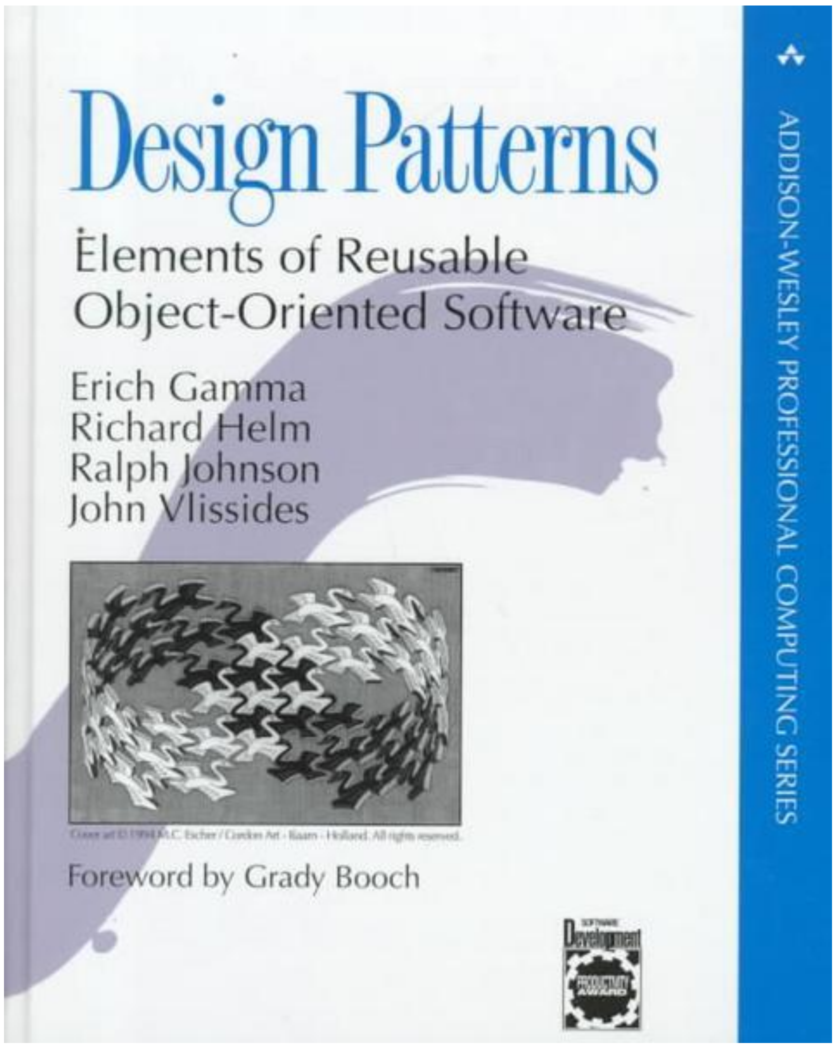
Design patterns

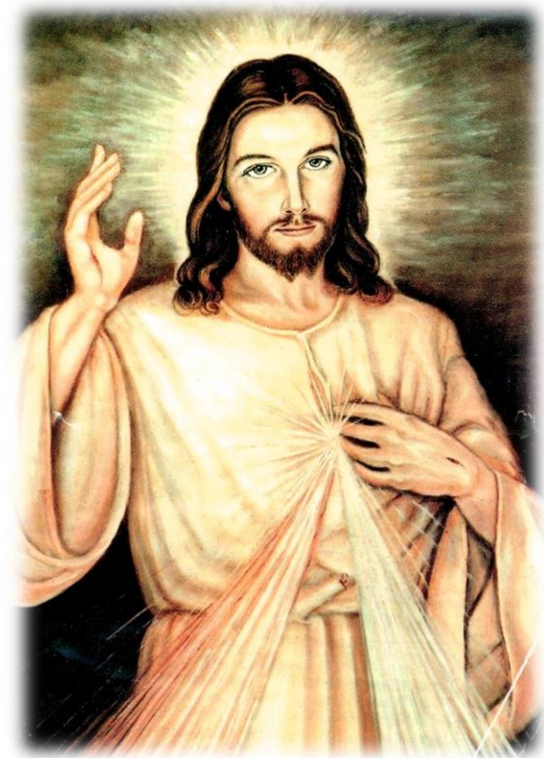
Paolo Burgio
paolo.burgio@unimore.it

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA
High Performance Real Time Lab

PROGRAMMING

70% THINKING
5% CODING
25% DEBUGGING





The Gang of Four

Elements of reusable Object Oriented Software *

Elements

of reusable

Object Oriented

Software

** Cit. Wikipedia*

Elements of reusable Object Oriented Software *

Elements

- Simple, basic parts of

of reusable

- We did mistakes, we learned from them

Object Oriented

- Years of mistakes

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 patter 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 and classes

- Defining object implementations…
- …and the relations among them (inheritance between interfaces, or between classes?)

Implement reusable code

- Better inheritance, or composition/aggregation?
- 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 classes
- You explicitly call methods, to implement an high-level operation
- You have strong dependencies on HW and SW platforms (e.g., middleware)
- Your classes depend on internals of another class
- Your code might depend on algorithms that you implement
- Tightly coupling among components/entities/classes/…
- Always use subclasses to extend functionality/specialize behavior
- (not actually a mistake) you might need to modify a "closed" class
- …

The so-called Code smells (we'll see them later)

A brief recap...

so that we can go beyond

Dependency inversion principle

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

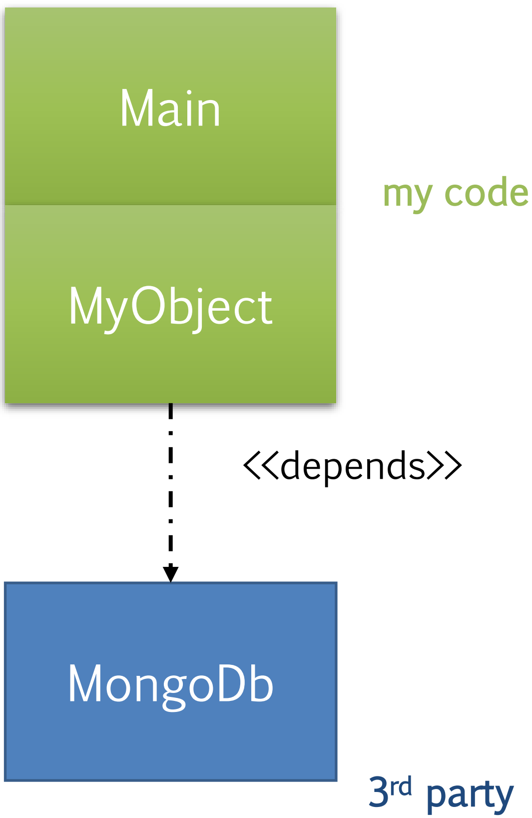
- Design wrappers around your dependencies
  - (This is **NOT** "dependency injection"...but its good friend)

- **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)

Dependency inversion principle

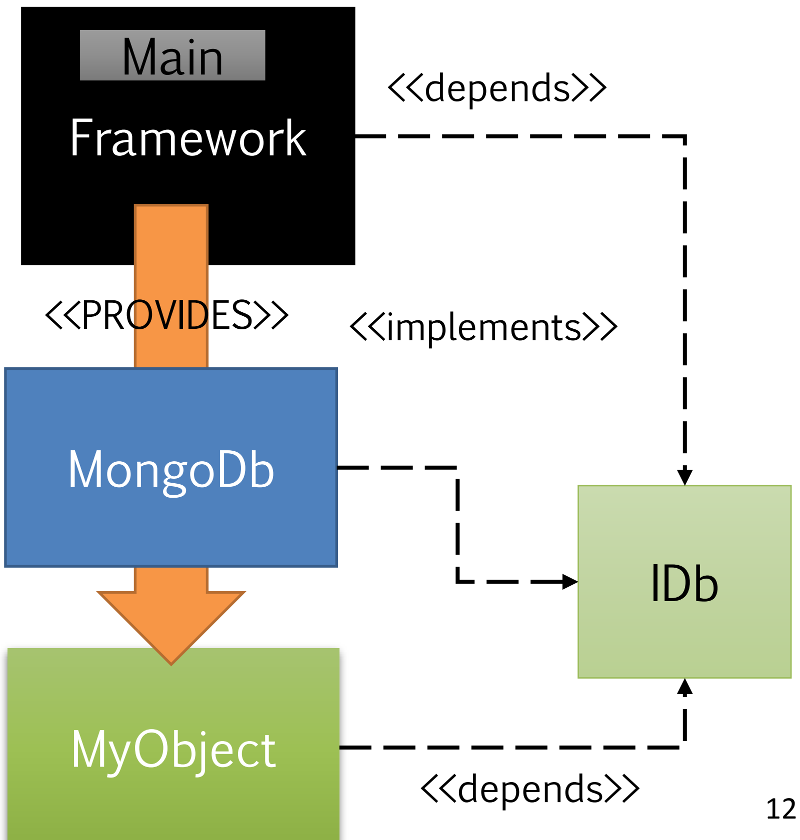
Library/Toolkit

- Tied to 3rd party code



Framework

- Inversion of control
- Dependency injection



The software journey, so far…

Aka: welcome to the real world

- (You didn't know, but) you've been designing software according to one of this schemes
- Depending on the complexity of your system, you have these tools

(To tackle) system complexity

3) Framework

2) Toolkit

1) Application

1) Application

Self-contained software artifacts

- In two words: it has its own `main()`, and few dependencies
- The only hard dependency might be on HW platform (e.g., uses Ethernet, or GPU w/CUDA), or SW platform (uses GNU/Linux vs. Win API)
- Typical of small projects (because it's **hard** to make it grow bigger)

How design patterns can help you

- Reduce dependencies among app internals
- Loosely coupling among (sub)modules increases reusability/debugging/testing
- Wrapping HW/SW platform increases portability

2) Toolkit

Self-contained application

› ..but it heavily uses runtimes and libraries

› You call them to get basic functionalities such as filesystem, I/O towards peripherals, etc..

  – Examples: stdlib, JRE

› Projects can get bigger, components are reusable (e.g., runtime libs)

How design patterns can help you

› Same as before

› Moreover, by wrapping libraries, you ensure that disrupting changes in their structure/API won't affect your code

3) Framework

A set of classes that constitutes the architecture/structure of an application

- Most of the part of application is already written...you often don't even write/own the `main()` function!
- Of course, frameworks are written for a specific application domain (e.g., Web servers)
- Heavily relies on Dependency inversion / Inversion of Control

How design patterns can help you

- They are implemented in frameworks
- If you want to interact with it...well, you'd better stick to them, to code faster

What is the difference?

- They work at higher abstraction level
- They are small architectural bricks to build bigger applications (e.g., how to build a door, a stair, etc)
- They are not specialized for an application domain

(Incomplete) taxonomy of design patterns

Creational

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

Structural

- **Adapter**
- Bridge
- Composite
- Façade
- Proxy
- Decorator
- FlyWeight

Behavioral

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

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!)

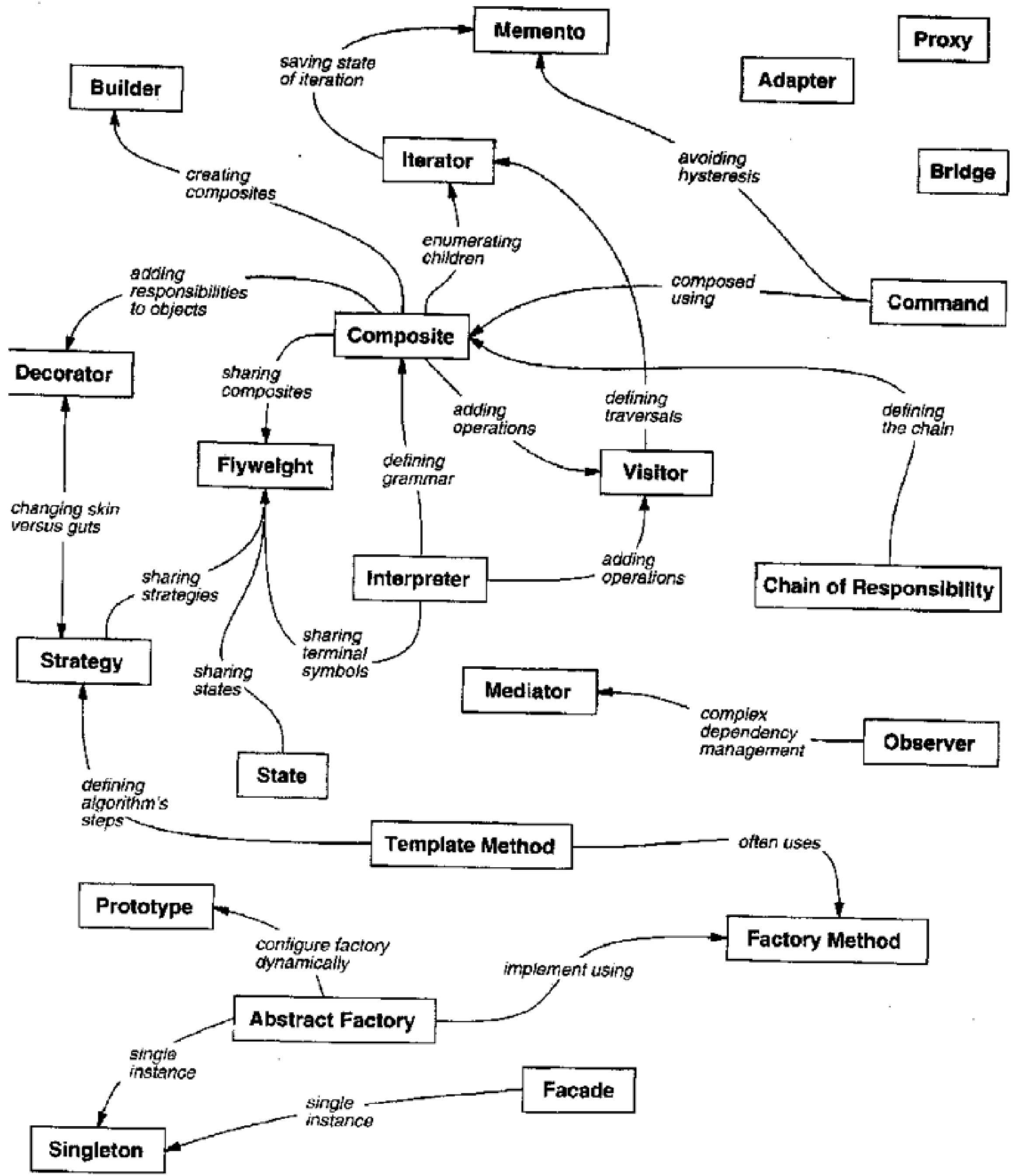
A full set of example/code snippets to implement it

- 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 3-4 four of them
- Advanced (LM?) courses can give you a full
- Coding, coding, coding

Relationships between patterns



Singleton

How to code it

Let's code!

I won't give you any practical example, I'll let you do it

- Create a simple application that creates an object, and calls a method on this object
- Just, make sure it is possible to instantiate only only one object of this class

- TODO for home
- Concurrent creatzion (w/multiple threads)
- The Object Pool patterns: instead of one, I want to create at most *N* objects

Singleton

A **creational** pattern

Purpose

- Make sure that there is only one instance (object) of a class active in the system

Motivation

- You might need to abstract single resources (e.g., printing queues, DBMS, ...)
- The class itself shall be responsible to instantiate the singleton
- No other instance (i.e., object of the same class) shall exist

Applicability

- When you need a single point of access to an instance of a class

Consequences/side effects

- You give controlled access to the single instance, which is a bottleneck in your system
  - You can handle access to its internals via queues…
  - Need to handle concurrency, via locks, mutexes, etc..
- You reduce the namespace (no global vars)
- Still, easy to specialize via subclassing
- You can extend it to provide a limited set (pool) of instances instead of one
  - Goes towards the Factory pattern
- More flexibility wrt class-wise operations and members (aka: `static`)

Factory

Factory/Factory method/Virtual constructor

A **creational** pattern

Purpose

- Defines an interface for the creation of an object, leaving to subclasses the choice of which class to instantiate (basically, it forbids you using the constructor anywhere in code)

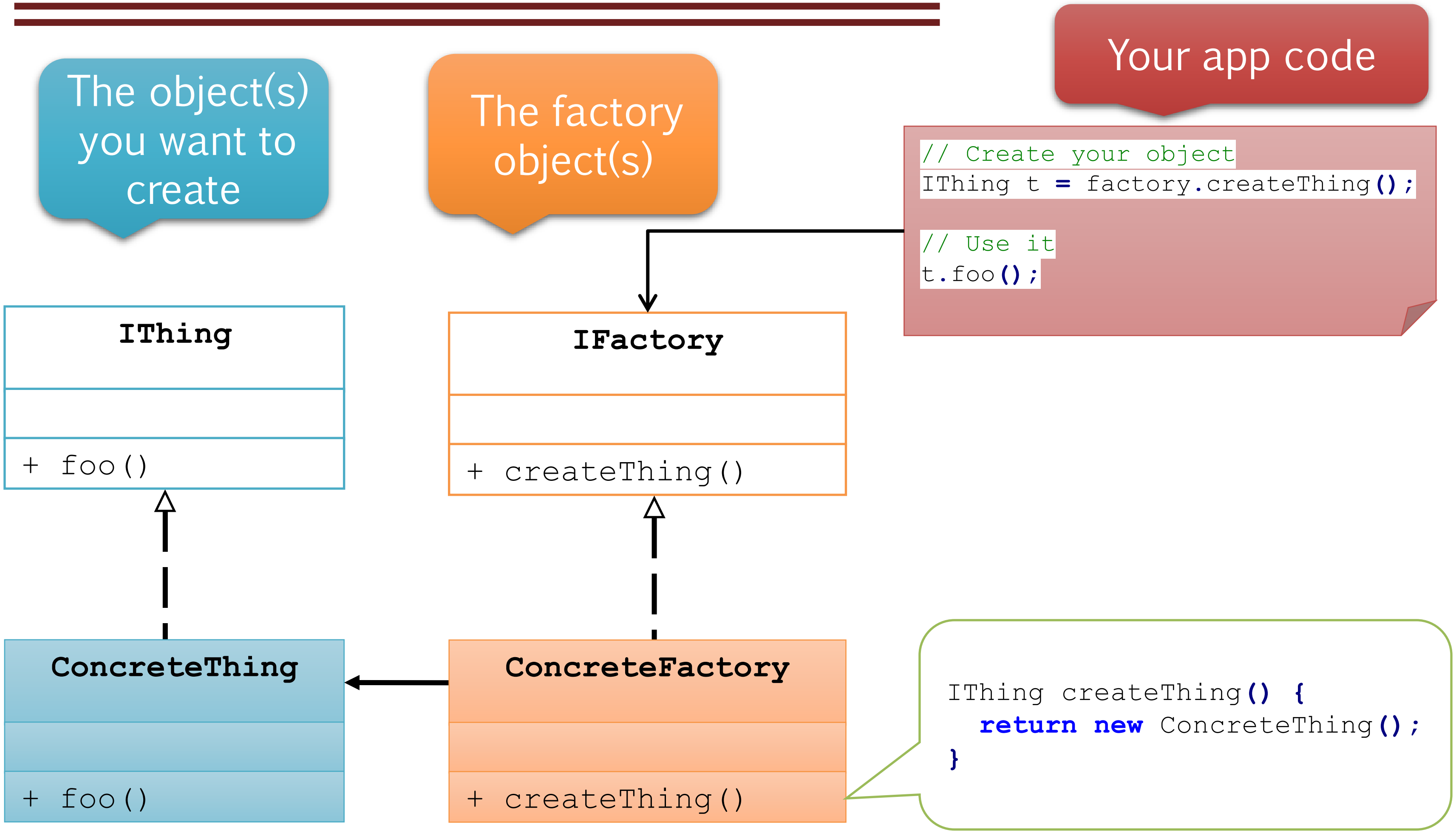
Motivation

- Most of modern frameworks rely on abstract classes/interfaces, and maintain the relations among them

Applicability

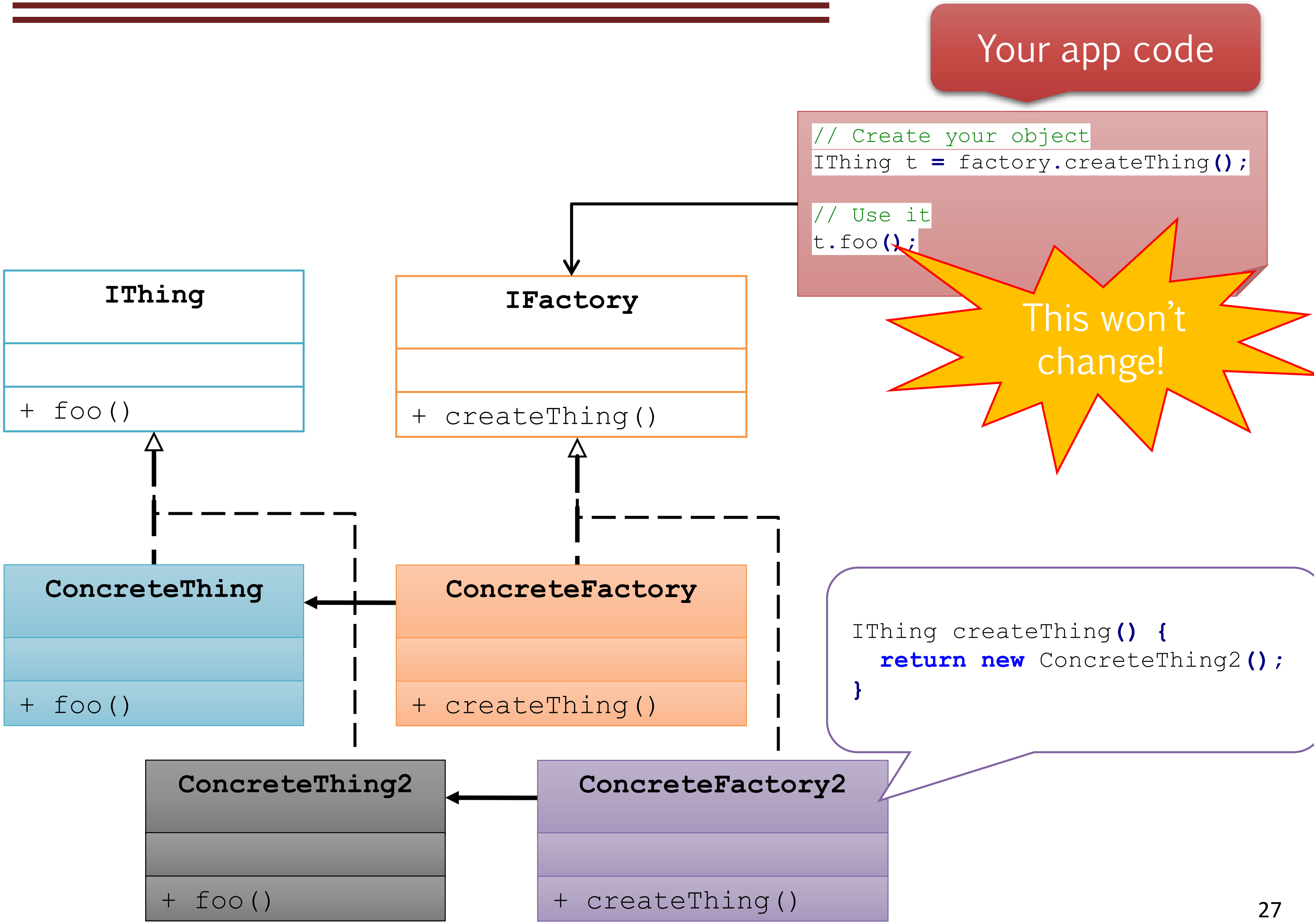
- When you don't (want to) know which actual class you shall instantiate
- You can choose among multiple objects that implement the same contract (interface)
- Single point of access for delegates

Class diagram



IMPORTANT: The entities that you see here are the <u>actors</u> of the pattern, often named <u>roles</u>

How to extend this



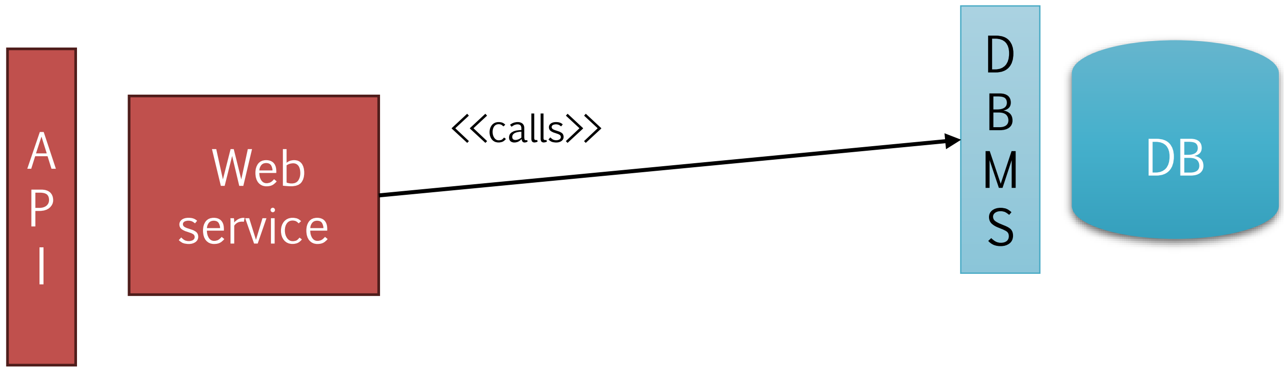
Consequences/side effects

- You **decouple** the created object from the utilizer (it *uses* `IThing`, you *provide* `ConcreteThing`)
- ...and the choice of which class to instantiate is **responsibility** of another class, which is itself segregated behind an interface (e.g., `IFactory`) to enable scalability
- You can easily specialize/alter functionalities (`ConcreteThing2`), with minimal modifications to code
- You can provide parallel/alternate implementations of the same functionality

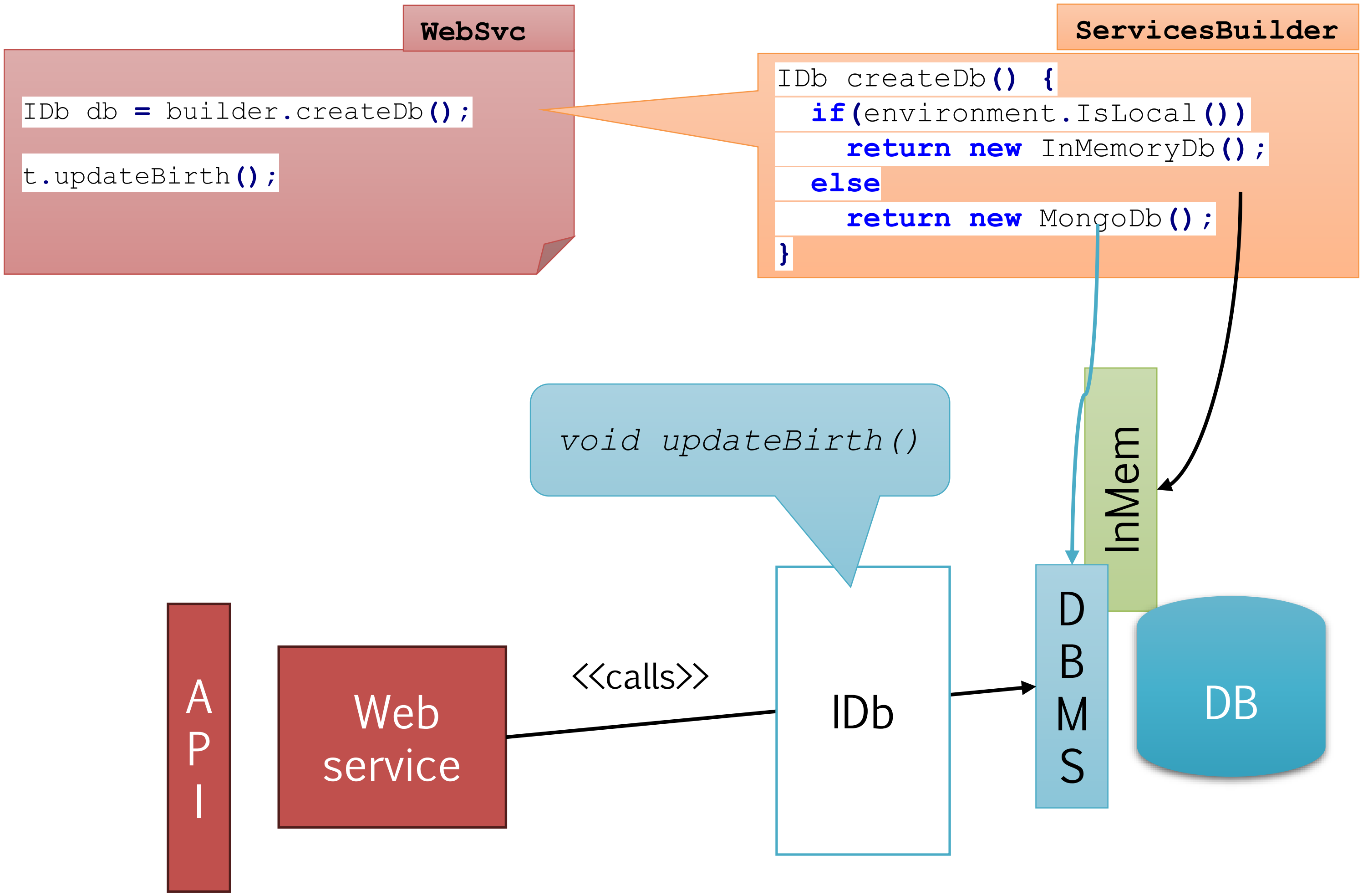
Practical example: Unit & Integration testing

Problem: I need to implement & test an application that has a DB

- **I don't want to run a DB every time I am testing!**
- In Unit tests, I only want to test a single functionality (e.g., check for age)
- In Integration tests, I don't need a persistent storage!

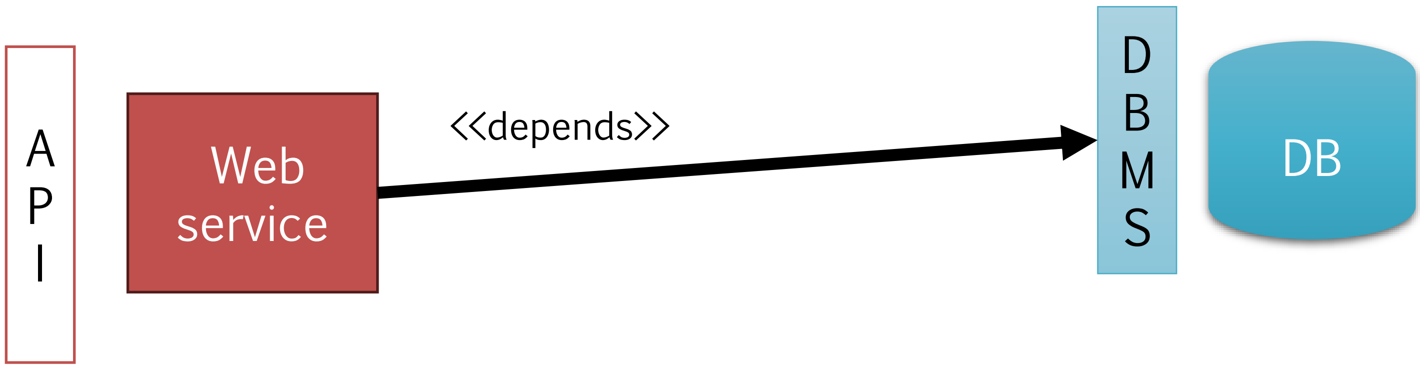


Integration tests: InMem DB vs. real storage

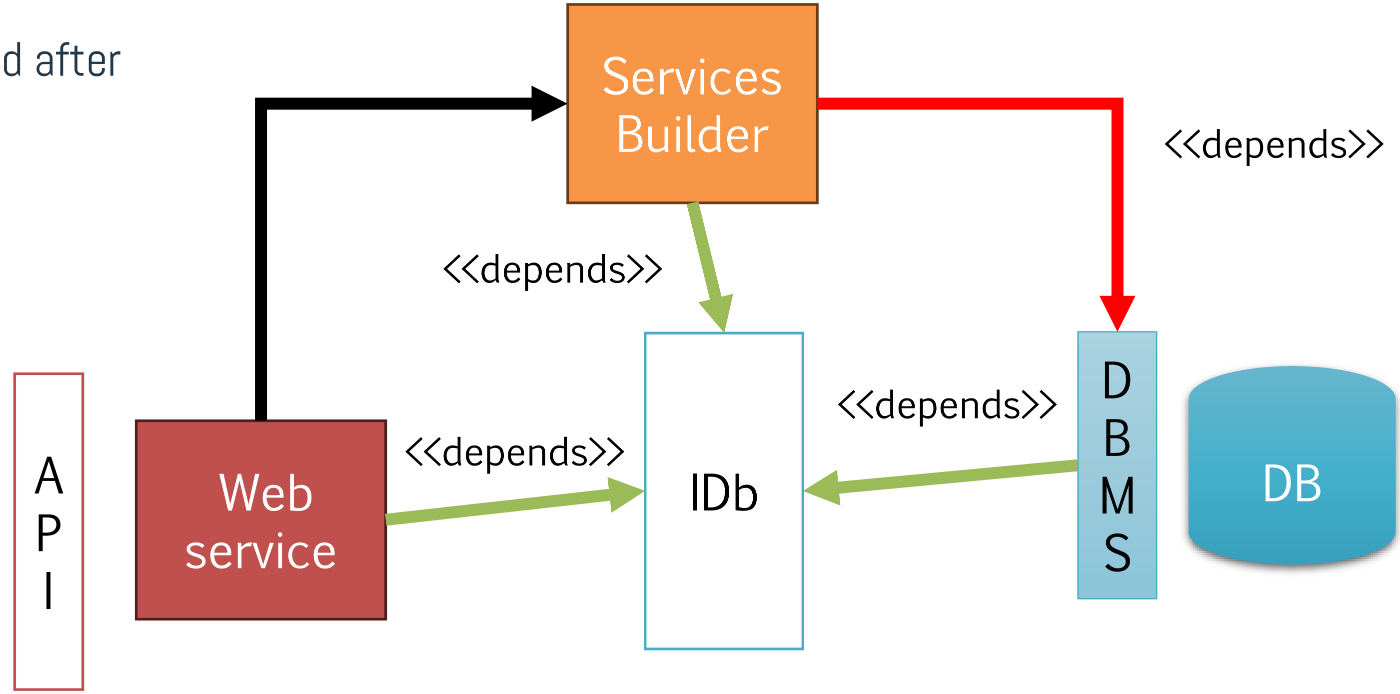


Interface segregation & Dependency inversion

- Before…



- ..and after



The set-up/build responsibility

ServicesBuilder

```
IDb createDb() {
  if(environment.IsLocal())
    return new InMemoryDb();
  else
    return new MongoDb();
}
```

We cannot completely remove the dependency towards MongoDb (of course!)

- Someone shall know about which class to create!
- In this case, `ServicesBuilder` class has the <u>responsibility</u> of setting up the application services
- This is a typical pattern for set-up / bootstrap in highly-scalable systems

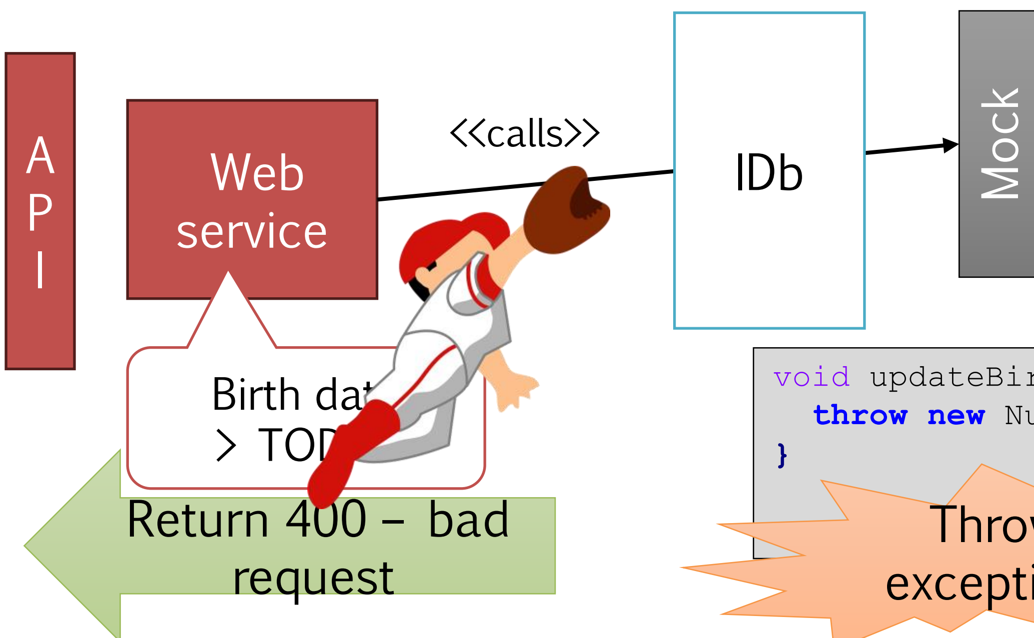
Another example: mocking objects

WebSvc

```
IDb db = builder.createDb();

t.updateBirth();
```

ServicesBuilder

```
IDb createDb() {

  if(environment.IsUnitTest18734())
    return new MockThatThrowsException();
  else if(environment.IsLocal())
    return new InMemoryDb();
  else
    return new MongoDb();
}
```



MockThatThrowsException

```
void updateBirth(int key, int age) {
  throw new NullvalueException();
}
```

Throw exception

Another example: mocking objects

WebSvc

```
IDb db = builder.createDb();

t.updateBirth();
```

API

Web service

ServicesBuilder

```
IDb createDb() {

  if(environment.IsUnitTest18734())
    return new MockThatThrowsException();
  else if(environment.IsLocal())
    return new InMemoryDb();
  else
    return new MongoDb();
}
```

Hello there *, I am the code smell watchdog

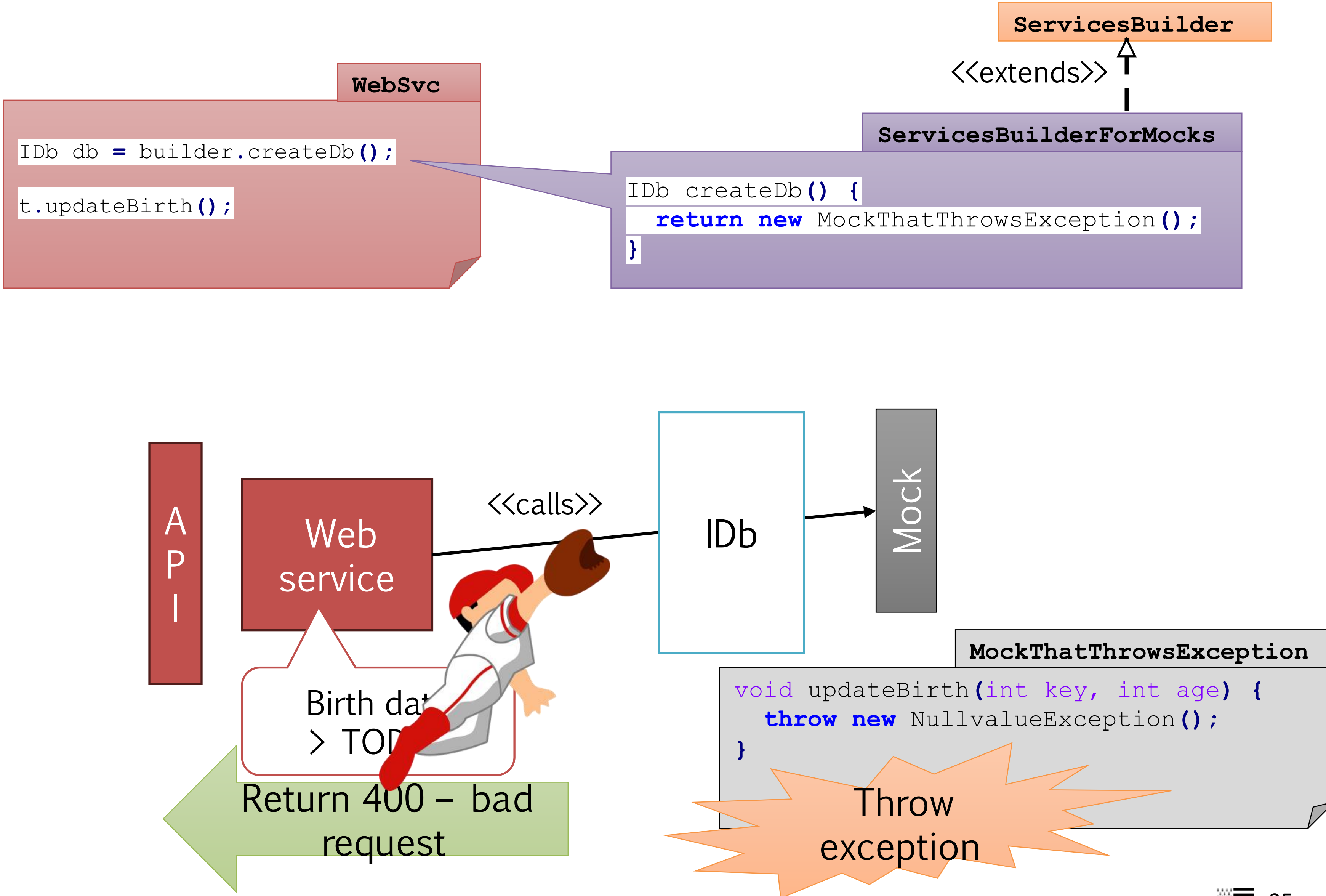
...and your class smells a lot, because it has too many `if-else`

Moreover, are you sure you want to hardcode a unit-test feature in a builder, that goes in production?

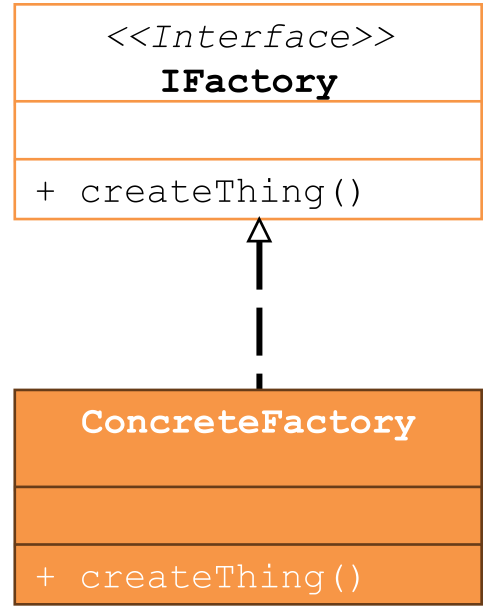
exception

* General Kenobi

Mocking objects: you're doing it well

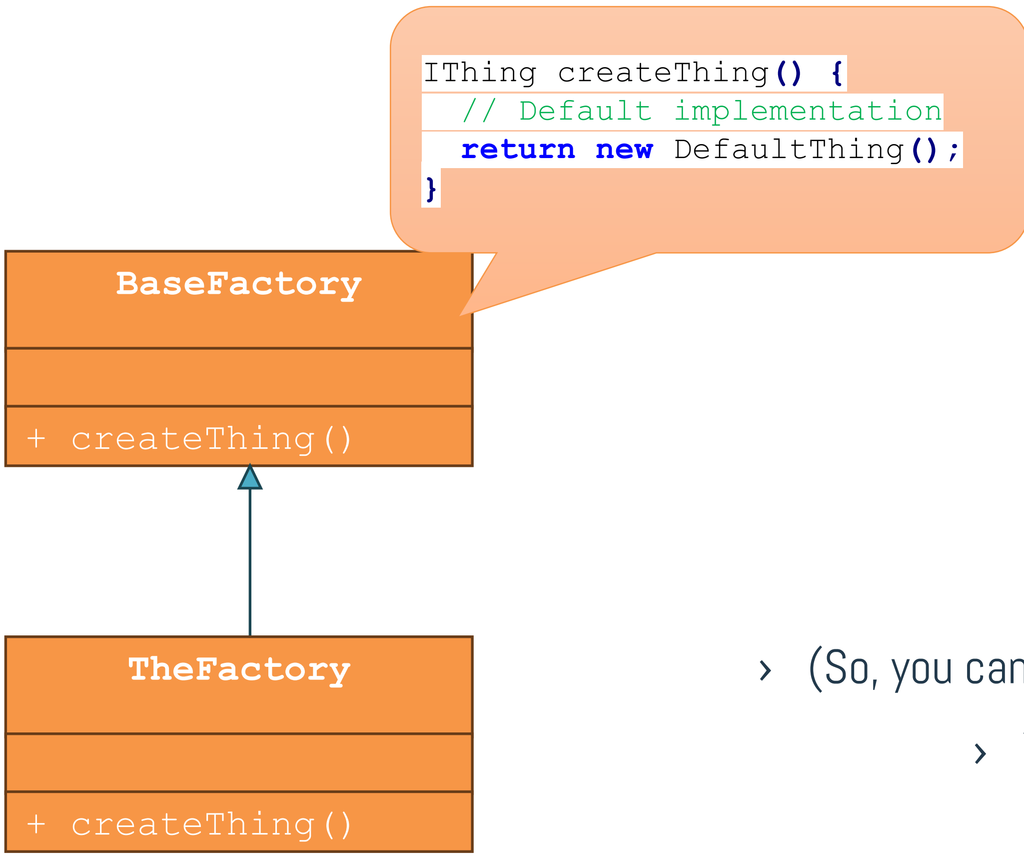


Variants: abstract vs. concrete class



A completely/partly abstract class...

- (i.e., Interface)



...or a concrete class?

- (So, you can provide, ex: default implementation)
  - You don't need to add a class for this

Variants: parametrical methods

Practical problem: how do I provide `Environment` to the `Builder`?!

- We don't like global vars!!!
- Add it as a parameter

ServicesBuilder

```
IDb createDb() {
  if(environment.IsLocal())
    return new InMemoryDb();
  else
    return new MongoDb();
}
```

ServicesBuilder

```
IDb createDb(IEnv environment) {
  if(environment.IsLocal())
    return new InMemoryDb();
  else
    return new MongoDb();
}
```

ServicesBuilder
+ createDb(in environment)

Variants: use templates/generics

- When the problem is simple, avoid creating subclasses
- Here, we use template 'T' to specify the default type
- Optionally, "hide" actual DB implementation using concrete subclasses

WebSvc

```
IDb db = builder<DefaultDb>.createDb();

t.updateBirth();
```

ServicesBuilder

```
// Note: 'T' shall be declared to implement
// IDb, otherwise this doesn't compile
public class ServiceBuilder<T> {
  IDb createDb() {
  if(environment.IsProduction())
    return new MongoDb();
  else
    return new T();
  }
}
```

WebSvc

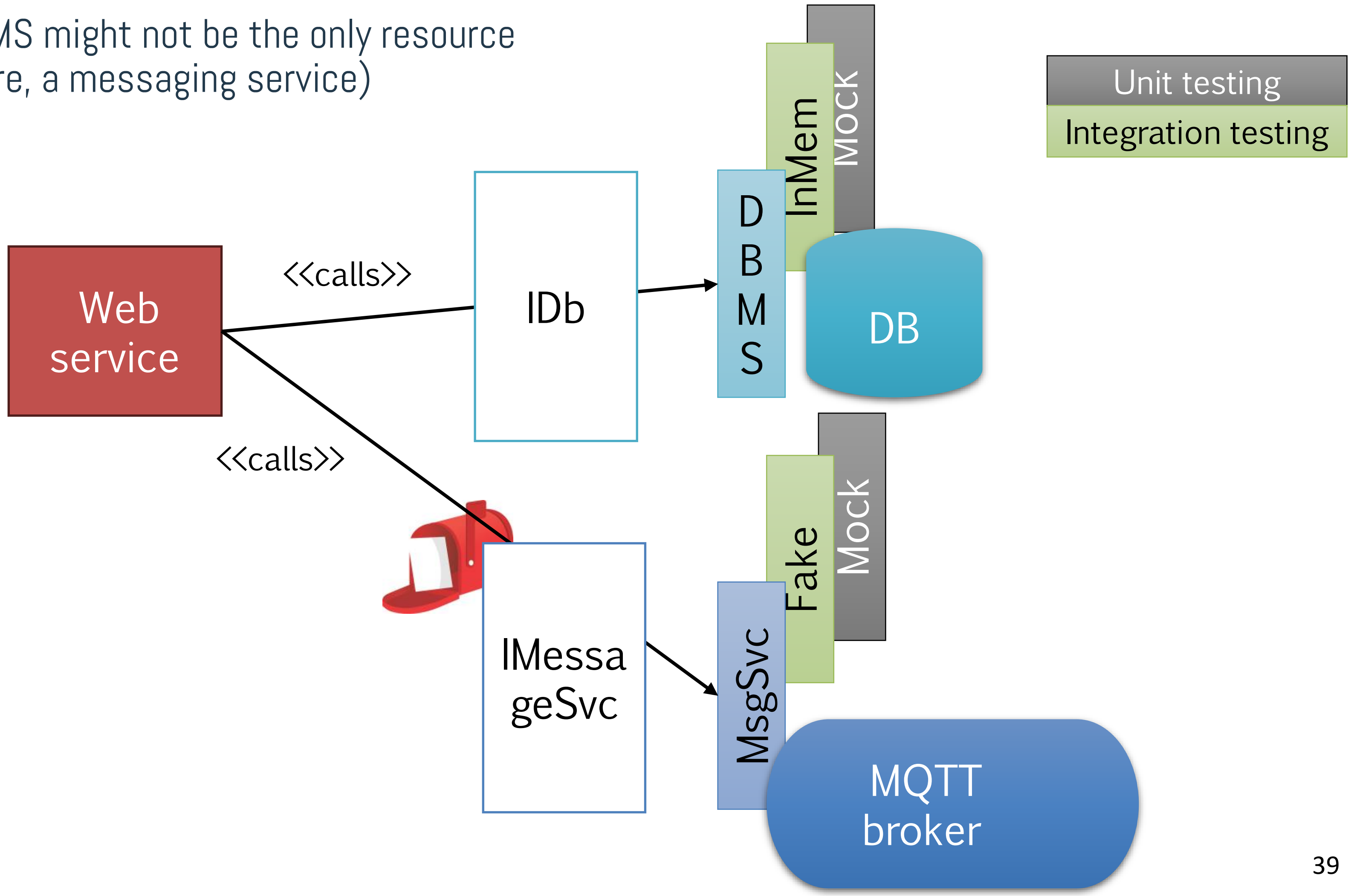
```
IDb db = builderWithDefault.createDb();

t.updateBirth();
```

ServicesBuilderWithDefault

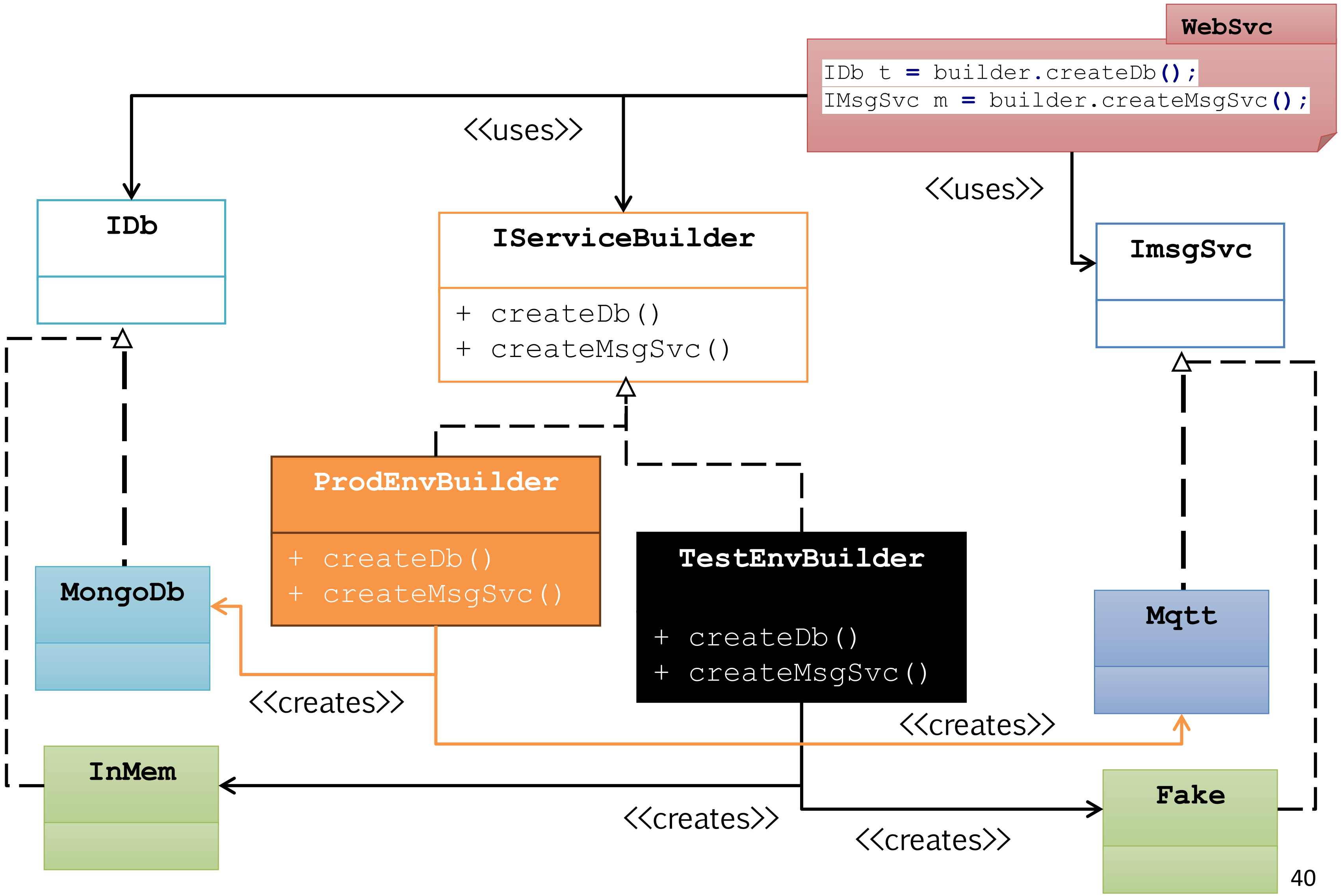
```
// Another option, to hide the "DefaultDb"
// class choice
public class ServiceBuilderWithDefault
       extends ServiceBuilder<DefaultDb> {
}
```

...local environment?

- We could **group** classes for local environment, for DEV environment, etc
- DBMS might not be the only resource
(here, a messaging service)

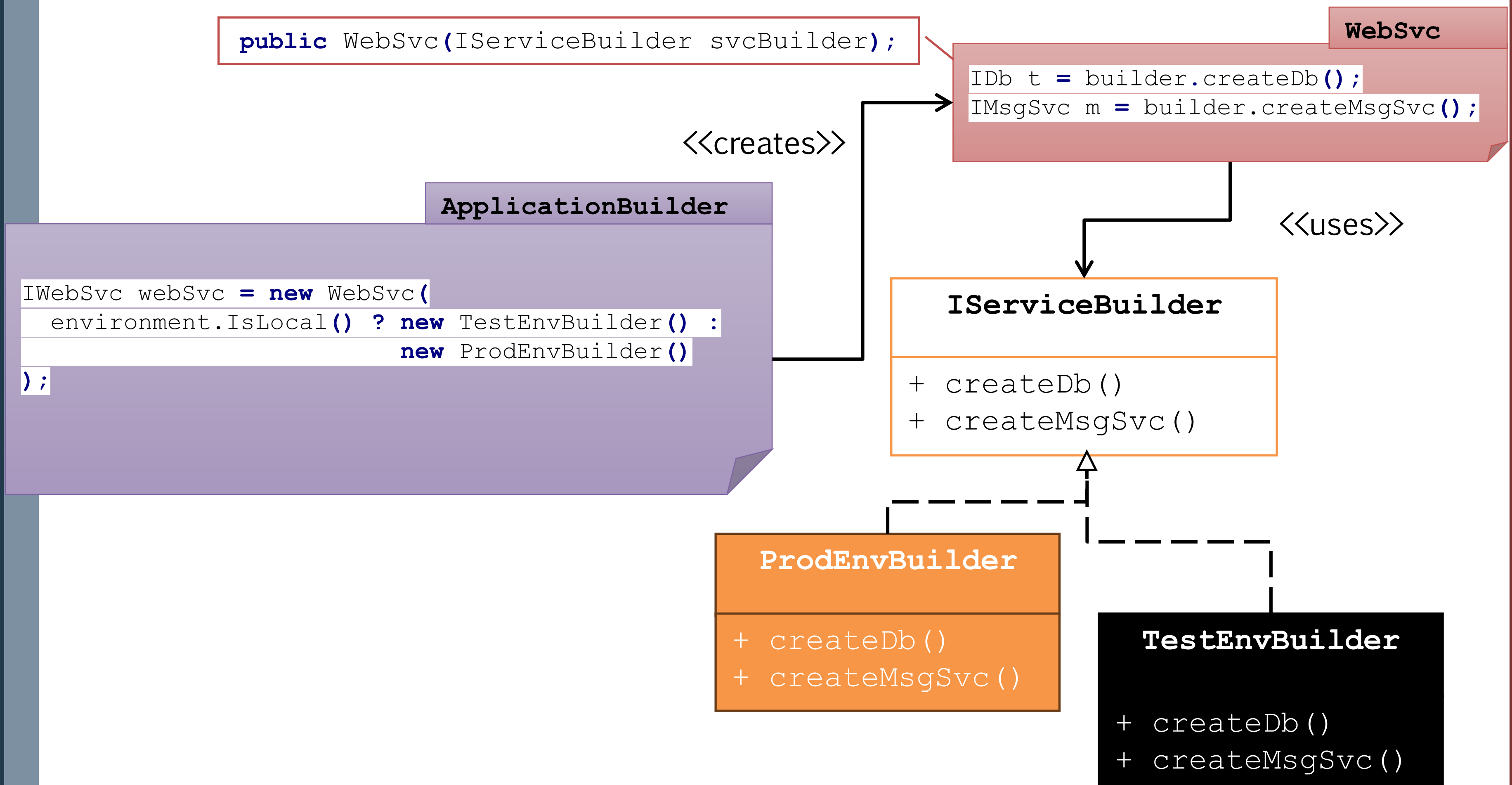


Environment-specific builders



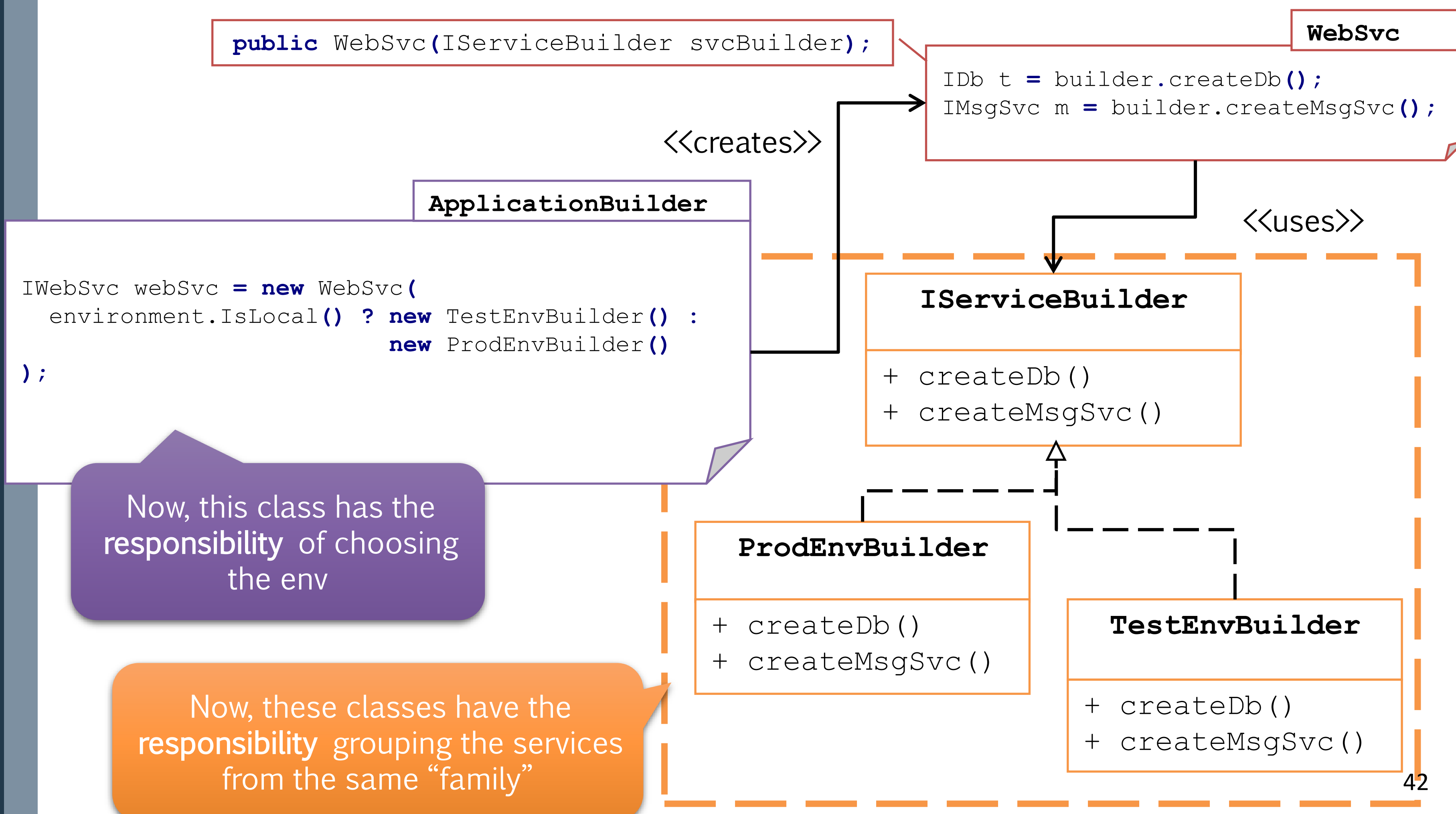
Dependency injection

- A consequence/nice side effect of Dependency inversion / Inversion of Control



Dependency injection

› A consequence/nice side effect of Dependency inversion / Inversion of Control



Growing up: Abstract Factory (aka: Kit)

A **creational** pattern

Purpose

- Defines an interface for the creation of objects that are correlated among them, without specifying the actual classes

Motivation

- Classes for which we provide multiple variants/overridings, are often related among them

Applicability

- There are multiple “families” of objects/services that a system shall use
- Objects belonging to the same “family” are related among them (e.g., depending on the environment)
- The system shall be independent on actual implementation of its services

Consequences/side effects

- Same as Factory Method
- But you can quickly change the “family” of services you are using
- You typically shall use all classes from one “family” at the same time

Warning!

- Adding new classes implies modifying the factory Interface, hence, all factories/classes that implement that interface!

Notes

- Typically, every factory is a Singleton
- Not only related to OOP! See the example of runtime libraries

Adapter
(and variants)

Adapter

A **structural** pattern

Purpose

- Convert the interface of a class into another interface, as requested by the client (i.e., the object who uses it)

Motivation

- Eventually, you might be able to use a given interface (e.g., from a library) because the client application cannot use it

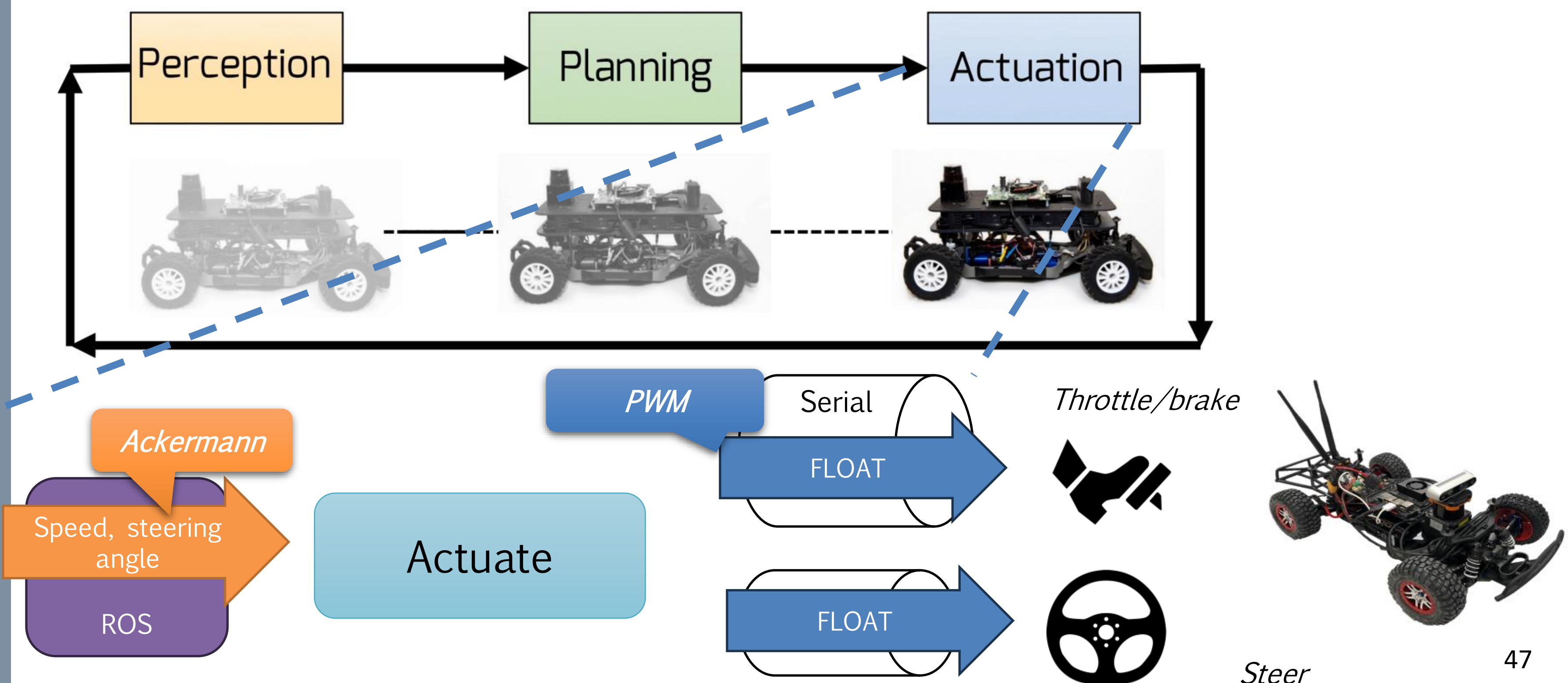
Applicability

- Whenever you have “compatibility” issue between two objects, because the client uses an interface that the source object does not declare it

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



Motivational example: F1/10

AckermannMsg.java

```java
public class AckermannMsg {
  float steering_angle           // virtual angle (rad)
  float steering_angle_velocity  // rate of change (rad/s)

  float speed                    // forward speed (m/s)
  float acceleration             // acceleration (m/s^2)
  float jerk                     // jerk (m/s^3)
 }
```

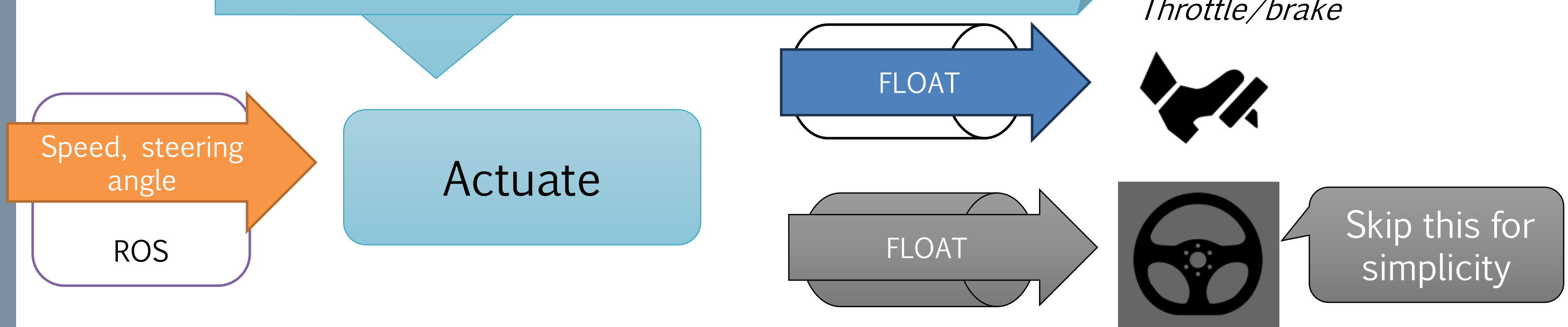
IRosReceiver.java

```java
public interface IRosReceiver {
  // let’s skip this, ok? }
```

ISerialPwm.java

```java
public interface ISerialPwm {
  /* Param pwr. A float number to
   * express the % of engine power
   */
  public void send(float pwr);
}
```

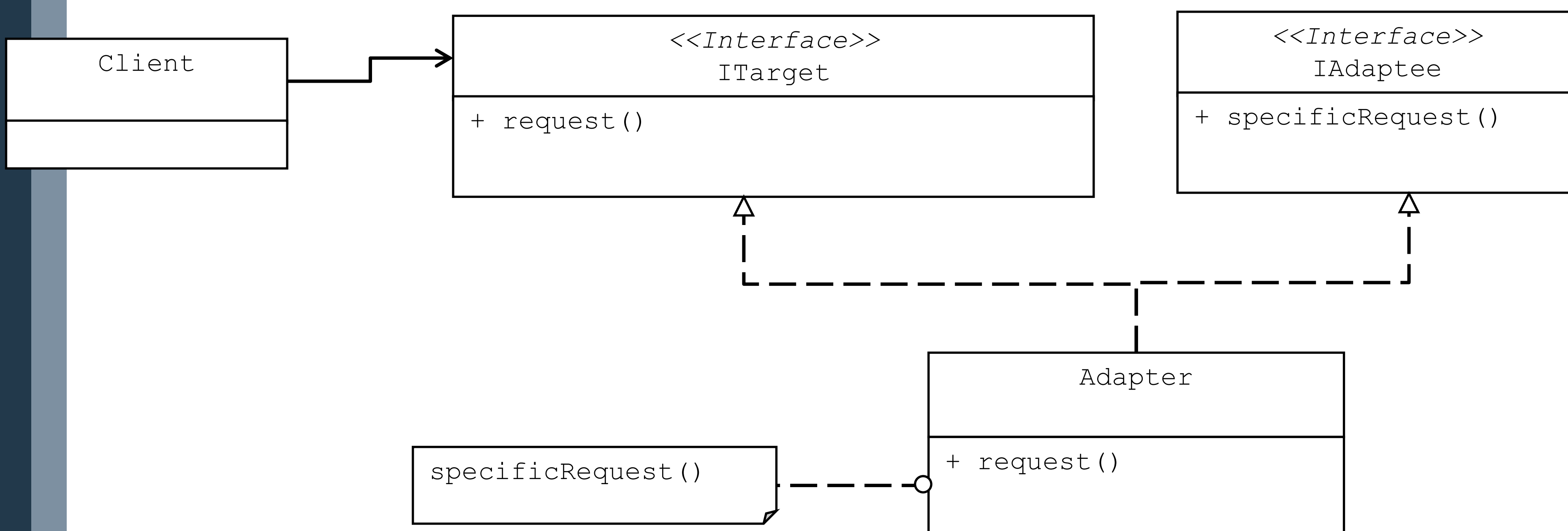
```java
public class VehicleActuation {

  public void driveVehicle(AckermannMsg msg)  {
     /*  ...? */
  }
}
```



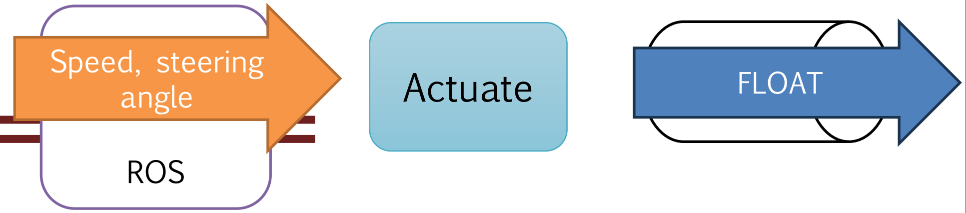
Basic structure: class-based

Overloads interfaces/abstract classes

› Here, interfaces

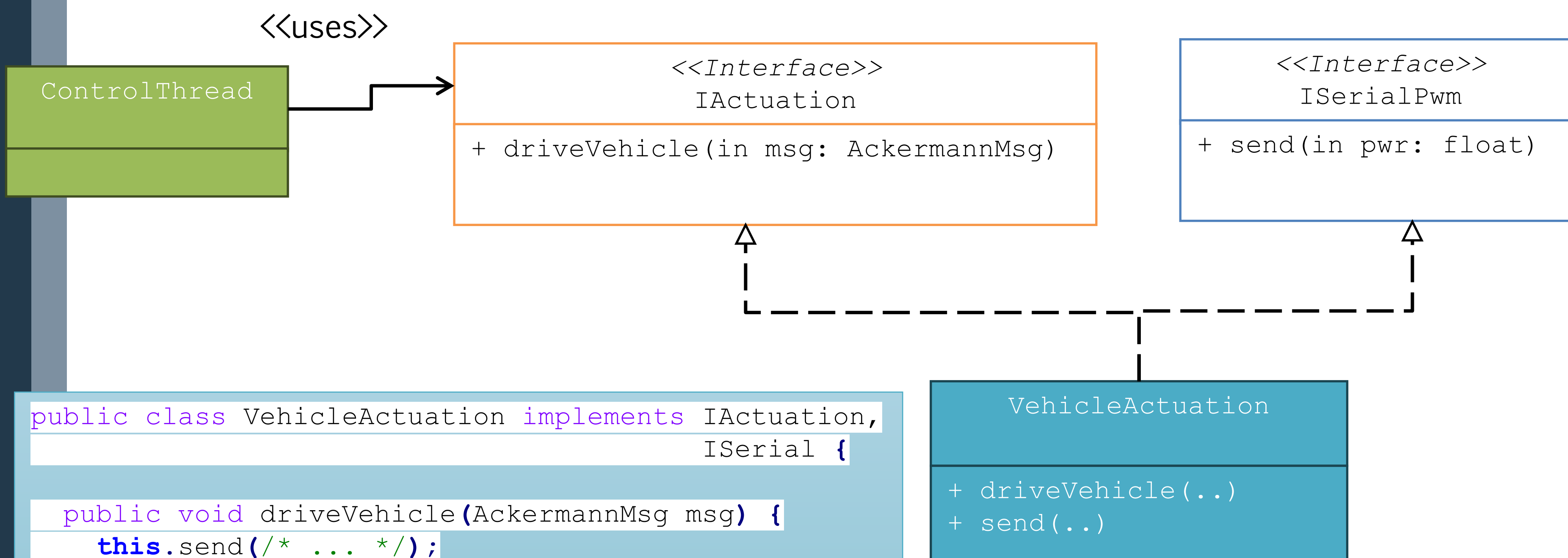


In the F1/10 case



Note

› It might look like we're breaking the Single Responsibility principle...

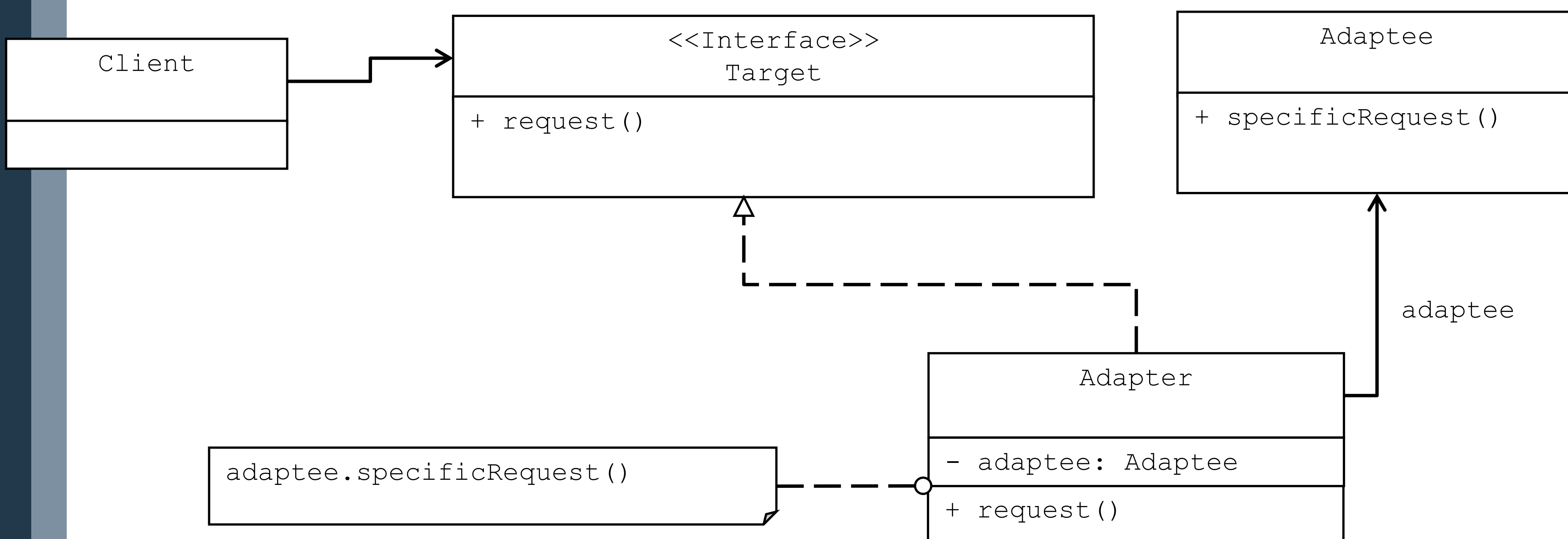



```java
public class VehicleActuation implements IActuation,
                                         ISerial {

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

  public void send(float pwr) {
    /* ... */
  }
}
```

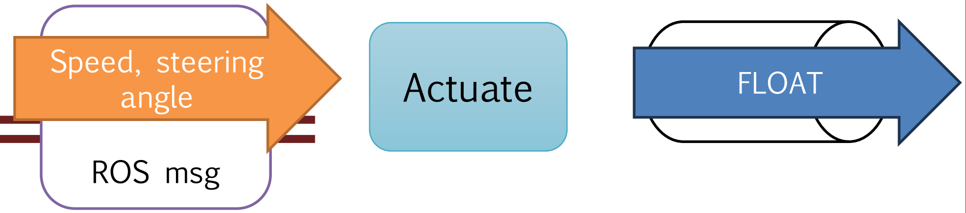
Basic structure: object-based

Overloads concrete classes

- Here, `target` is an interface, and `Adaptee` is not for the purpose of clarity

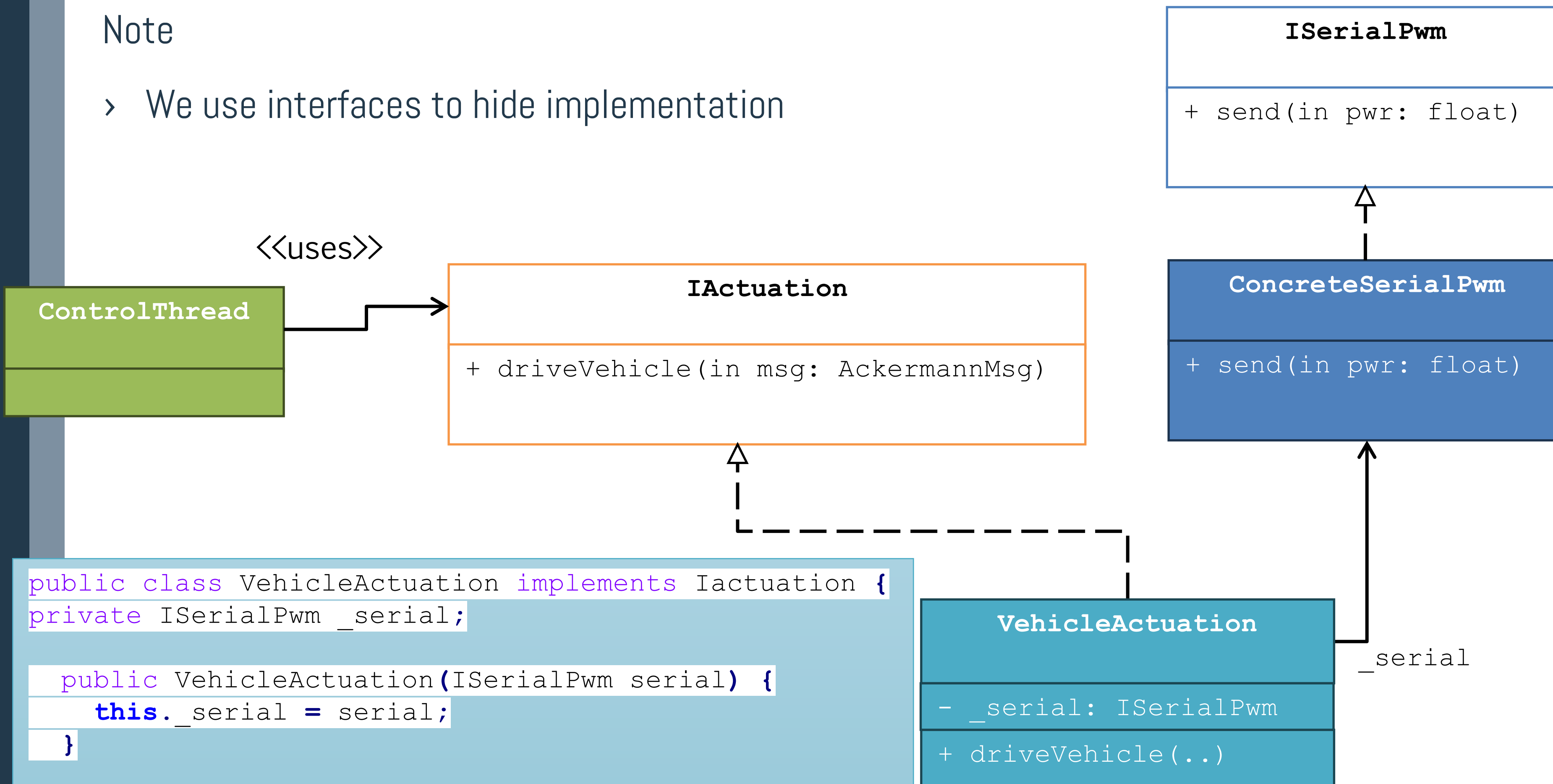


In the F1/10 case



Note

- We use interfaces to hide implementation



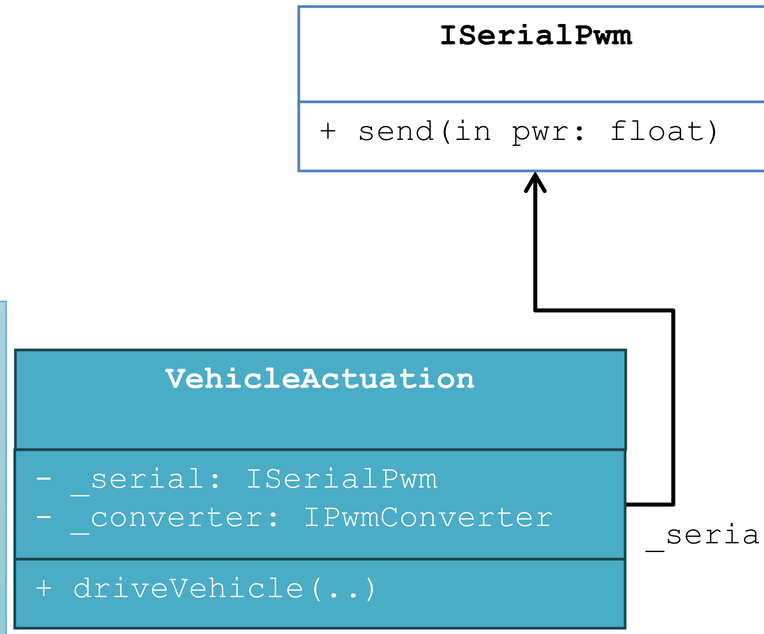

```java
public class VehicleActuation implements Iactuation {
private ISerialPwm _serial;

  public VehicleActuation(ISerialPwm serial) {
    this._serial = serial;
  }

  public void driveVehicle(AckermannMsg msg) {
    this._serial.send(/* Convert format */);
  }
}
```

Go beyond...

How do we convert the format?

- Is this responsibility of the Vehicle Actuation class?
- Is this responsibility of another class?

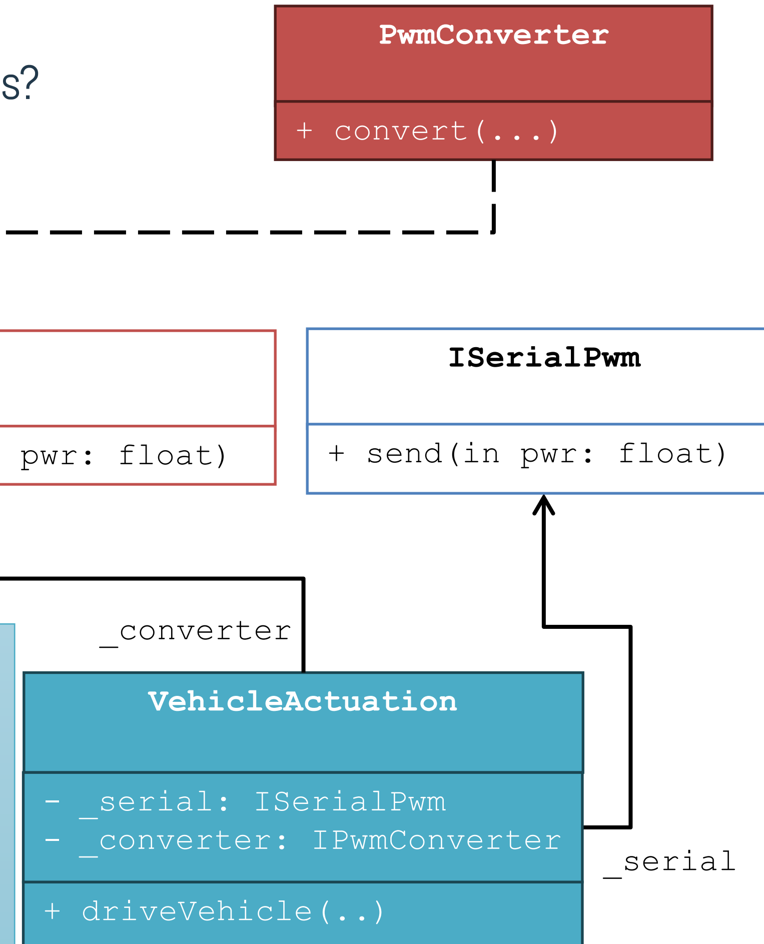
```
public class VehicleActuation implements Iactuation {
private ISerialPwm _serial;

  public VehicleActuation(ISerialPwm serial) {
    this._serial = serial;
  }

  public void driveVehicle(AckermannMsg msg) {
    this._serial.send(/* Convert format */);
  }
}
```



Go beyond...

How do we convert the format?

- Is this responsibility of the Vehicle Actuation class?
- Is this responsibility of another class?
- Here, I use another class/interface




```
public class VehicleActuation implements Iactuation {
  private ISerialPwm _serial;
  private IPwmConverter _converter;

  public void driveVehicle(AckermannMsg msg) {
    this._serial.send(this._converter.convert(msg));
  }
}
```

Consequences/side effects

- The amount of work an adapter shall do depends on the difference between the two interfaces to adapt: you might want (and I did) use multiple adapters
- You can group them into a "family" of adapters (see also Abstract Factory) to enable multiple targets (e.g., serial/PWM vs another protocol)
- We should implement many smaller interfaces, rather than few, big ones, to enable "seeing the same class from different perspectives"
  - The Single Responsibility principle applies also to interfaces! And it's even more important than for classes (**why?**)

Notes

- In the example, I used the interfaces as much as I could. See also the `Bridge` pattern (or the "I" principle)

Consequences/side effects (cont'd)

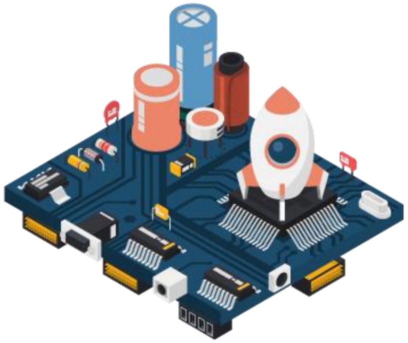
The class-based implementation

- Does not break the "S" principle, because...it's exactly the **responsibility** of the adapter!
- It requires only one object
- Makes it easier to subclass

The object-based approach

- Lets you providing a "default implementation/behavior"
- Makes the adapter operating with multiple adaptees (the F1/10 has two serials, one for throttle, one for steering!)

In embedded systems
(Adapter variants)



What's special about embedded programming?

You typically have less generalized, more purpose-specific circuits and systems

- Real-time constraints (e.g., Cyber-Physical Systems) call for hard requirements
  - BTW...The good news: requirements collection is highly structured and standardized
  - Specialized OSes (e.g., RT-Oses)
- Hardware might have specific features
  - How do we abstract them?
- Tight Size, Weight and Power constraints (SWaP), cause low computational power

We typically program them in C, C++, or reduced set of C (or even ASM!)

- OOP might be traded for performance reasons, for functional/structured programming
- Classes are "hacked" via structs, and functions; header files specify contracts/interfaces
- Finite State Machines as paradigm/pattern to ensure formal correctness

Closer to HW

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

› Note: this is just a possible example

App
Libs

Docker

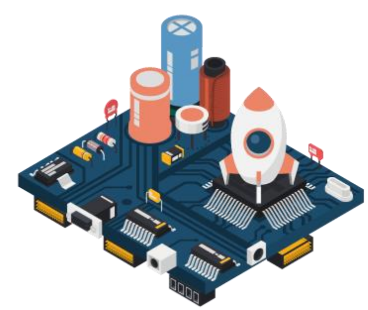
Runtime libs
Framework

OS
Hyper-V

BSP

Can even be compiled all together

App
SDK libs
(OS- MicroK)??
BSP



The challenge: abstracting the HW

- Cores and caches are hidden, however specific functionalities might exist (ex: RISC-V extensions)
- Memory is explicitly managed: no Garbage Collector!
- 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”

- Does this remind of something?

The challenge: abstracting the HW

Adapter

- Does this remind of something?

Hardware Proxy

A **structural** pattern

Purpose

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

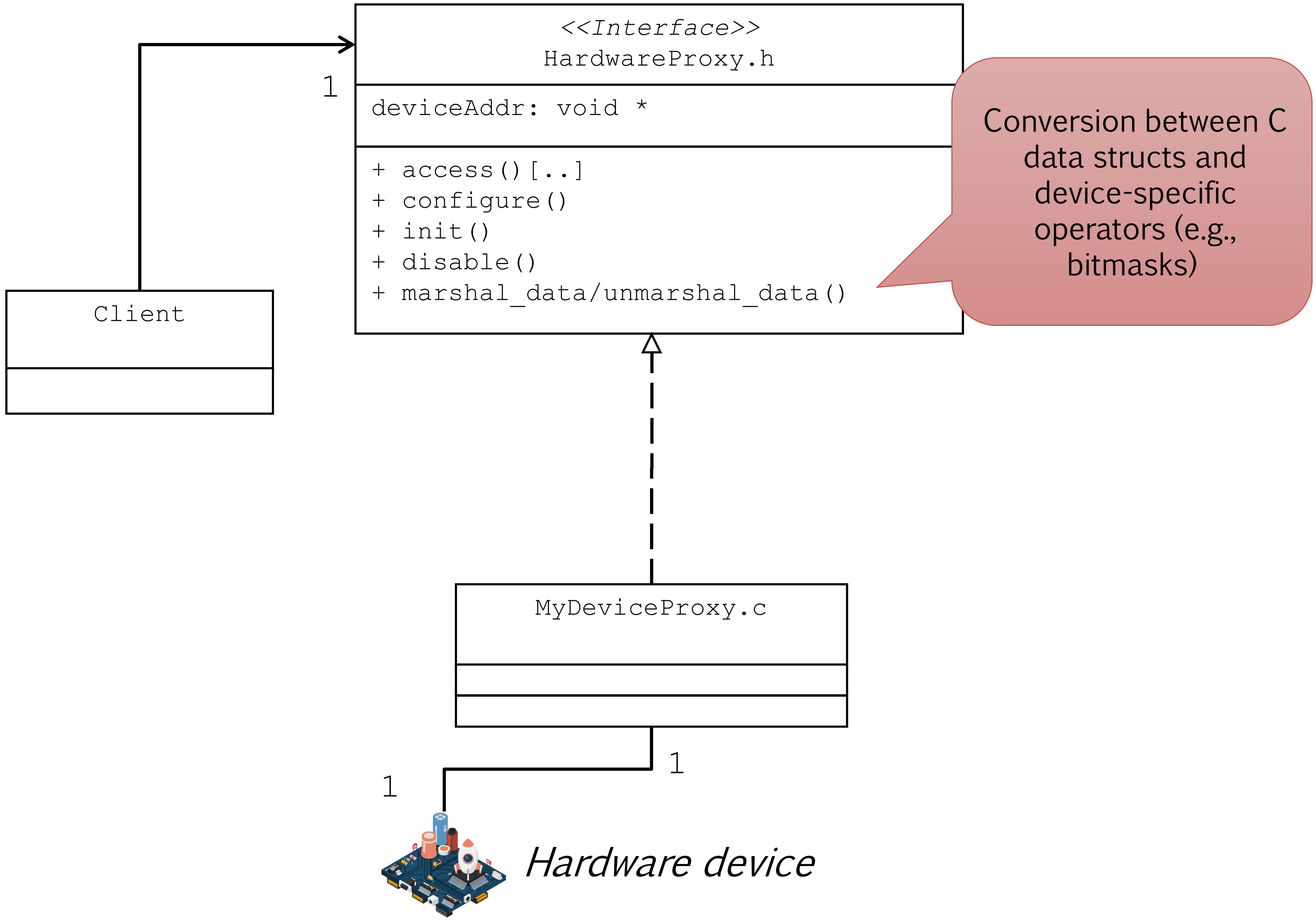
Motivation

- 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



Consequences/side effects

Same as previously seen in Adapter, plus

- You have to handle concurrency (with locks, critical regions...)
- 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!

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!

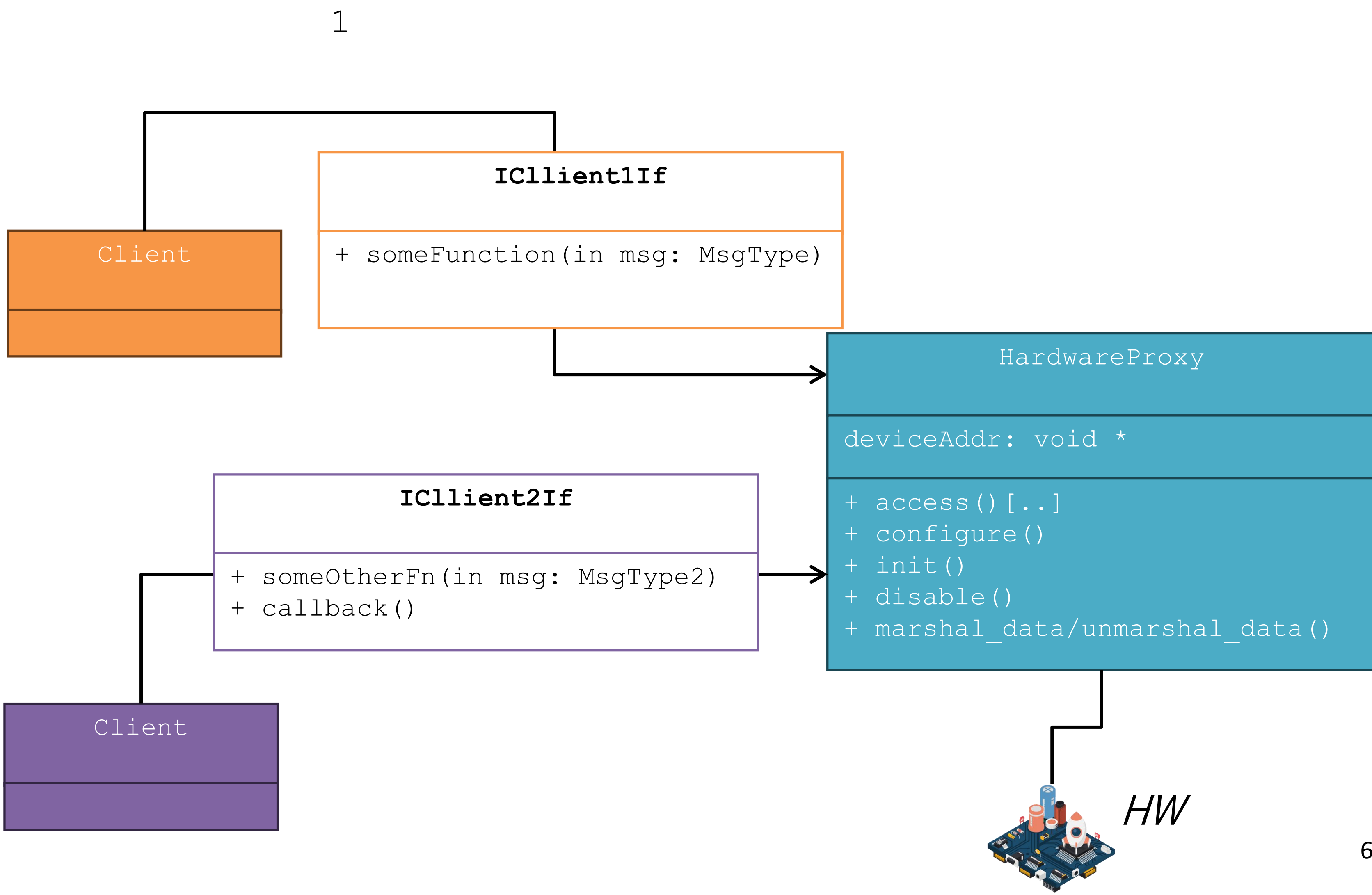
› Actually, it is typically used together with Proxy!

Applicability

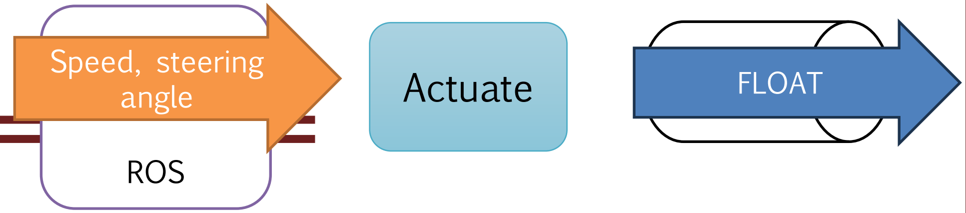
› When you need to adapt application data structs to HW

Roles

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

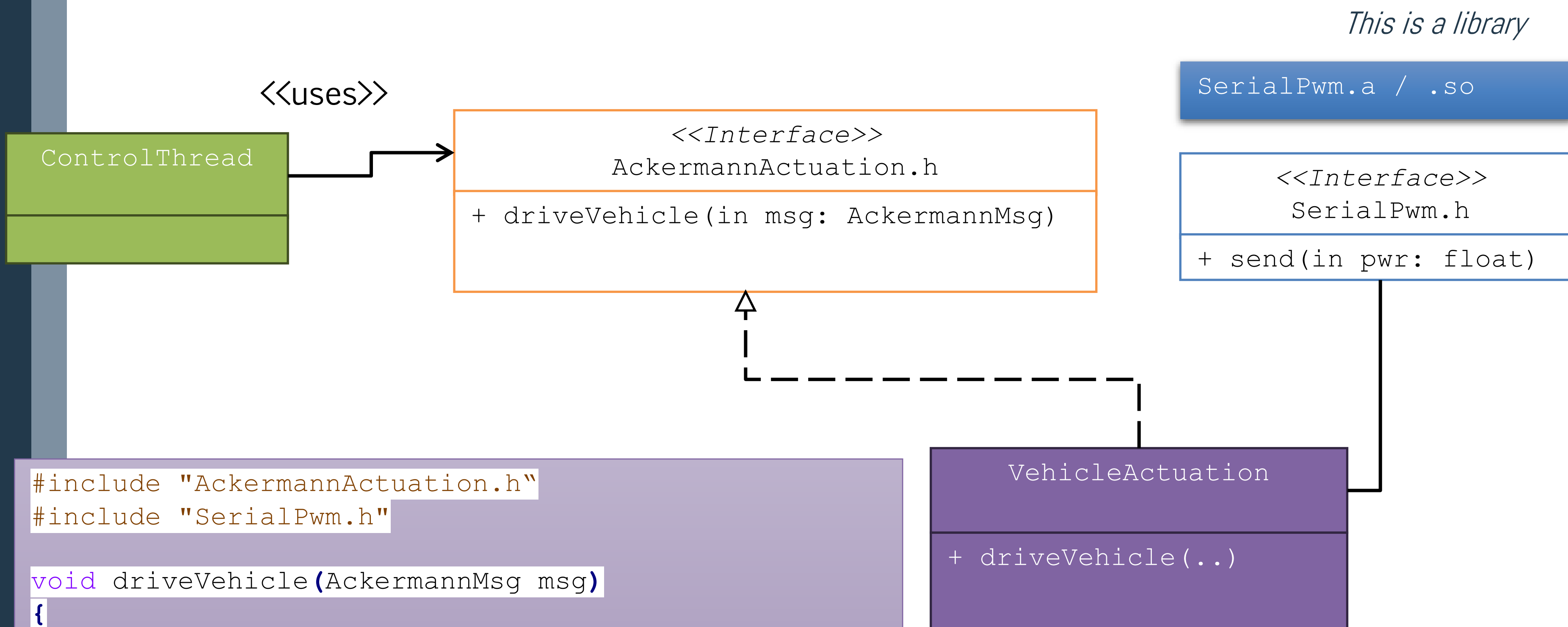


Example: the F1/10



Note

- Here, I implemented using C-style primitives




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

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

Code smells

Code smells

"Any characteristic in the source code of a program that possibly indicates a deeper problem." (cit. Wikipedia)

- It's just a "warning" that "probably something is going wrong"
- Typically, a wrong/stale design choices (yes, project evolve over time), or you're breaking SOLID principles, or some design pattern is not applied them
- You can probably solve it by using design patterns

The definition of Anti-pattern

- A **commonly-used** process, structure or pattern of action that, despite **initially appearing to be an appropriate** and effective response to a problem, **has more bad consequences than good ones**.
- Another solution exists to the problem the anti-pattern is attempting to address. This solution is documented, repeatable, and proven to be effective where the anti-pattern is not.
- "Rule-of-three": you should witness at least three times in your code

The bad news, and the good ones

In 2015, an automated analysis * for half **a million source** code commits, and the manual examination of 9,164 commits, found that:

- There are only anecdotal evidence as to how, when, or why “technical debt” occurs, it cannot be formally analyzed (hence, there are no tools that can 100% identify it)
- Typically, caused by **urgent** maintenance activities and **pressure** to deliver features while *prioritizing time-to-market over code quality*

These were the good news

- The bad news is that you have **no control** on management..but still you can force you (and your team) to try to follow good coding guidelines
- Typically, +20-25% of coding time
- Providing a single, (declared) unstable version of an SW components, as proof-of-concept, is a good idea (you can refine it later), but the overall architecture must be dell designed!
- The usage of frameworks and well-known technologies forces, at least, to adhere to a SW architecture

** Tufano, Michele; Palomba, Fabio; Bavota, Gabriele; Oliveto, Rocco; Di Penta, Massimiliano; De Lucia, Andrea; Poshyvanyk, Denys (2015). "When and Why Your Code Starts to Smell Bad" (PDF). 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. pp. 403–414. CiteSeerX 10.1.1.709.6783."*

Typical smells

Bloaters

- Code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. They typically accumulate over time as the program evolves.
- Few examples are: long methods, big classes, too many params in ctors, methods...

OO abuse/misuse

- When you apply the OO principles in a wrong manner
- Ex: two classes that basically do the same thing; too many `if`s or `switch`es...

Changes preventers

- A single change/bugfix/added functionalities, requires too many modifications in different places
- Ex: when you create a subclass for a class, you need to create a subclass to another class
- Does this remind of something?

Typical smells

Dispensables

- You have, in your code, something that you don't really need
- Ex: dead code, duplicate code, overload of comments (we'll talk about this..), too many public fields in a class

Couplers

- Two or more classes are too much dependant one another
- Ex: *Feature Envy* - one class accesses more the methods of another class, than its own (ant it's not an aggregation)

There are typical patterns to solve each of these problems

References

Course website

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

Course website

- 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/

My contacts

- paolo.burgio@unimore.it
- http://hipert.mat.unimore.it/people/paolob/