System Architecture and Design

I2ISE

System Architectural Design

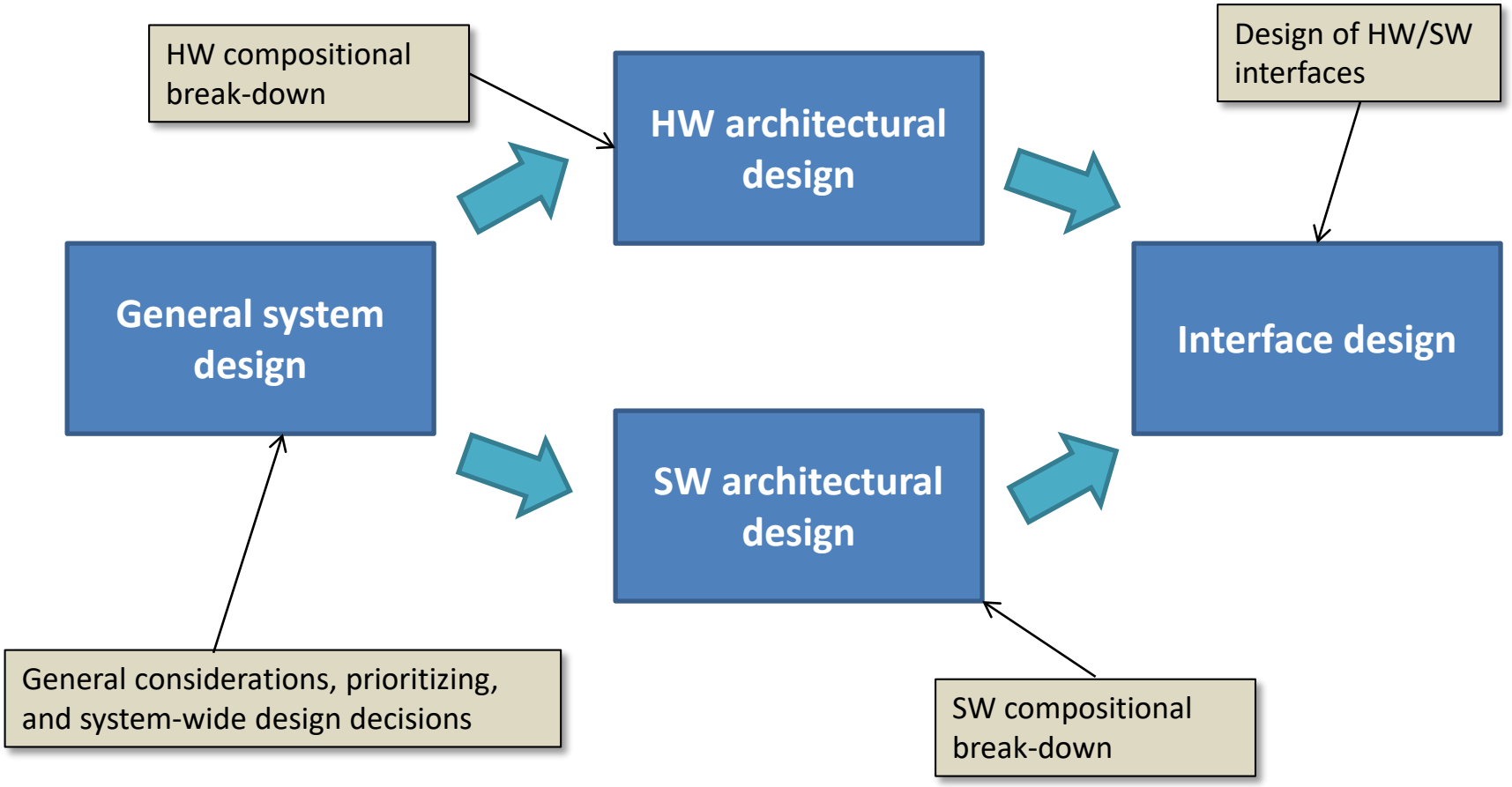
- Using our specification and System Design (BDD, IBD) as a starting point, we will now elaborate on the architectural design of the system

- The architectural design covers design decisions, priorities, HW and SW decomposition, etc. of our system

- This requires a number of different considerations which will be covered

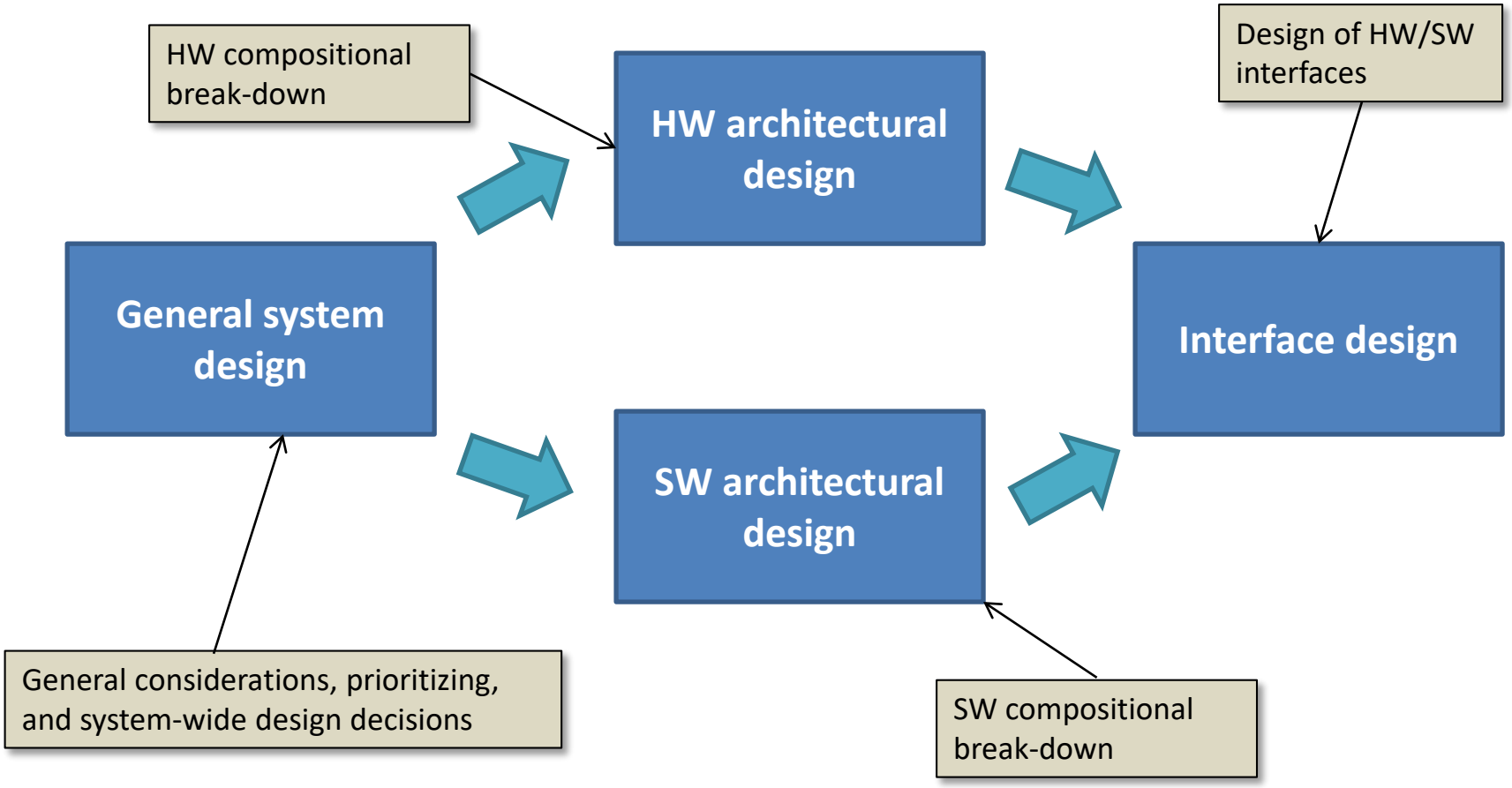
Architectural design activities - overview

- We will investigate 4 related architectural design activities:



General system design

- Today, we look at *General system design*



General system design

- In *general system design* we do design considerations

- There are important and hard choices to make

Fast, cheap, good – chose any two!

- But before we get to the hard choices, a couple of *design principles* that will help no regardless of choice

System design principles

- The system design principles are your new best friends!

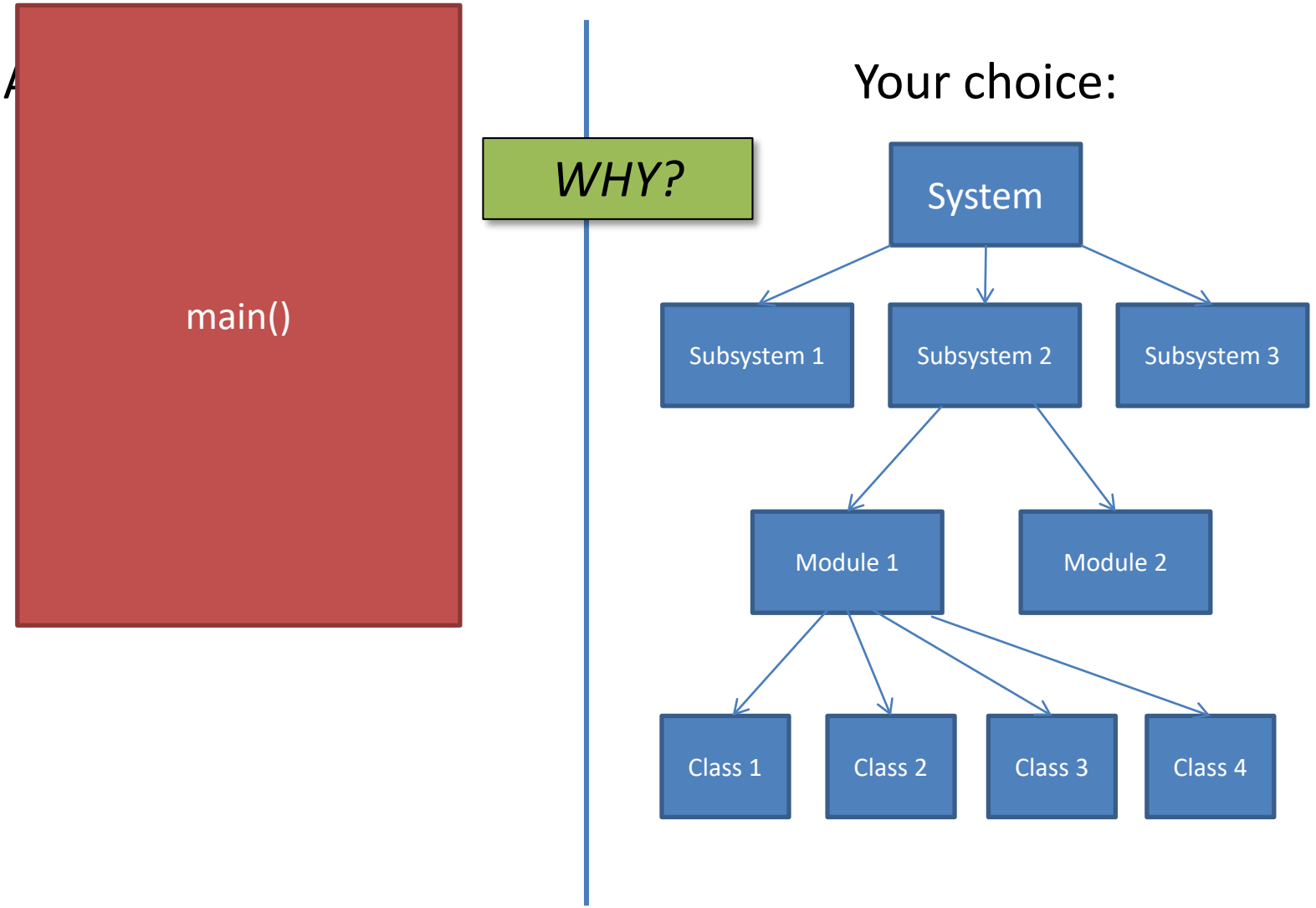
- They will help you construct a maintainable, scalable, easy-to-understand system

- The design principles will become second nature later in your studies. Right now, however, they are new and hard to understand and use.

System design principles

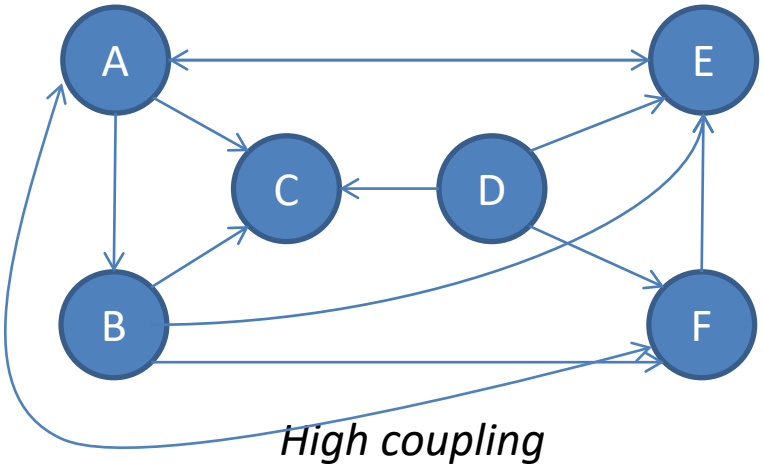
- The system design principles include:
  - *Decomposition*
  - L*ow coupling (kobling/binding)*
  - High *cohesion (samhørighed)*
  - Use *abstractions*
  - *Re-use* existing design solutions
  - Ensure *testability*

Design principles – Decomposition

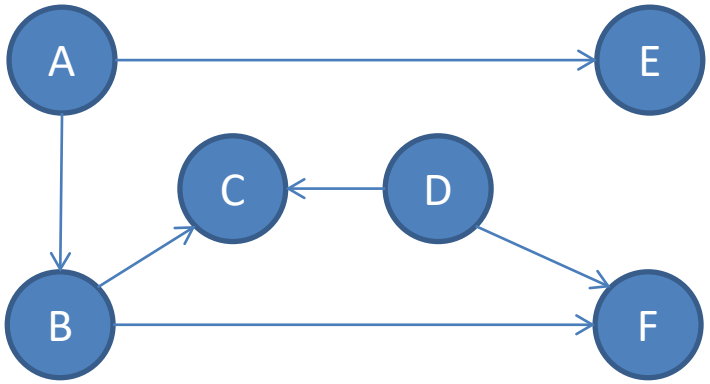


Design principles – low coupling

- *Coupling* is a measure of how dependent a {SW|HW} module is of other modules



High coupling

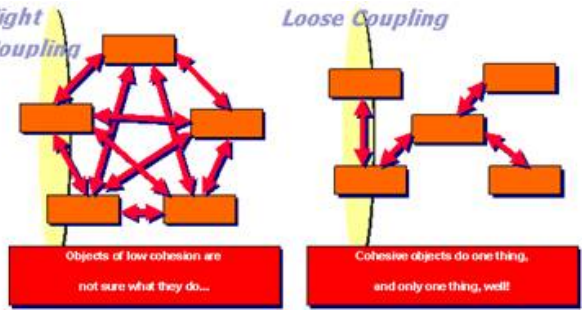


Low coupling

- Lowering the coupling entails a number of benefits – which?

High Coupling

- Difficult to debug during development
- Difficult to trace errors in running system
- Difficult to maintain
- Difficult to modify with new functionality
- ……



Advice

- Remove and reduce dependencies
- Minimize amount of information exchange between components
- Do not use global variables
- Keep design simple

Design principles – low coupling

- High coupling may be only indirectly evident:

```
void print(Report r)
{
    mySecretary.getTaskQueue().addPrintTask(new PrintTask(r));
}
```

Any changes to the secretary, the task queue, or the print task will impact our implementation of `print()` ☹

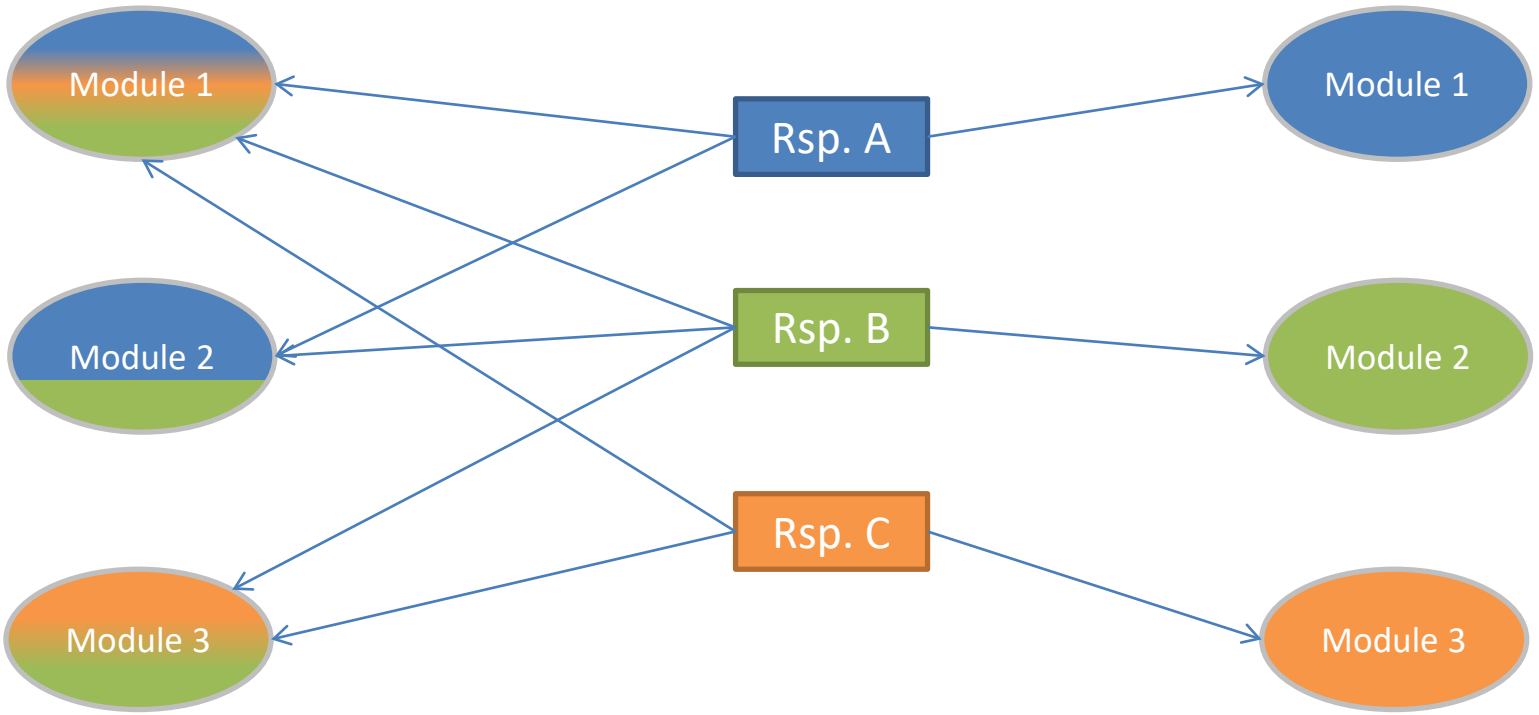
E.g. renaming `PrintTask()` -> `PrintJob()`: changes *everywhere!*

```
void print(Report r)
{
    mySecretary.print(r);
}
```

Only changes to the secretary will impact our implementation of `print()`. Whether she uses a task queue, stack, or whatever is not our concern ☺

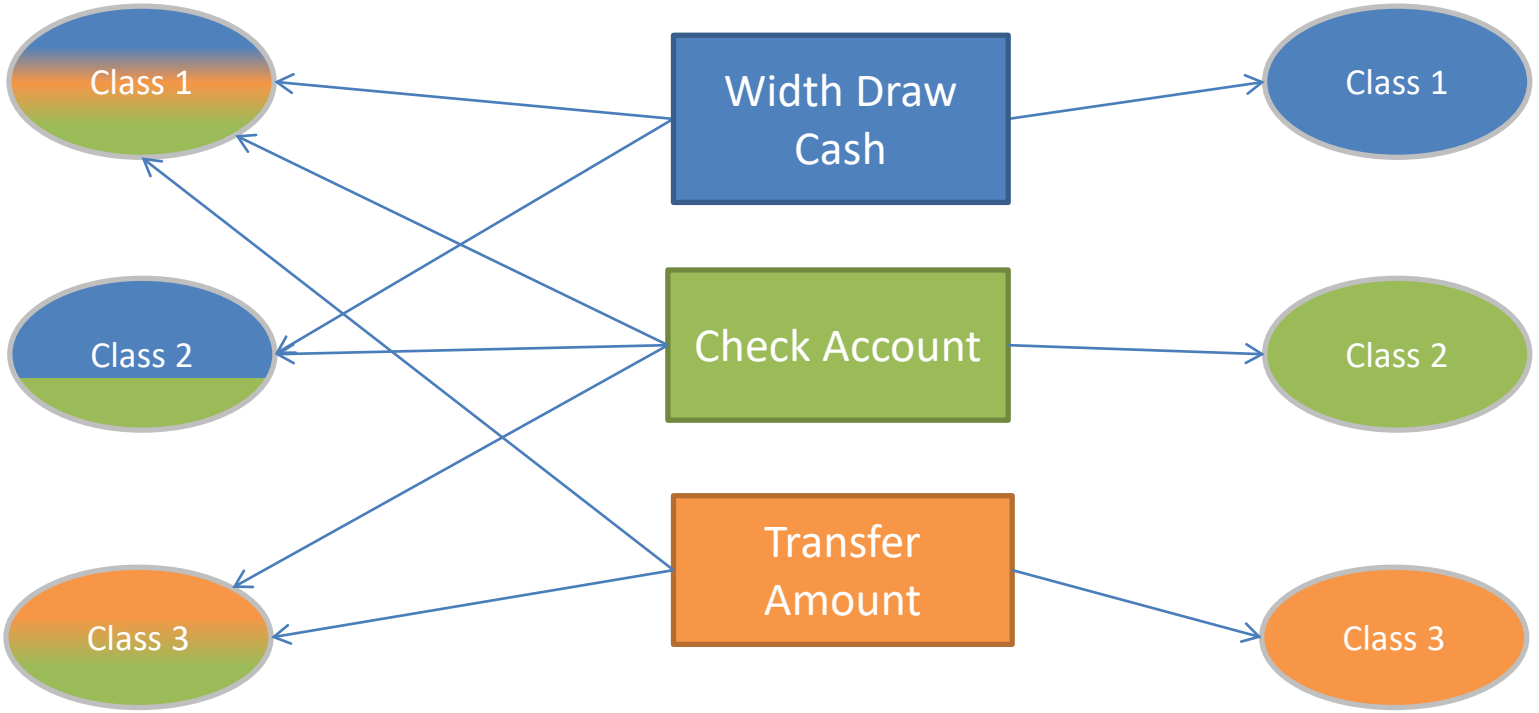
Design principles – high cohesion

- *Cohesion* is a measure of how well a given responsibiliy is encapsulated in a module or component



Design principles – Application Model (ATM)

- *Functionality specified by use cases, that is realized by controller classes, is an example of achieving high cohesion*

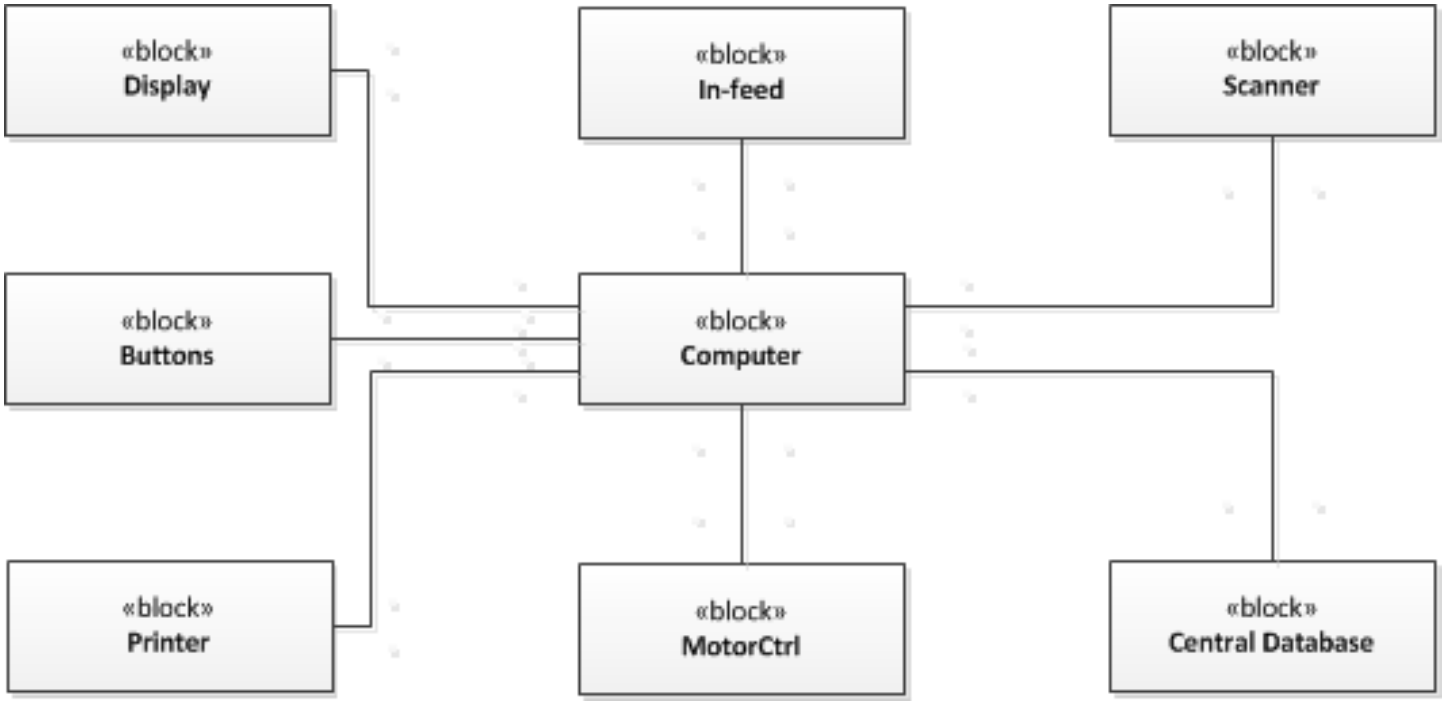


Types of Cohesion

- **Functional** – all elements contributes to execution of a specific task
- **Sequential** – output from one procedure becomes the input for the next
- **Communication** – procedures that operates on the same set of input data

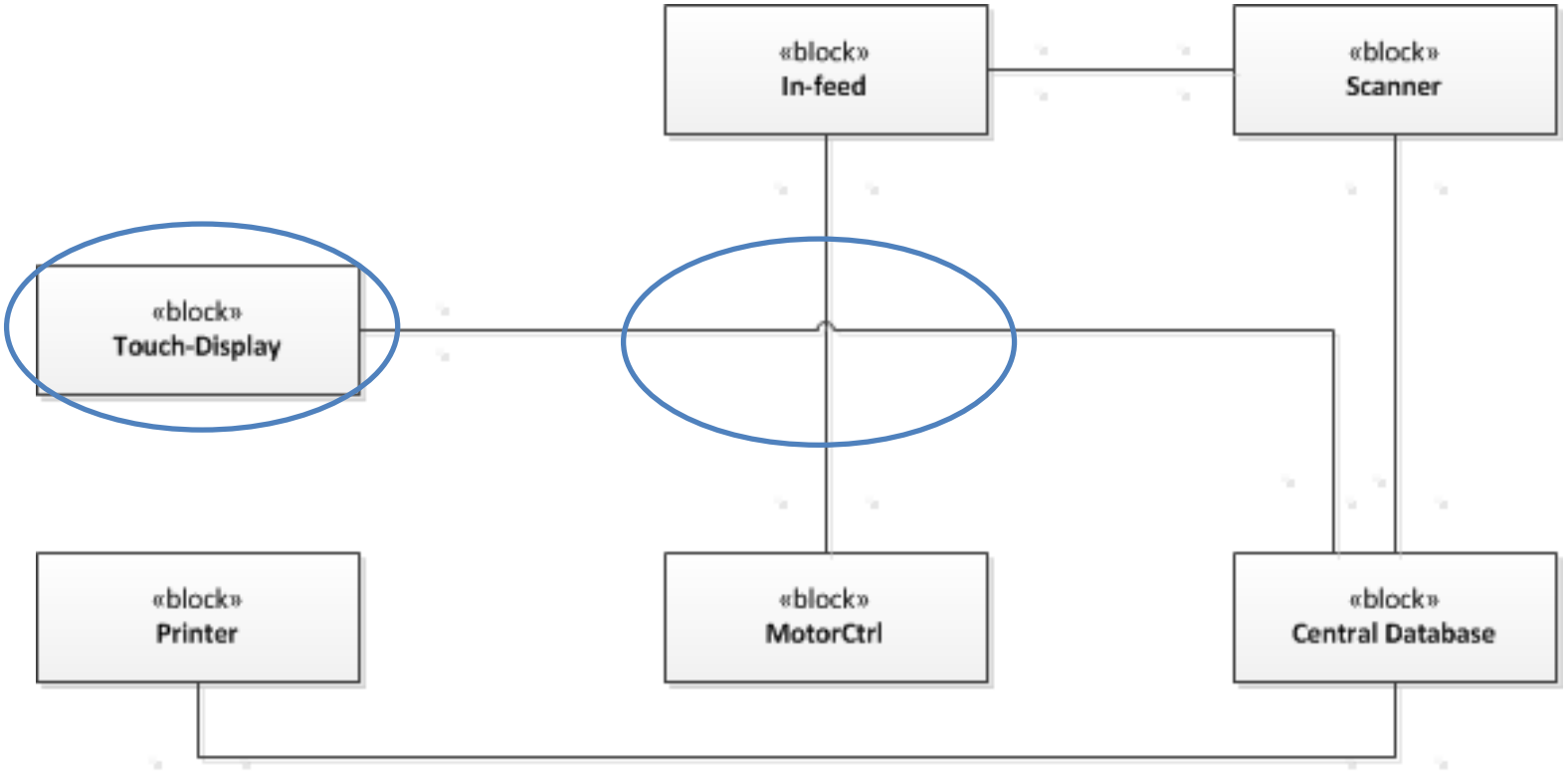
RVM Architecture 1

- Cohesion / Coupling ?



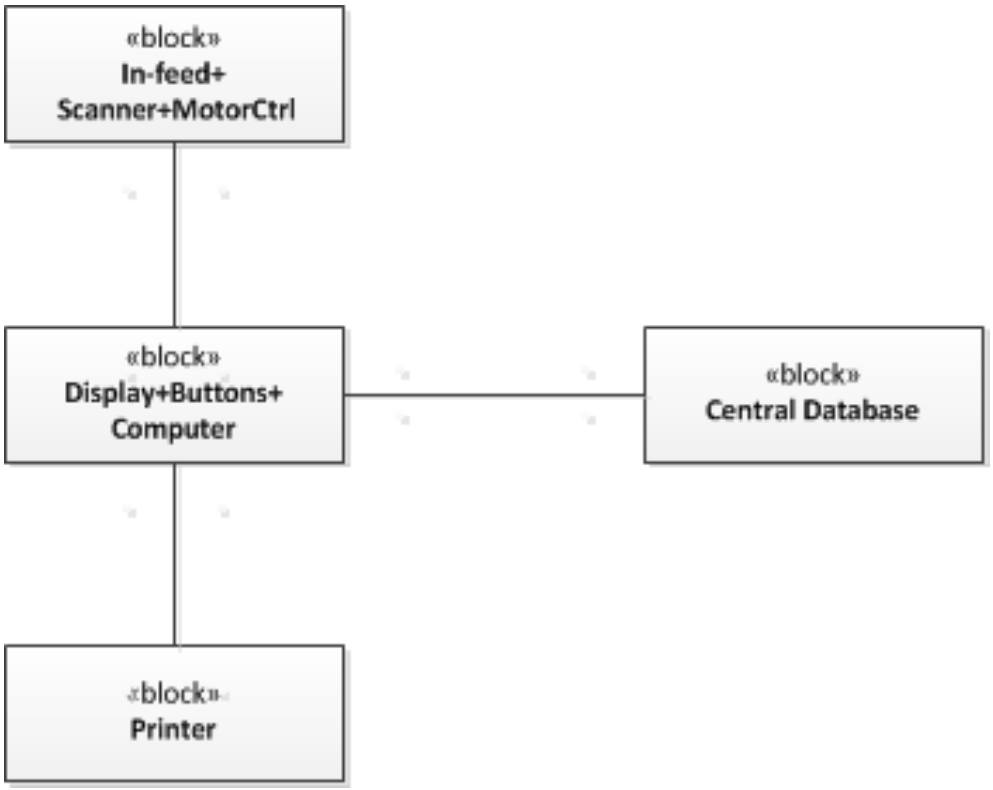
RVM Architecture 2

- Cohesion / Coupling ?
- Is the same functionality possible to realize as in RVM Architecture 1?



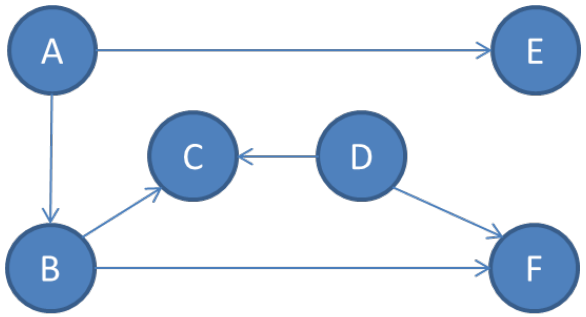
RVM Architecture 3

- Cohesion / Coupling ?
- What about errors and replacement of components?

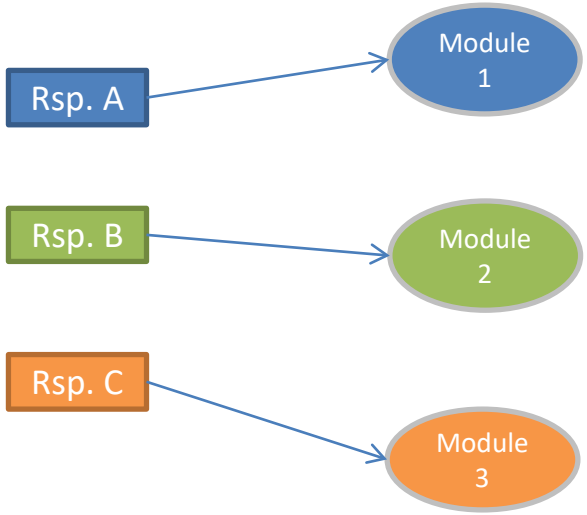


Low coupling, high cohesion – your turn!

- 5 minutes: Discuss…
  - …the benefits of high cohesion in a design
  - …how low coupling supports high cohesion
  - …how low cohesion hinders low coupling



Low coupling



High cohesion

Forklar kort hvilken betydning begreberne ”coupling” og ”cohesion” har i forhold til et godt arkitekturdesign.

- *Design principperne”coupling” (kobling) og ”cohesion” (samhørighed) er vigtige for et godt design. Der skal være lav kobling mellem klasser/komponenter i designet og stor samhørighed i de enkelte klasser/komponenter. Den lave kobling gør at de enkelte komponenter er lettet at vedligeholde, udskifte, modificere og teste. Samhørigheden internet i klassen/komponenten kan omhandle emner som: funktionsmæssig sammenhæng, sammenhæng mellem sekvens af operationer, kommunikation og logisk sammenhæng.*

Abstractions

- Using *abstractions* help you achieve low coupling and high cohesion
  - Disregard *irrelevant details*, focus on only some aspects
  - Data or control abstraction

- Using abstractions properly will also increase cohesion

- An example: My abstraction of the garage that repaired my car's parking sensor

Abstractions – parking sensor

```
Car fixMyCar(Garage garage, Car car)
{
    Mechanic kian = garage.findMechanic();
    Lift lift = garage.getAvailableLift();
    kian.moveCarToLift(car, lift);
    lift.liftCar();
    kian.removeRearBumper(car);
    if(kian.inspectElectronicsUnderRearBumper(car) == BROKEN)
    {
        ParkingSensor ps =
                    kian.getNewParkingSensor(garage.getStock());
        kian.install(ps, car);
    }
    else
        kian.cleanParkingSensorHeads(car);
    kian.attachRearBumper(car);
    lift.lowerCar();
    kian.moveCarToParkingLot(garage.getParkingLot());

    return car;
}
```

Far too many details that I (as car owner) do not care about. I just want my car fixed ☹.

```
Car fixMyCar(Garage garage, Car car)
{
    garage.repair(car, ”PARKING SENSOR BROKEN”);
    return car;
}
```

Much better! Let the garage worry about *how* to fix my car - they can use Kian, Mary, a robot or ninja smoke for all I care. I just want my car fixed ☺

Automation – Production

Airbus A380 – Flight deck

Design for test – Ensure testability

- Plan how to test hardware components and software classes at the same time as designing the system
  - V-model
- Unit and component test
  - Interfaces
  - Functionality
  - Test cases: stimuli and expected response
- Integration test
  - Top-down or Bottom-up
  - Stubs and drivers

Meanwhile, in the system design cave…

System design - activities

- We are going to make some tough decisions. These include:

List of decisions:

1. Prioritize design criteria
2. Architectural design strategy / strategies
3. Data storage strategies
4. Software control strategies
5. Initiation/termination strategies
6. Error handling
7. Self-test and backup functions
8. …

- The choices we make will impact the system architecture as a whole.
- We will investigate some of these in the following

Prioritizing design criteria

List of decisions:
1. **Prioritize design criteria**
2. Architectural design strategy / strategies
3. Data storage strategies
4. …

- A system's selected design criteria are the foundation upon which design decisions are later made
- Welcome to the real (imperfect) world!
  - Design criteria are diverse and contradicting - "*Fast, cheap, good…*"
- The selected design criteria are driven by the *intended market* and the *existing technology*
- Some example criteria on next slide

Prioritizing design criteria

List of decisions:
1. **Prioritize design criteria**
2. Architectural design strategy / strategies
3. Data storage strategies
4. …

Cost
Performance
Time-to-market
Safety
Security
…

Extensibility
Functionality
Reliability
Usability
Testability
Scalability

Prioritization

Decisions

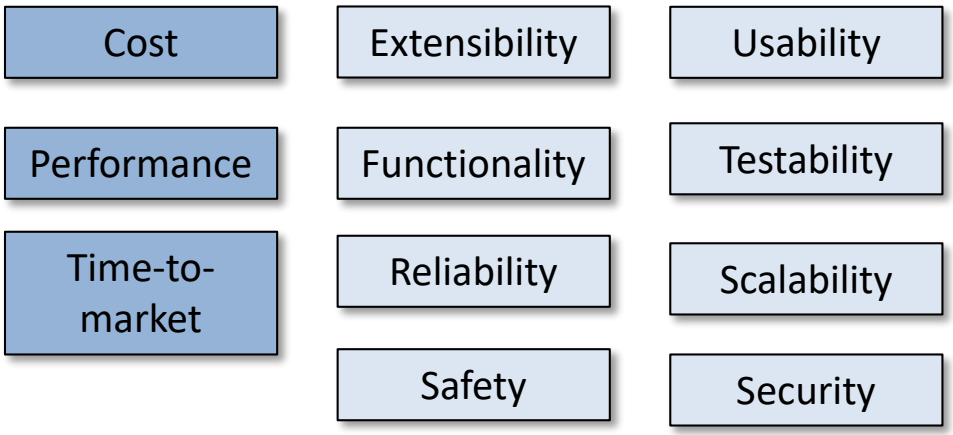
Exploitation

Available technology
(hardware, frameworks, operating systems, …)

Prioritizing design criteria - your turn!

List of decisions:
1. **Prioritize design criteria**
2. Architectural design strategy / strategies
3. Data storage strategies
4. …

- 10 minutes: Select top-4 design criteria if you are making…
  - An iPhone accessory
  - A nuclear power plant
  - A Reverse Vending Machine
  - ”Slusesystem”



Architectural design strategy

List of decisions:
1. Prioritize design criteria
2. **Architectural design strategy / strategies**
3. Data storage strategies
4. …

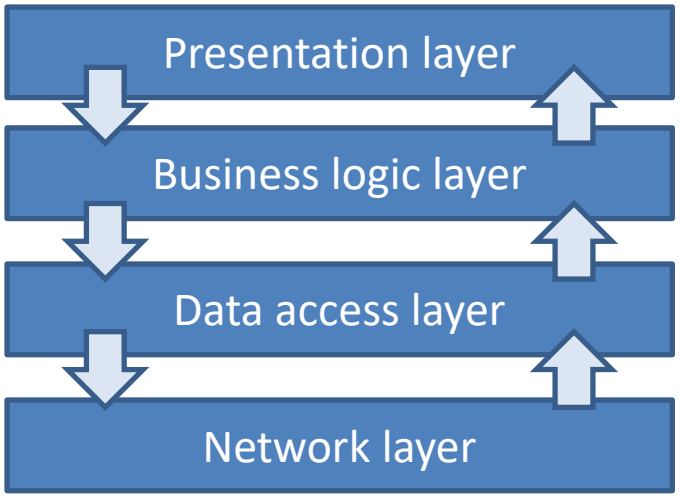
- An architectural design strategy is a strategy for the design of the system. This includes…
  - Selection of layering
  - Deciding the use of framework(s)
  - Network technologies
  - Database management
  - …

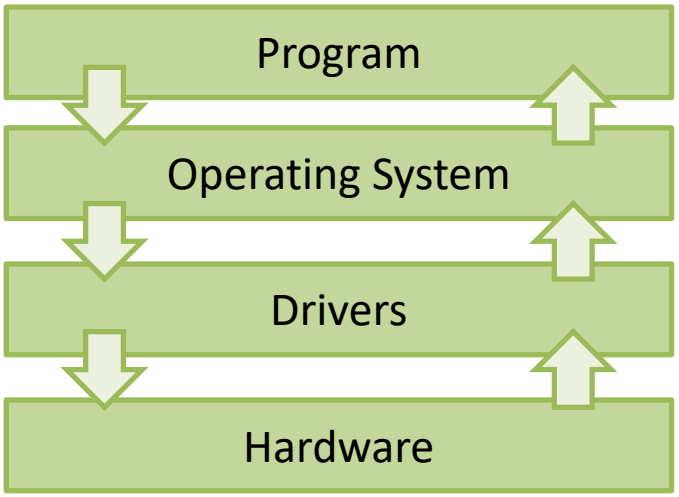
- Let’s take a look!

Architectural design strategy - Layering

List of decisions:
1. Prioritize design criteria
2. **Architectural design strategy / strategies**
3. Data storage strategies
4. …

- Layering the system is one of the most effective ways of achieving low coupling on a system-wide scale.

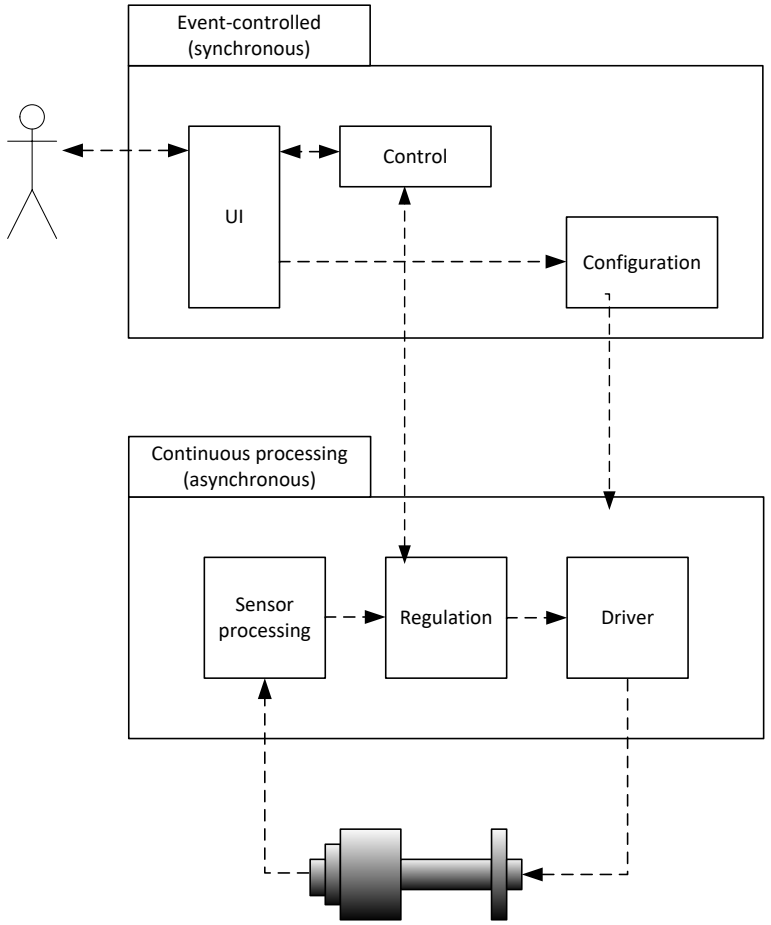




Architectural design strategy - Half-sync, half-async

List of decisions:
1. Prioritize design criteria
2. **Architectural design strategy / strategies**
3. Data storage strategies
4. …

- For continous (e.g. control) systems, *half-sync, half-asynch* can be a fine approach



Frameworks

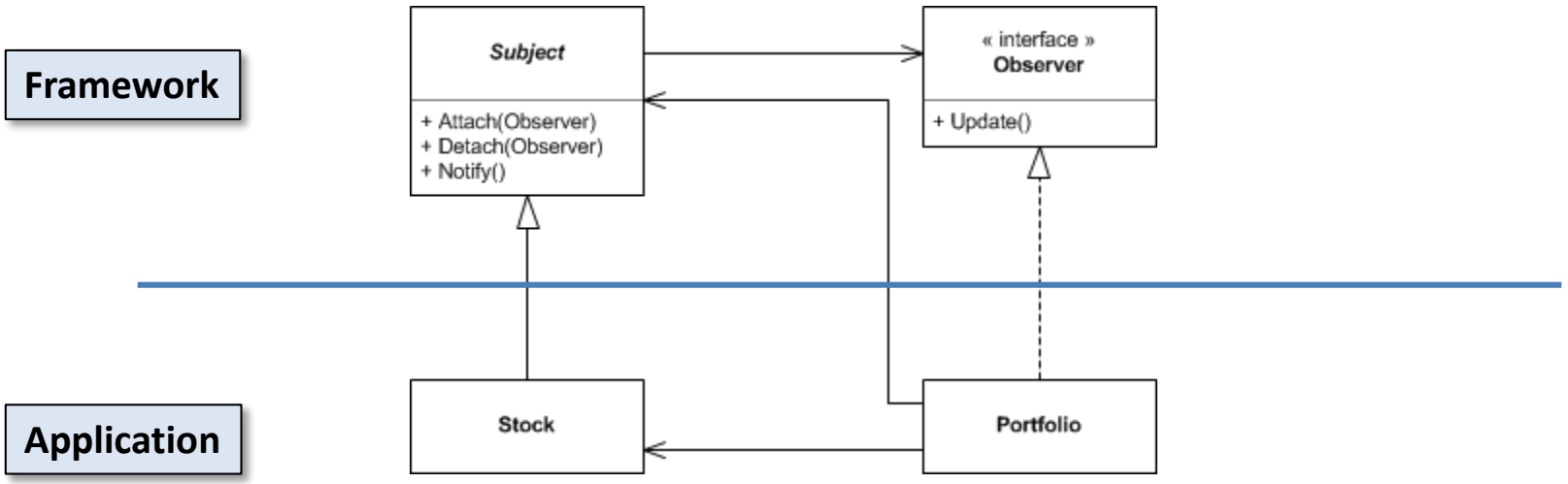
List of decisions:
1. Prioritize design criteria
2. **Architectural design strategy / strategies**
3. Data storage strategies
4. …

- Frameworks are ready-made software systems which you instantiate, typically by extension
  - Declare inheritance from a framework class
  - Implement framework (abstract) methods

- Frameworks provide a lot of functionality and decoupling, and typically hides irrelevant details

Frameworks – example: A stock market

List of decisions:
1. Prioritize design criteria
2. **Architectural design strategy / strategies**
3. Data storage strategies
4. …




```
Stock::setValue(double v)
{
  value = v;
  Notify();
}
```

```
Portfolio::addStock(Stock s)
{
  s.Attach(this);
}
```

```
Portfolio::Update()
{
  // Do something with the stock
}
```

Initiation/termination strategies

List of decisions:
4. …
5. **Initiation/termination strategies**
6. Error handling
7. Self-test and backup functions

- You should also consider how the system is initiated and terminated

- Initiation:
  - Start order?
  - Access to configuration data?
  - Power-On Self-Test?

- Termination:
  - Termination order?
  - Configuration storage?

Error handling

List of decisions:
4. …
5. Initiation/termination strategies
6. **Error handling**
7. Self-test and backup functions

- Hardware in system →System will eventually fail
- Error handling is critical to get into the system from day 1
- You need a *strategy* for this.
  - How do you *detect* errors?
  - How do you *handle* detected errors?

Some strategies for error detection and handling

List of decisions:
4. …
5. Initiation/termination strategies
6. **Error handling**
7. Self-test and backup functions

- Detection:
  - Watchdogs – detect software deadlocks
  - Voting systems
  - Self tests / self-diagnostics
    - Power-On Self-Test (POST)
    - Continuous Built-In Test (CBIT)

- Handling
  - User intervention
  - Limp mode
  - Redundant systems