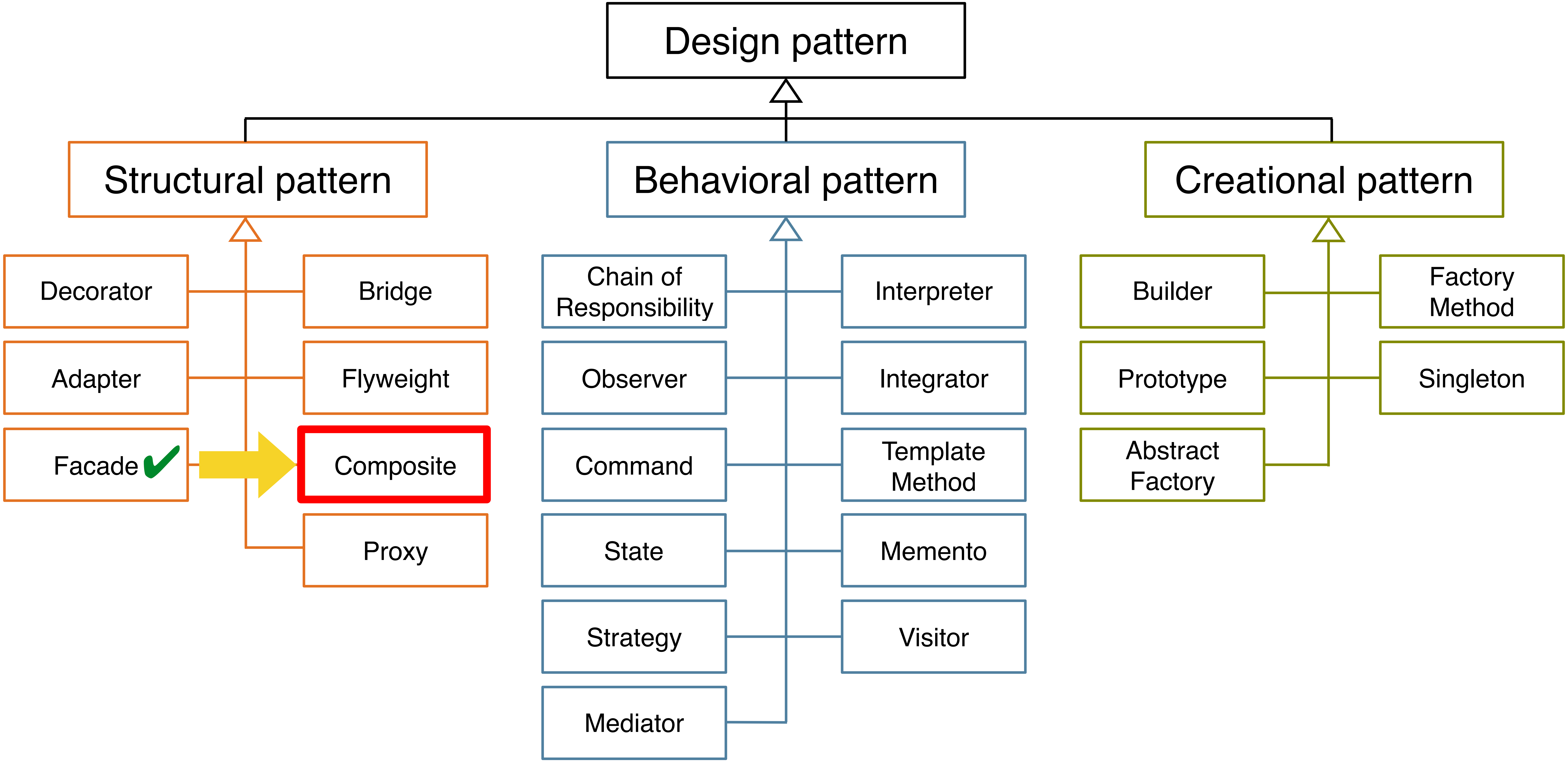
Outline

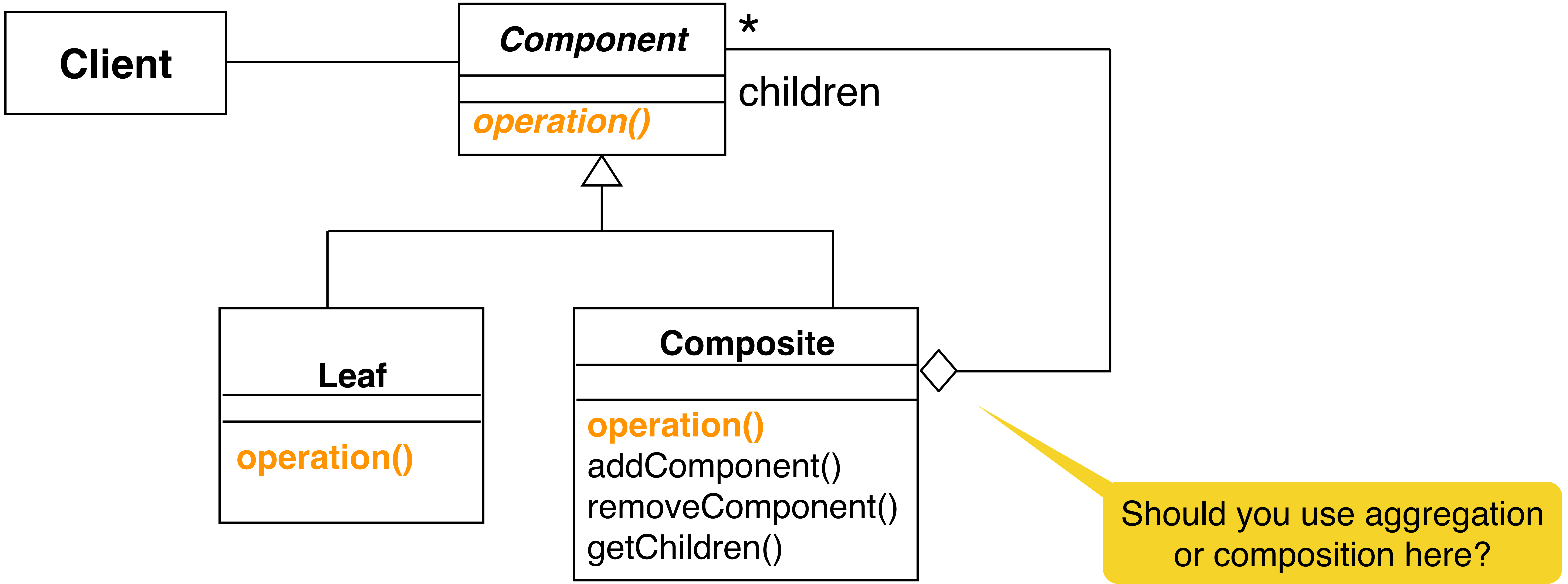
- Object design
- Reuse
- Generalization vs. specialization
- Design patterns
  - **Composite pattern**
  - Bridge pattern
  - Proxy pattern

Design patterns taxonomy



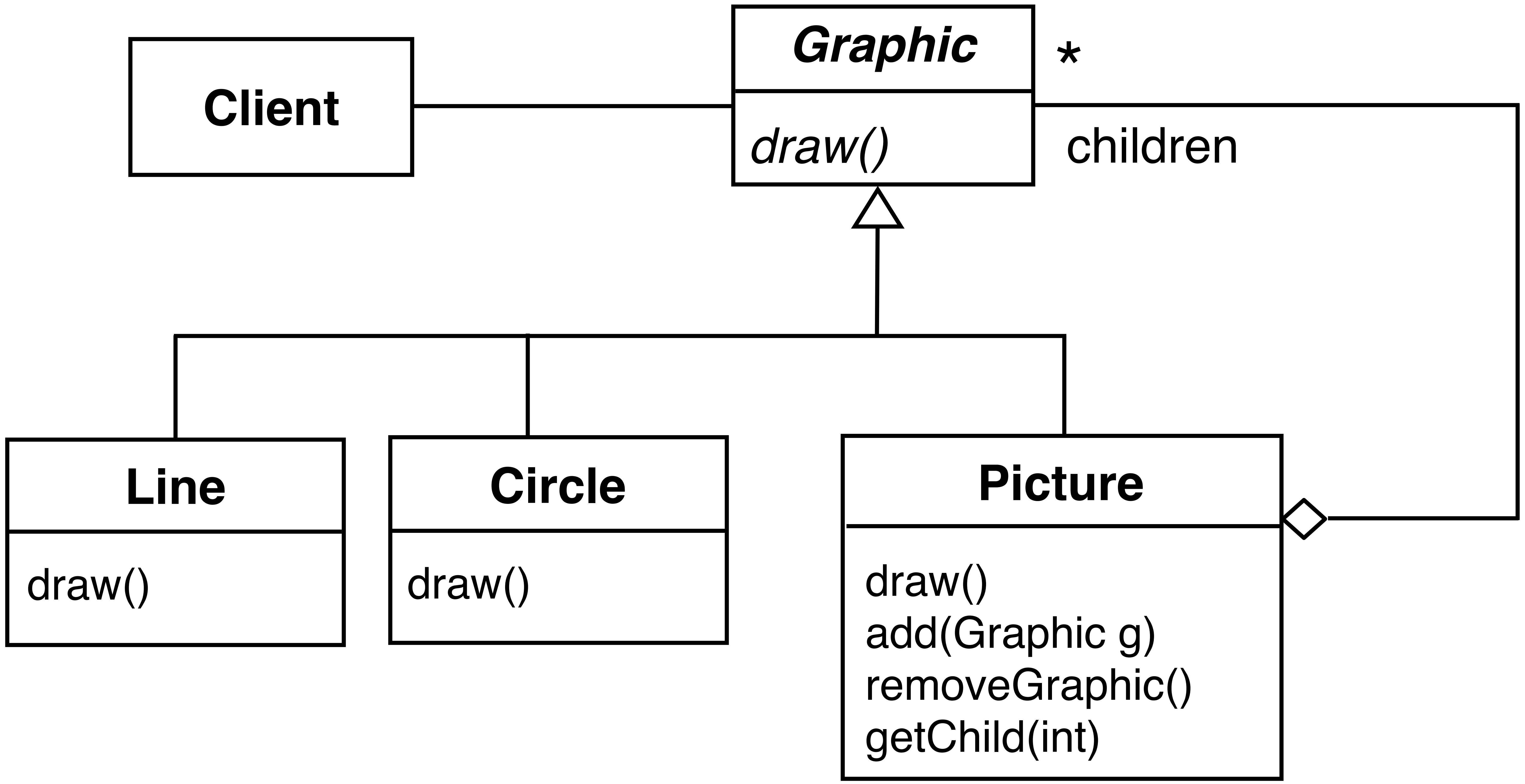
Composite pattern

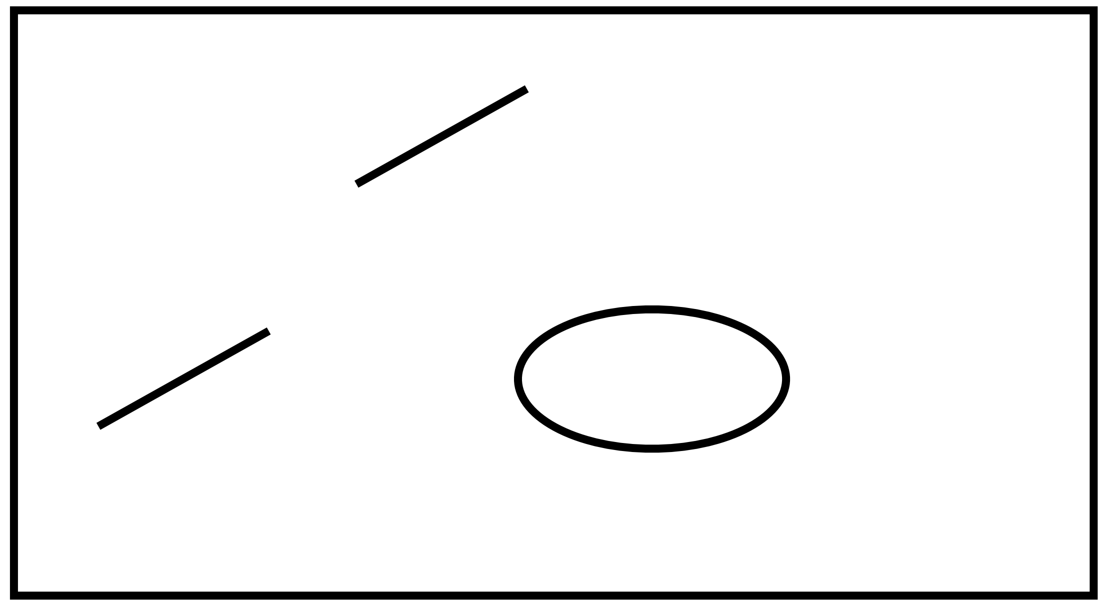
- **Problem:** there are hierarchies with arbitrary depth and width (e.g. folders and files)
- **Solution:** the composite pattern lets a **`Client`** treat an individual class called **`Leaf`** and **`Compositions`** of **`Leaf`** classes uniformly



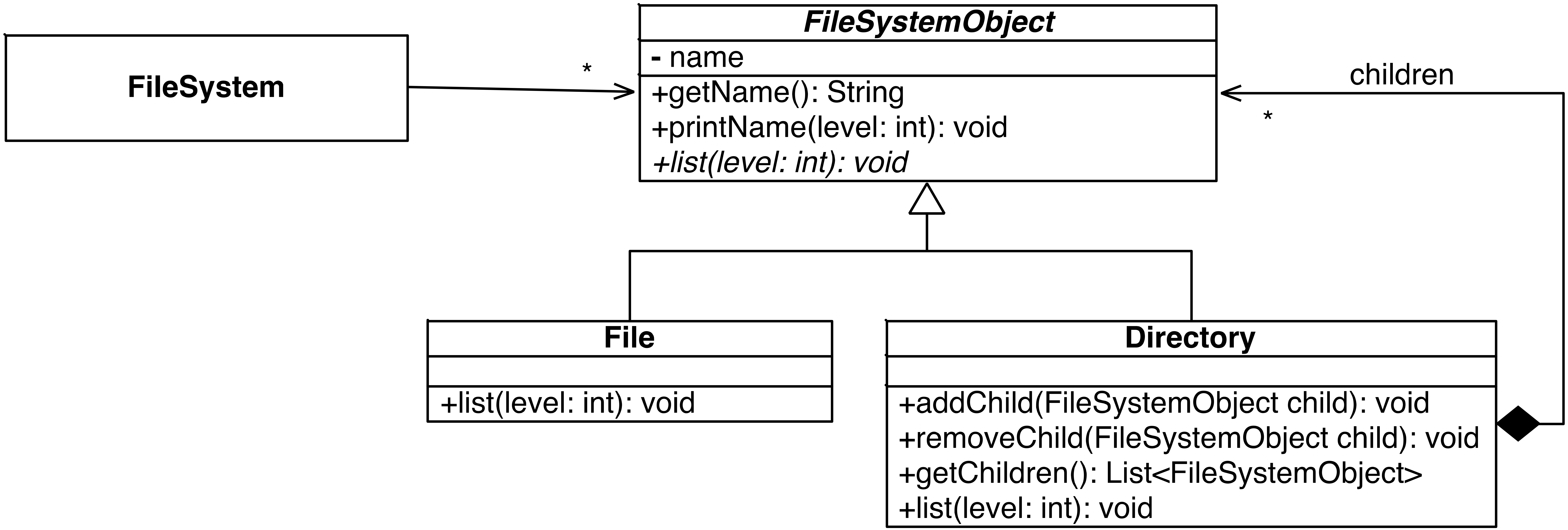
Example: composite pattern in graphic applications

Graphic represents both **Line** and **Circle** (Leaf), as well as **Picture** (Composite)





Example: composite pattern in file systems



L06E02 Composite Pattern

Start exercise

Easy

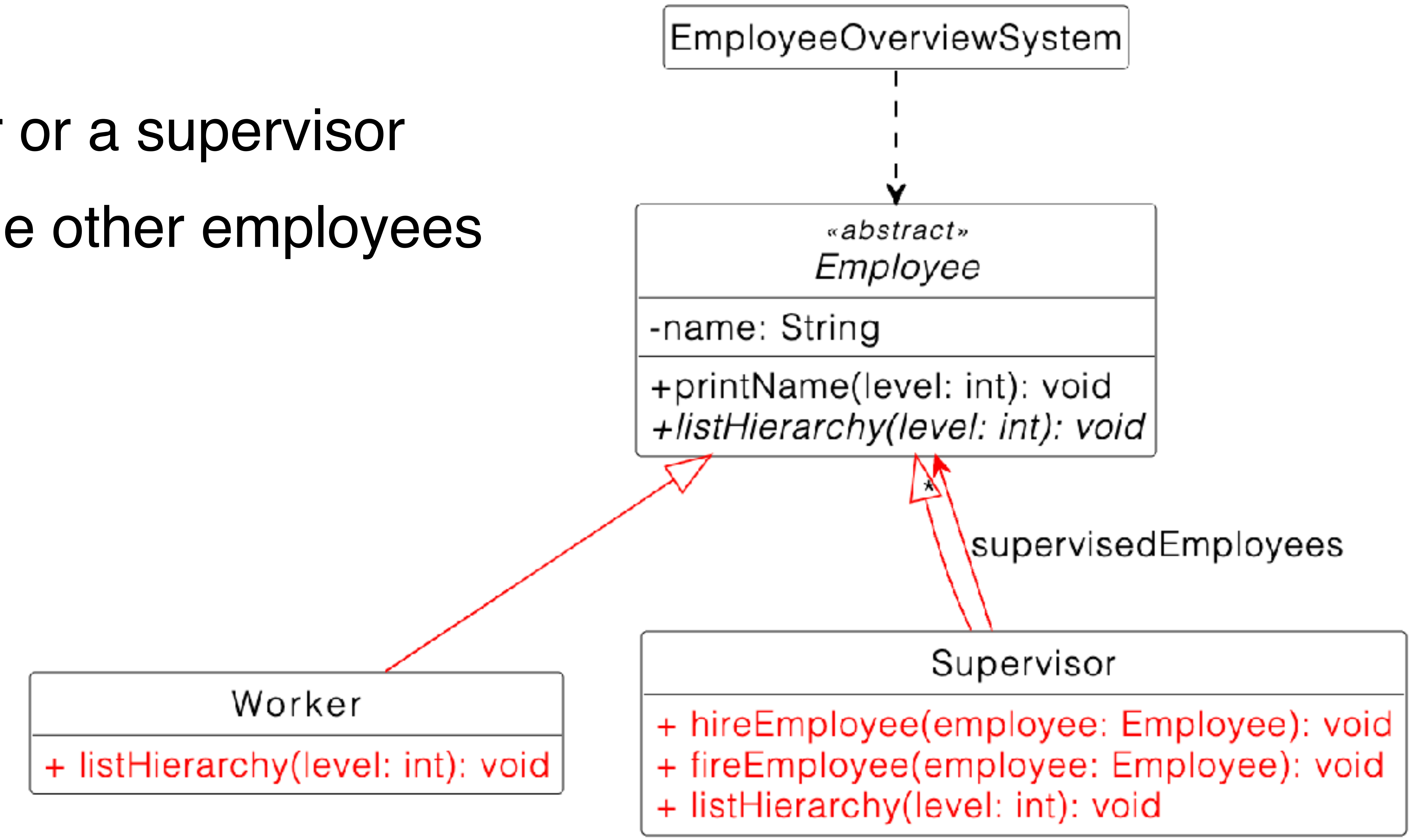
Not started yet.

Due date: end of today

10 min

4 pts

- **Problem statement**
  - You design a simple employee overview system which consists of employees
  - Employees can either be a worker or a supervisor
  - A supervisor can supervise multiple other employees



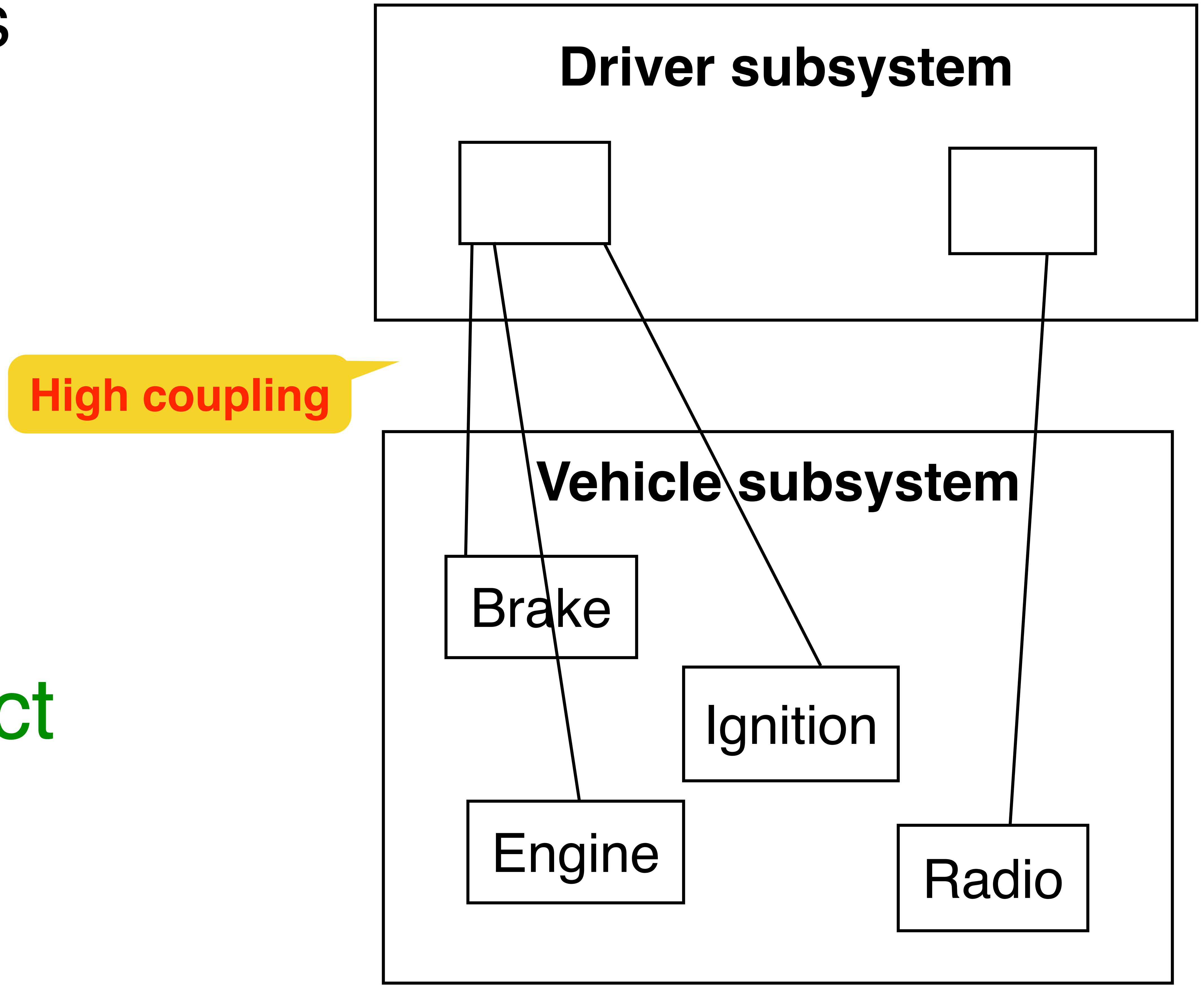
Outline

- Overview of system design
- Design goals
- Hints for system design
- Subsystem decomposition
- **Façade pattern**
- Architectural styles
  - Layered architecture
  - Client server architecture
  - REST architectural style
- UML component diagrams

Another design example

The **driver subsystem** can call any class operation in the **vehicle subsystem**

➡ **Problem: Spaghetti design!**



Solution: subsystem interface object

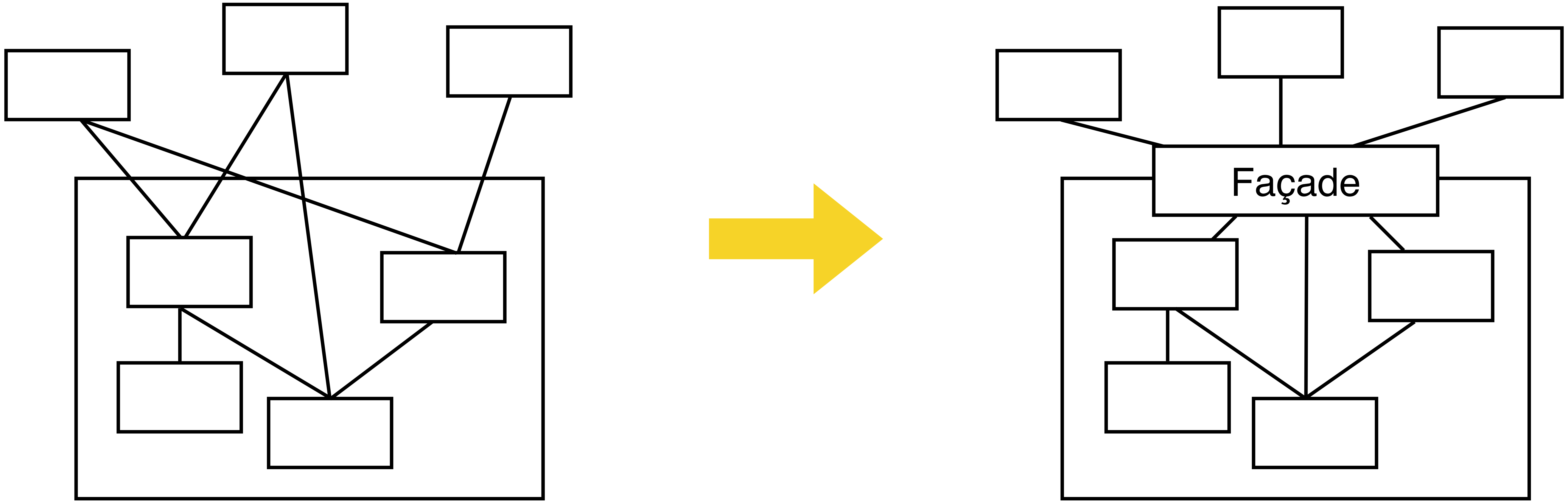
Subsystem interface object

- The set of **public operations** provided by a subsystem
- The subsystem interface object describes **all services** of the subsystem interface
- A subsystem interface object can be realized with the **façade design pattern**

Façade design pattern: reduces coupling

More details on **design patterns** in **Lecture 07** on **Object design**

- Provides a **unified interface** for a subsystem
  - Consists of a set of public operations
  - Each public operation is delegated to one or more operations in the classes behind the façade
- Defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts away the gory details)
- Allows to **hide** design spaghetti from the caller

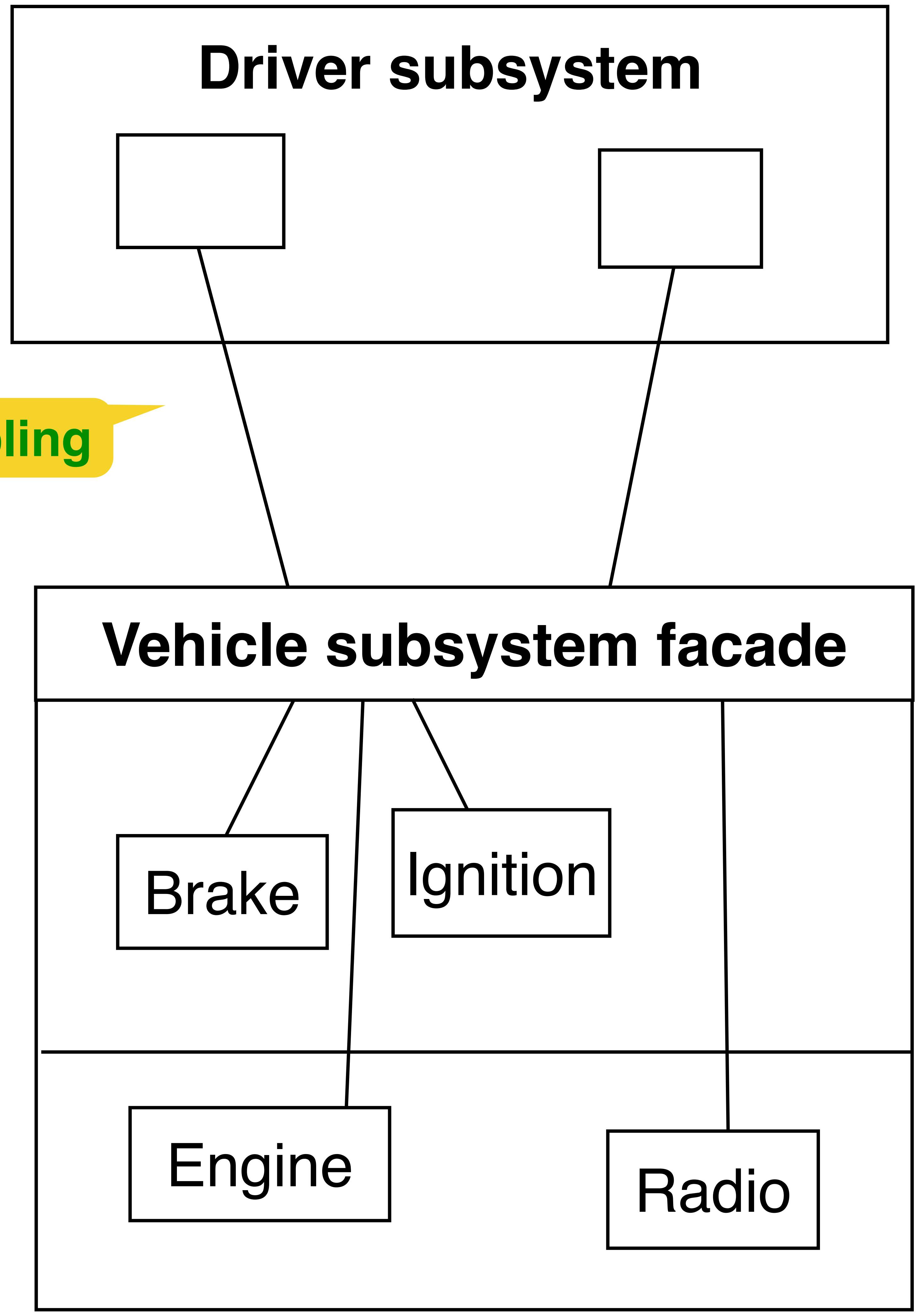


Opaque architecture with a façade

- The **vehicle subsystem** decides exactly how it is accessed with the **vehicle subsystem façade**
- **Advantages**
  - Reduced complexity
  - Fewer recompilations
  - A façade can be used during integration testing when the internal classes are not implemented
  - Possibility of writing mock objects for each of the public methods in the façade

More details in
Lecture 08 on **Testing**

Lower coupling



L04E02 Facade Pattern

Not started yet.

Start exercise

Easy

Due Date: End of today (AoE)

10 min

5 pts

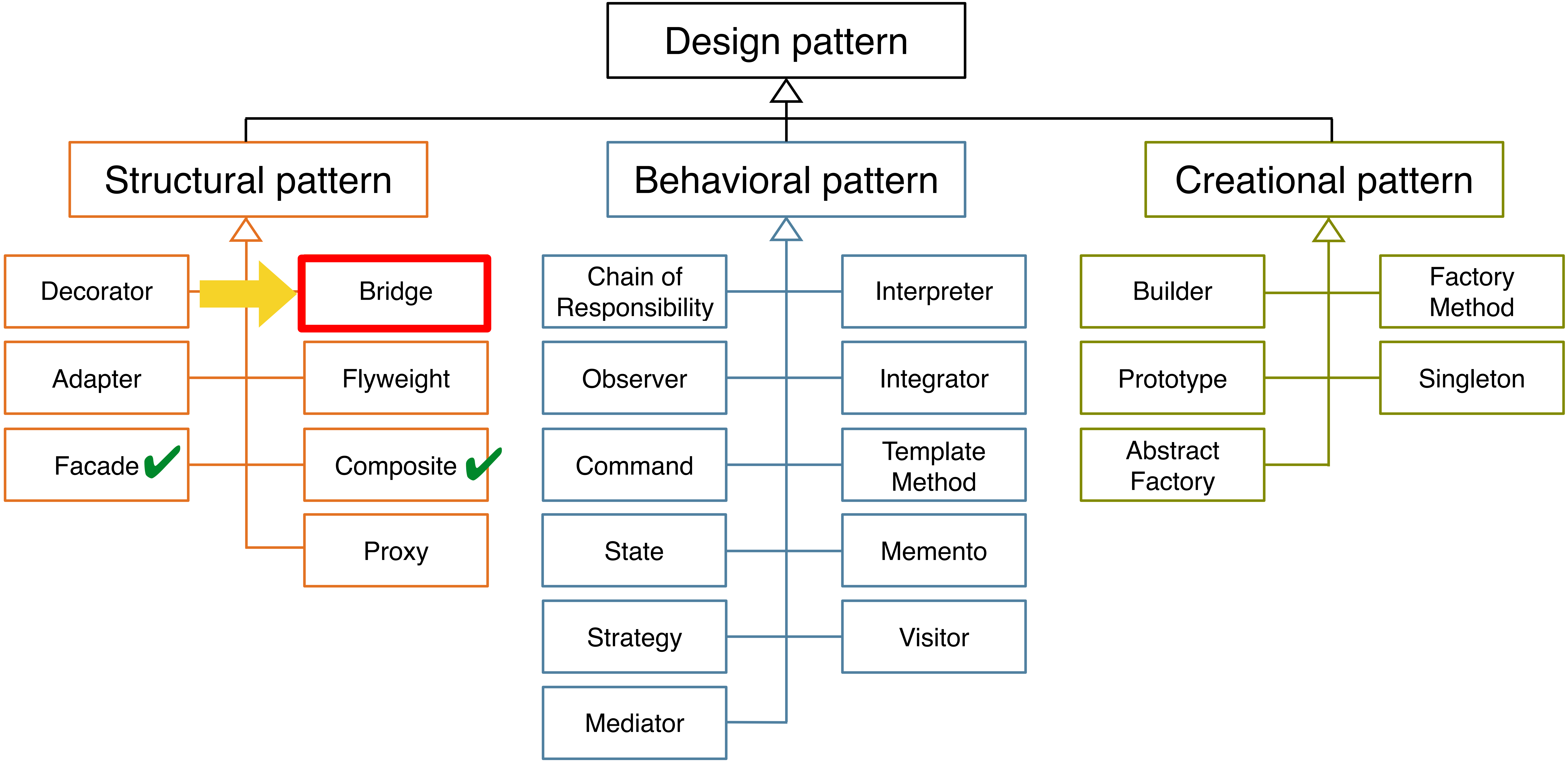
- **Problem statement**
  - Reduce the coupling between the two subsystems **store** and **ecommerce**
  - Implement the façade pattern to provide a unique interface for the **ecommerce** subsystem

Design principile:
low coupling

Outline

- Object design
- Reuse
- Generalization vs. specialization
- Design patterns
  - Composite pattern
  - **Bridge pattern**
  - Proxy pattern

Design patterns taxonomy

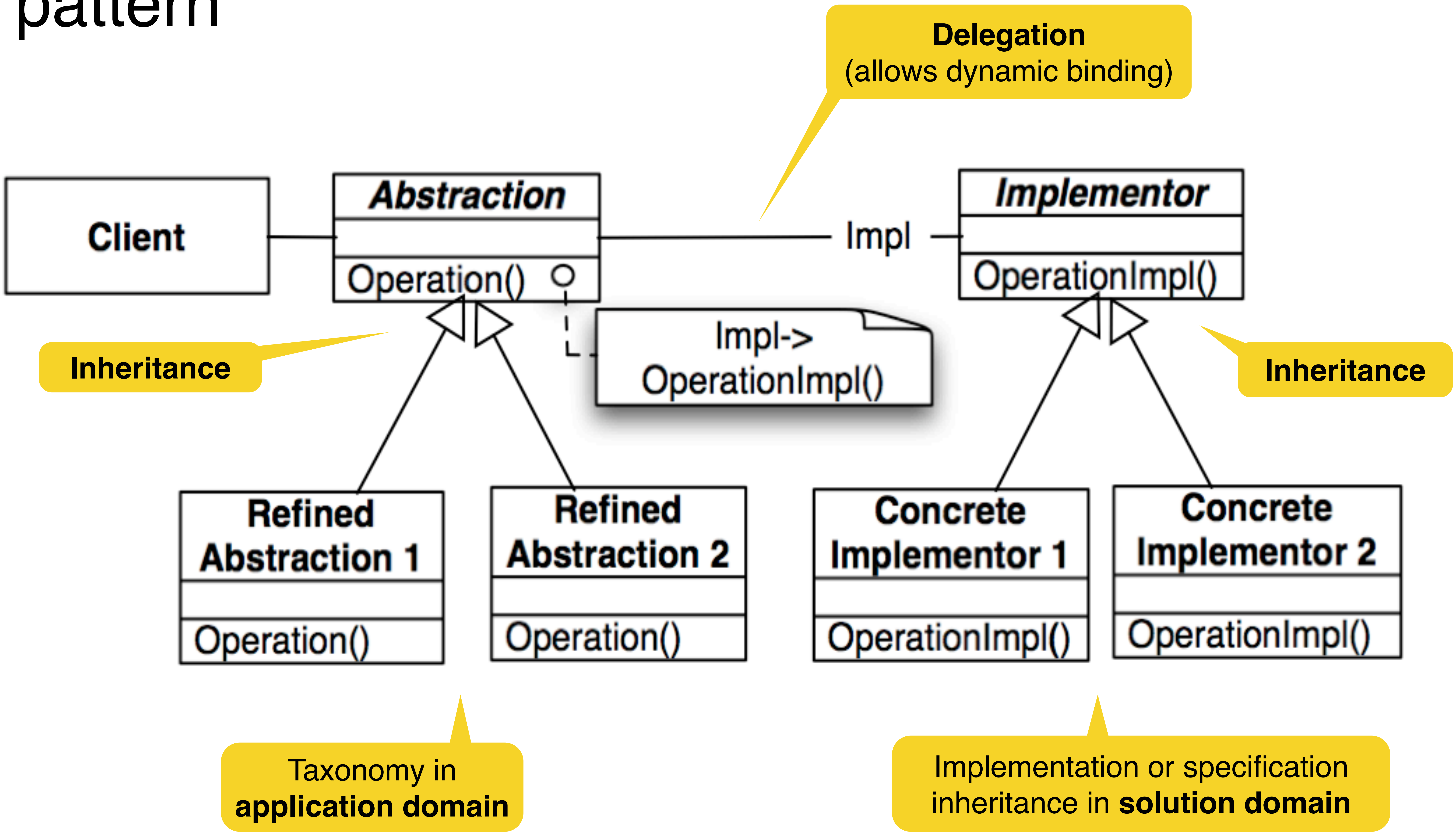


Bridge pattern

- **Problem:** many design decisions are made final at design time or at compile time
  - **Example:** binding a client to one of many implementation classes of an interface
- Sometimes, it is desirable to delay design decisions until runtime
  - **Example:** one client uses a very old implementation, the other client uses a more recent implementation of an interface
- **Solution:** the bridge pattern allows to delay the binding between an interface and its subclass to the startup time of the system

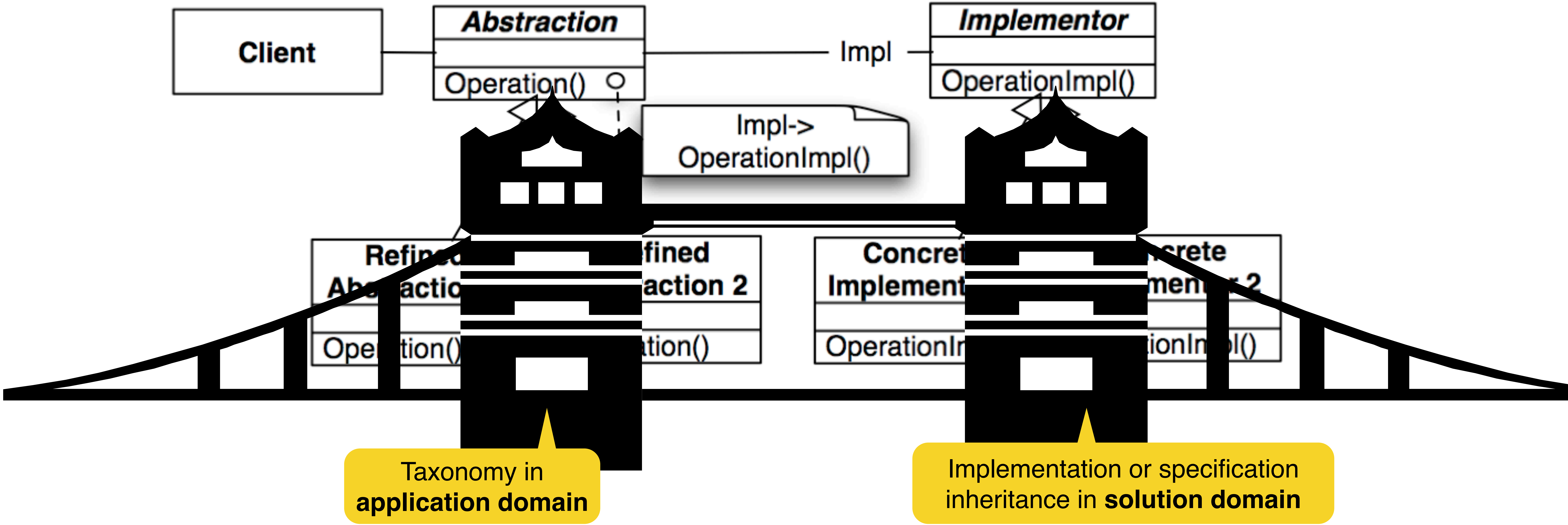
e.g. in the constructor of the implementation class

Bridge pattern

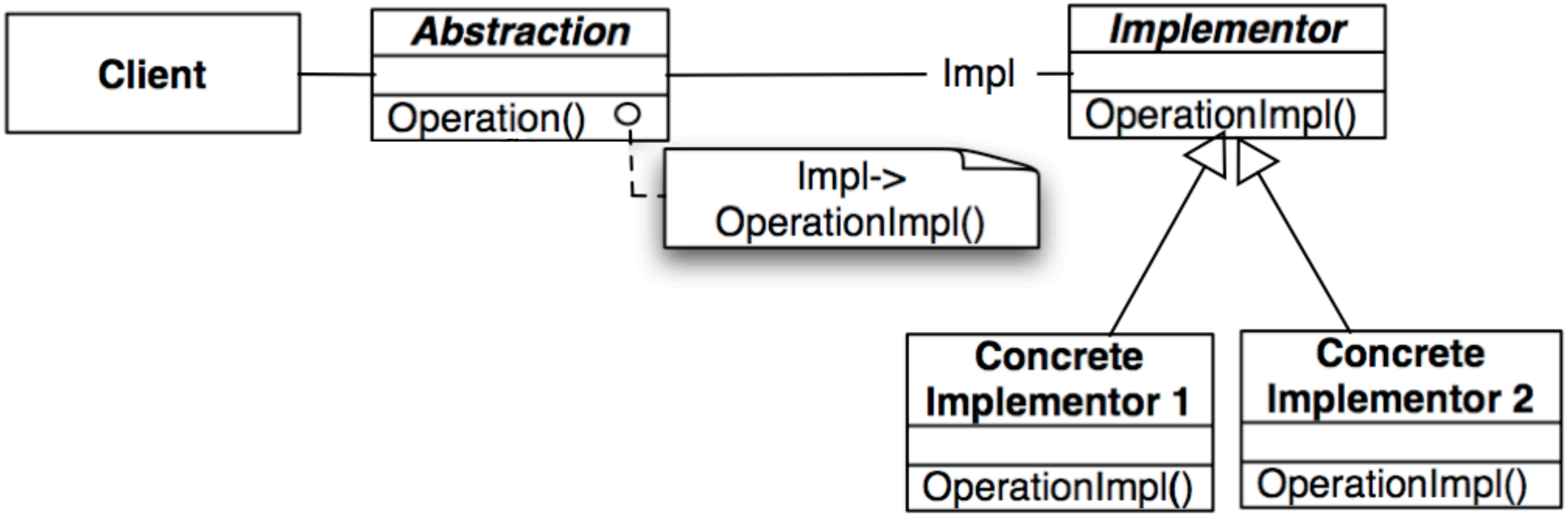


Why the name bridge pattern?

It provides a bridge between the abstraction (in the application domain) and the implementor (in the solution domain)



“Degenerated” bridge pattern: no application domain taxonomy

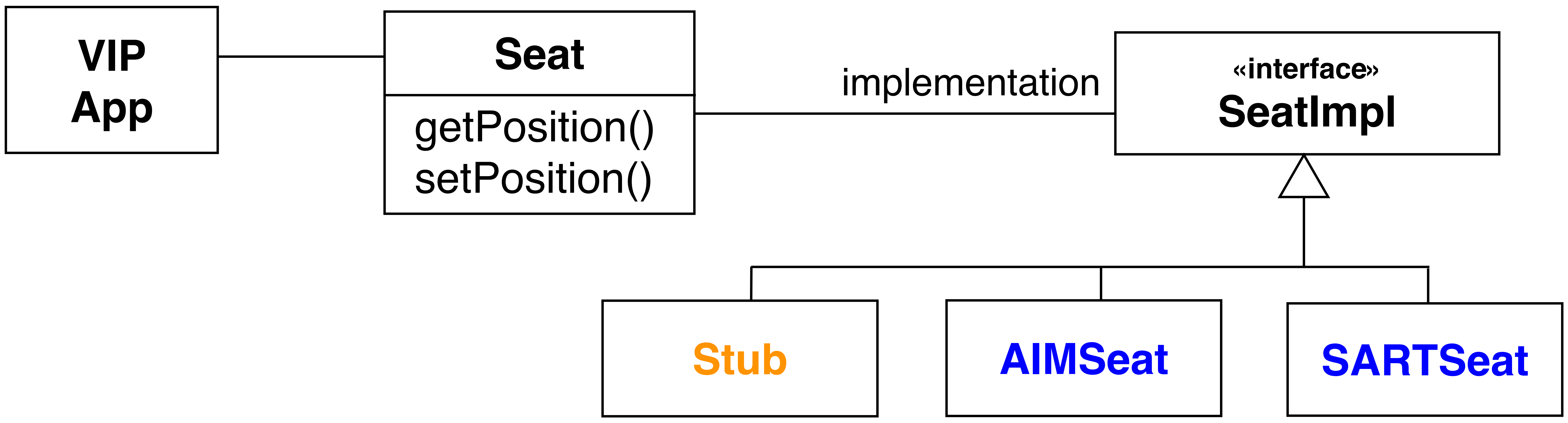


Implementation or specification inheritance in **solution domain**

Example of a degenerated bridge pattern

- Interface to an incomplete component (not yet implemented or unavailable)
- Typical situation in unit testing
- Example: a driver assistance system (VIP App) is developed which can adjust the position of the seat to the preference of the driver
  - The seat is not yet implemented, so we are using a stub
  - Then two-seat simulators become available: AIM and SART

More details in **Lecture 08** on **Testing**



L06E03 Bridge Pattern

Start exercise

Medium

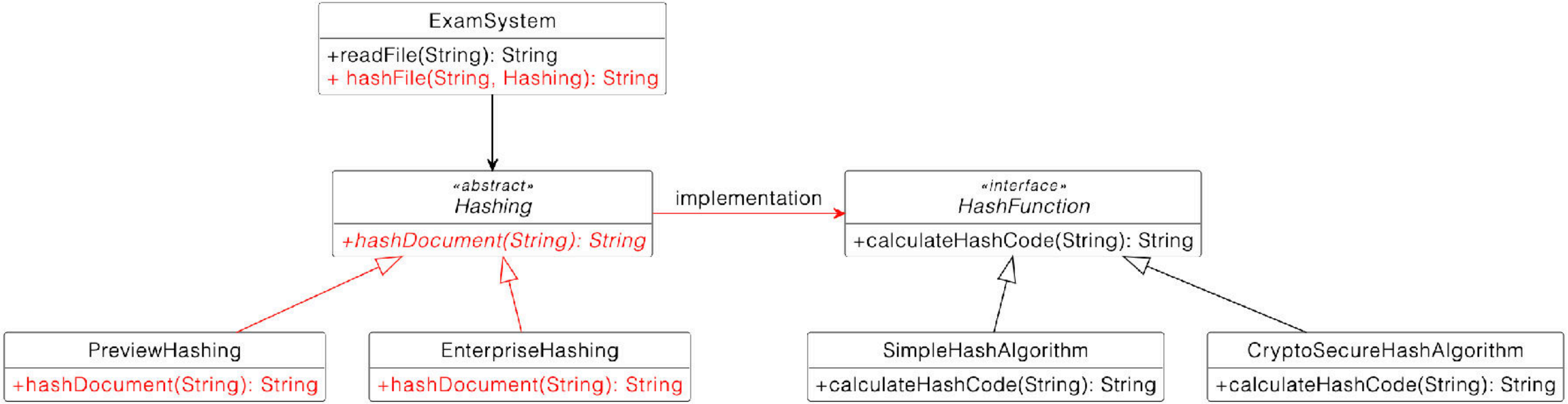
Not started yet.

Due date: end of today

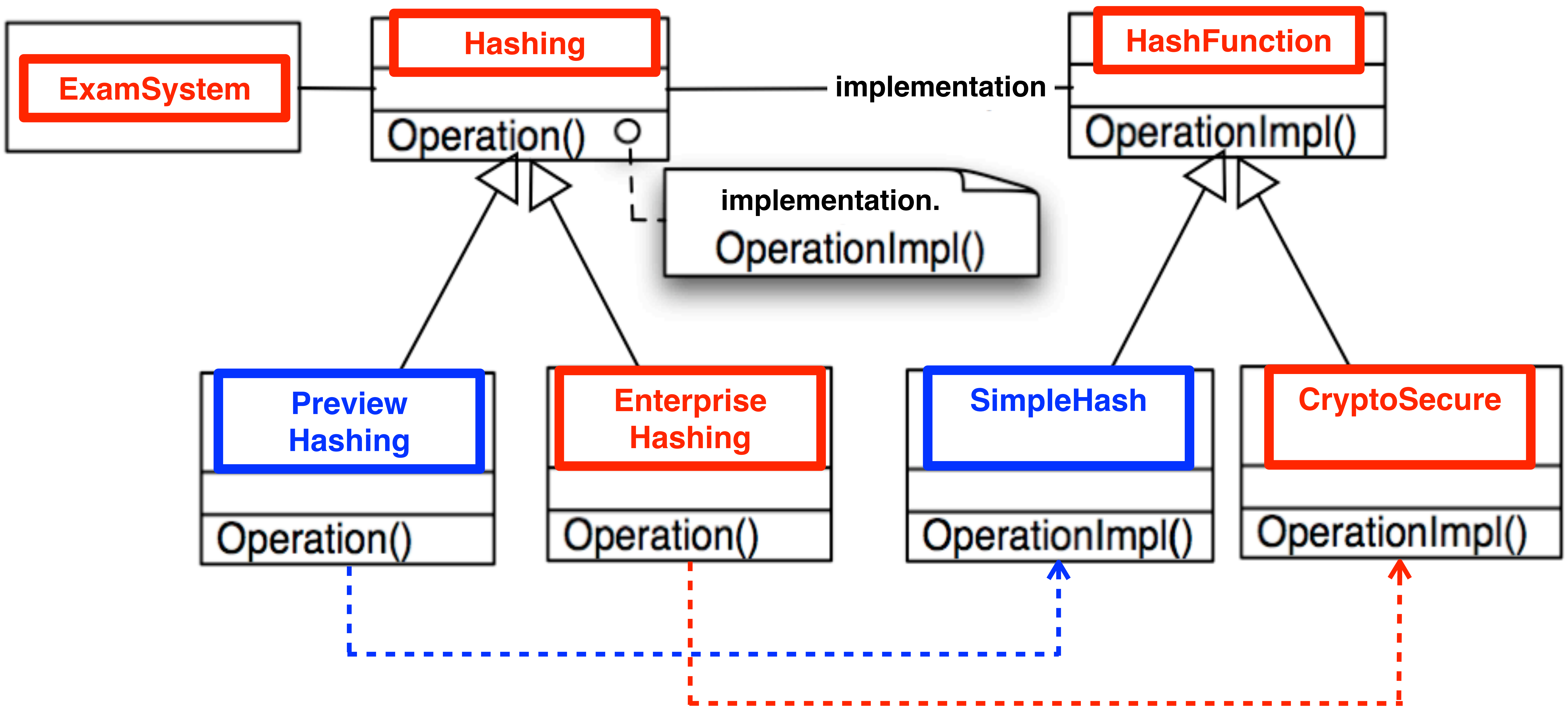
15 min

6 pts

- **Problem statement**
  - You develop an application for online exams
  - Two hashing algorithms are already implemented: **Simple**, **CryptoSecure** (solution domain)
  - You want to offer two versions of the application: **Preview** and **Enterprise** (application domain)
  - Both versions should be able to handle strings of text



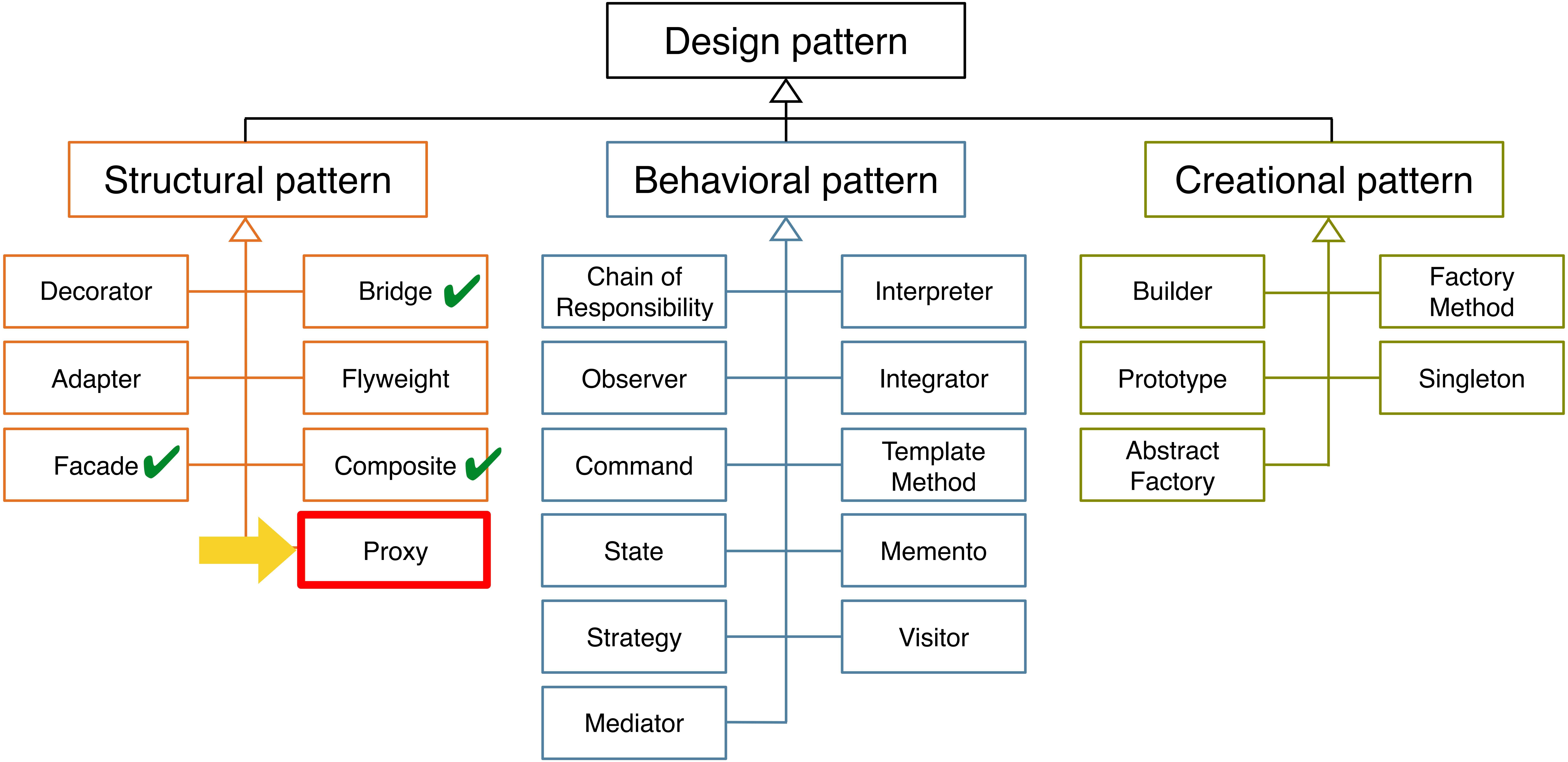
Hint: resulting UML diagram



Outline

- Object design
- Reuse
- Generalization vs. specialization
- Design patterns
  - Composite pattern
  - Bridge pattern
  - **Proxy pattern**

Design patterns taxonomy

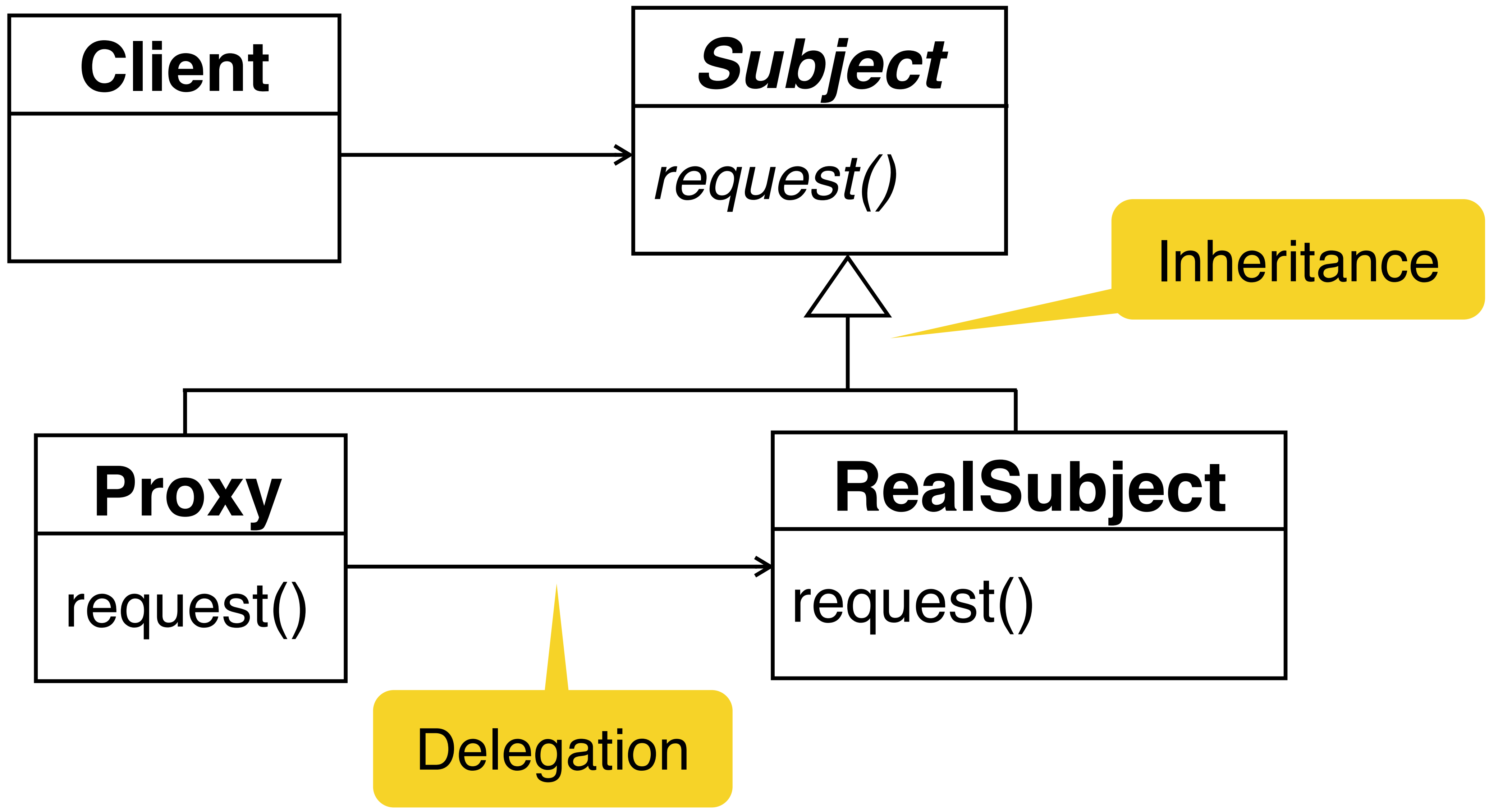


The proxy pattern solves two problems

- **Problem #1:** the object is complex, its instantiation is expensive
  - **Solution**
    - Delay the instantiation until the object is actually used
    - If the object is never used, then the costs for its instantiation do not occur
- **Problem #2:** the object is located on another node (i.e. on a web server), accessing the object is expensive
  - **Solution**
    - Instantiate and initialize a “smaller” local object, which acts as a representative (“proxy”) for the remote object
    - Try to access mostly the local object
    - Access the remote object only if really necessary

Proxy pattern

- **`Proxy`** and **`RealSubject`** are subclasses of the *abstract* class **`Subject`**
- The **`Client`** always calls **`request()`** in an instance of type **`Proxy`**
- The implementation of **`request()`** in **`Proxy`** then uses delegation to access **`request()`** in **`RealSubject`**



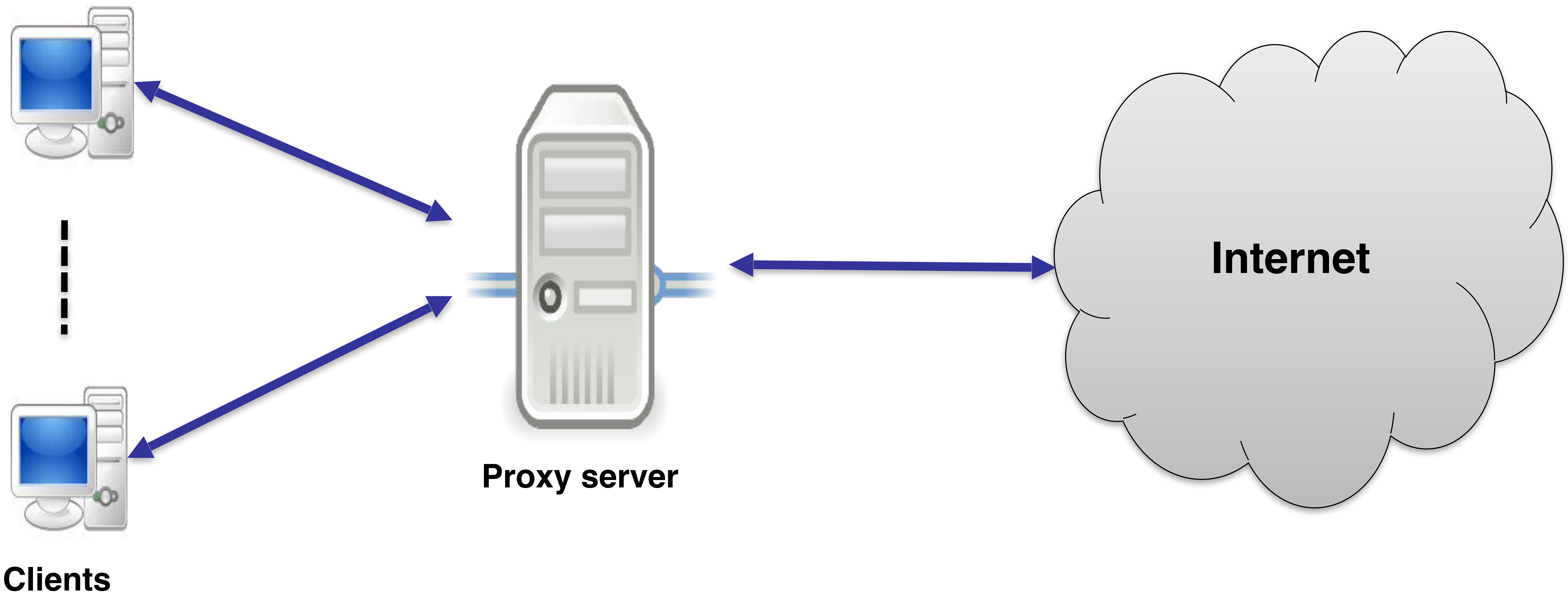
Applicability of the proxy pattern

- What is expensive in object-oriented systems?
    - Object creation
    - Object initialization
- The proxy pattern allows to defer **object creation** and **object initialization** to the time the object is needed
    - Reduces the cost of accessing objects
    - The proxy acts as a stand-in for the real object
    - The proxy creates the real object only if the user asks for it
    - Provides location transparency

Use cases of the proxy pattern

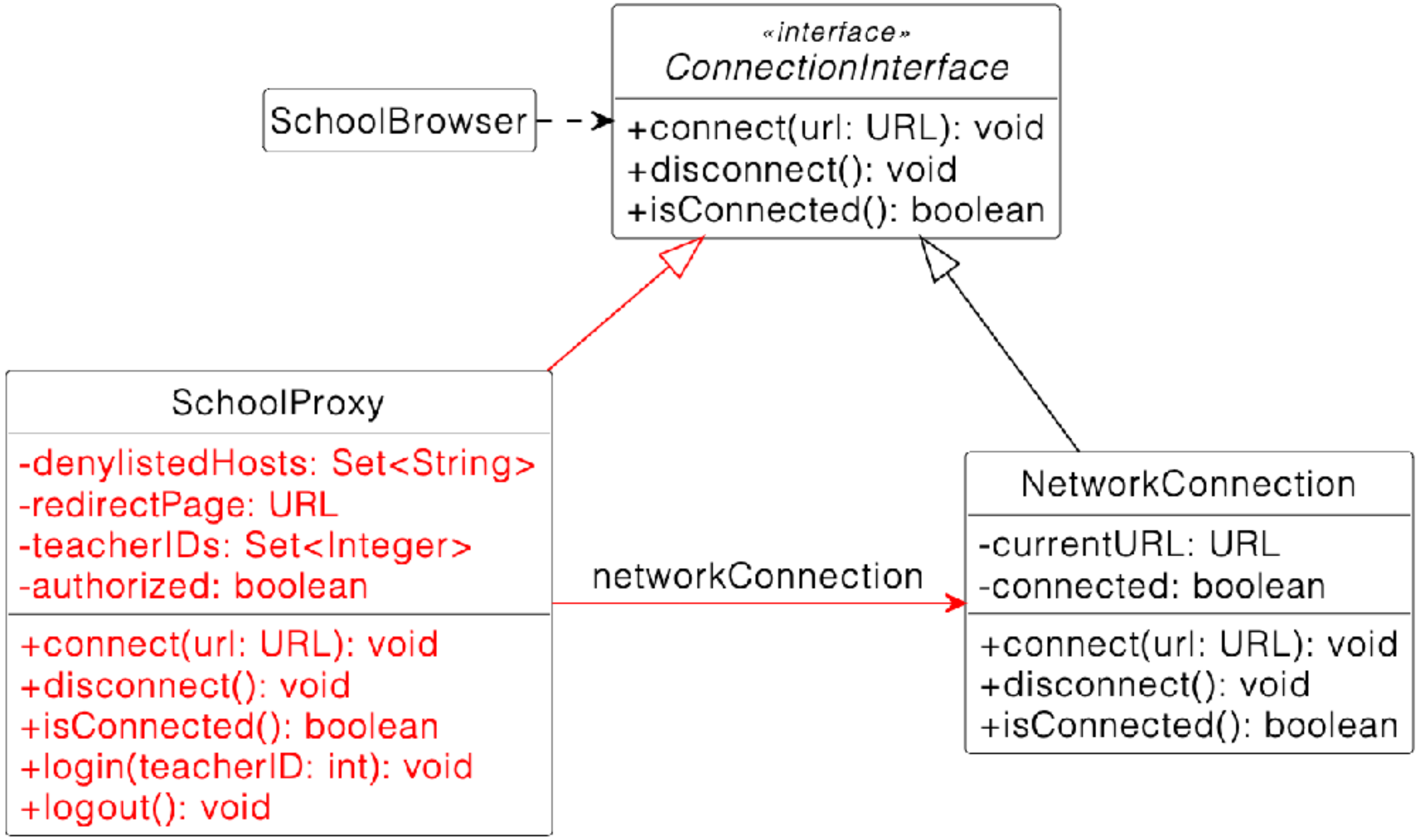
- **Caching** (remote proxy): the proxy object is a local representative for an object in a different address space
  - Caching is good if information does not change too often
  - If information changes, the cache needs to be flushed
  - **Example:** TUM Logo on Artemis
- **Substitute** (virtual proxy): the proxy object acts as a stand-in for an object which is expensive to create or download
  - Good for information that is not immediately accessed
  - Useful for objects that are not visible (not in line of sight, far away)
  - **Example:** Google Maps, Fog of War
- **Access Control** (protection proxy): the proxy object provides access control to the real object
  - Beneficial when different objects should have different access and viewing rights
  - **Example:** grade information shared by administrators, teachers, and students

Access control example



Homework **H06E01**: proxy pattern

- Access control with a school browser
- **Problem**
  - Implement a school browser with a proxy for deny-listed hosts
  - Restrict students from accessing **inappropriate** websites
  - Use the proxy pattern to separate functionality from access control



Homework

- **H06E01** Proxy Pattern (programming exercise)
- **H06E02** Choose a Design Pattern (modeling exercise)
- **H06E03** Inheritance vs. Delegation (text exercise)
- Read more about **design patterns** on https://sourcemaking.com (see Literature)

→ Due until 1h before the **next lecture**

Summary

- **Inheritance** can be used in analysis as well as in object design
  - During analysis: inheritance is used to describe taxonomies
  - During object design: Inheritance is used for interface specification and reuse
- Blackbox vs. whitebox **reuse**: composition vs. inheritance
- Interface specification: implementation inheritance, delegation, specification inheritance
- Discovering inheritance: generalization and specialization
- **Design patterns**
  - Provide solutions to common problems
  - Lead to extensible models and reusable code
  - Structural patterns, behavioral patterns, creational patterns
  - Composite pattern, bridge pattern, proxy pattern

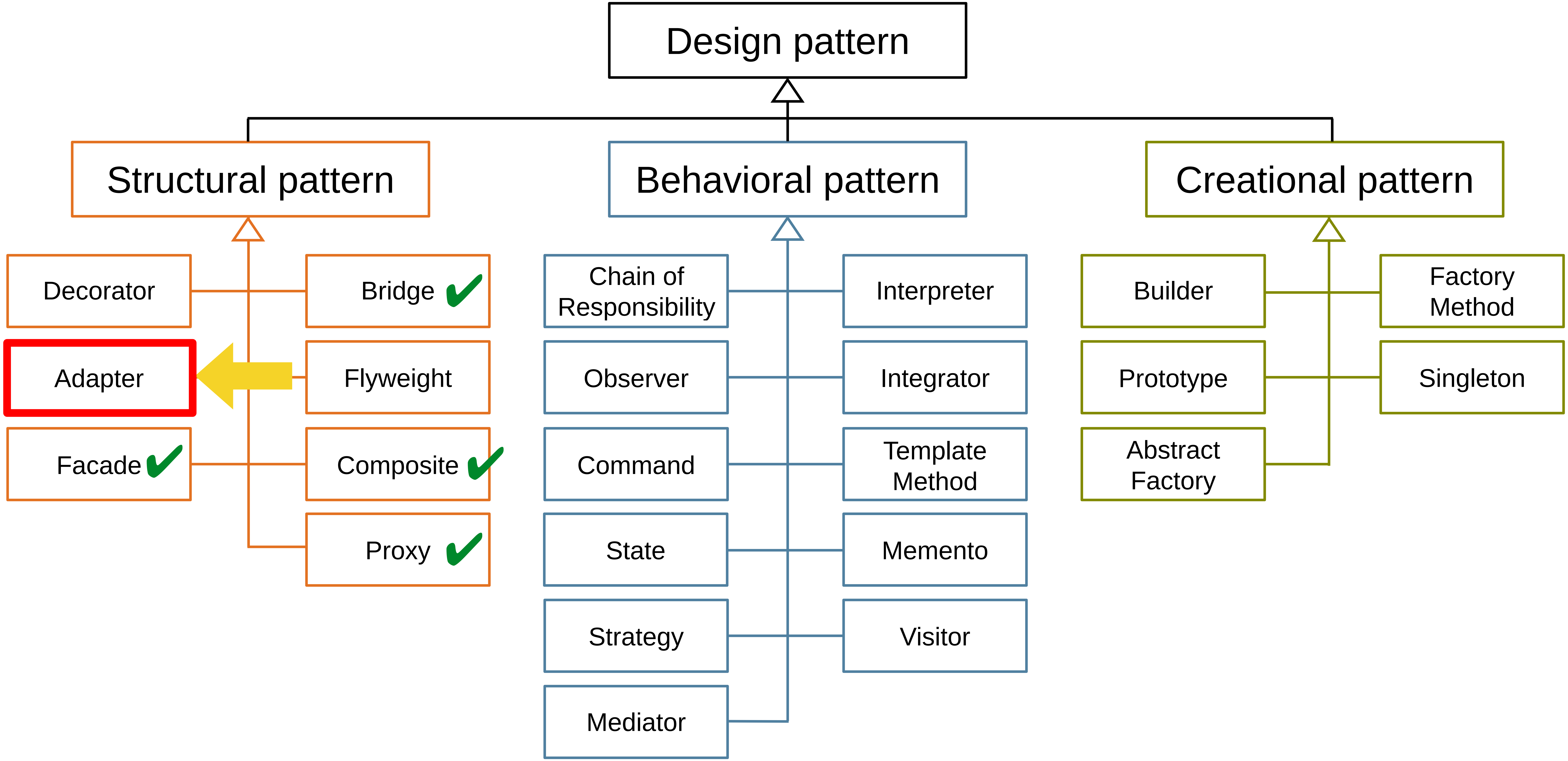
Literature

- Design Patterns. Elements of Reusable Object-Oriented Software – Gamma, Helm, Johnson & Vlissides
- Pattern-Oriented Software Architecture, Volume 1, A System of Patterns - Buschmann, Meunier, Rohnert, Sommerlad, Stal
- Pattern-Oriented Analysis and Design - Composing Patterns to Design Software Systems - Yacoub & Ammar
- https://sourcemaking.com

Outline

- **Adapter pattern**
- Observer pattern
- Winners of the Bumpers competition
- University course evaluation
- Strategy pattern

Design patterns taxonomy



Example: accessing a power charger

Scenario: Stephan is using a phone that requires power

Problem: Stephan's phone battery is empty, he has access to a US Charger that offers 110 Volt charging

Challenge: provide power to the US Charger in Germany

New system

?

Existing system

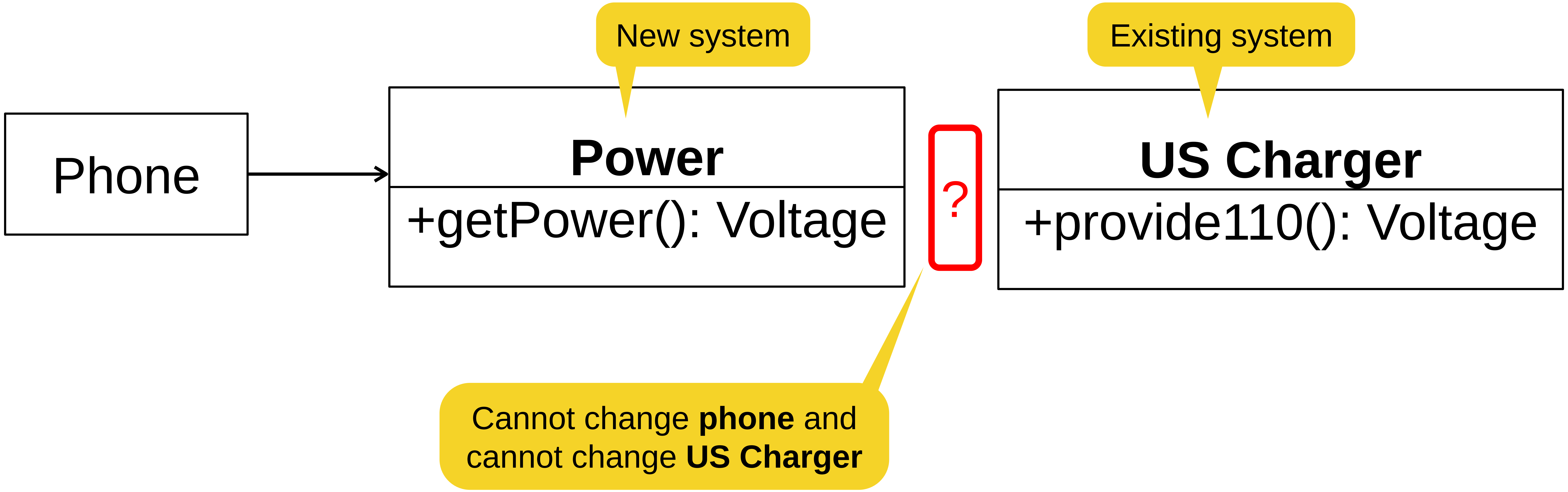
Cannot change **phone** and cannot change **US Charger**

Example: accessing a power charger

Scenario: Stephan is using a phone that requires power via the **`getPower()`** method

Problem: Stephan's phone battery is empty, he has access to a US Charger that offers 110 Volt charging via the **`provide110()`** method

Challenge: provide access to the US Charger class from the power class

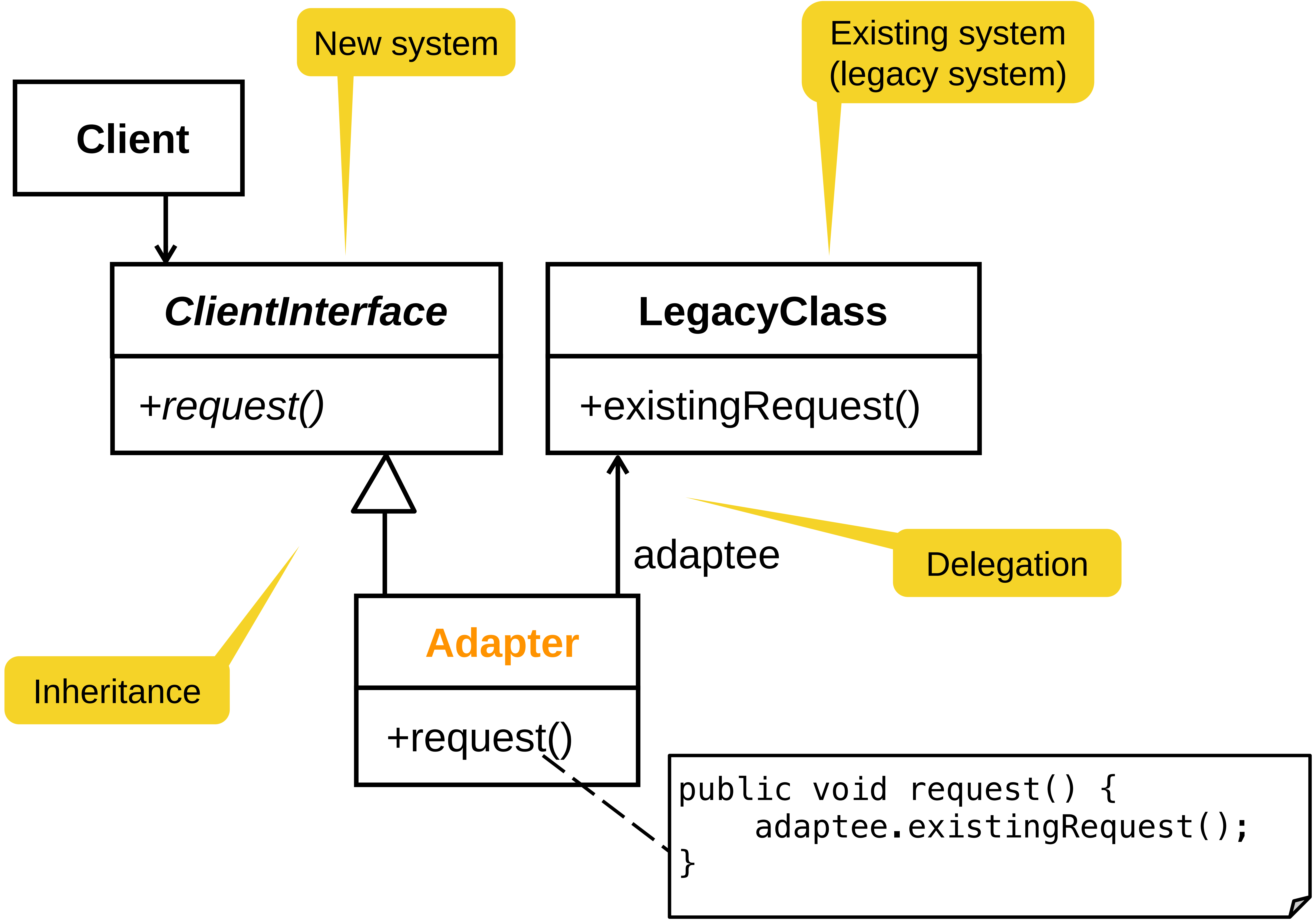


Adapter pattern

- **Problem:** an existing component offers functionality, but is not compatible with the new system being developed
- **Solution:** the adapter pattern connects incompatible components
  - Allows the reuse of existing components
  - Converts the interface of the existing component into another interface expected by the calling component
  - Useful in **interface engineering** projects and in **reengineering** projects
  - Often used to provide a new interface for a legacy system

→ Also called **wrapper**

Adapter pattern



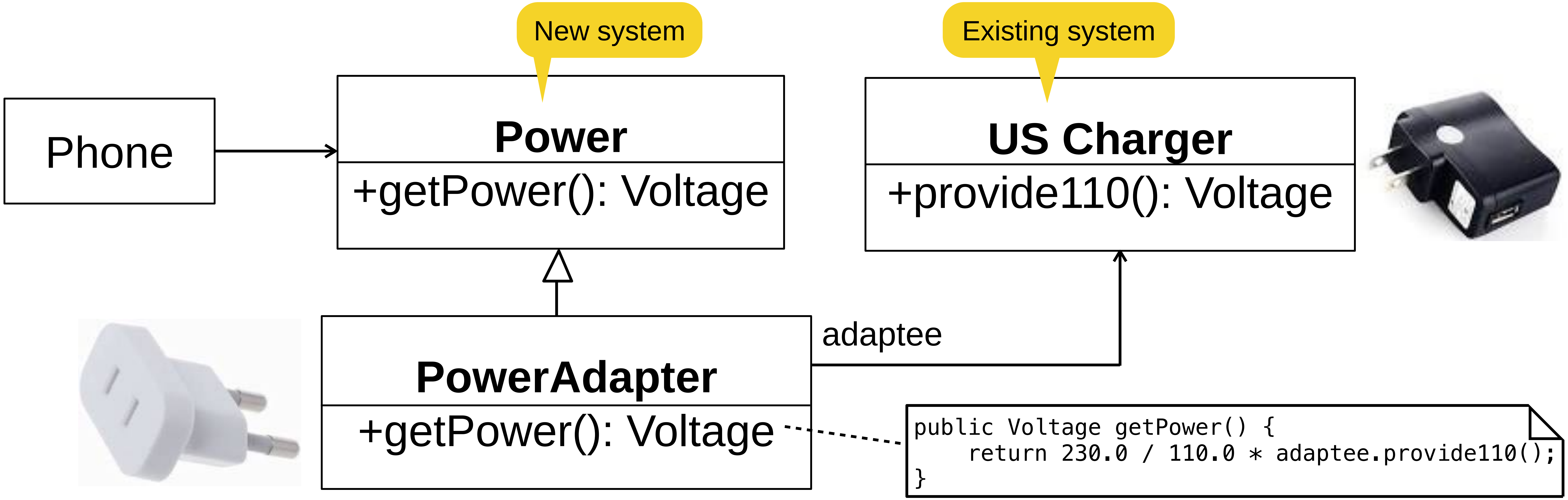
Example: accessing a power charger

Scenario: Stephan is using a phone that requires power via the **`getPower()`** method

Problem: Stephan’s phone battery is empty, he has access to a **`US Charger`** that offers 110 Volt charging via the **`provide110()`** method

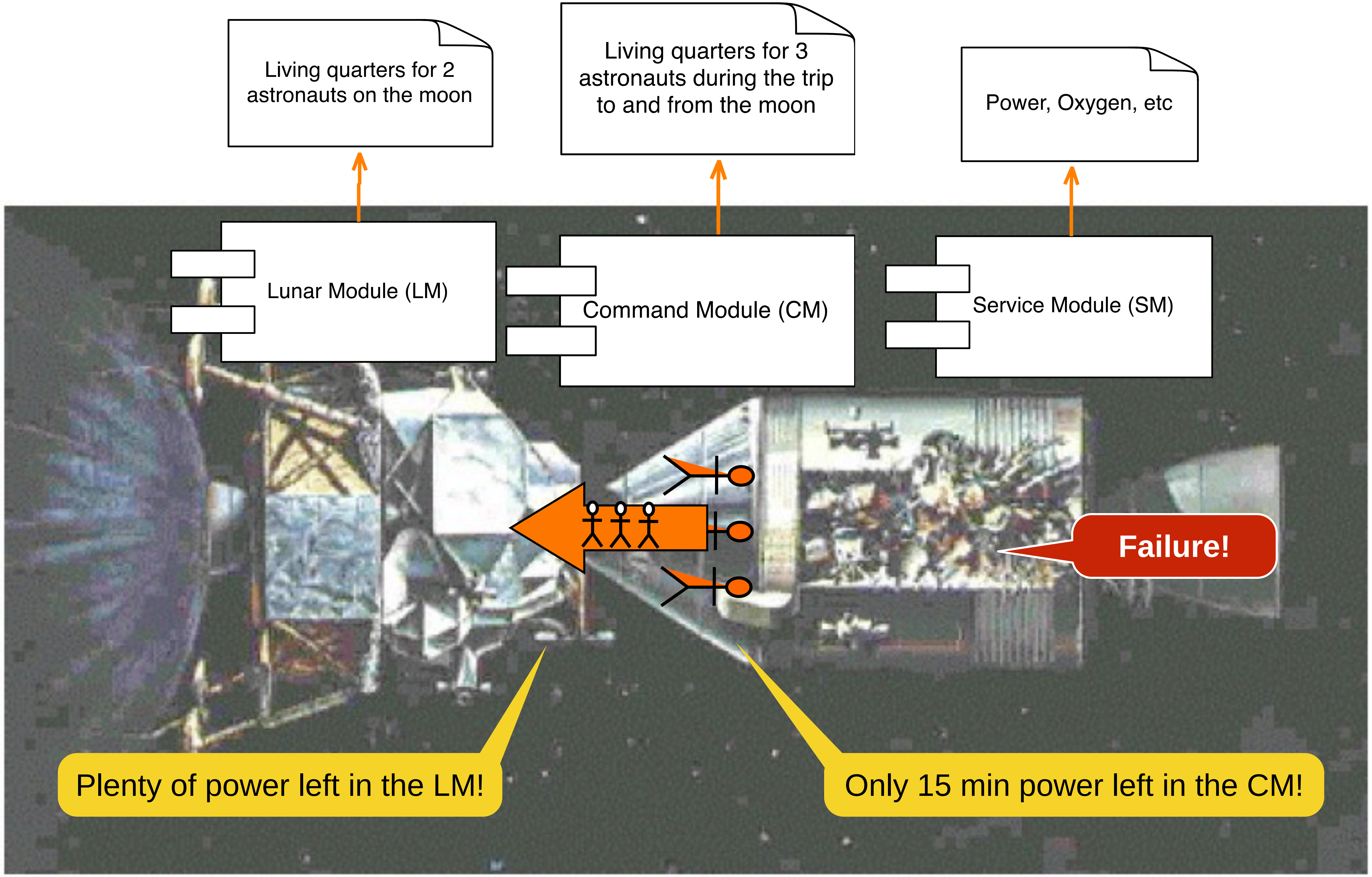
Challenge: provide access to the **`US Charger`** class from the **`Power`** class without changing the interface

Solution: use the adapter pattern to connect to the **`US Charger`**



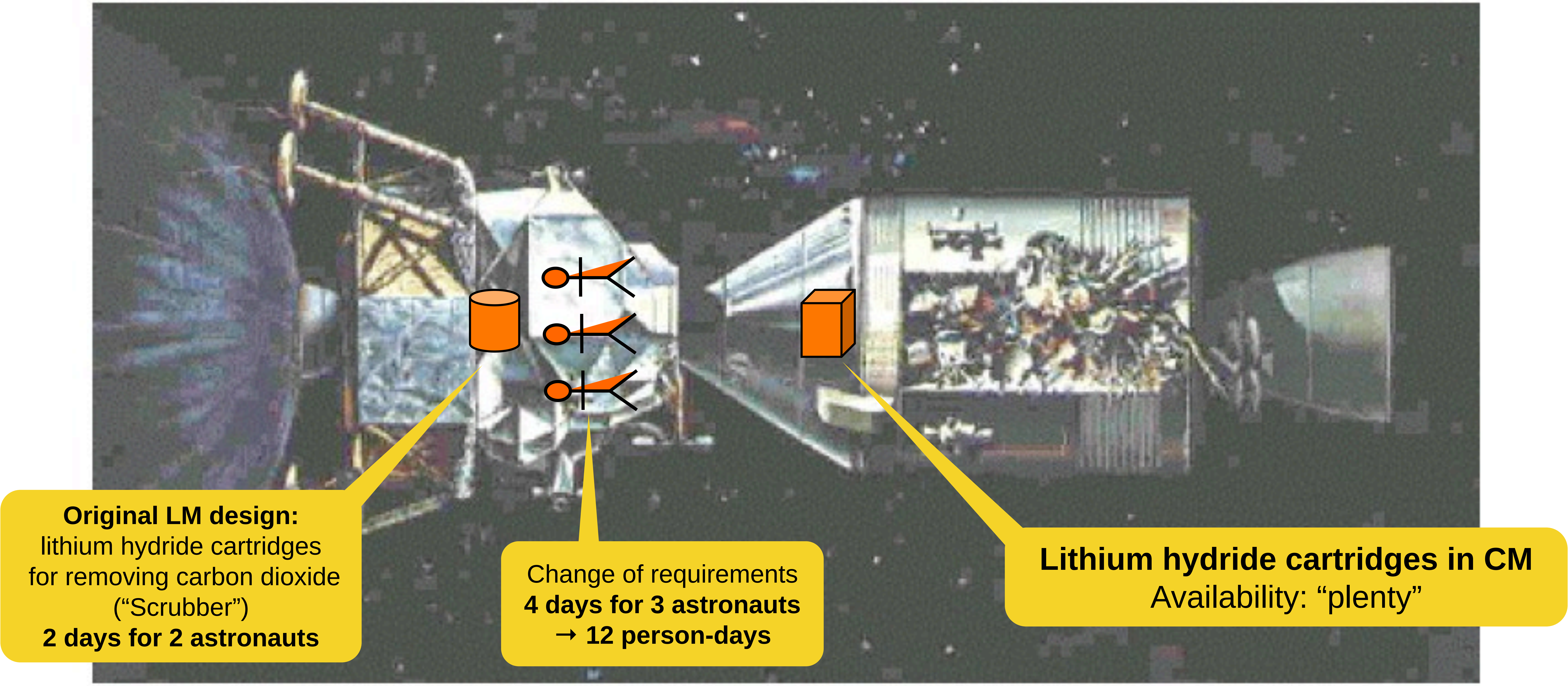
Another adapter pattern example

“Houston, we’ve had a problem!”



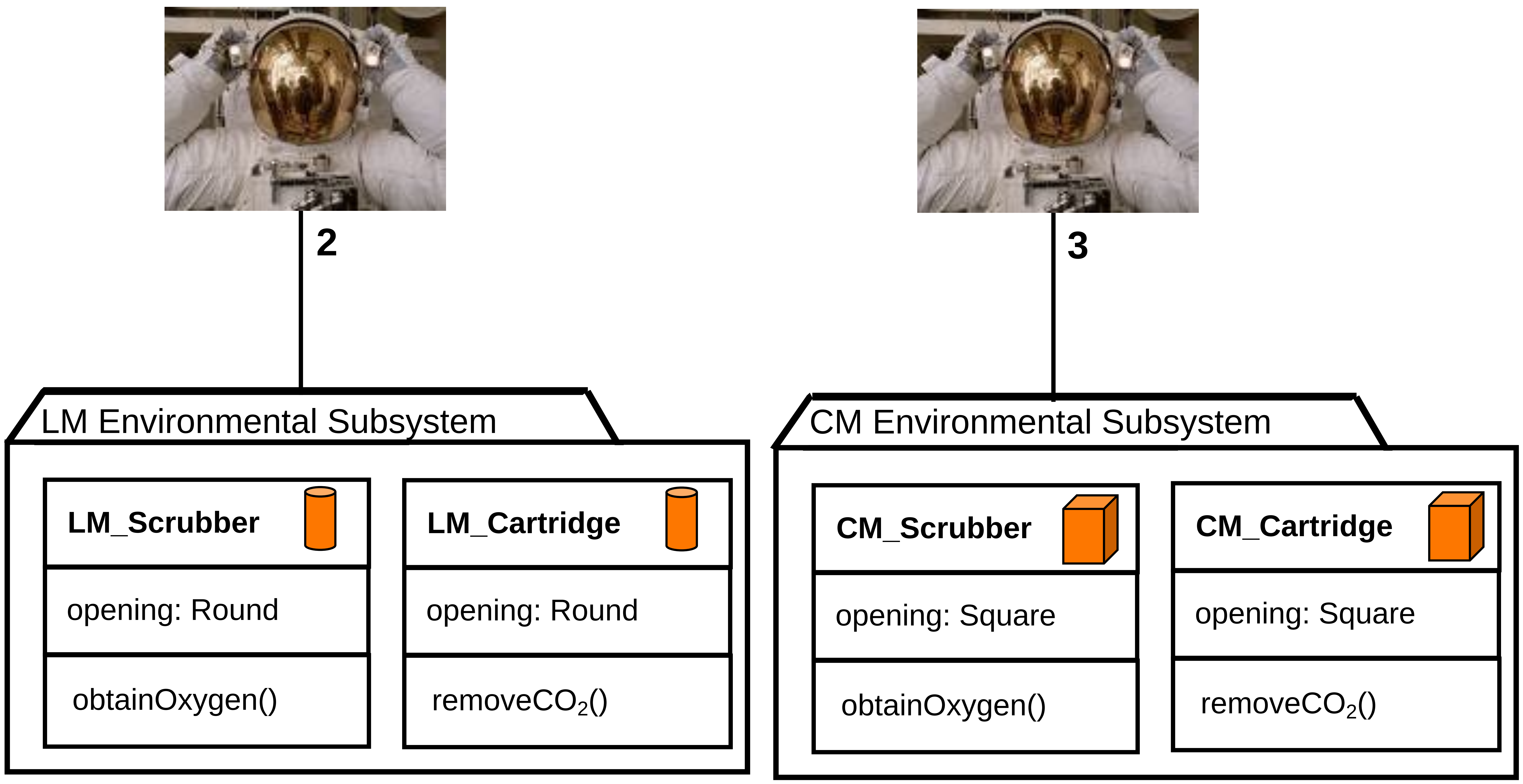
Subsystem decomposition of the Apollo 13 spacecraft

Apollo 13: “Houston, we’ve had a problem!”



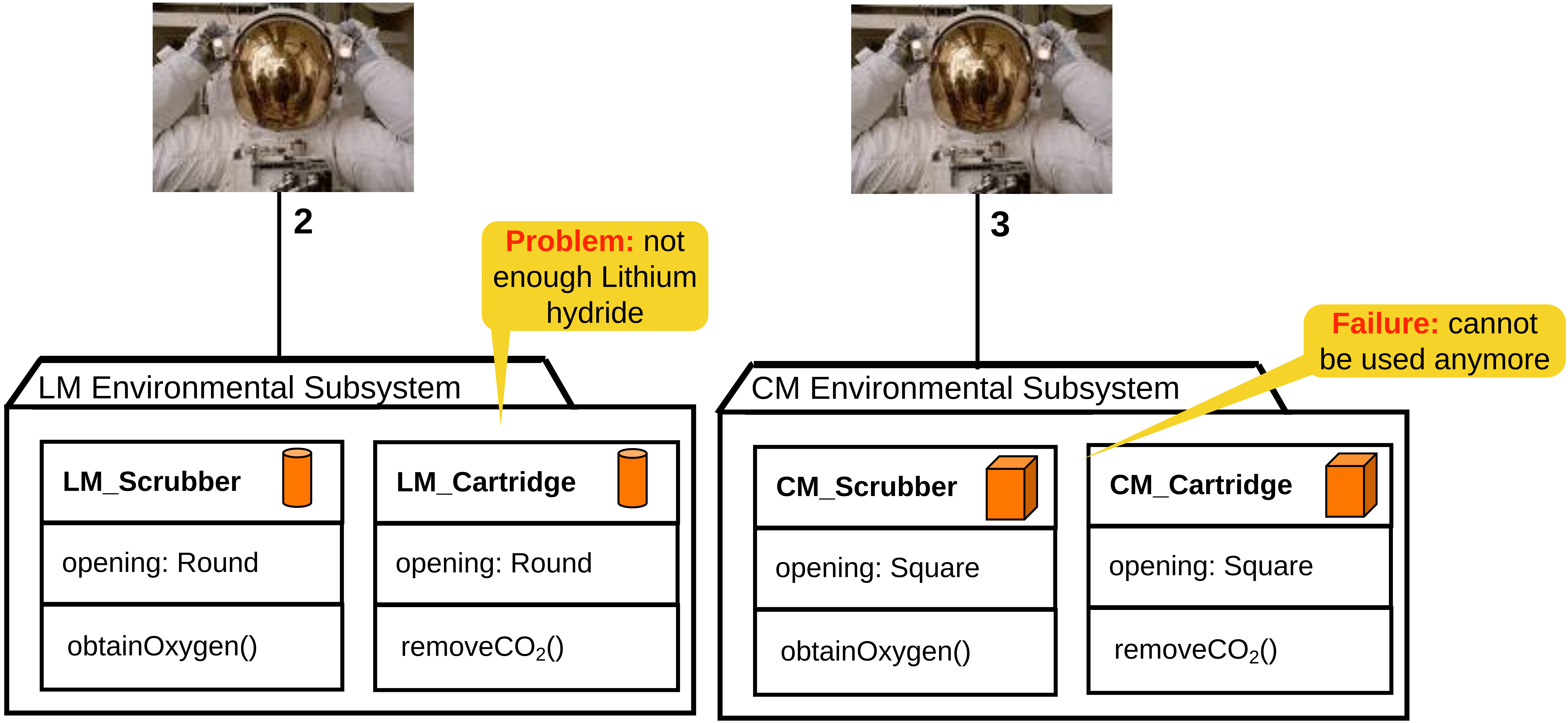
The LM was designed for 2 astronauts staying 2 days on the moon (4 person-days)
Redesign challenge: how can the LM be used for 12 person-days (reentry into Earth)?
Proposal from mission control: “use the lithium hydride cartridges from the CM to extend life in LM”
Problem: cartridges in CM are incompatible with the cartridges in the LM subsystem!

Original design of the Apollo 13 environmental system



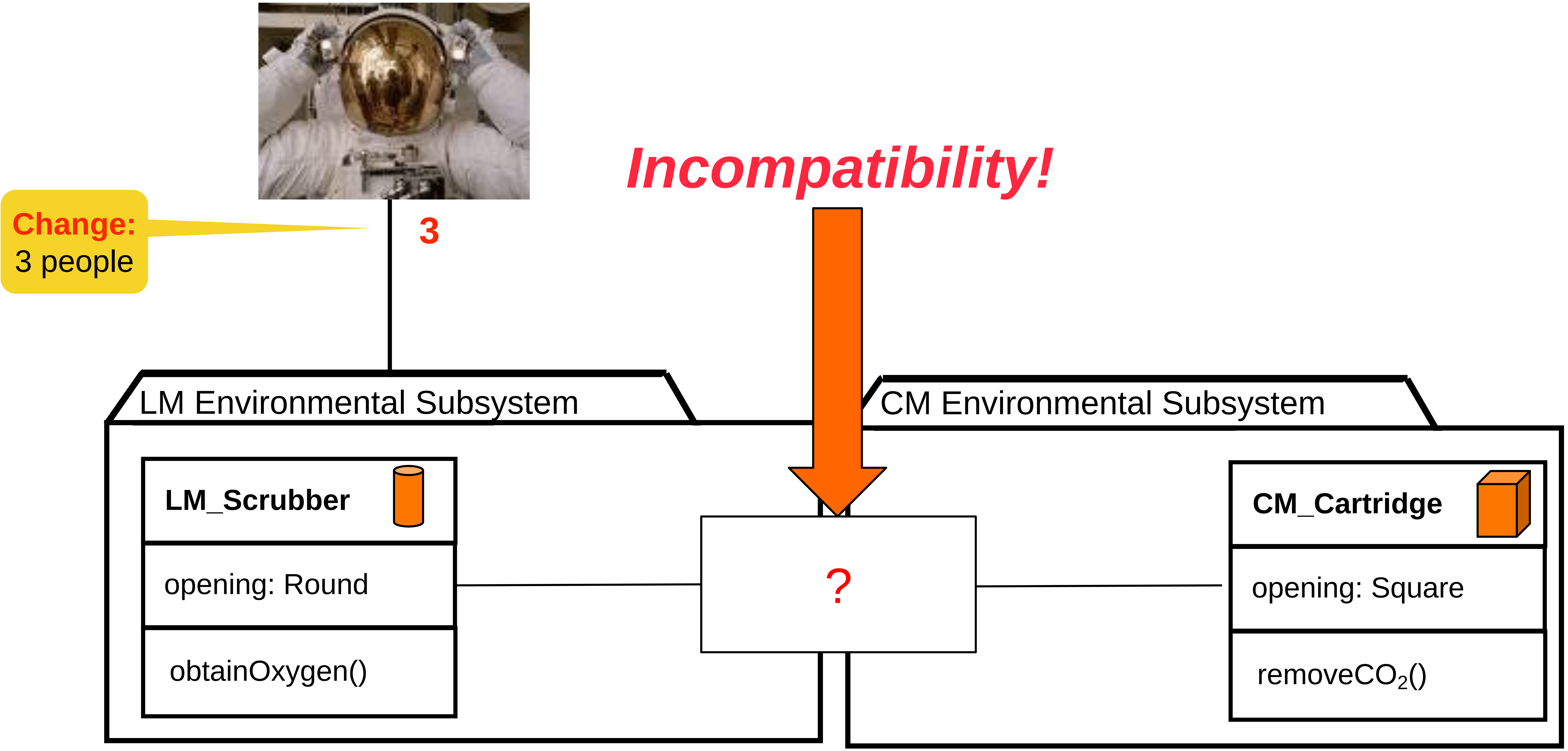
Lunar Module (LM) **Command Module (CM)**

Change!



Lunar Module (LM) **Command Module (CM)**

Can we connect the LM_Scrubber with the CM_Cartridge?

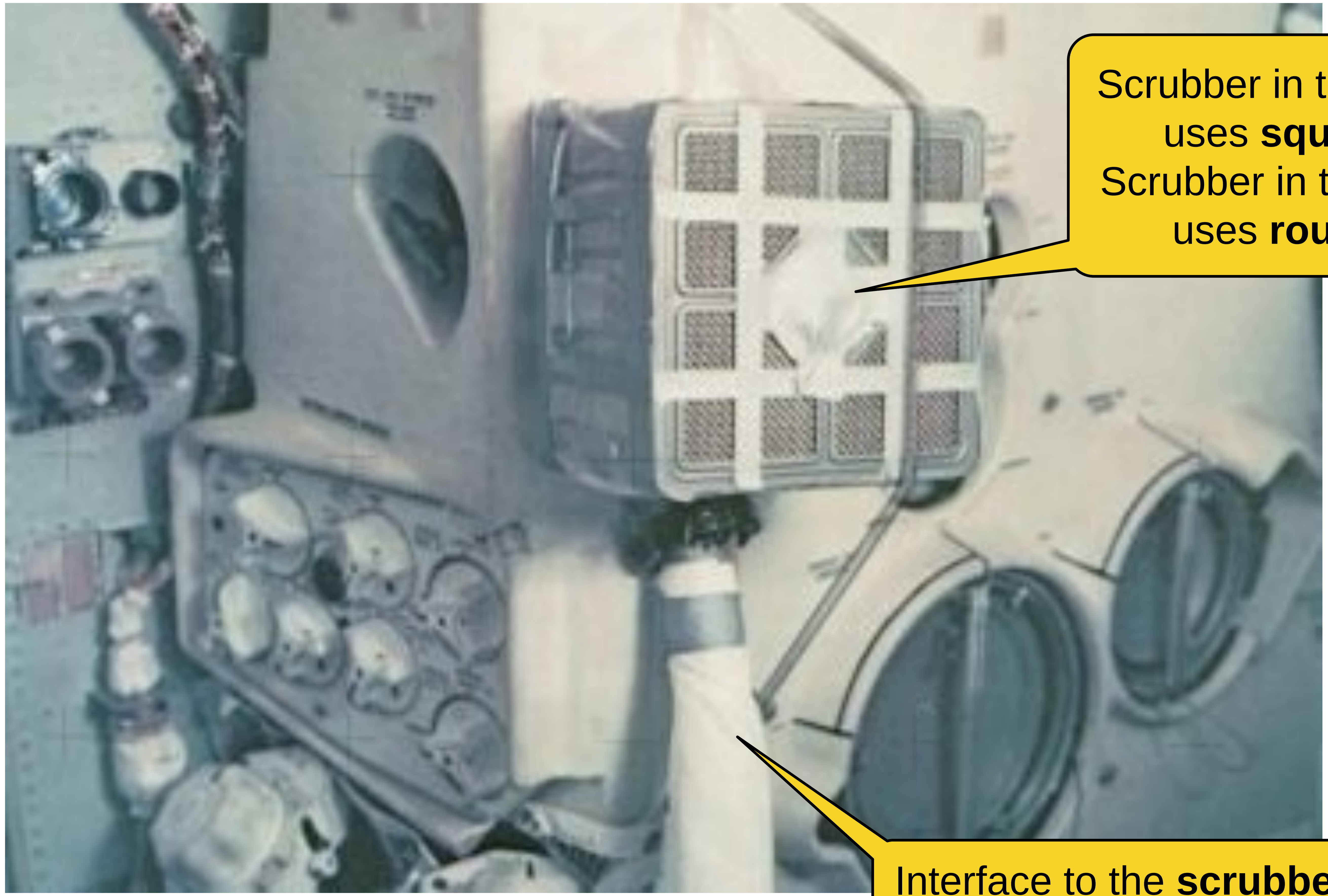


Lunar Module (LM) **Command Module (CM)**

Apollo 13: “Fitting a square peg in a round hole”

Source: http://www.hq.nasa.gov/office/pao/History/SP-350/ch-13-4.html

Object design challenge: Connecting incompatible components

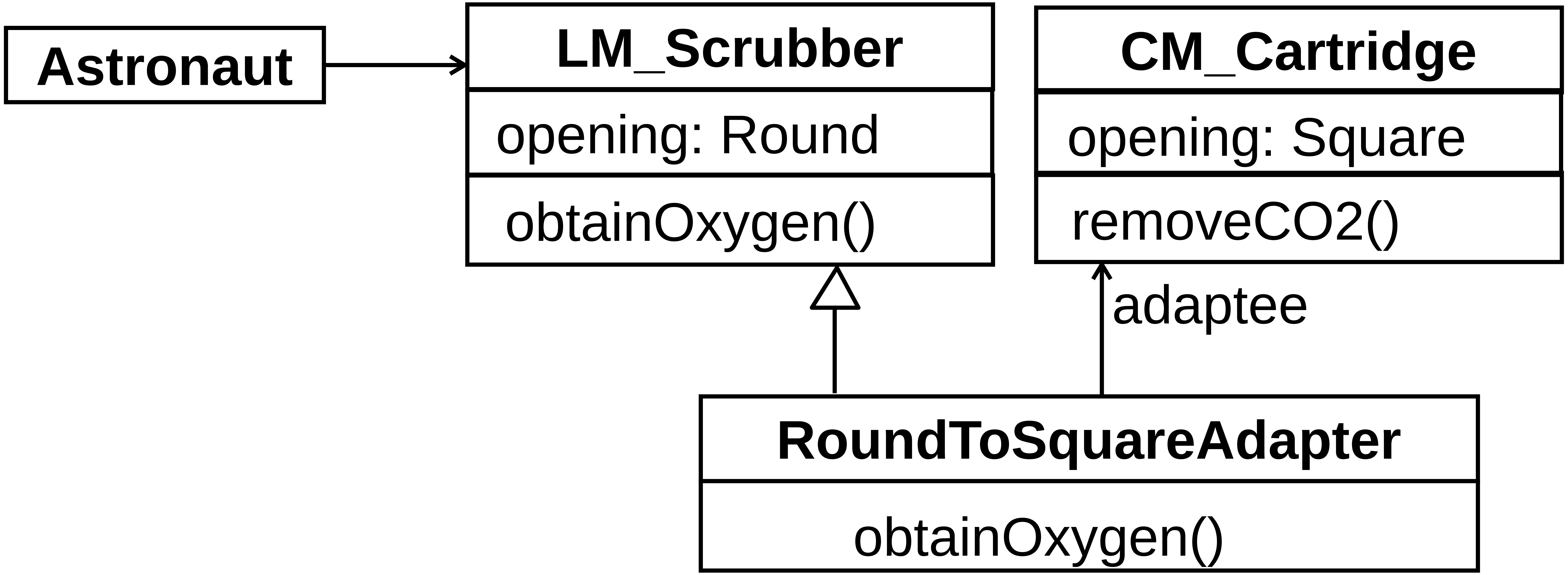


Scrubber in the CM subsystem uses **square openings**
Scrubber in the LM subsystem uses **round openings**

Interface to the **scrubber** in the LM subsystem

Source: http://www.hq.nasa.gov/office/pao/History/SP-350/ch-13-4.html

Adapter for scrubber in lunar module



➡ **Solution:** A carbon dioxide scrubber (round opening) in the lunar module LM using square cartridges from the command module CM (square opening)

Definition: legacy system

- An old system that continues to be used, even though newer technology or more efficient methods are now available
  - Evolved over a long time
  - Still actively used in a production environment
- Often designed without modern software design methodologies

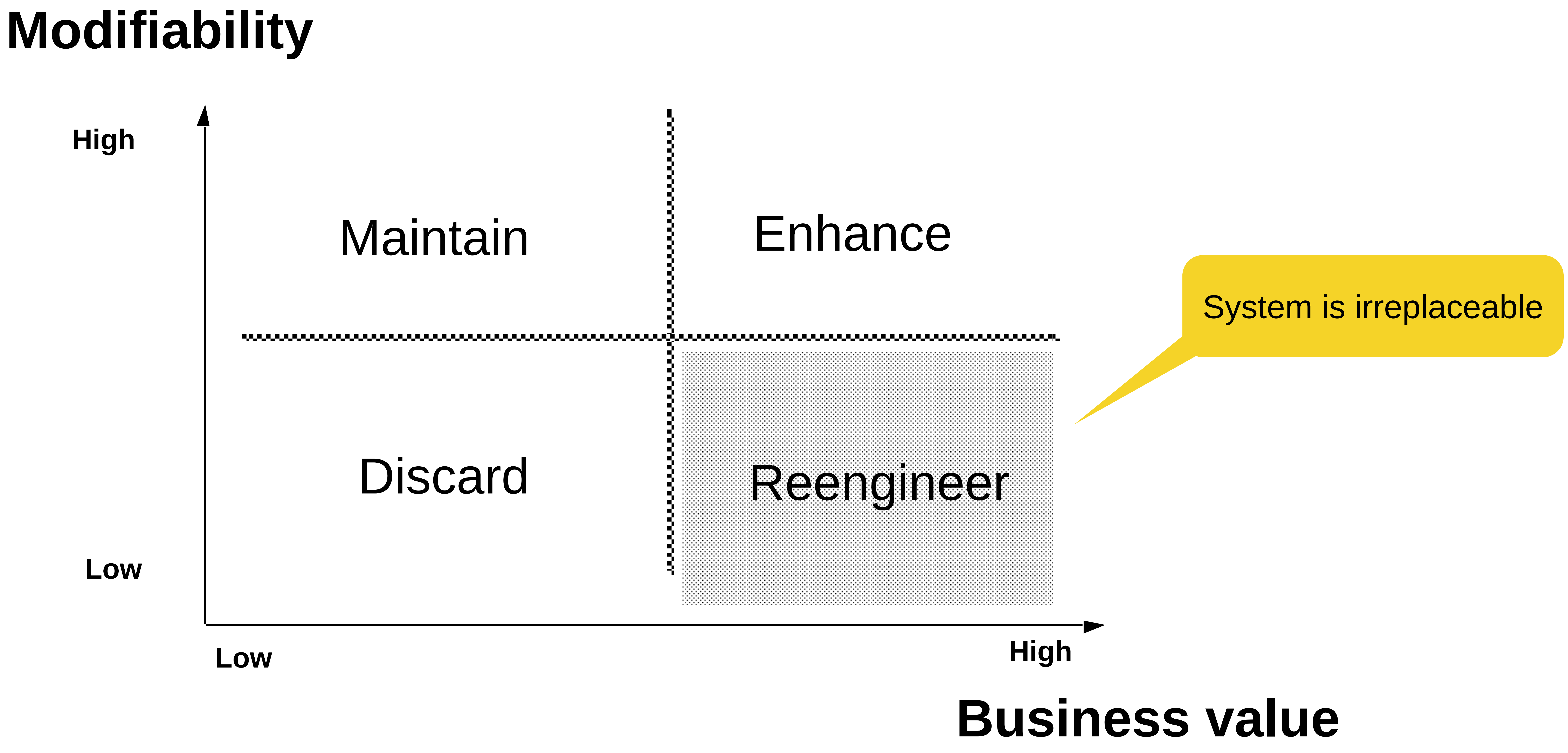
  → High maintenance cost
- Considered irreplaceable because a re-implementation is too expensive or impossible

Problems with legacy systems

- Reasons for the continued use of a legacy system
  - **System cost:** the system still makes money, but the cost of designing a new system with the same functionality is too high
  - **Poor engineering (or poor management):** the system is hard to change because the compiler is no longer available or source code has been lost
  - **Availability:** the system requires 100% availability and cannot simply be taken out of service and replaced with a new system
  - **Pragmatism:** the system is installed and working

- **But:** change is required due to new functional-, nonfunctional- or pseudo requirements

What to do with legacy systems?



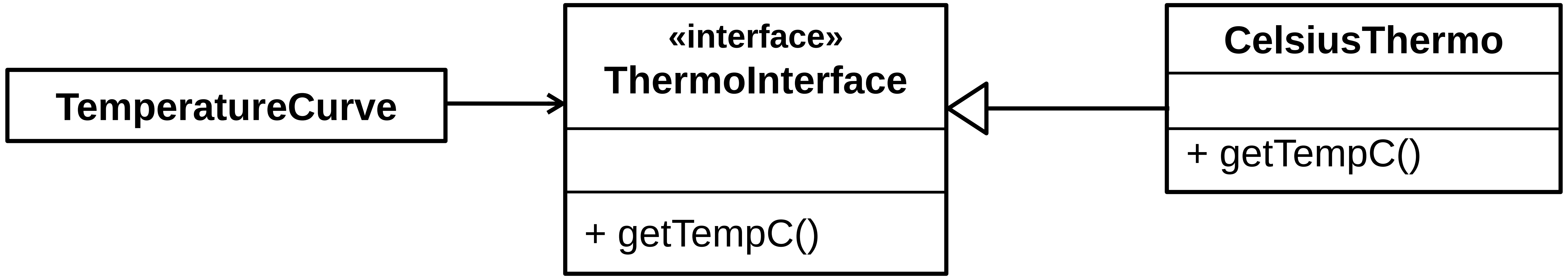
Comparison: adapter pattern vs. bridge pattern

- **Similarities**
  - Both hide the details of the underlying implementation
- **Differences**
  - Adapter: designed towards making unrelated components work together
    - Applied to systems that are already designed (reengineering, interface engineering projects)
    - **Inheritance → delegation**
  - Bridge: used up-front in a design to let abstractions and implementations vary independently
    - **Greenfield engineering** of an “extensible system”
    - New “beasts” can be added to the “zoo” (“application and solution domain zoo”), even if these are not known at analysis or system design time
    - **Delegation → inheritance**

Exercise: adapter pattern

Problem: replace a broken thermometer

- You are climbing Denali (6.193 m) and you need to reliably read the temperature for the last **n** hours (temperature curve) **in Celsius**
- You use a digital thermometer implemented in Java: **`TemperatureCurve`** uses **`ThermoInterface`**
- It connects to **`CelsiusThermo`** which provides the temperature in **Celsius**



- Somebody **broke** the Celsius thermometer (`CelsiusThermo`)
- There is one more thermometer, but it measures the temperature in **Fahrenheit**

L07E02 Adapter Pattern

Not started yet.

Start exercise

Easy

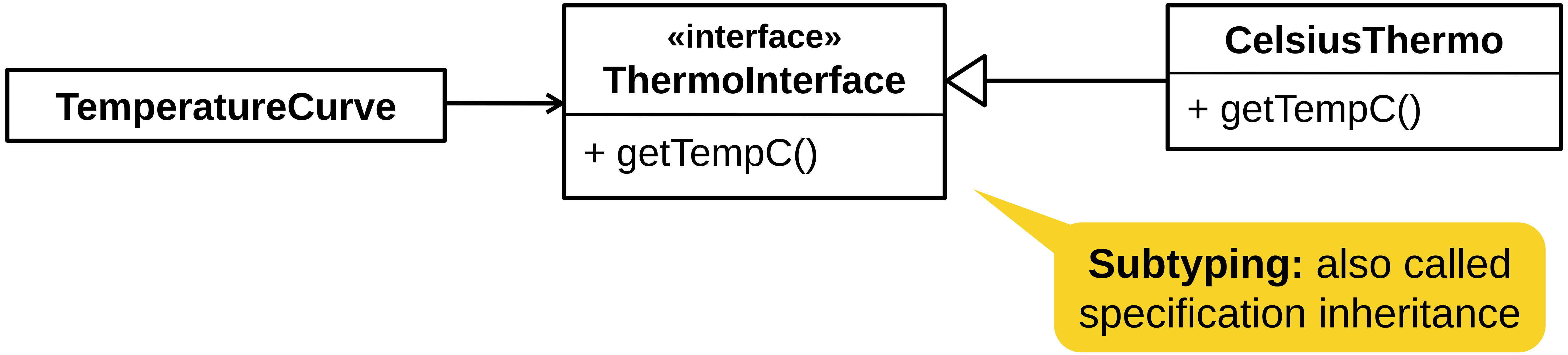
Due date: end of today

10 min

4 pts

- **Solution**
  - Write an adapter called **`ThermoAdapter`** that reuses the code from **`FahrenheitThermo`** while still providing temperatures in **Celsius** in **`TemperatureCurve`**

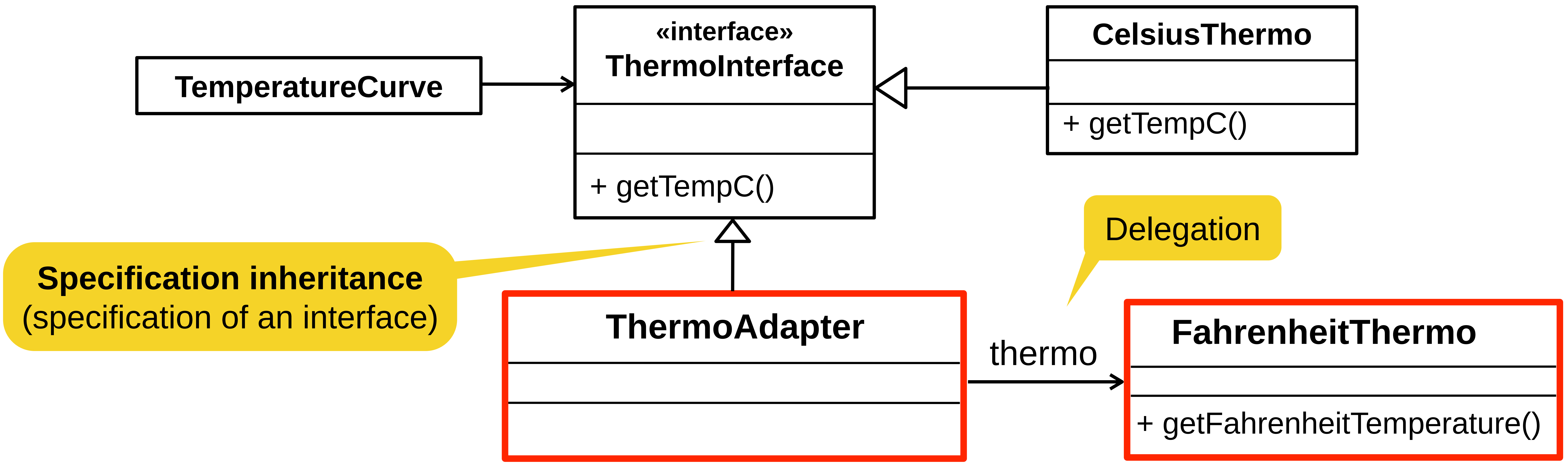
    ```
    tempCelsius = (tempFahrenheit – 32.0) * (5.0 / 9.0)
    ```
  - **Constraint**: the **`TemperatureCurve`** code should only be minimally changed
  - Call the **`getFahrenheitTemperature()`** method in the **`FahrenheitThermo`** class (delegation)



Hint: inheritance in Java

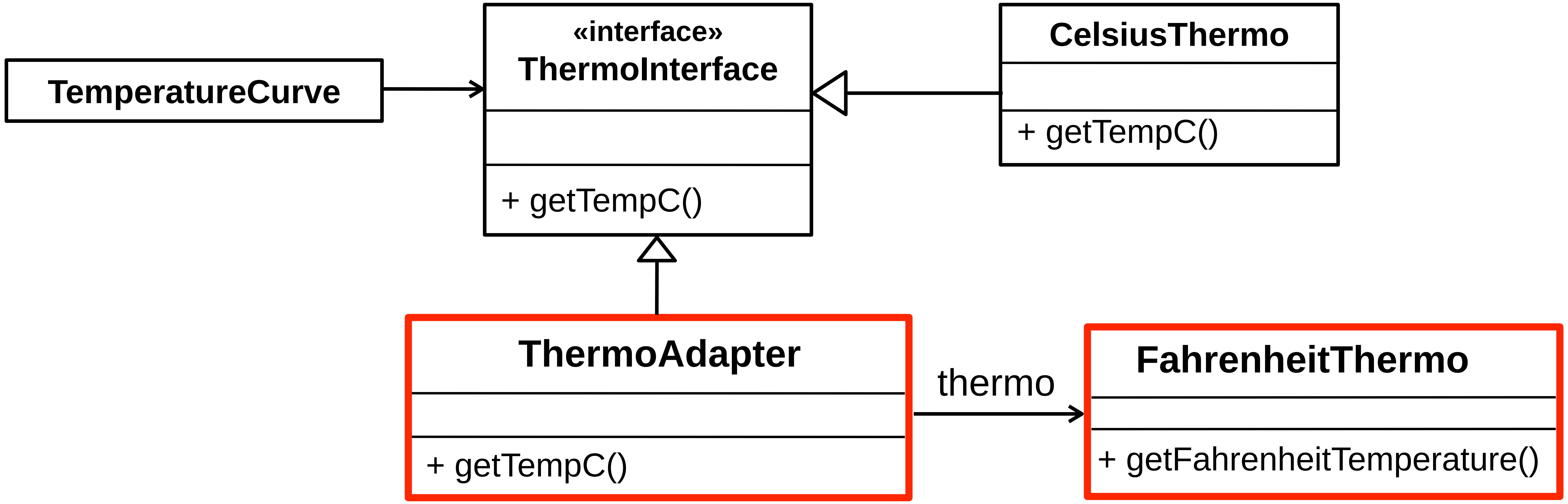
- Specification inheritance (subtyping)
  - Specification of an interface
  - Java keywords: **`abstract, interface, implements`**
- Implementation inheritance (subclassing)
  - Overriding of methods is allowed
  - No keyword necessary: overriding of methods is the default in Java
- Specialization and generalization
  - Definition of subclasses
  - Java keyword: **`extends`**
- Simple inheritance
  - Overriding of methods is not allowed
  - Java keyword: **`final`**

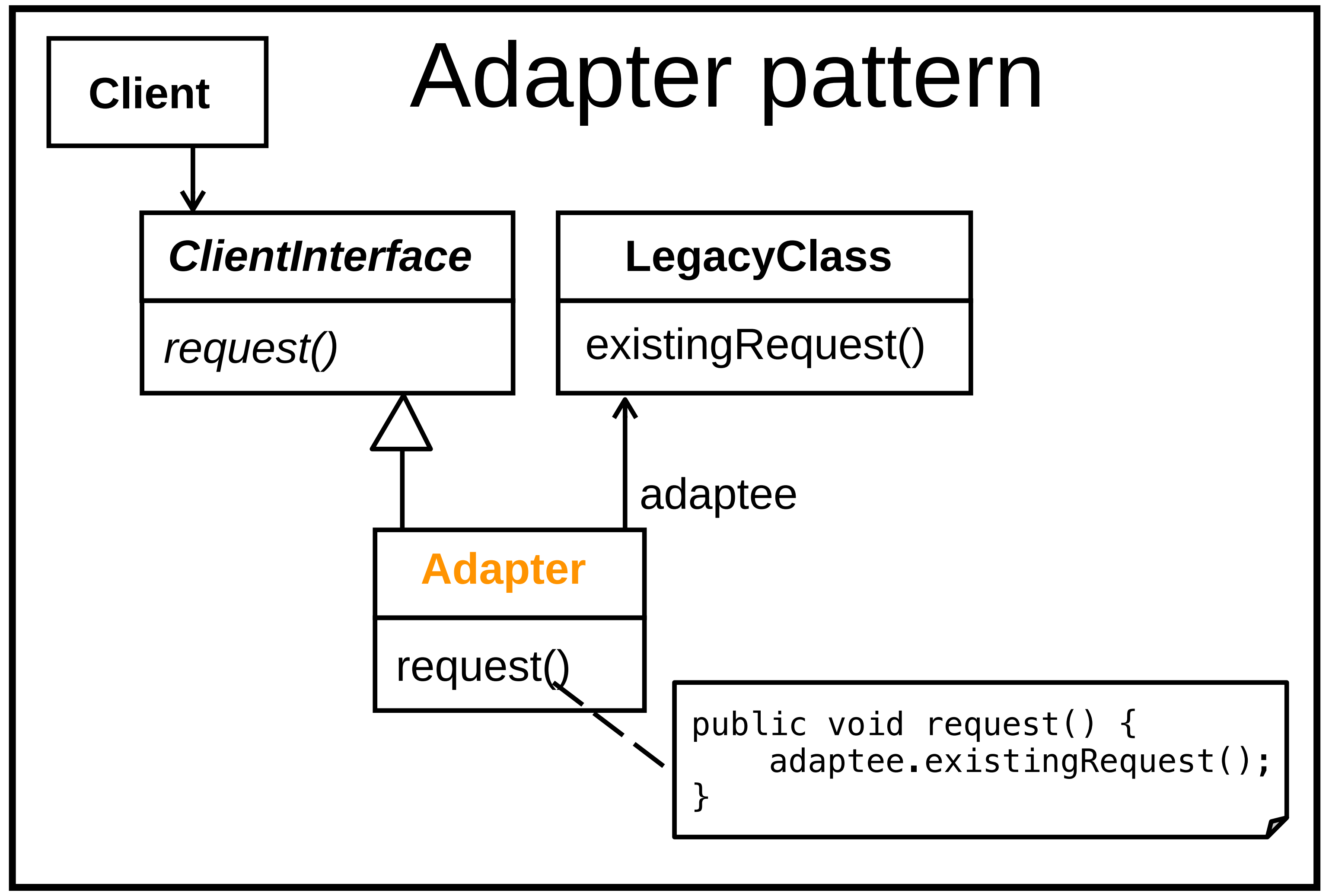
Hint: ThermoAdapter




```
public class ThermoAdapter implements ThermoInterface {
 ....
}
```

Solution: ThermoAdapter





Java Code

```java
public class ThermoAdapter implements ThermoInterface {

    private FahrenheitThermo thermo;

    public ThermoAdapter() {
        thermo = new FahrenheitThermo();
    }

    public double getTempC() {
        return (thermo.getFahrenheitTemperature() - 32.0) * (5.0 / 9.0);
    }
}
```