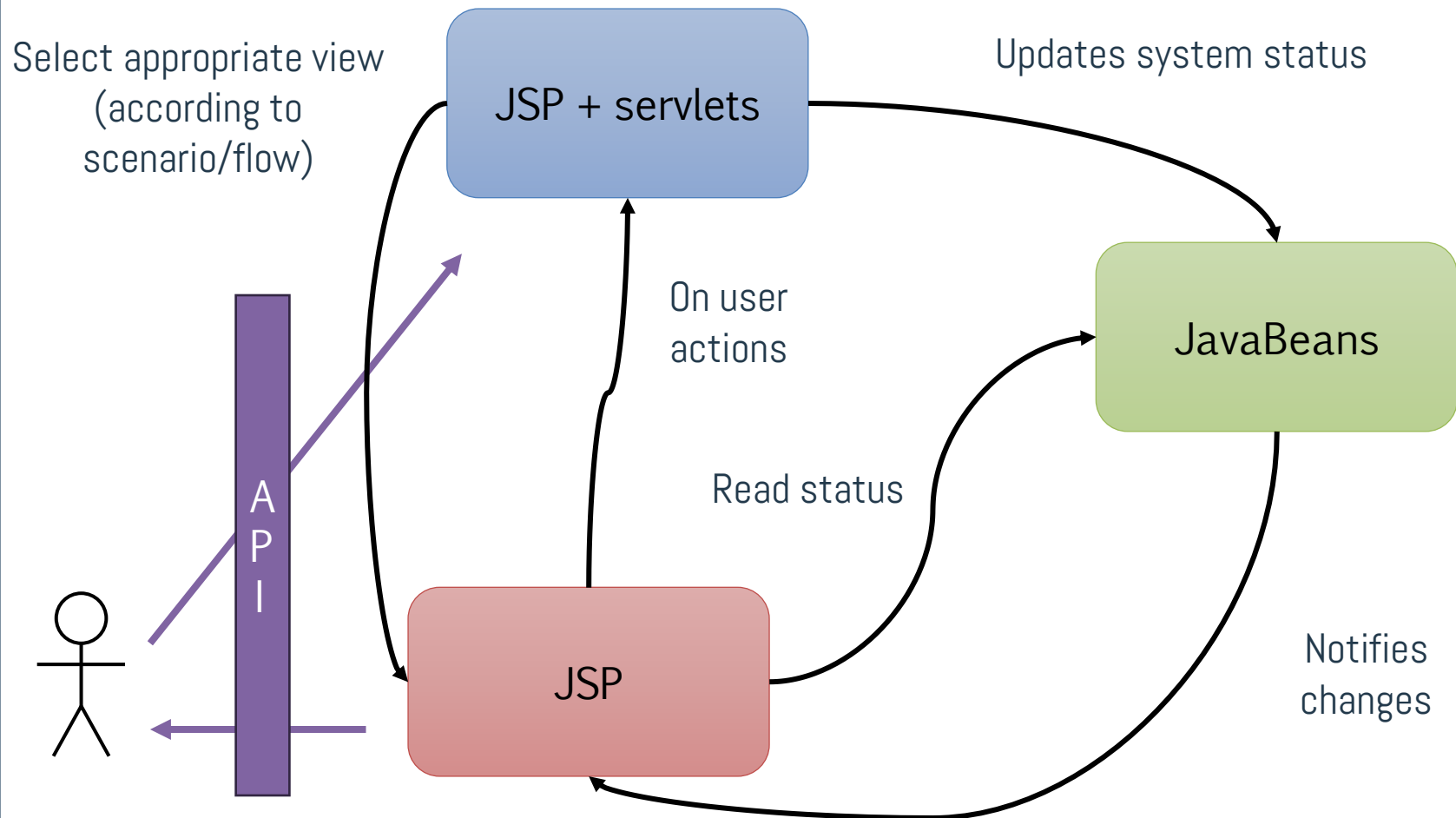
- › Control stands as code-behind of a View
- › It injects (processed) data into it
- › And triggers modification to Model, as response to user interaction

# Integrating MVC parts



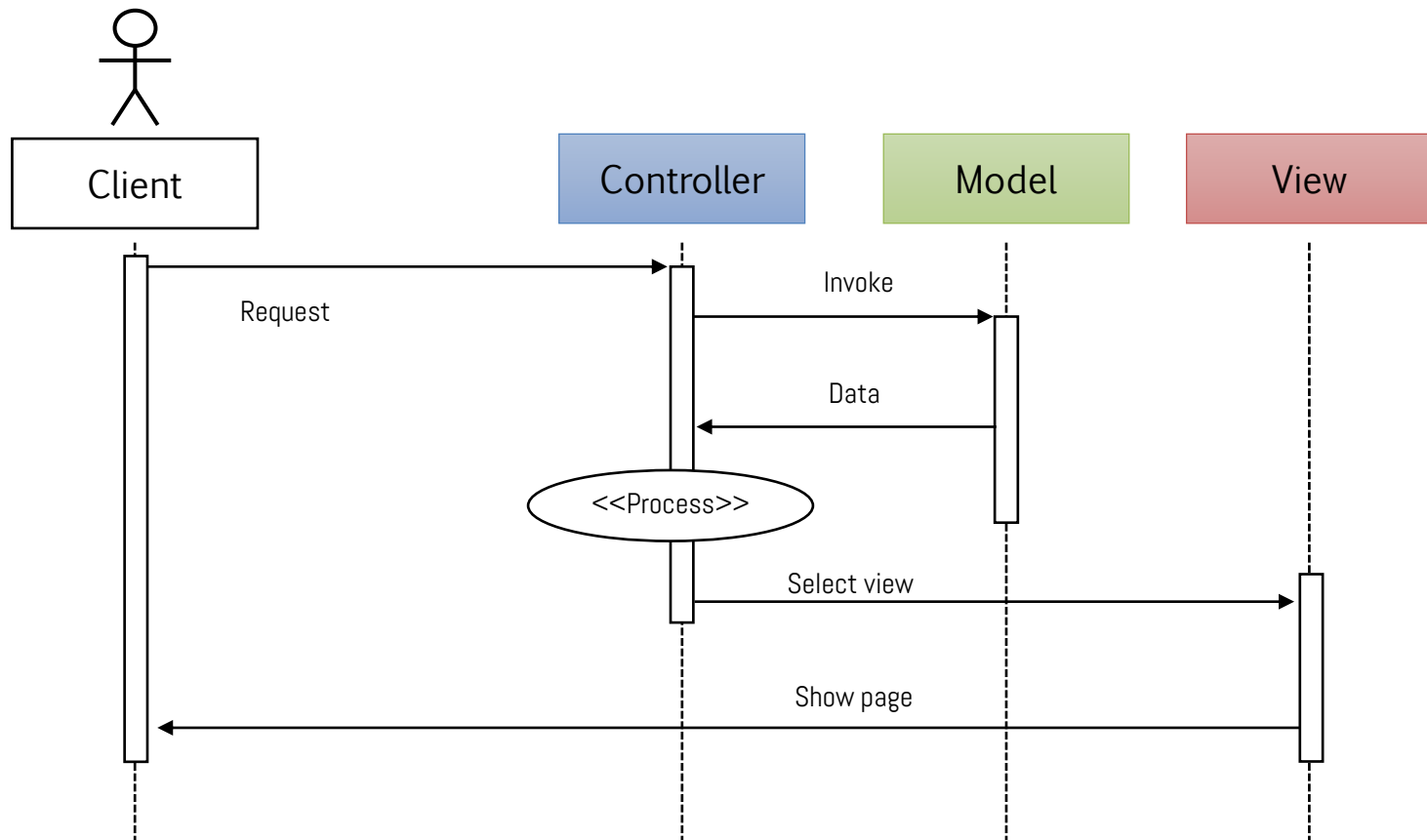


# Integrating MVC parts - JEE





# MVC sequence diagram





# Why MVC?

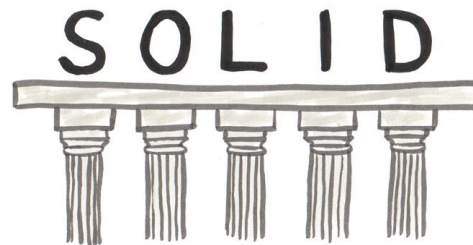
---

## Pros

- › Isolation between component improves modularity and reusability
- › E.g., we can switch from JSP/Web view to a mobile app, written using another technology

## Cons

- › Architecture is more complex, with more files and components
- › But this is not too much of a problem, as we will see..





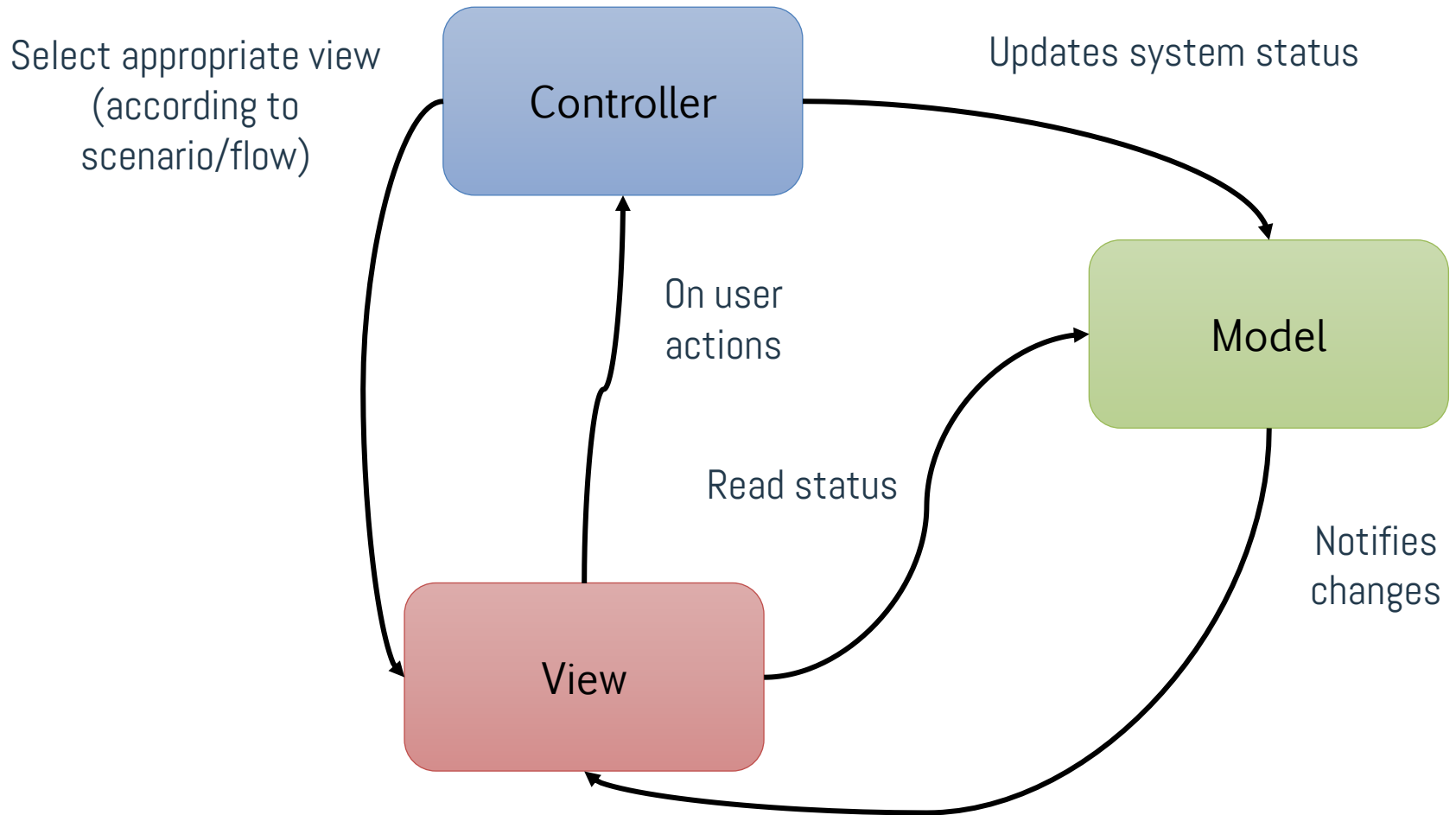
---

# Model View ViewModel

*Aka: il figliol prodigo*



# ..but we can do better!







# ..but we can do better!

Select appropriate view  
(according to  
scenario/flow)

Controller

Where do we put data  
checks? Are they part of  
Controller, or model?

Ex:  $\text{age} < 0$

The same technology is  
used for V and C

UX designers also must  
write the controller!

Ex: JSP/XML and Java..

On user  
actions

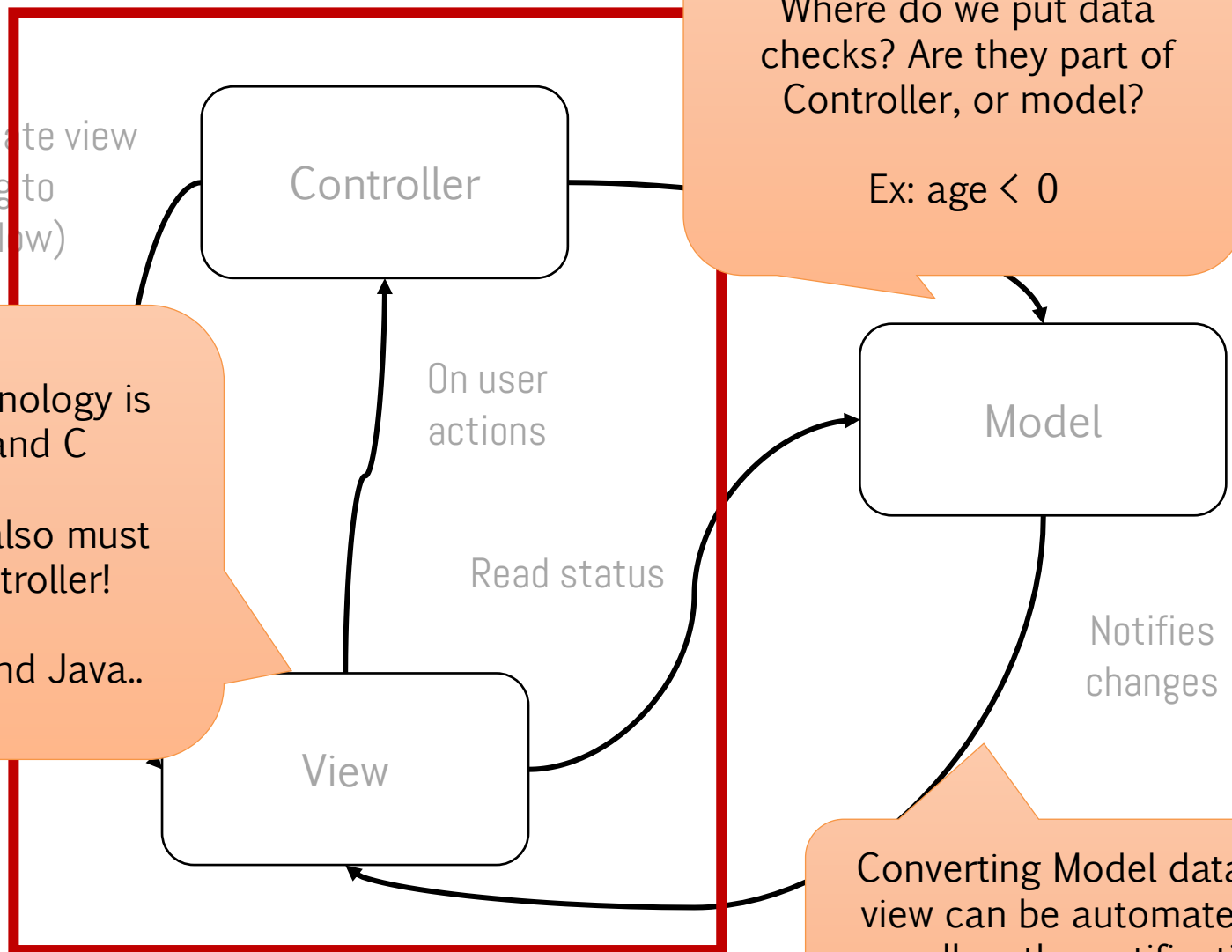
Read status

Model

Notifies  
changes

View

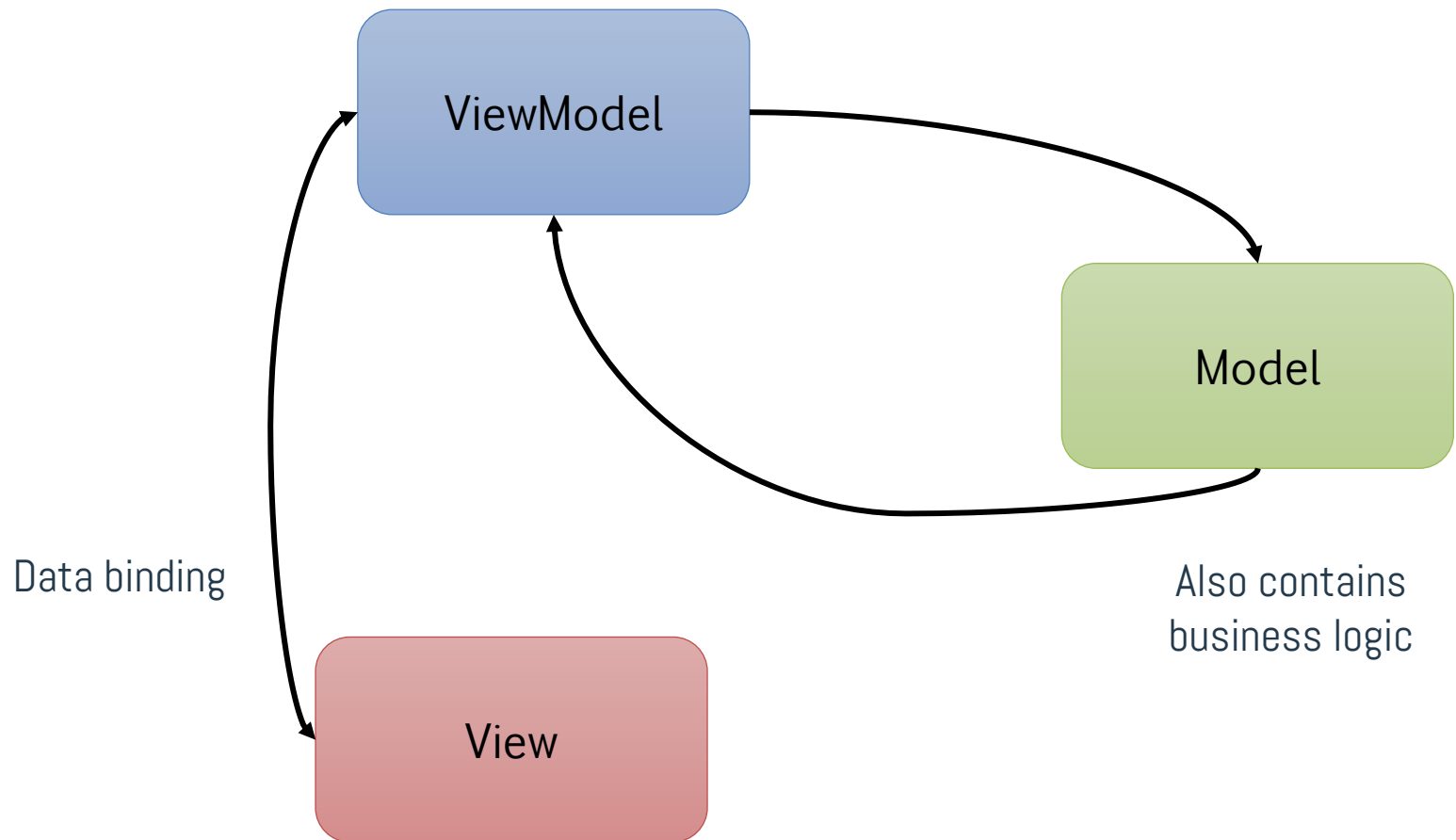
Converting Model data into  
view can be automated, as  
well as the notification  
engine





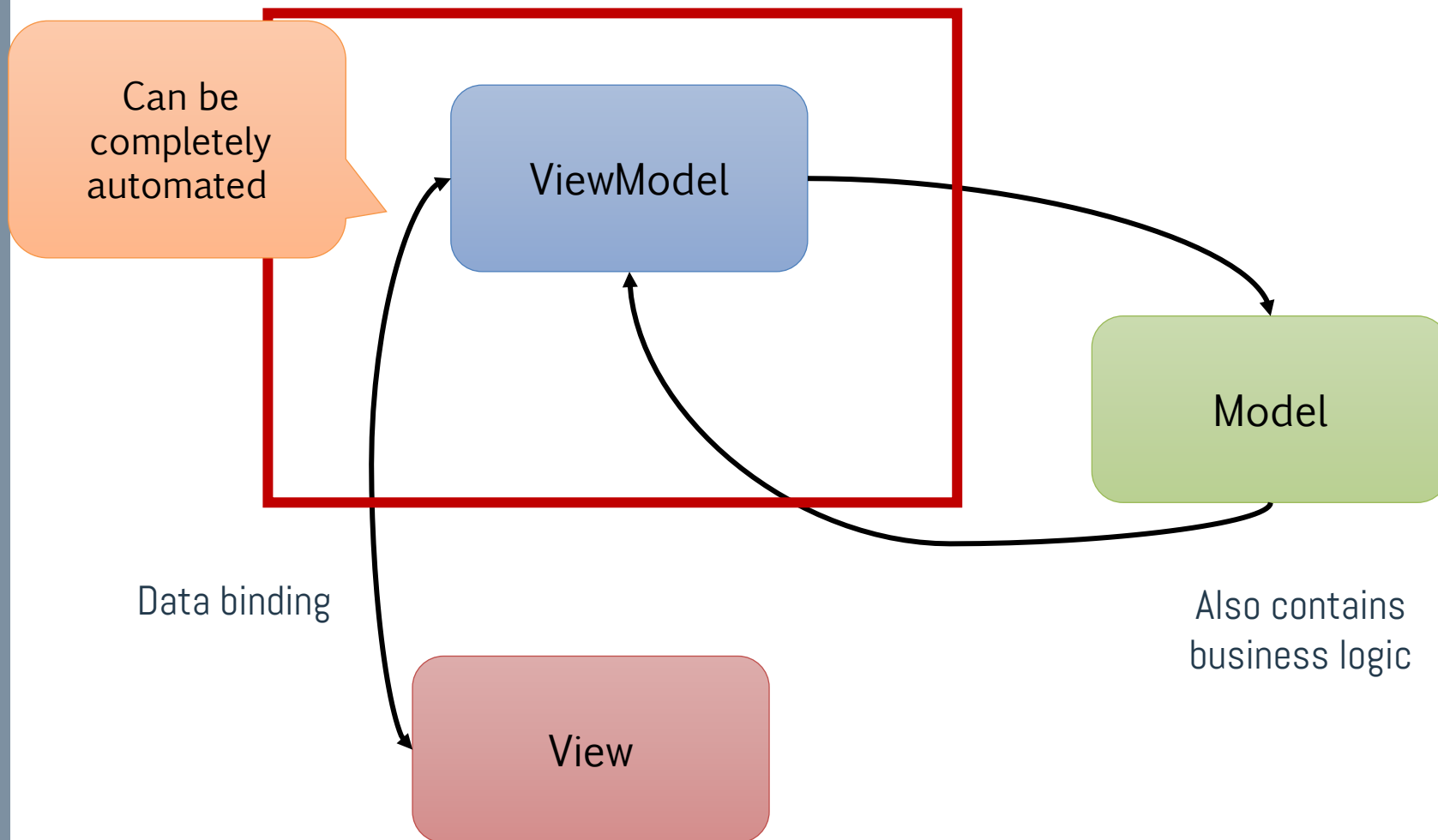
# MVVM structure

---





# MVVM structure



BY MICROSOFT

# References

---



## Course website

- › <http://hipert.unimore.it/people/paolob/pub/ProgSW/index.html>

## Book

- › I. Sommerville, "Introduzione all ingegneria del software moderna", Pearson
  - Chapter 3
- › For MVVM - <https://learn.microsoft.com/en-gb/archive/blogs/johngossman/advantages-and-disadvantages-of-m-v-vm>
- › Any book that teaches SOLID principles

## My contacts

- › [paolo.burgio@unimore.it](mailto:paolo.burgio@unimore.it)
- › <http://hipert.mat.unimore.it/people/paolob/>
- › <https://github.com/pburgio>