
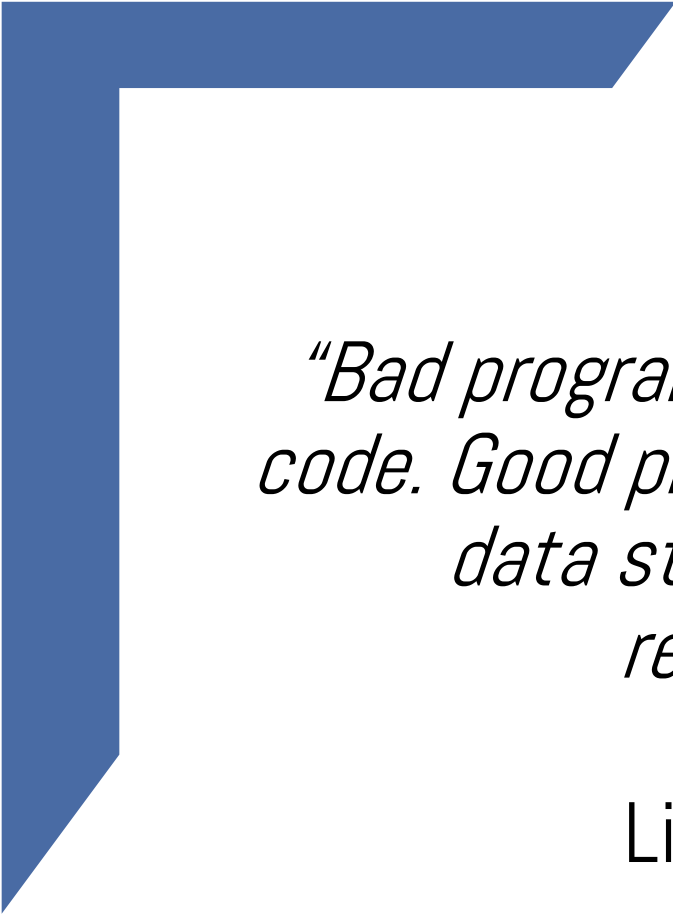


# Engineering embedded software

---

Paolo Burgio  
paolo.burgio@unimore.it



*"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."*

Linus Torvalds





# What will we see?

---

How to code properly

- › “Properly?”
- › In such a way that your code becomes scalable, maintainable, robust..
- › Do engineer’s job!

Three pillars

- › SOLID programming ← *Disclaimer: these two are not born for embedded systems*
- › Design patterns (for embedded systems)
- › CLEAN code architecture ←



# SOLID programming



# SOLID programming

---

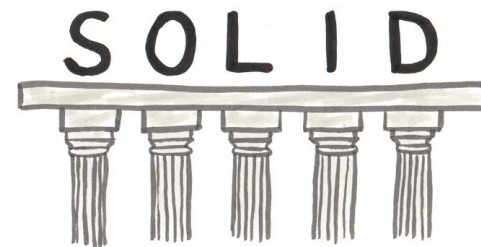
Aka: Object Oriented Design

Five principles that ~~save your life~~ make the difference between a programmer/coder and a software architect

› (Or between an happy person and a sad person)

Applicable to object-oriented programming (but not only)

1. Single Responsibility
2. *Open/Close principle*
3. *Liskov substitution*
4. Interface segregation
5. Dependency inversion





# Single responsibility principle

---

*A software entity should have only one reason to change*

- › Aka: every class should have a single responsibility or single job or single purpose
- › **Answers to:** "What should I put into a class?"
- › **Pros:** you always know where/what/how to change your code, and don't mess up things
- › (**Cons:** increase number of classes and effort...)

How to implement this in embedded code?

- › Replace "class" with "module" in the above sentence



# Open/Close principle

---

*Entities should be open for extension, but closed for modification*

- › Aka: you should never change anything, always adding new behavior using polymorphism
- › **Answers to:** "How should I extend my code?"
- › **Pros:** reduce the number of bugs and headaches
- › **(Cons:** N/A)

How to implement this in embedded code?

- › Might not be easy to implement





# Liskov substitution principle

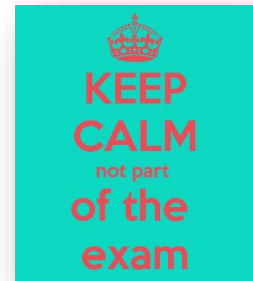
---

*You should be able to substitute any parent class with any of their children without any behavior modification*

- › Aka: "Rectangles vs Squares"
- › **Answers to:** "When and what should I inherit?"
- › **Pros:** code is scalable, and minimize changes upon modifications
- › (**Cons:** you have to think before you code)
  - not actually a con...

How to implement this in embedded code?

- › Replace "class" with "module" in the above sentence







# Interface segregation principle

---

*Many client specific interfaces are better than a big one*

- › Aka: Interfaces should be the minimal set of behaviors you need
- › **Answers to:** "When should I create an interface?"
  - Spoiler: "as much as you can"
- › **Pros:** you minimize dependencies in your code (thus, programming effort)
- › (**Cons:** trust me...NONE)

Always remember: an interface is a contract. You can build (automatic) tests over contracts!

How to implement this in embedded code?

- › Replace "interface" with "C[++] header" in the above sentence



# Dependency inversion principle

---

*Your project shouldn't depend of anything, make those things depend of interfaces*

- › Design wrappers around your dependencies
  - (This is **NOT** “dependency injection”...but its good friend)
- › **Answers to:** “How can I avoid getting crazy with dependencies?”
- › **Pros:** isolation between code components; your code reflects the analysis/model of business
- › (**Cons:** additional programming effort)

How to implement this in embedded code?

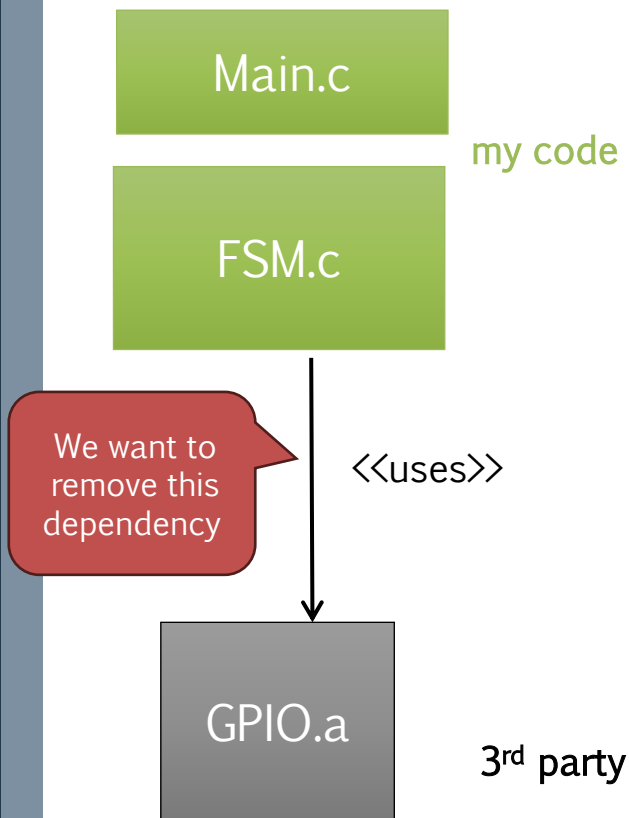
- › We'll see this now....



# Dependency inversion principle

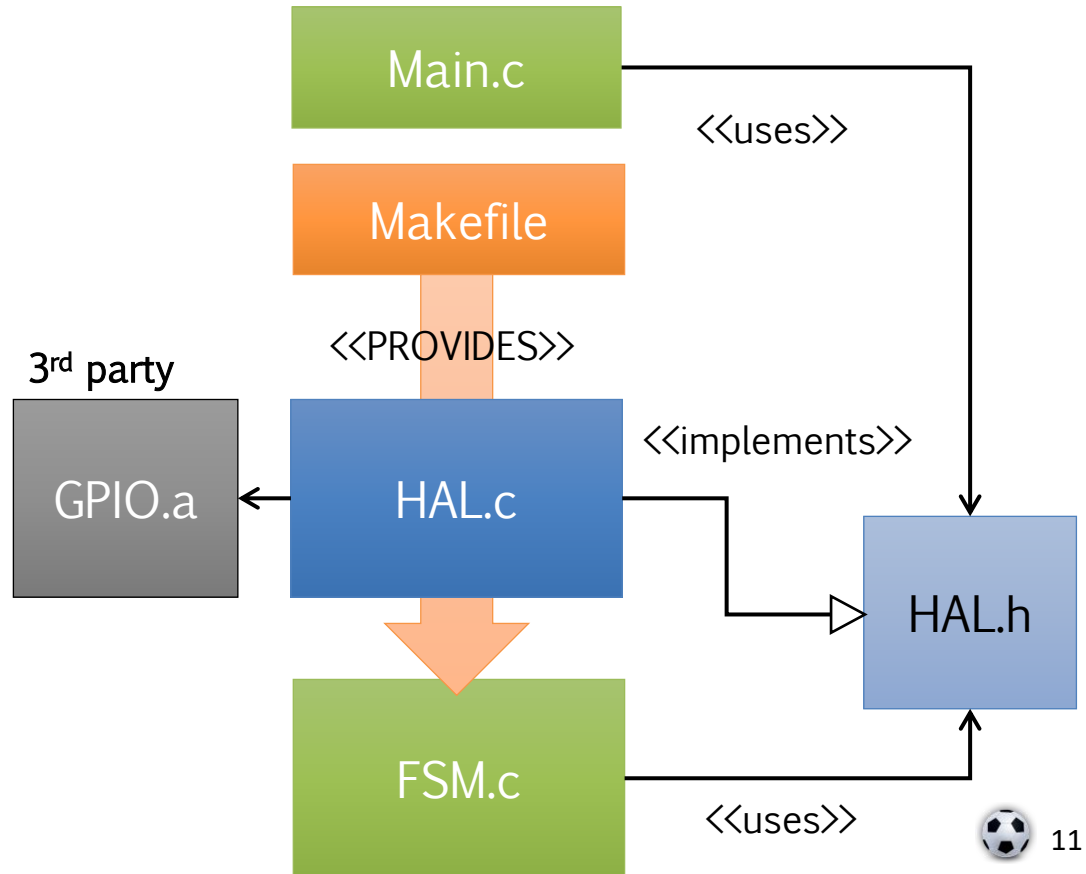
## Library-like approach

- › Tied to 3<sup>rd</sup> party code



## Framework-like approach

- › Inversion of control
- › Dependency injection





# Dependency inversion principle

HAL.c

```
#include "GPIO.h" // 3rd party dependency

/* Implementation */
void toggle_led(int led_id, bool onoff) {
    // ...
}
```

## Framework-like approach

- › Inversion of control
- › Dependency injection

HAL.h

```
/* Turns on the blue led */
void toggle_led(int led_id, bool onoff);
```

FSM.c

```
#include "HAL.h"

/* Computes output.
   Moore machine. */
void mfn(int currState) {
    switch(currState) {
        case 0:
            toggle_led(BLUE, true);
            break;
        case 1:
            // ...
    }
}
```

GPIO.a

Makefile

<<PROVIDES>>

HAL.c

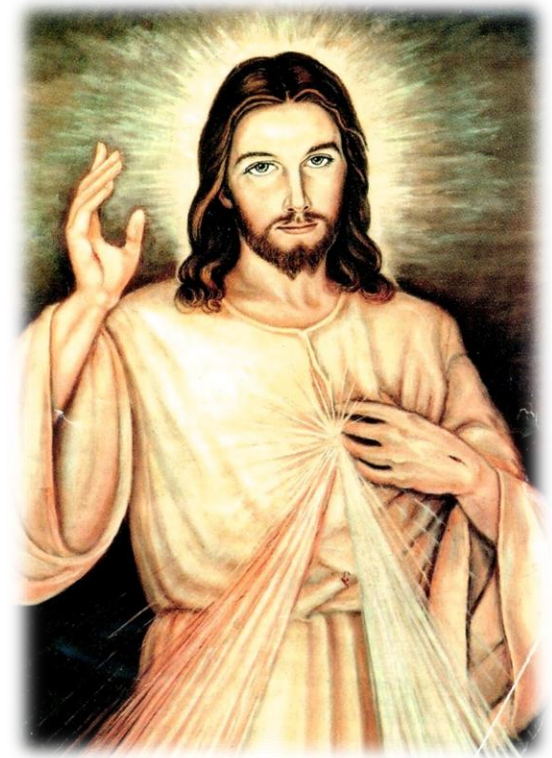
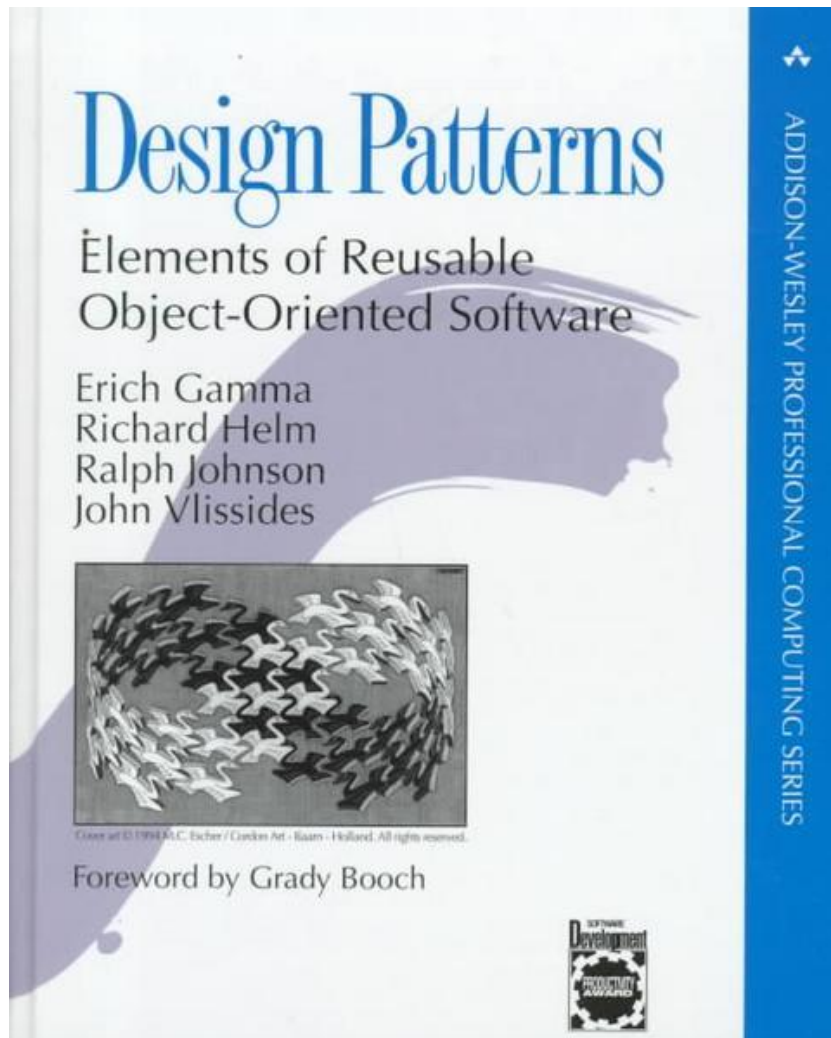
<<implements>>

HAL.h

FSM.c



# Design patterns



The Gang of Four



# Elements of reusable Object Oriented Software \*

---

## Elements

- › Simple, basic parts of

## of reusable

- › We did mistakes, we learned from them

## Object Oriented

- › Yet can be reused in non-OOP structure

## Software

- › ....

\* *Cit. Wikipedia*



# As simple as that

---

Your parents, grandparents, teachers, ancestors faced problems

They found solutions

- › ..smart solutions...

This is their (our) legacy

- › Hundreds of know problems, with known solutions
- › All of them build upon basic principles
- › Sync/vs async, de-coupling, SOLID, etc





# Ok, let's be clear

---

What design patten **can** give you

- › A common, known vocabulary
- › Solve complex problems way ahead of time
- › Provide solid ground to motivate your design choices

What they **cannot** give you

- › Exact solution: each problem/project is unique
- › Full-fledged solution for every design/programming problem

But they can save you a lot of headaches!



# How do they help you?

---

They force you to

- › Find appropriate objects to model your domain (aka: decomposition)
- › Determine objects granularity (e.g., *Creational* patterns such as *Factory*)

Clearly define interfaces (.h) and modules (.c/.cpp)

- › Defining implementations...
- › ...and the relations among them

Implement reusable code

- › Separate modules for separate functionalities
- › Delegation (e.g., *Adapter*, *Strategy*, *Visitor*) implements loose coupling among SW entities
- › “*Who has control?*”, “*Who creates objects?*” ...focus on the **role** of your SW entities!



# Commonly known (design) mistakes

---

...you didn't know about

- › You explicitly declare object and modules
- › You explicitly call methods, to implement an high-level operation
- › You have strong dependencies on HW and SW platforms (e.g., embedded)
- › Your module depend on internals of another module
- › Your code might depend on algorithms that you implement
- › Tightly coupling among components/modules/classes/...
- › Always use subclasses to extend functionality/specialize behavior
- › (not actually a mistake) you might need to modify a "closed" modules
- › ...



The so-called Code smells



# (Incomplete) taxonomy of design patterns

---

## Creational

- › Factory
- › *Singleton*
- › Builder
- › Prototype

## Structural

- › **Adapter / HAL**
- › Bridge
- › Composite
- › Façade
- › **Proxy**
- › Decorator
- › FlyWeight



## Behavioral

- › Chain of Responsibility
- › Command
- › Iterator
- › Interpreter
- › Mediator
- › Memento
- › Observer
- › **State**
- › Strategy
- › Visitor

...and...

- › **CLEAN**



# The typical structure of a design pattern

---

1. Name, purpose, aliases
  2. Motivation - *Why the hack should I do so?*
  3. Applicability - *Where it applies, and where it doesn't*
- => What to do (Personal note: even if you don't know why...use them!)

A full set of example/code snippets to implement it

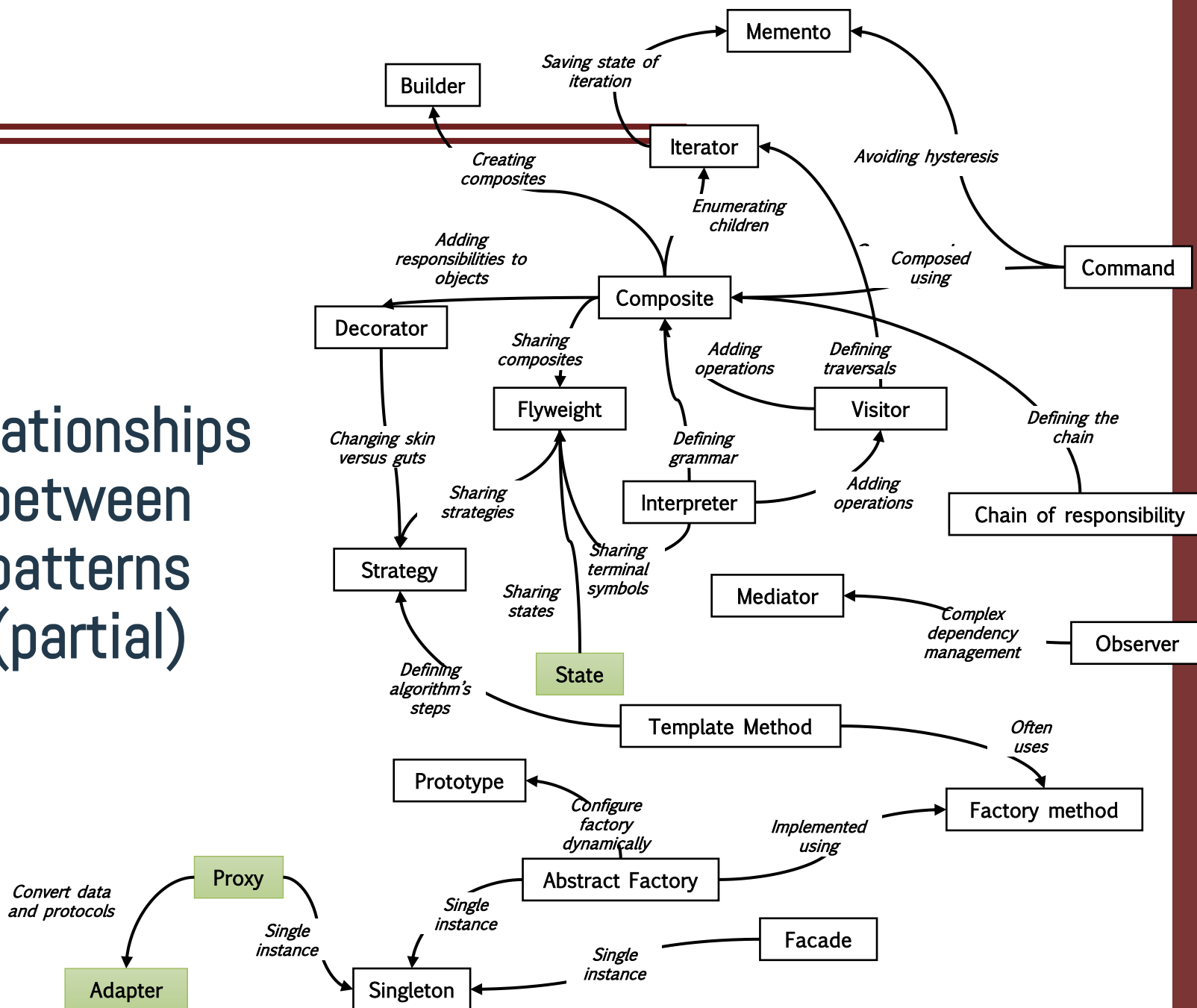
- › With known examples
- › With related patterns (everything is part of a bigger picture!)
- › With (wanted or unwanted) side effects

The bad news

- › I will only teach you 2-3 of them
- › Advanced (LM?) courses can give you a full
- › Coding, coding, coding



# Relationships between patterns (partial)





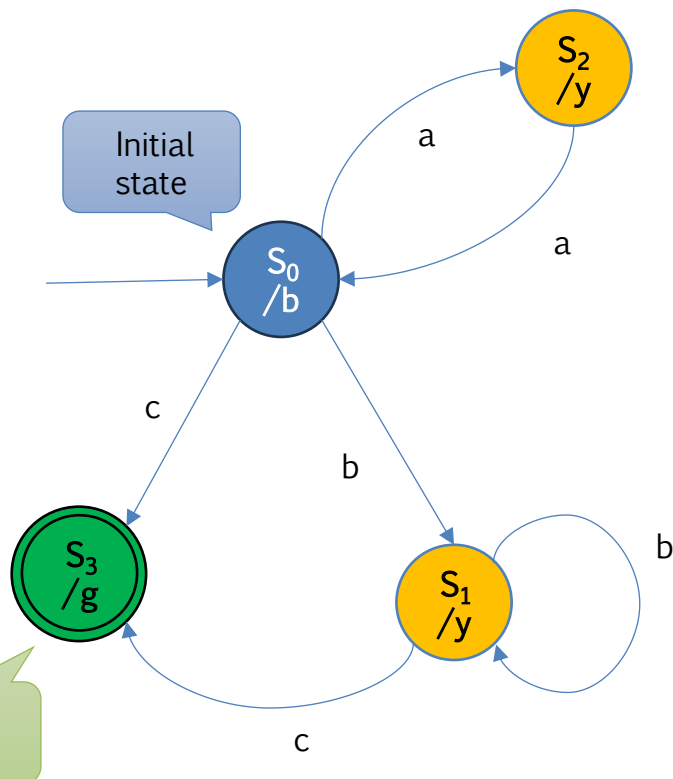
# Finite State Machine Pattern(s)



# Finite State Machines

Automata/FSMs are **not** a pattern!

- › But there are patterns that help implementing them
- › Basic implementation is typically a switch or a table
- › Can implement more scalable constructs



FSM.c

```
switch(state) {  
  case 0:  
    switch(input) {  
      case 'a':  
        state = 2;  
        output = YELLOW;  
        break;  
      default:  
        state = -1;  
        output = RED;  
        break;  
    }  
    break;  
  // ...  
}
```





# State pattern

---

A **behavioral** pattern

Also known as

Object state

## Purpose

- › Let an entity change its behavior because of a change of its internal state (i.e., an FSM)

## Motivation

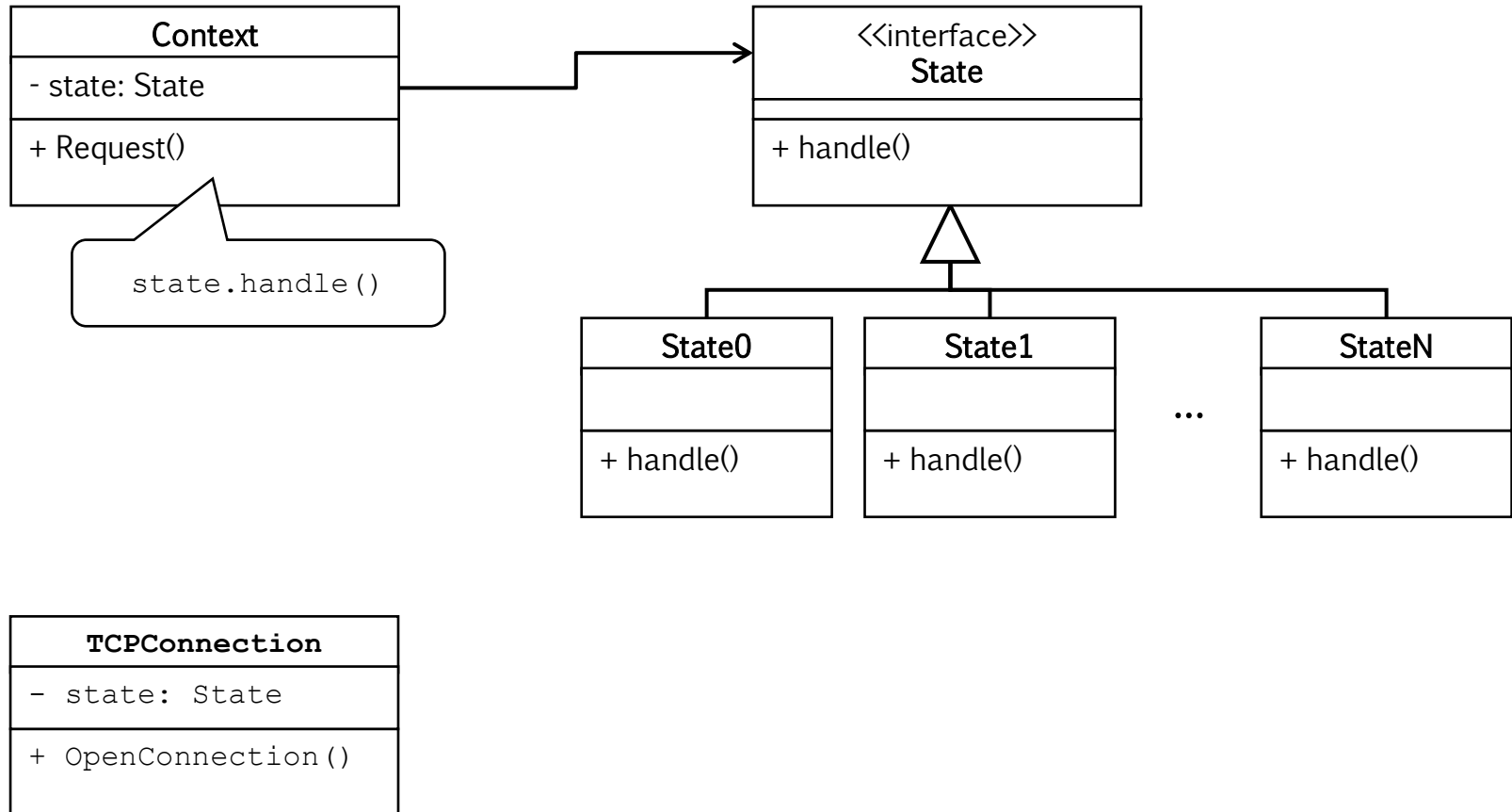
- › Providing a scalable and maintainable (and elegant) way of implementing FSMs

## Applicability

- › When the FSM has many states, and you would be forced to use multiple nested `switch` or `if`
- › When you want to acknowledge that STATE and FSM are two distinct entities
- › (Remember the Single Responsibility principle..)

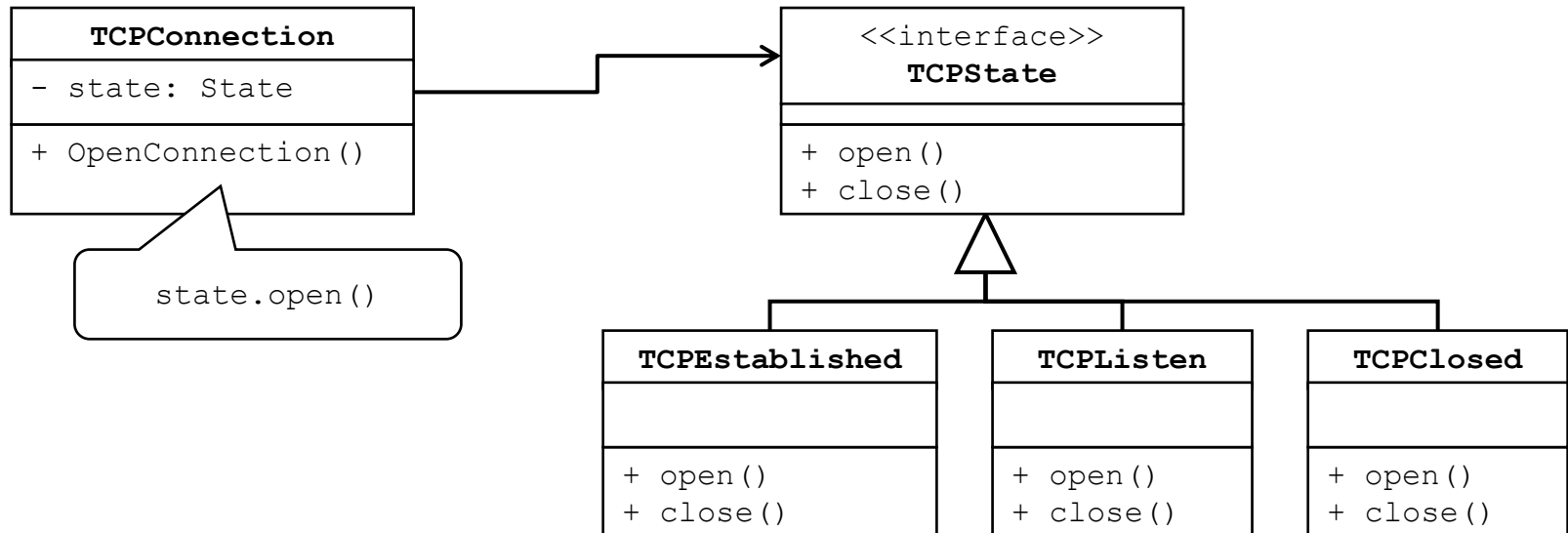


# State pattern: class diagram



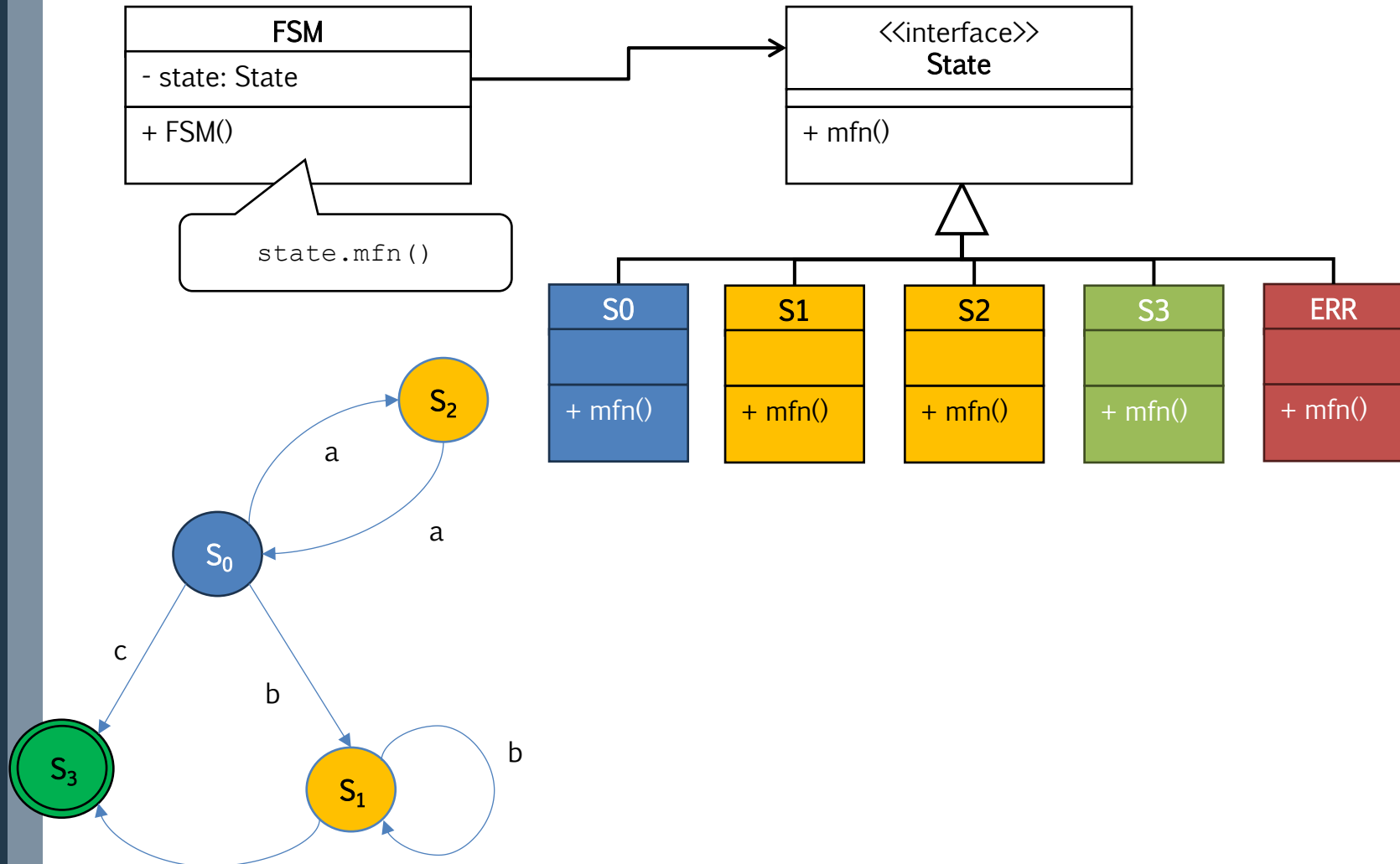


# State pattern: TCP connection





# State pattern: automata1 (only MFN)





# Hardware Abstraction Layer

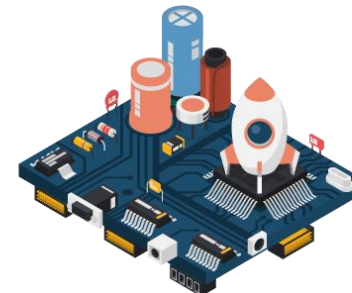
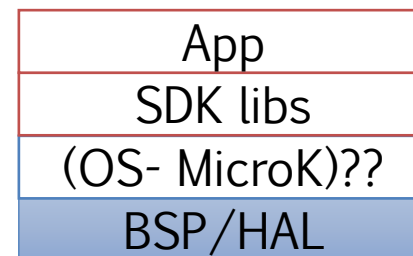
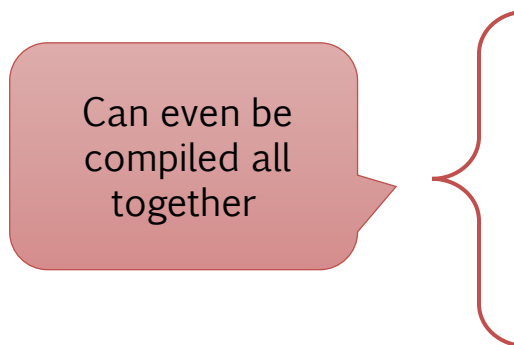
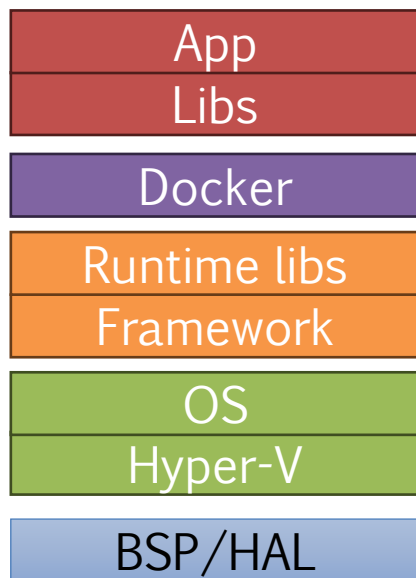
(aka: Hardware Proxy, Hardware Adapter...)



# We are closer to HW than ever

Software stack for General-purpose/HPC systems vs. Embedded systems

› Note: this is just a possible example





# The challenge: abstracting the HW

---

- › Cores and caches are hidden, however specific functionalities might exist (ex: RISC-V extensions)
- › Explicit memory management: call `free/delete` after `malloc/new`
- › HW devices are typically memory-mapped: I/O space
- › We speak with them setting-unsetting bits, registers, using masks, etc

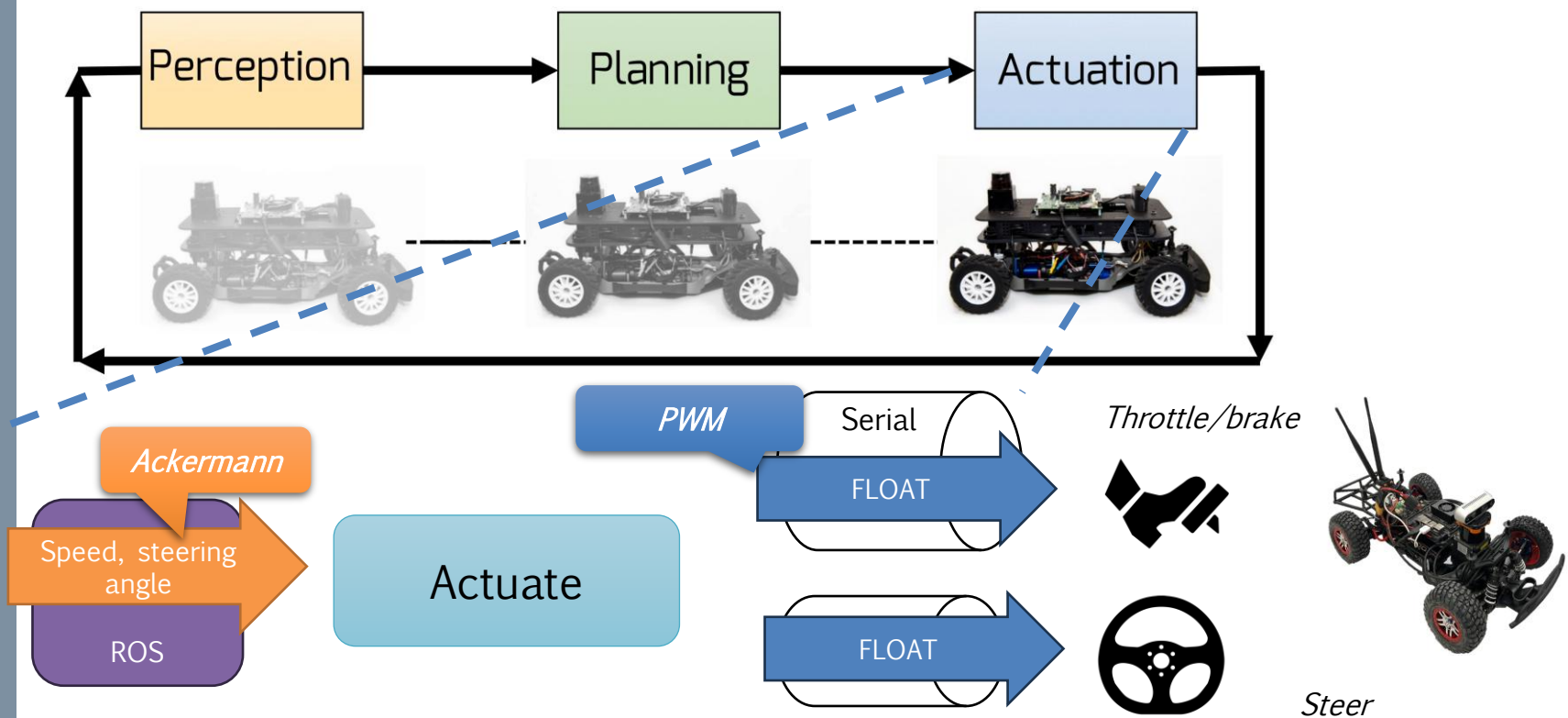
Every device has a specific protocol!

- › Actually, also GP system have this issue...but they have full-fledged OS such as GNU/Linux and Win
- › How can we convert low-level drivers/protocols into high level protocols?
- › E.g.; "Set a bit here" => "Activate the robotic arm"

# Motivational example: F1/10 Roboracer

- › The engine controller (aka: VESC) speaks *PWM* protocol, via Serial
- › Driving system runs using *Ackermann* control protocol, via ROS2

Different protocols, different data formats







# Hardware Proxy / Hardware Abstraction Layer

---

A structural pattern

## Purpose

- › Represent a given device with specific (C) structure and primitives, that provide access to it

## Motivation

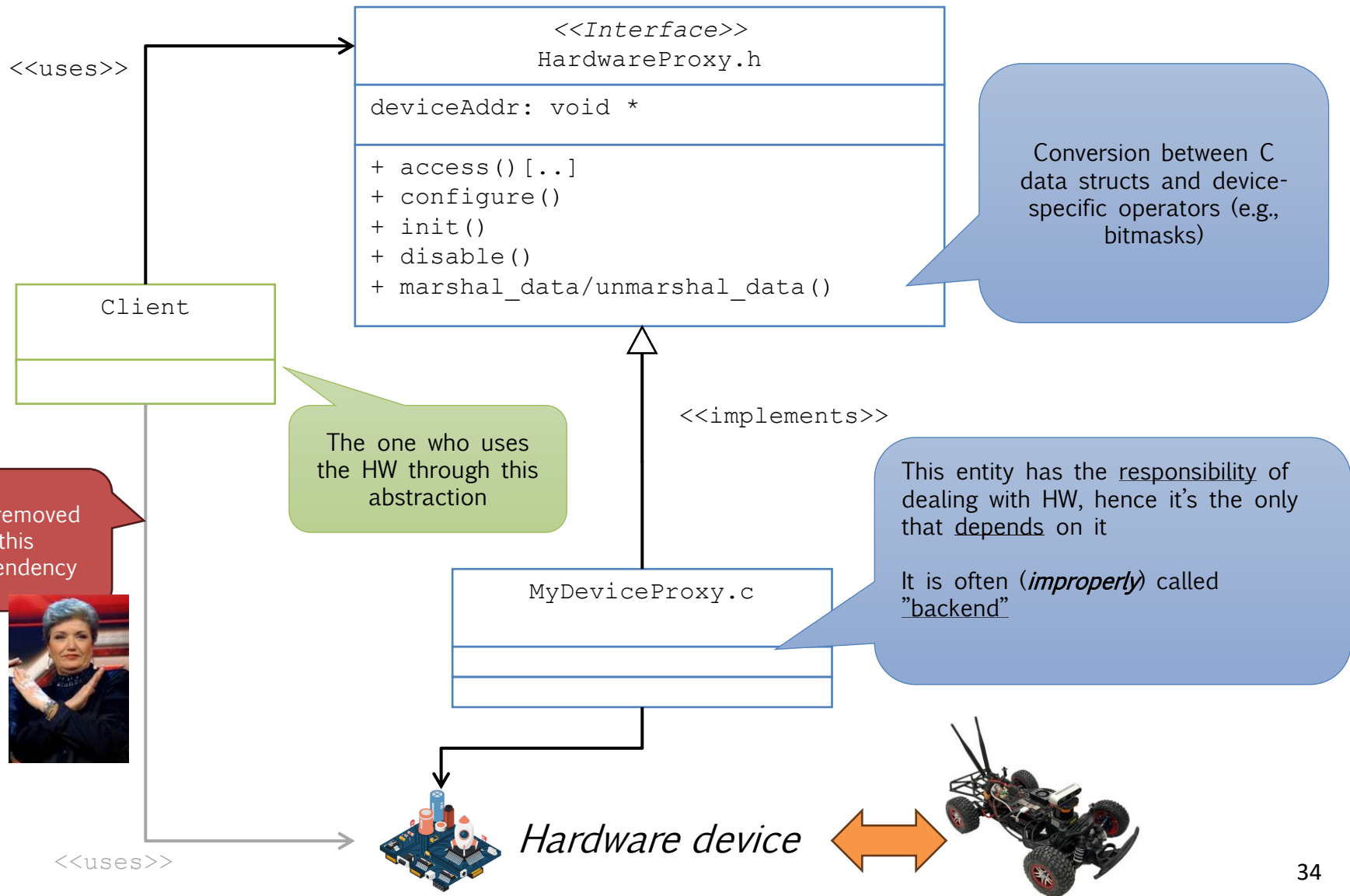
- › If we access HW directly, changes to HW might affect our code, so we wrap it in a **proxy**

## Applicability

- › Whenever you need to abstract HW which is not “standard” in the sense that there exist no standard representation for it (ex: threads are an abstraction for CPU cores)

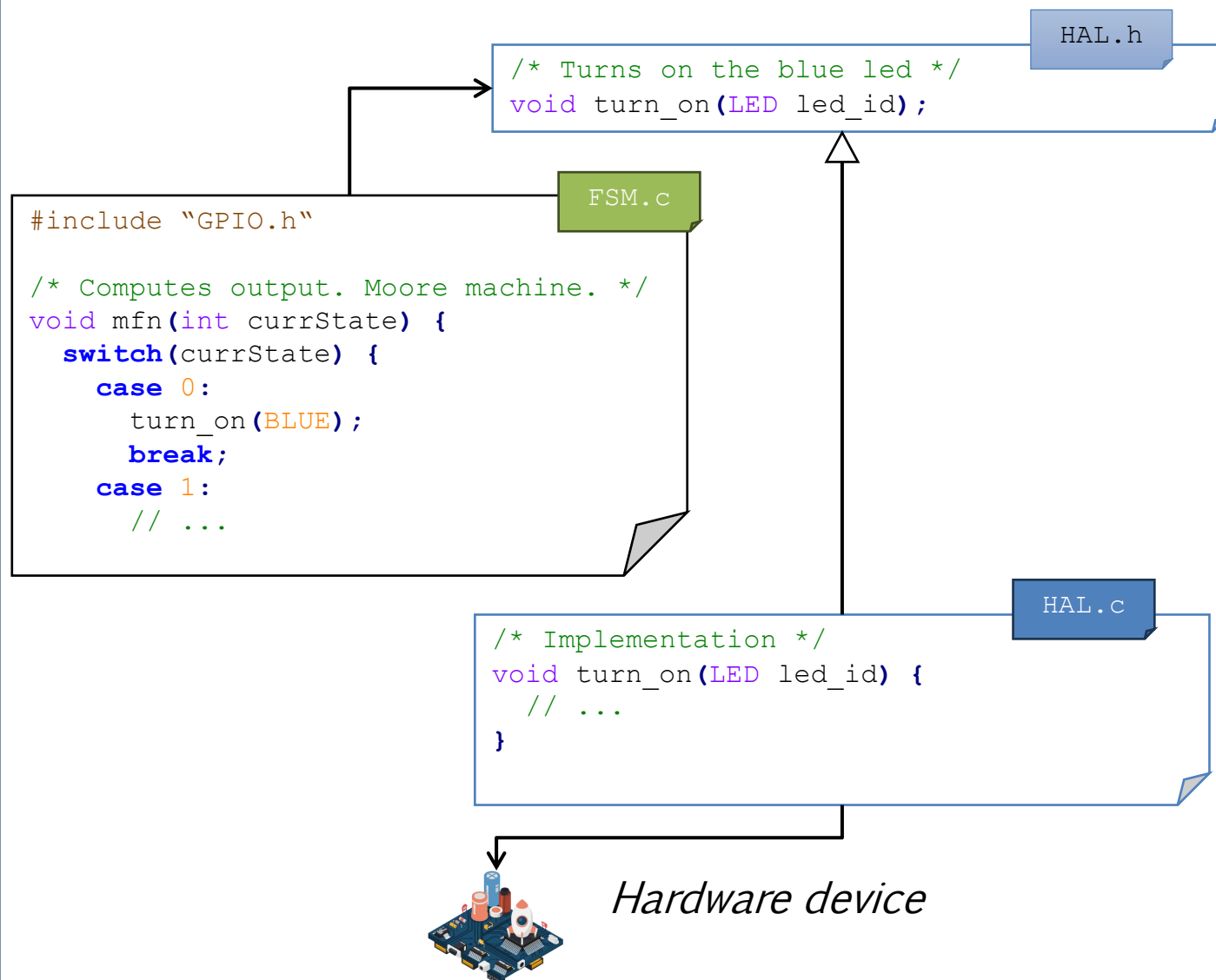


# Pattern structure





# Does this remind of something?





# Hardware Adapter pattern

---

A **structural** pattern

## Purpose

- › Adapt the specific HW interface to the format required by the application

## Motivation

- › While all HW interfaces have similar operations (see HW Proxy pattern), their data format might certainly differ!
- › Actually, it is typically used together with Proxy!

## Applicability

- › When you need to adapt application data structs to HW



# Consequences/side effects

---

Same as previously seen in Adapter, plus

- › You have to handle concurrency (with locks, critical regions...we'll see this)
- › You shall implement interrupt-base device-to-app communication (e.g., callbacks)
- › Format conversion might add delays (which, in embedded systems, are extremely unwanted!)

Notes

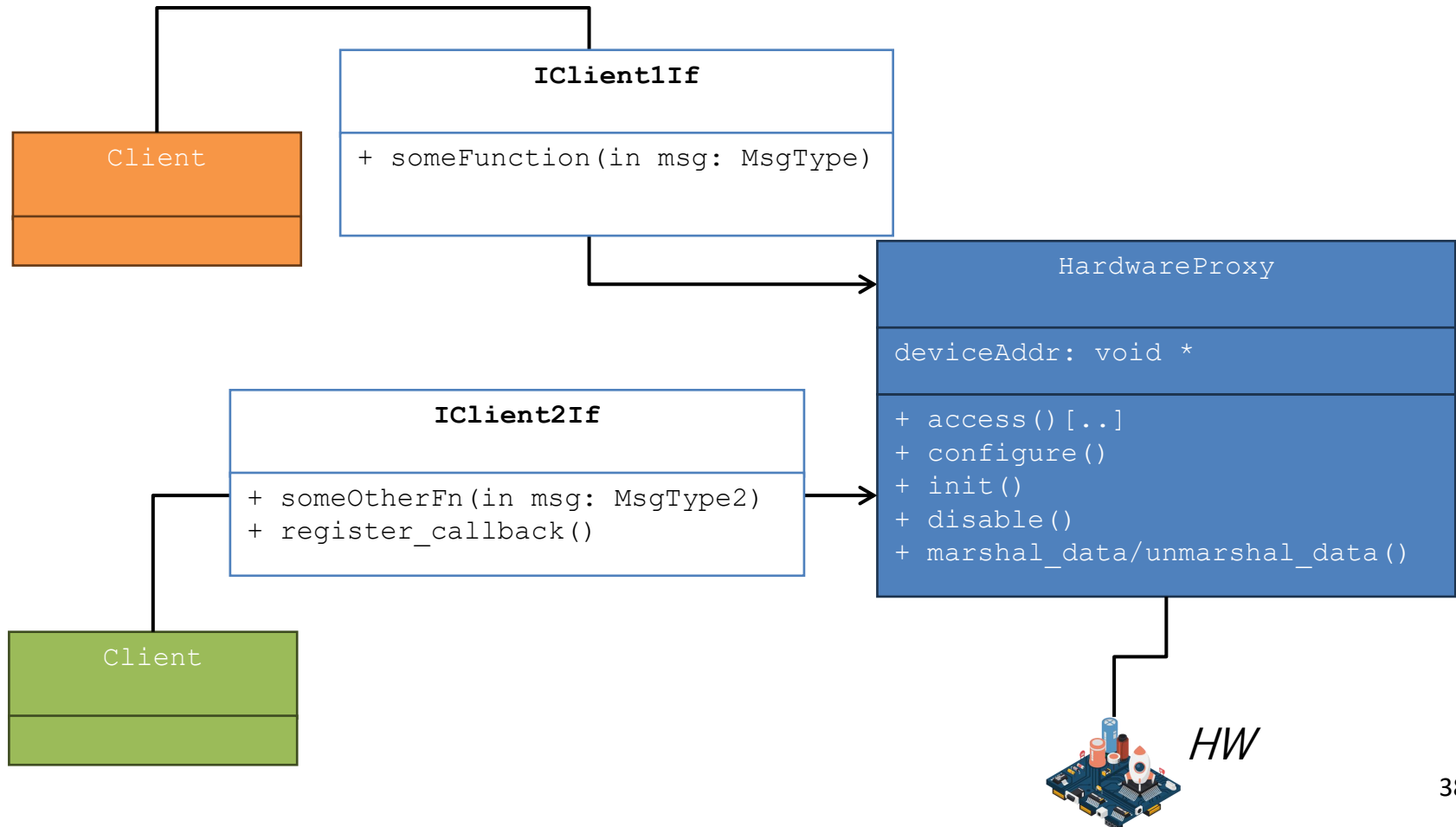
- › In C coding, headers contain contracts, hence, interfaces!



# Roles

- › Note. Here, I omit the structure of Proxy for the sake of readability

1

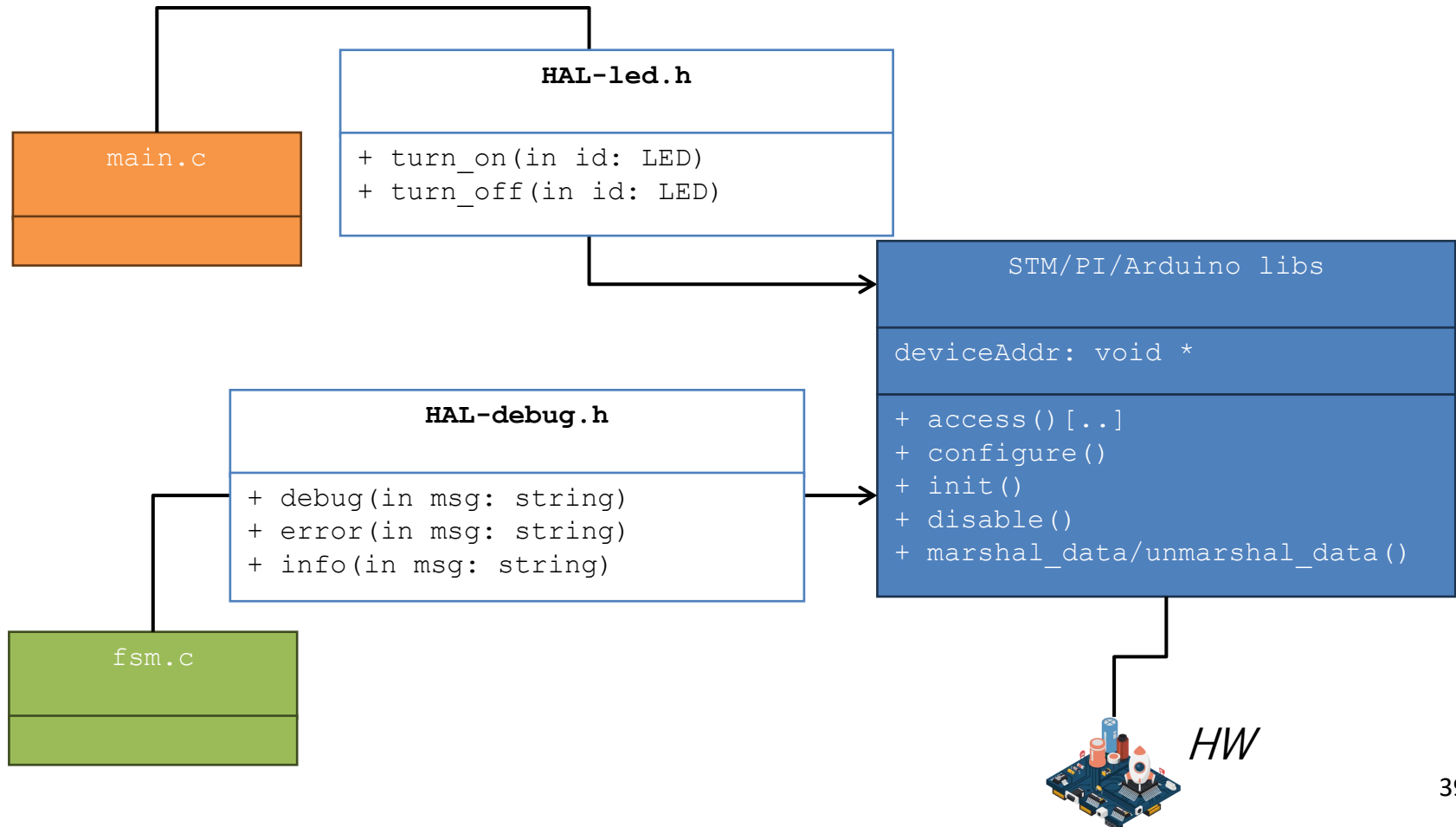




# Roles (in our example)

› Note. Here, I omit the structure of Proxy for the sake of readability

1

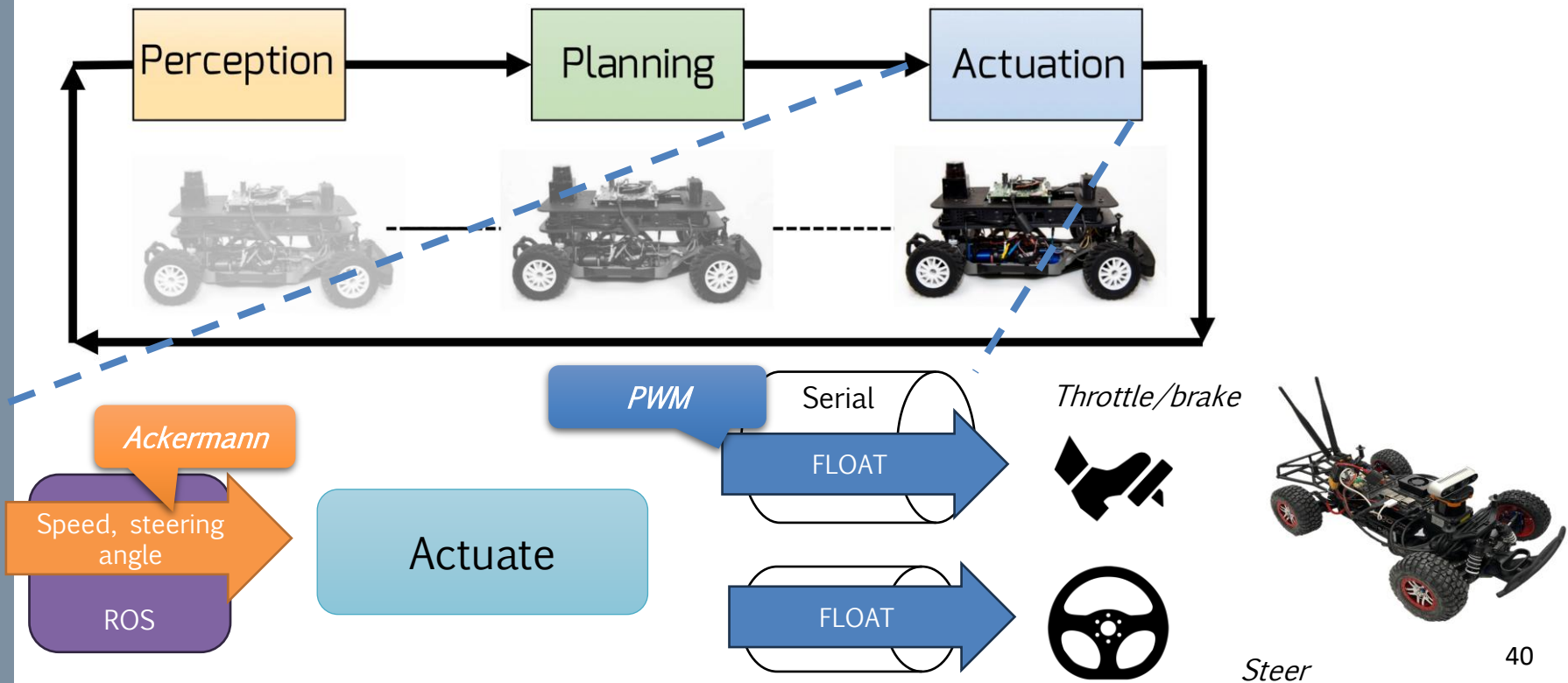




# Motivational example: F1/10

- › The engine controller (aka: VESC) speaks *PWM* protocol, via Serial
- › Driving system runs using *Ackermann* control protocol, via ROS2

Different protocols, different data formats







# Example: the F1/10

Speed, steering  
angle

ROS

Actuate

FLOAT

Note

- › Here, I implemented using C-style primitives

*This is a library*

SerialPwm.a / .so

ControlThread

<<Interface>>  
AckermannActuation.h

+ driveVehicle(in msg: AckermannMsg)

<<Interface>>  
SerialPwm.h

+ send(in pwr: float)

<<implements>>

VehicleActuation

+ driveVehicle(..)

```
#include "AckermannActuation.h"
#include "SerialPwm.h"

void driveVehicle(AckermannMsg msg)
{
    send(/* ... */);
}
```



CLEAN code architecture



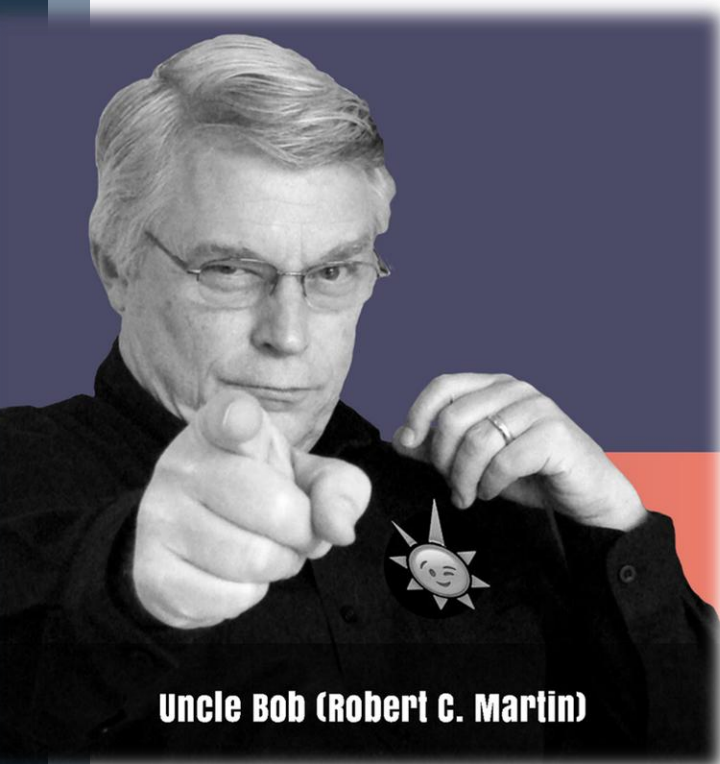
# What is it?

---

A code architectural pattern

- › A structure that enables building software that is more scalable, testable, maintainable
- › Built upon/heavily relies on good coding practices (e.g., SOLID, design patterns..)
- › Disclaimer: +15-20% dev time overhead

- › Formalized by “Uncle Bob”
- › Started his blog in 2011
- › Adopted by nearly all mid- and large-scale projects

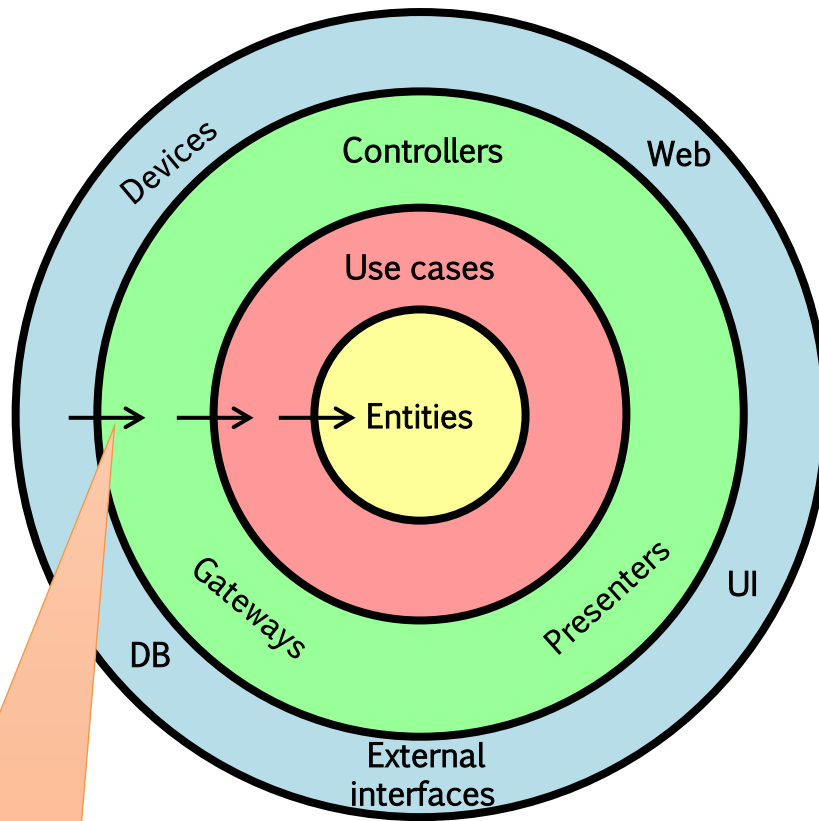


**Uncle Bob (Robert C. Martin)**



# As simple as this

› Aka: "Onion Architecture"



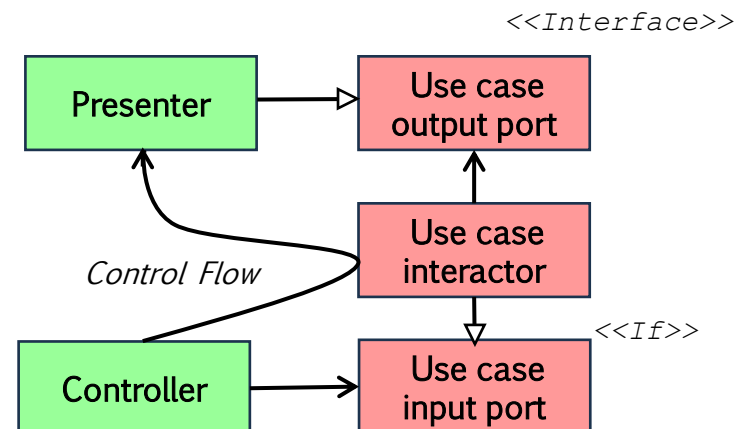
Dependencies go from  
"out" to "in"

Enterprise business rule

Application business rule

Interface Adapters

Frameworks & Drivers

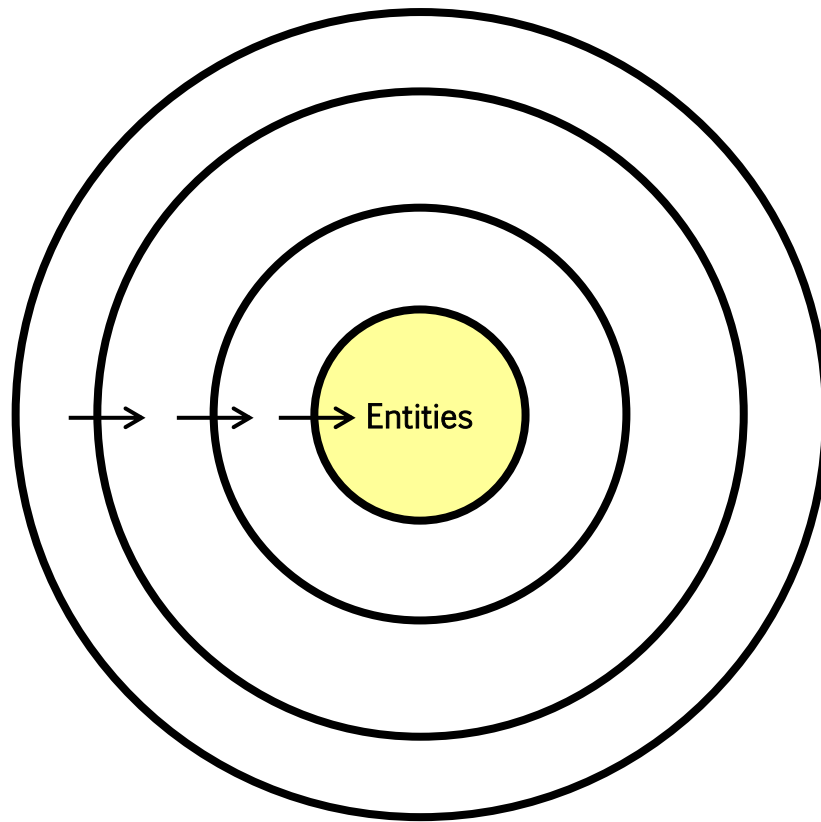




# The Model

---

- › Our view of the world: just field, and basic operations (get, set..)



Enterprise business rule

- › Everything depends on them/includes them, they do not depend on anything
- › Why is this so important?

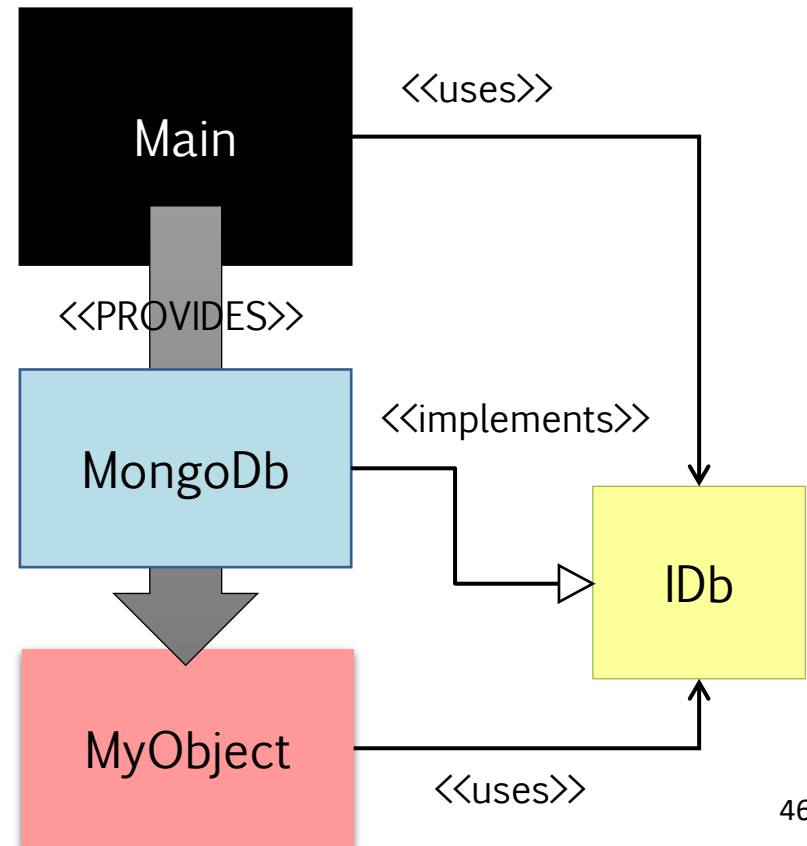
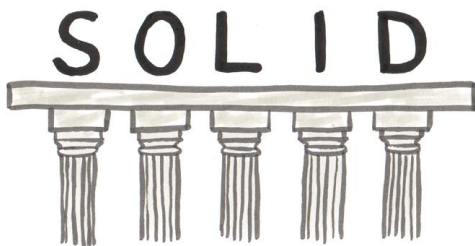


# Dependency Inversion

- › Reduce coupling
  - Avoids unnecessary dependencies that ultimately make the code hard to modify
- › Enables fast testing and debugging
- › Wraps functionalities (Interface Segregation)

(Only one issue)

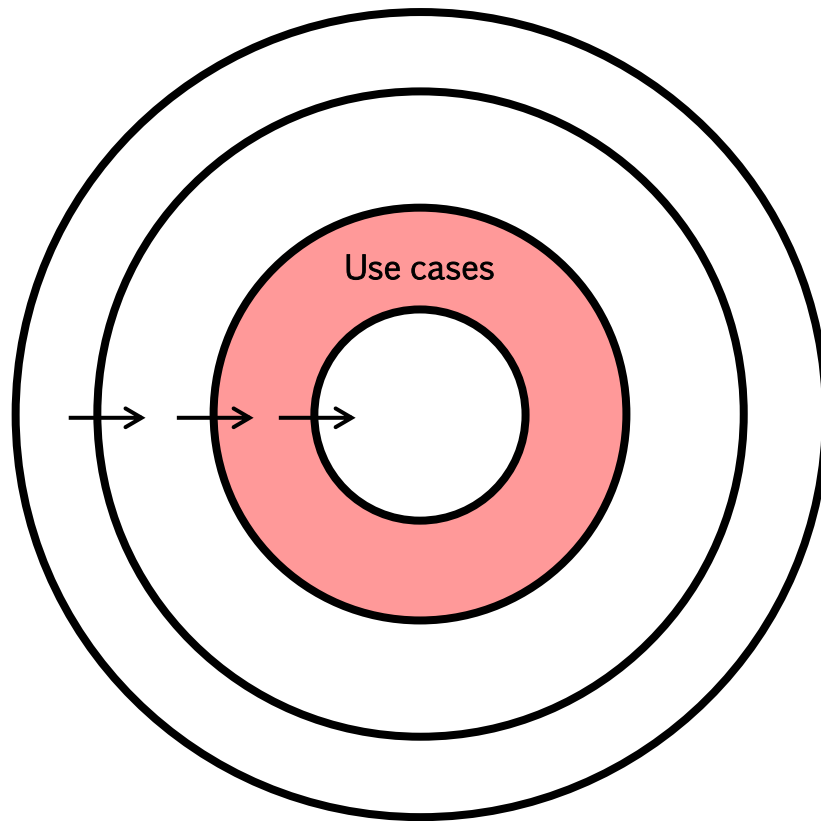
- › You need to find a (elegant) way to provide the required services
- › Dependency Injection!



# Straight from requirements

---

- › Application specific logics: functionalities



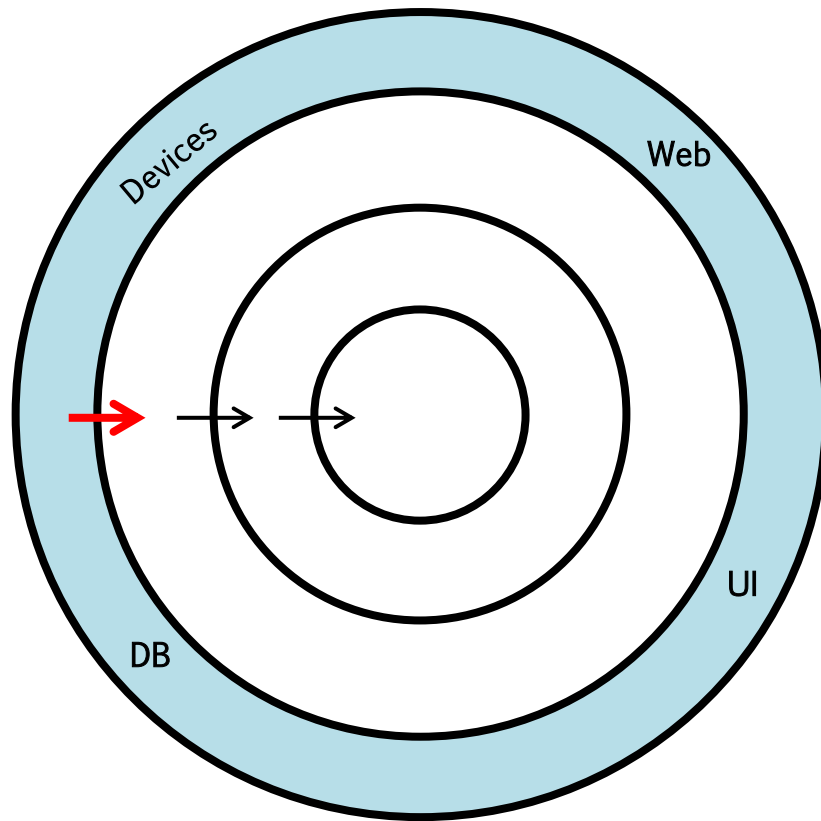
Application business rule



# "The bad world"

---

- › This layer represents, and wraps, "external" dependencies, e.g., DTOs, MongoDB...



Frameworks & Drivers

- › How do we implement the dependency?

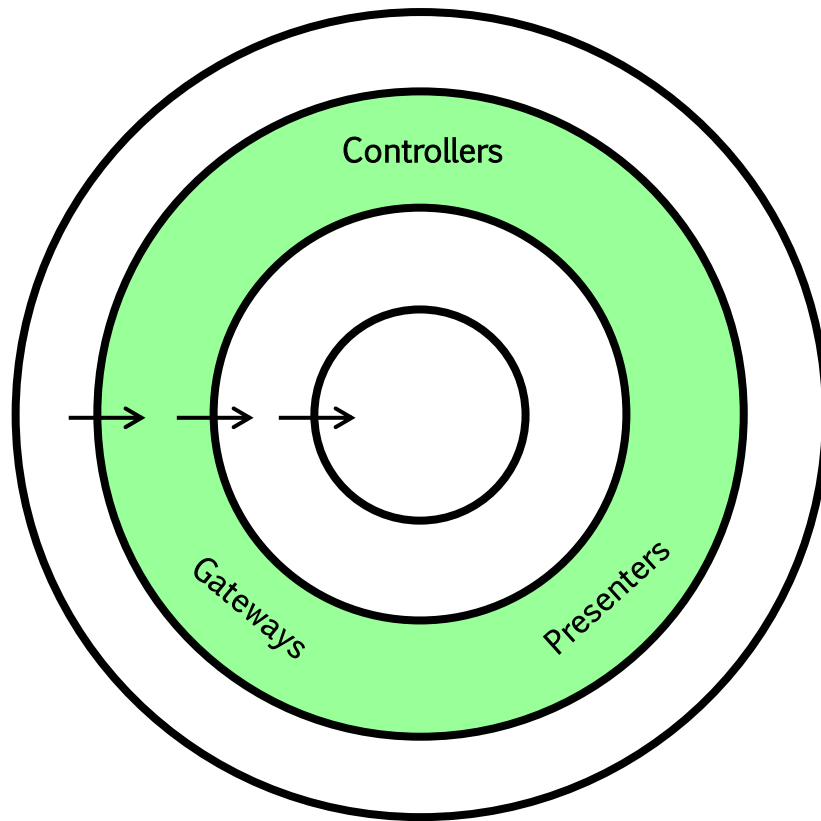




# Our good old friend

---

› Aka: "Onion Architecture"

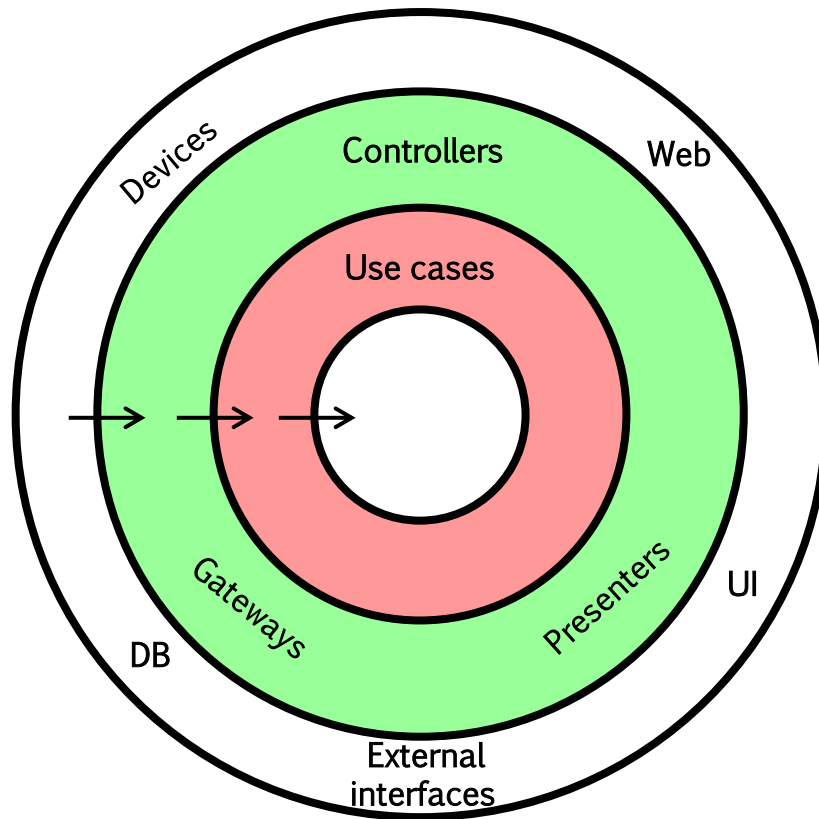


Interface Adapters



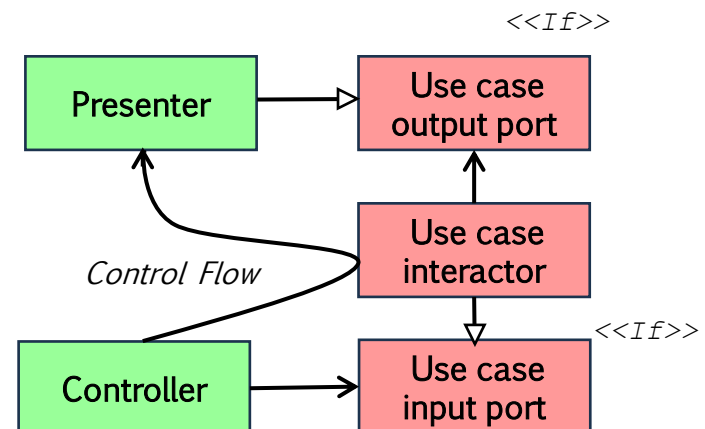
# Control flow, and class diagram

- › Note how we use Interfaces, and (consequently) Dependency Injection



Application business rule

Interface Adapters





# References

---



## Course website

- › [http://hipert.unimore.it/people/paolob/pub/Industrial\\_Informatics/index.html](http://hipert.unimore.it/people/paolob/pub/Industrial_Informatics/index.html)

## My contacts

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

## Resources

- › Gamma, et.al «Design Patterns – Elements of reusable Object Oriented Software», Addison Wesley
- › Douglass – «Design Patterns for Embedded Systems in C», Newnes
- › Fowler, Martin (1999). "Refactoring. Improving the Design of Existing Code. Addison-Wesley". ISBN 978-0-201-48567-7.
- › <https://refactoring.guru/>
- › <https://springframework.guru/solid-principles-object-oriented-programming/>
- › <https://blog.cleancoder.com/uncle-bob/2011/11/22/Clean-Architecture.html>
- › A "small blog"
  - <http://www.google.com>