Introduction to Software Engineering

07 Object Design II

Stephan Krusche

14 June 2022
Technical University of Munich
https://ase.in.tum.de

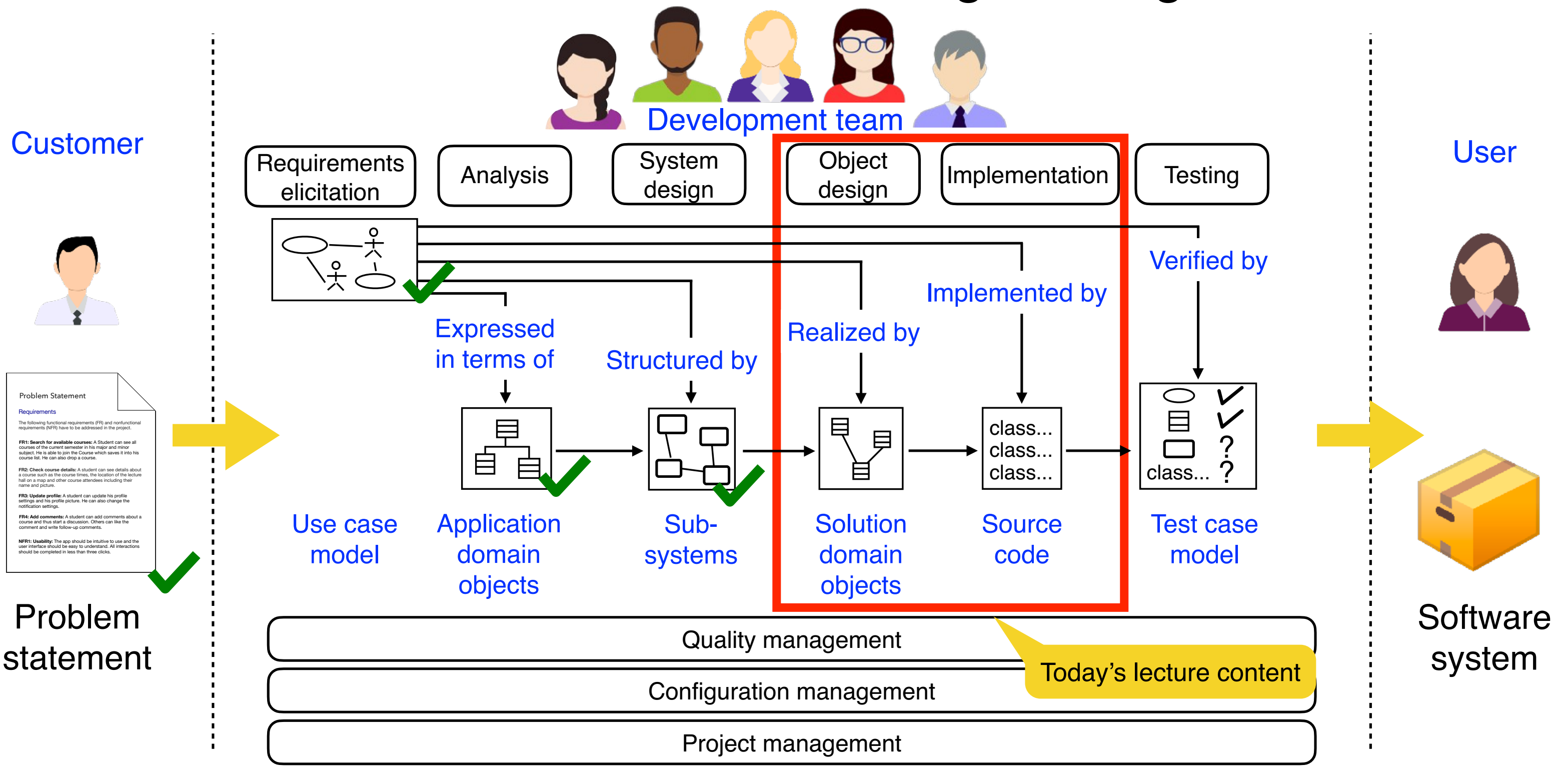
Roadmap of the lecture

- **Context and assumptions**
  - We completed requirements elicitation, analysis, and system design
  - You know the most important activities of model-based software engineering
  - You understand Scrum, UML diagrams, JavaFX, Gradle, REST, and MVC
  - You have an overview of object design activities and design patterns
- **Learning goals: at the end of this lecture you are able to**
  - Understand and apply the adapter pattern
  - Understand and apply the observer pattern
  - Understand the strategy pattern

Course schedule (Garching)

#	Date	Subject
1	26.04.22	Introduction
2	03.05.22	Model-based Software Engineering
3	10.05.22	Requirements Analysis
4	17.05.22	System Design I
5	24.05.22	System Design II
6	31.05.22	Object Design I
	07.06.22	**Holiday (no lecture, no tutor groups)**
7	14.06.22	Object Design II
8	21.06.22	Testing
	28.06.22	**no lecture, no tutor groups**
9	05.07.22	Software Lifecycle Modeling
10	12.07.22	Software Configuration Management
11	19.07.22	Software Quality Management
12	26.07.22	Project Management

Overview of model based software engineering



Problem Statement

Requirements

The following functional requirements (FR) and nonfunctional requirements (NFR) have to be addressed in the project.

FR1: Search for available courses: A Student can see all courses of the current semester in his major and minor subject. He is able to join the Course which saves it into his course list. He can also drop a course.

FR2: Check course details: A student can see details about a course such as the course times, the location of the lecture hall on a map and other course attendees including their name and picture.

FR3: Update profile: A student can update his profile settings and his profile picture. He can also change the notification settings.

FR4: Add comments: A student can add comments about a course and thus start a discussion. Others can like the comment and write follow-up comments.

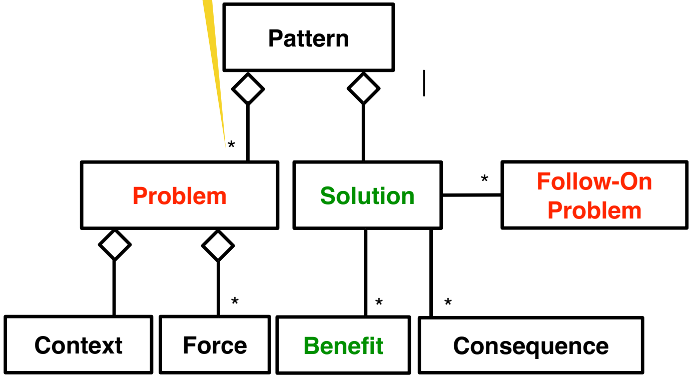
NFR1: Usability: The app should be intuitive to use and the user interface should be easy to understand. All interactions should be completed in less than three clicks.

Today's lecture content

Review: modeling a pattern in UML

- The **Problem** explains the actual situation in form of context and forces
  - The **Context** sets the stage where the pattern takes place
  - **Forces** describe why the problem is difficult to solve
- The **Solution** resolves these forces with benefits and consequences
  - **Benefits** describe positive outcomes of the solution
  - **Consequences** explain effects, results, and other outcomes of the application of the pattern
- **Follow-On Problems** can occur when you apply the solution

One type of problem, but many (slightly) different instances

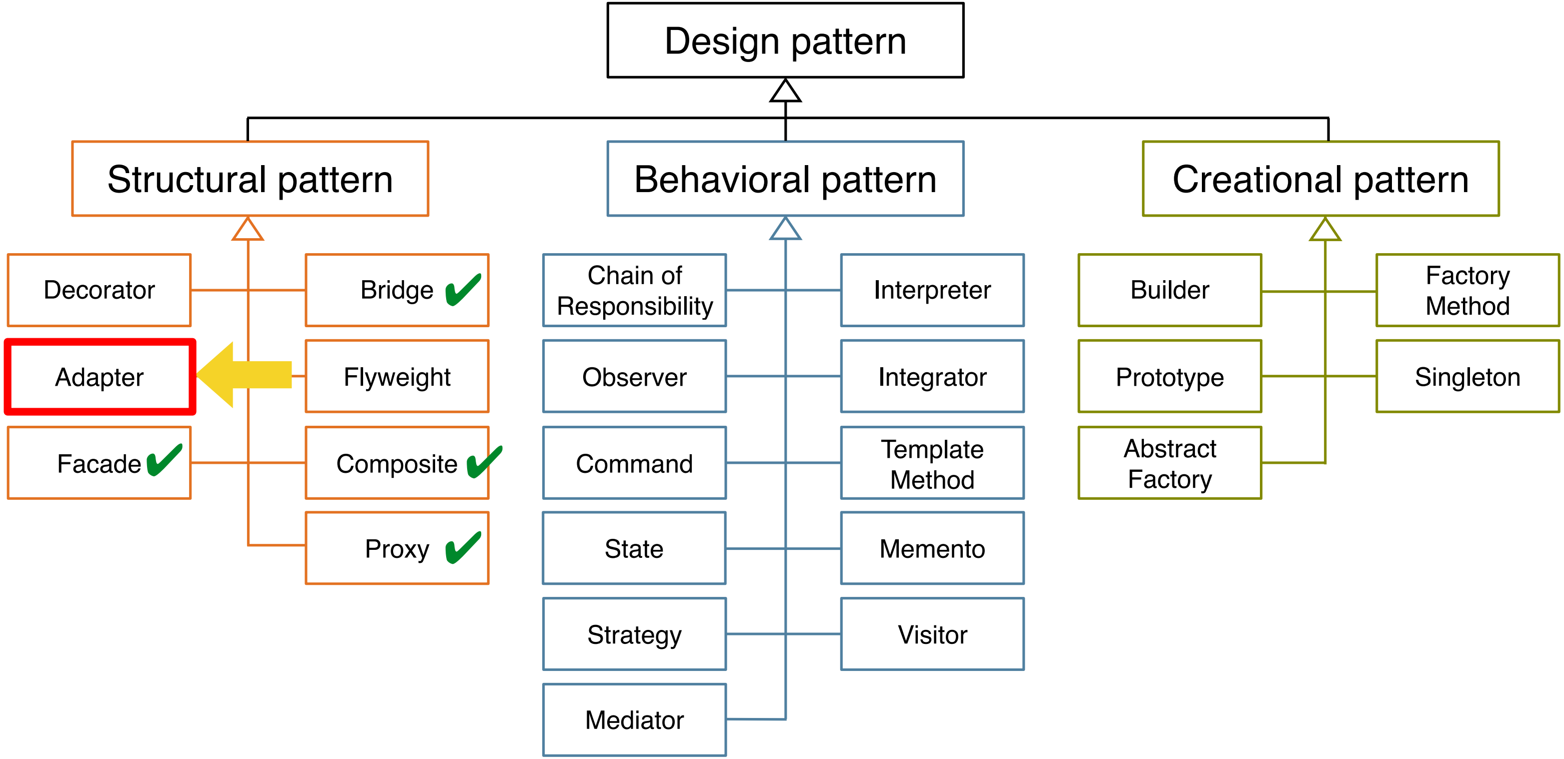


Outline

➡ **Adapter pattern**

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

Design patterns taxonomy

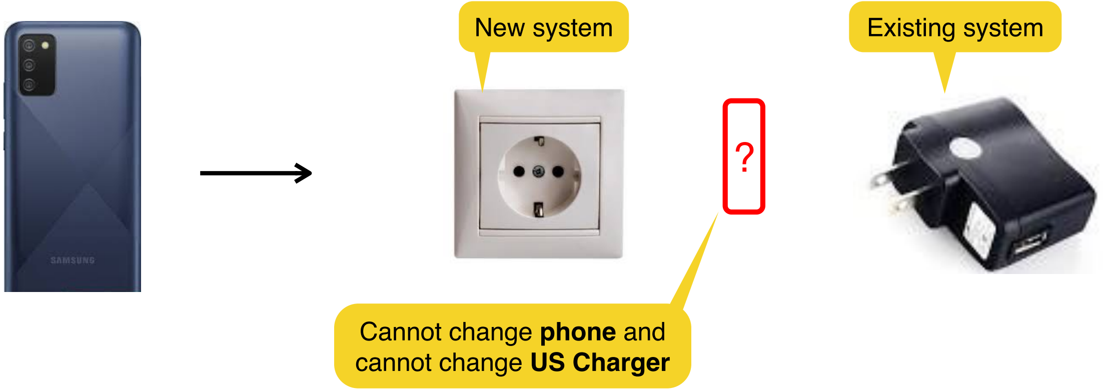


Example: accessing a power charger

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

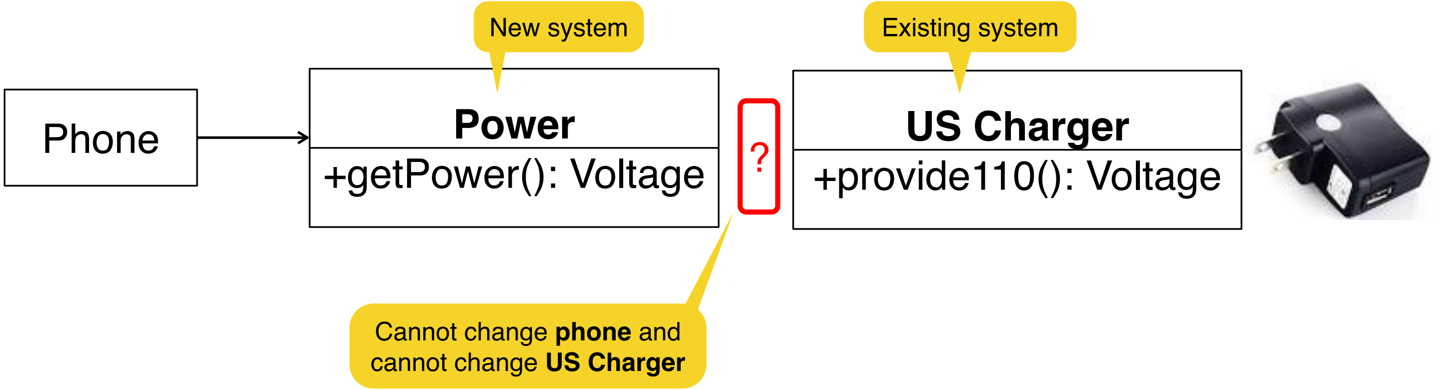
Cannot change **phone** and cannot change **US Charger**

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

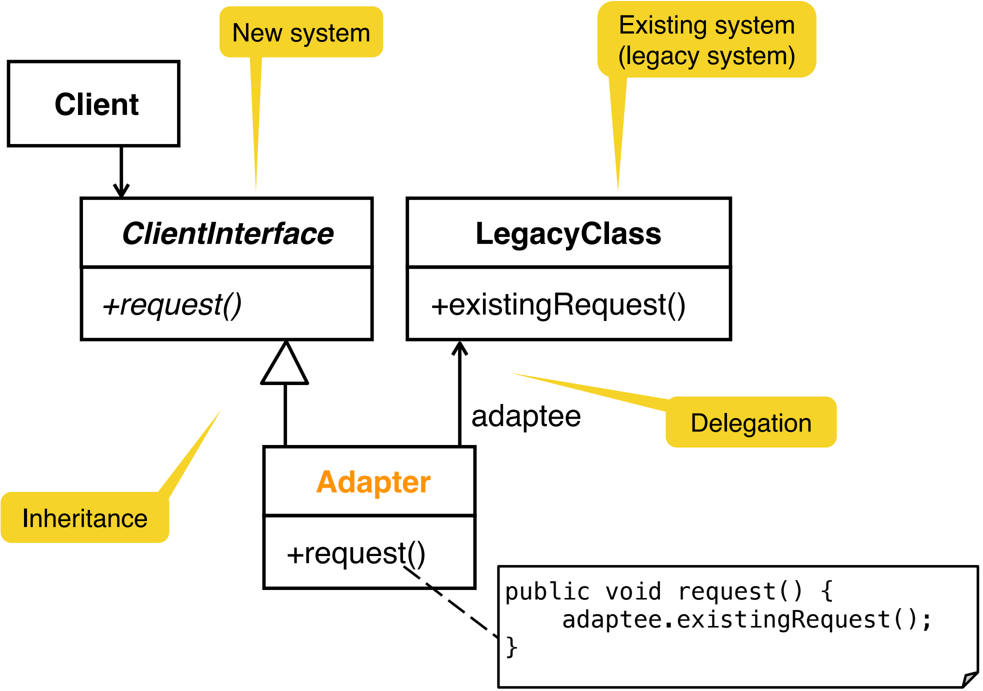


Adapter pattern

- **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



```
public void request() {
    adaptee.existingRequest();
}
```

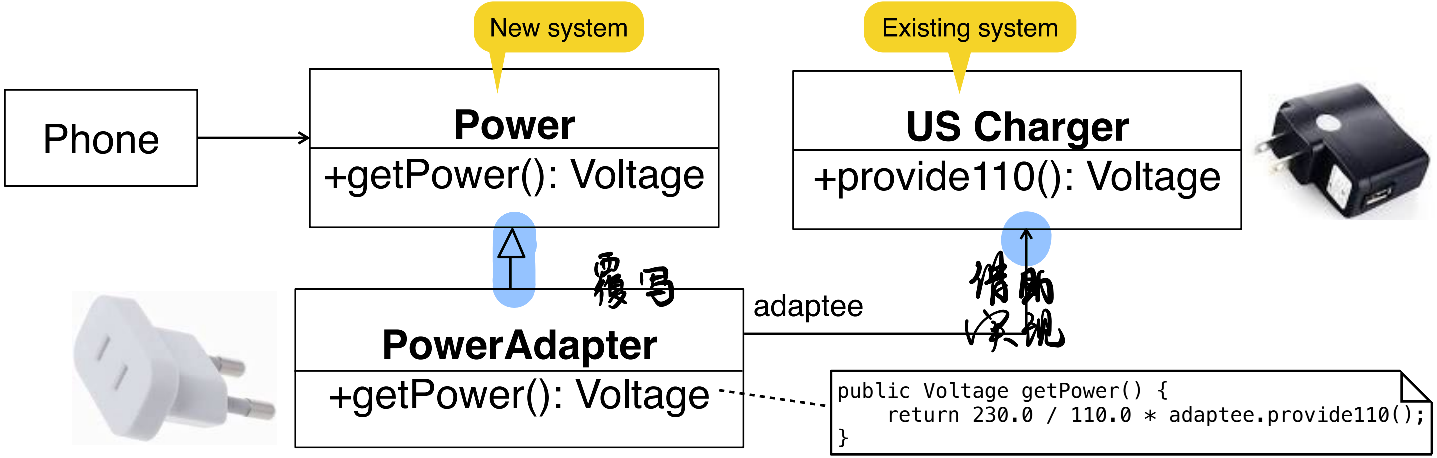
Example: accessing a power charger

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

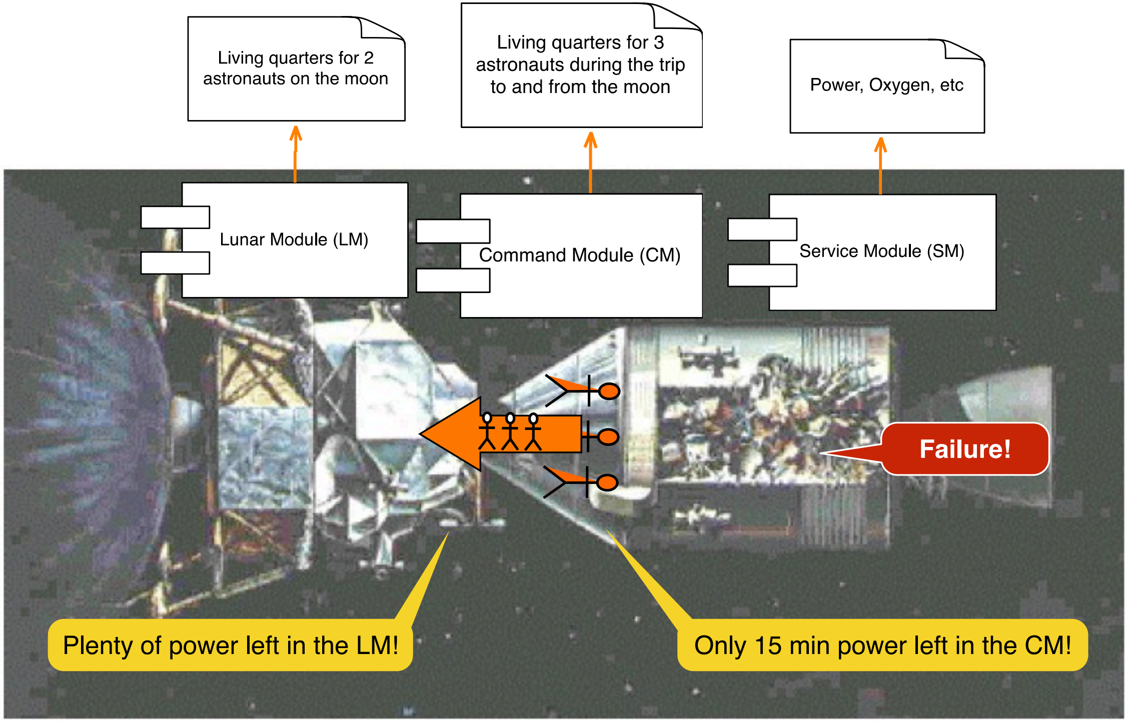
Solution: use the adapter pattern to connect to the **`US Charger`**



```
public Voltage getPower() {
    return 230.0 / 110.0 * adaptee.provide110();
}
```

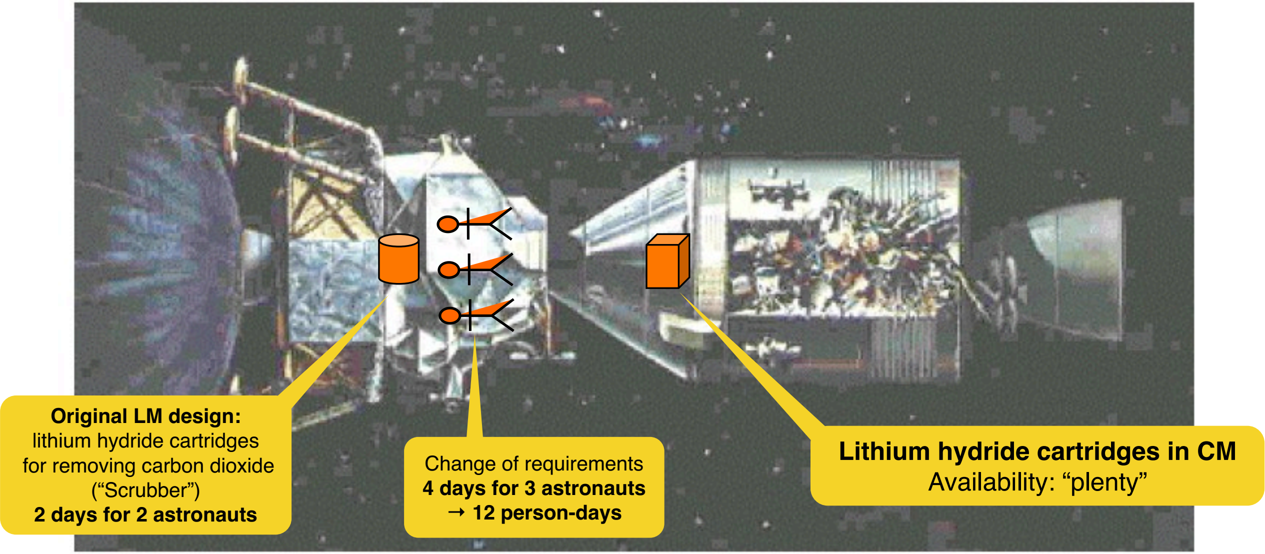
Another adapter pattern example

“Houston, we’ve had a problem!”


Living quarters for 2 astronauts on the moon
Living quarters for 3 astronauts during the trip to and from the moon
Power, Oxygen, etc
Lunar Module (LM)
Command Module (CM)
Service Module (SM)
Failure!
Plenty of power left in the LM!
Only 15 min power left in the CM!

Subsystem decomposition of the Apollo 13 spacecraft

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



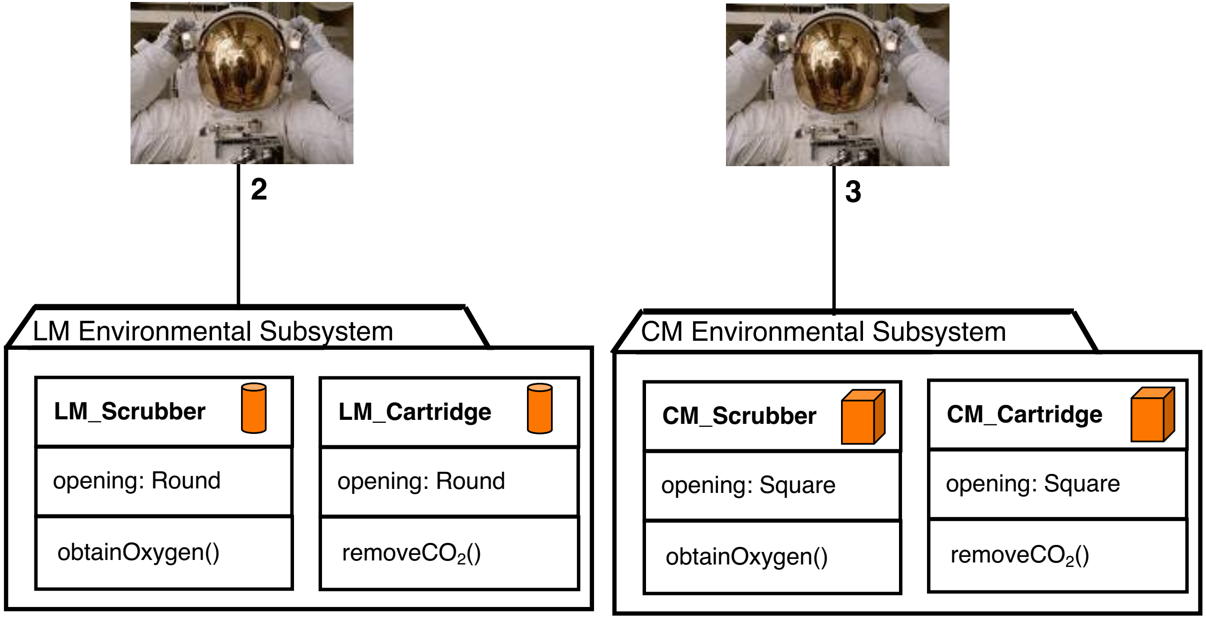
Original LM design:
lithium hydride cartridges
for removing carbon dioxide
(“Scrubber”)
2 days for 2 astronauts

Change of requirements
4 days for 3 astronauts
→ 12 person-days

Lithium hydride cartridges in CM
Availability: “plenty”

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



2

LM Environmental Subsystem

LM_Scrubber

opening: Round

obtainOxygen()

LM_Cartridge

opening: Round

removeCO$_2$()

3

CM Environmental Subsystem

CM_Scrubber

opening: Square

obtainOxygen()

CM_Cartridge

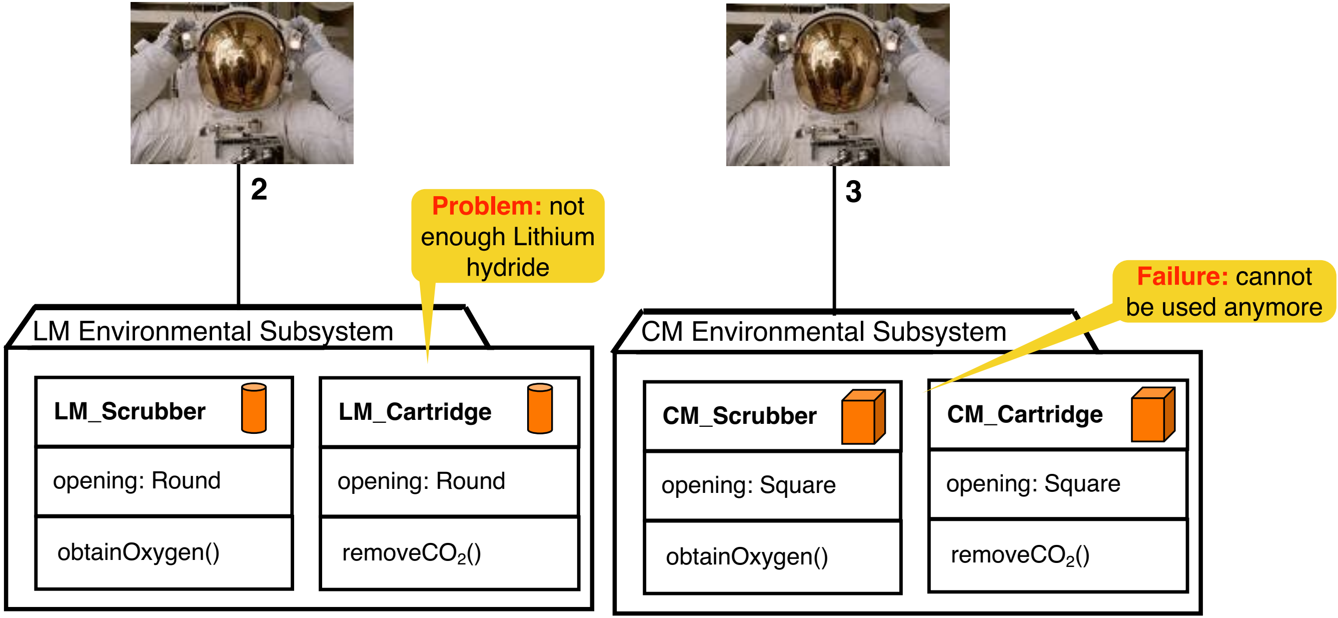
opening: Square

removeCO$_2$()

Lunar Module (LM)

Command Module (CM)

Change!



2

3

Problem: not enough Lithium hydride

Failure: cannot be used anymore

LM Environmental Subsystem

LM_Scrubber

opening: Round

obtainOxygen()

LM_Cartridge

opening: Round

removeCO_2()

CM Environmental Subsystem

CM_Scrubber

opening: Square

obtainOxygen()

CM_Cartridge

opening: Square

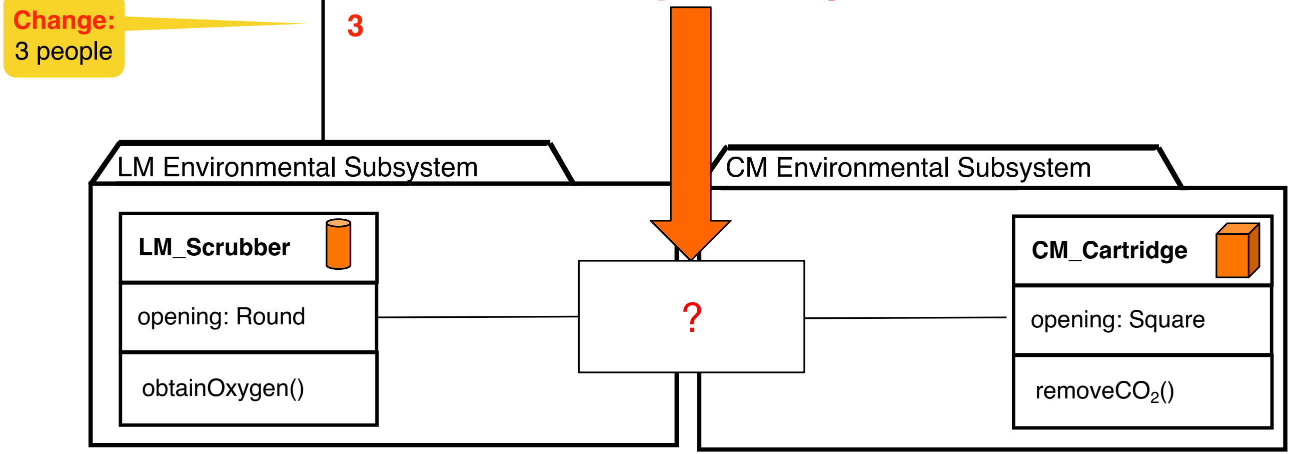
removeCO_2()

Lunar Module (LM)

Command Module (CM)

Can we connect the LM_Scrubber with the CM_Cartridge?

Incompatibility!



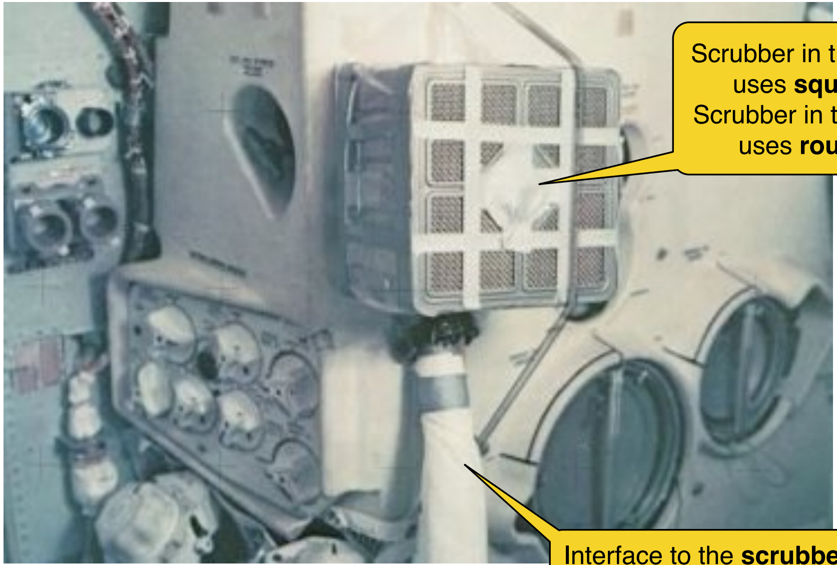
Lunar Module (LM)

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

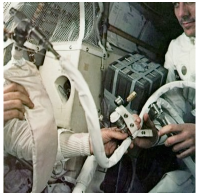


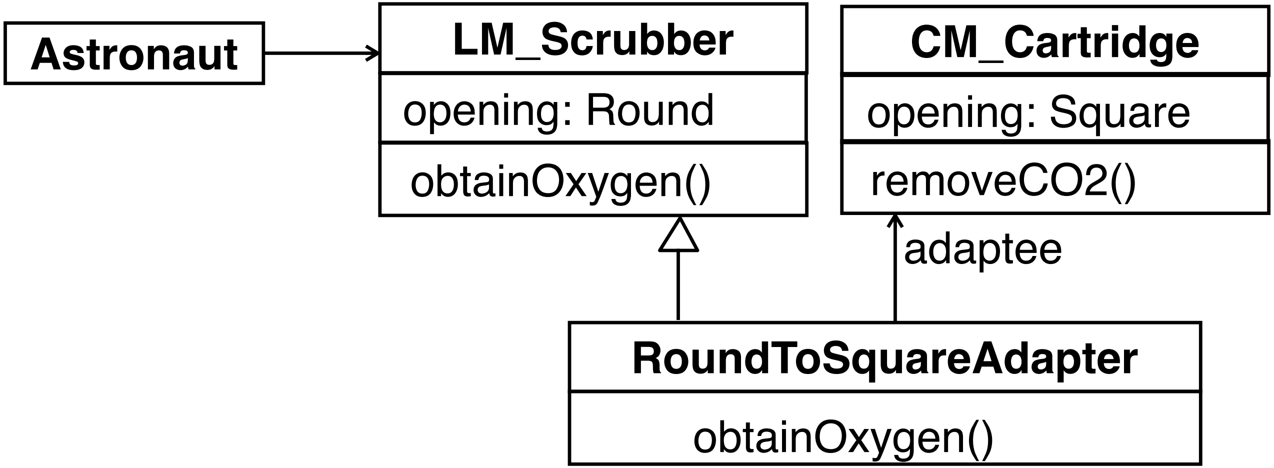
Scrubber in the CM subsystem uses **square openings**
Scrubber in the LM subsystem uses **round openings**

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

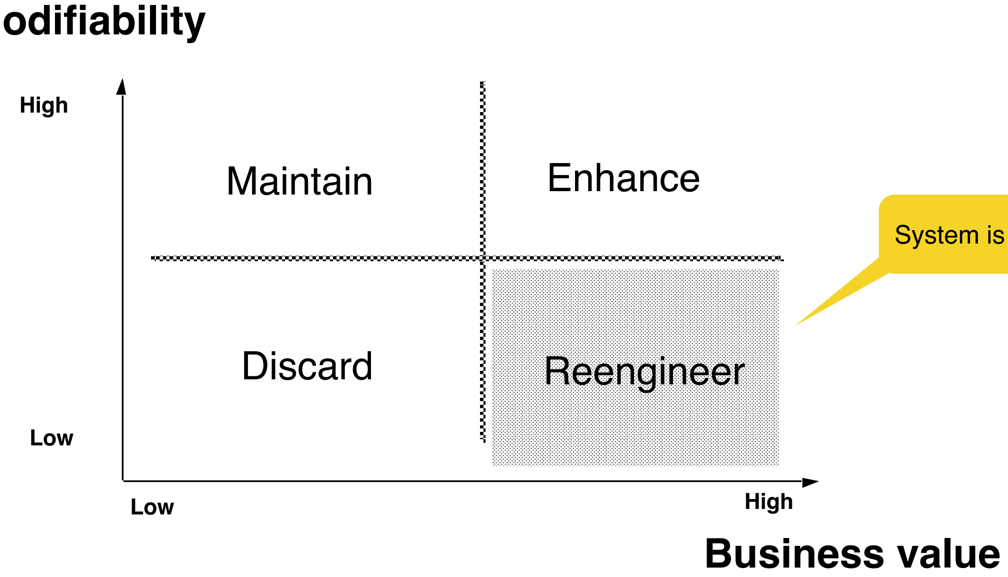
  → High maintenance cost
- Considered irreplaceable because a re-implementation is too expensive or impossible

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?



System is irreplaceable

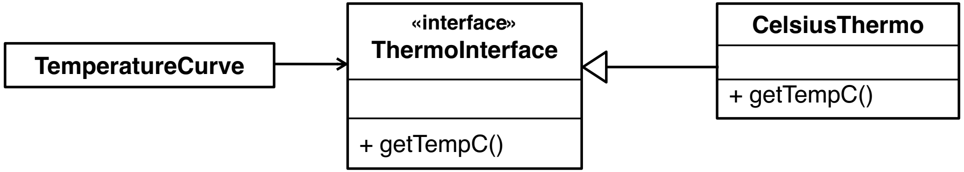
Comparison: adapter pattern vs. bridge pattern

- **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**

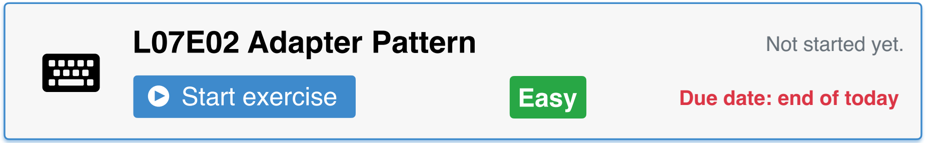
Exercise: adapter pattern

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

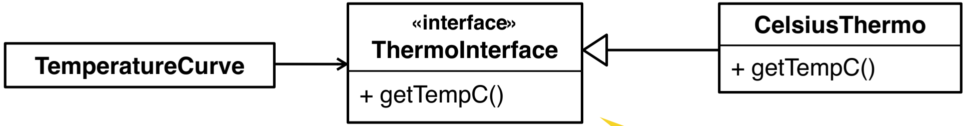
10 min

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)

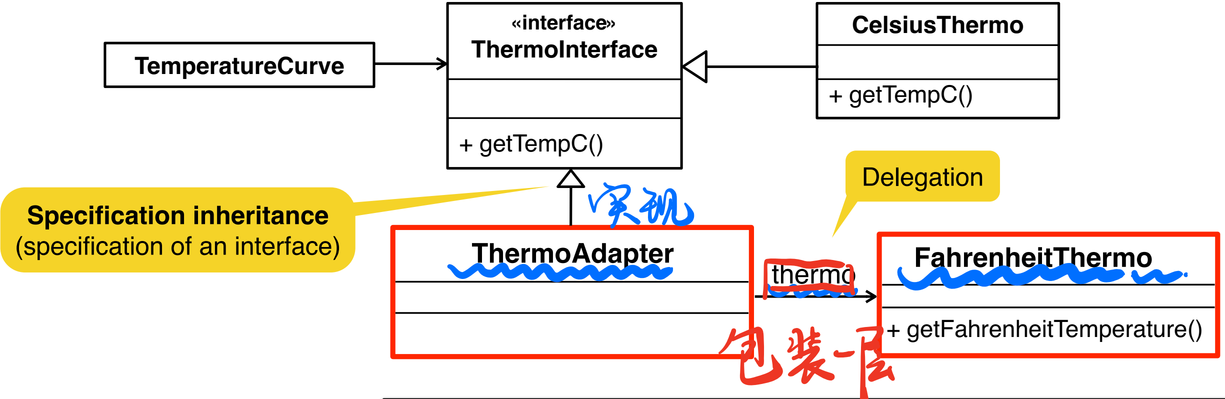


Subtyping: also called specification inheritance

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

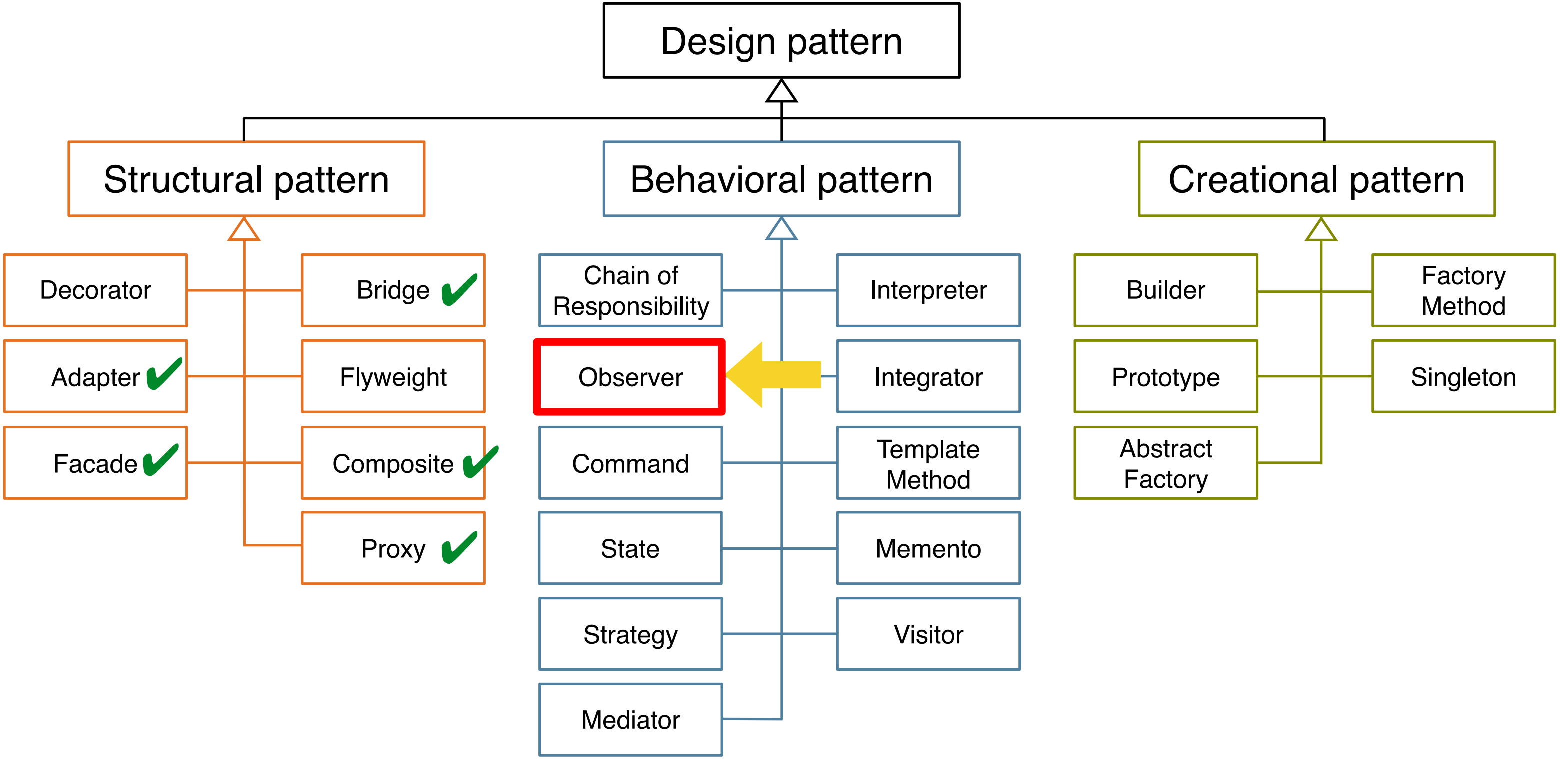


```
public class ThermoAdapter implements ThermoInterface {
  ....
}
```

Outline

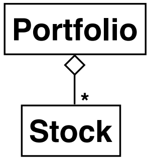
- Adapter pattern
- ➡ **Observer pattern**
- Winners of the Bumpers competition
- University course evaluation
- Strategy pattern

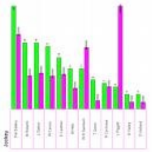
Design patterns taxonomy

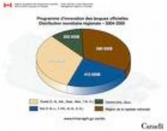


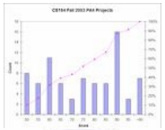
Observer pattern

- **Problem**
  - An object that changes its **state** often
    - **Example:** a portfolio of stocks
  - Multiple views of the current **state**
    - **Example:** histogram view, pie chart view, timeline view
- Requirements
  - The system should maintain consistency across the (redundant) views, whenever the state of the observed object changes
  - The system design should be highly extensible
  - It should be possible to add new views - for example, an alarm - without having to recompile the observed object or existing views





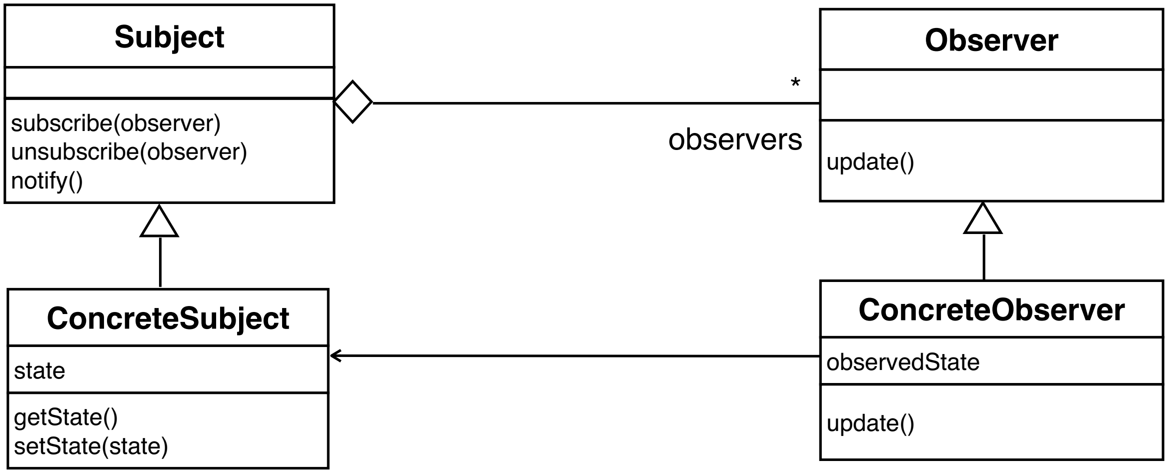




Observer pattern

- **Solution:** model a 1-to-many dependency between objects
  - Connect the state of an observed object, the **subject** with many observing objects, and the **observers**
- **Benefits**
  - Maintain consistency across redundant observers
  - Optimize a batch of changes to maintain consistency
- Also called **Publish and Subscribe**

The observer pattern decouples a subject from its observer



- The **Subject** represents the entity object
  - The state is contained in the subclass **ConcreteSubject**
- **Observers** attach to the **Subject** by calling **subscribe()**
- Each **ConcreteObserver** has a different view of the **state** of the **ConcreteSubject**
  - The state can be obtained and set by the subclasses of type **ConcreteObserver**

Variants of the observer pattern

3 variants for maintaining the consistency

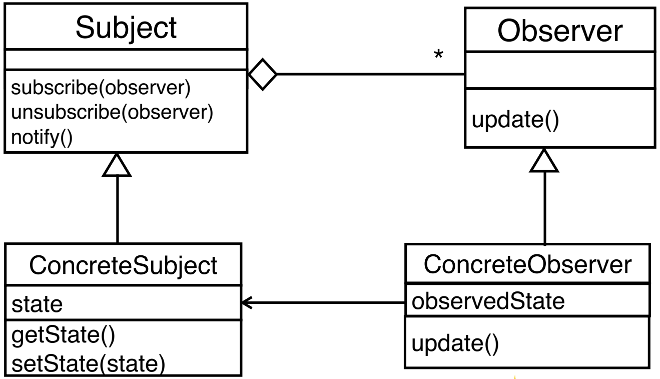
1. **Notification + pull:** every time the state of the `Subject` changes, `notify()` is called which calls **`update()`** in each `Observer`
   An observer can decide whether to pull the state of the `Subject` by calling `getState()`

Used in the **pull notification variant** of the MVC architectural style

2. **Notification + push:** the `Subject` also includes the `state` that has been changed in each **`update(state)`** call

Used in the **push notification variant** of the MVC architectural style

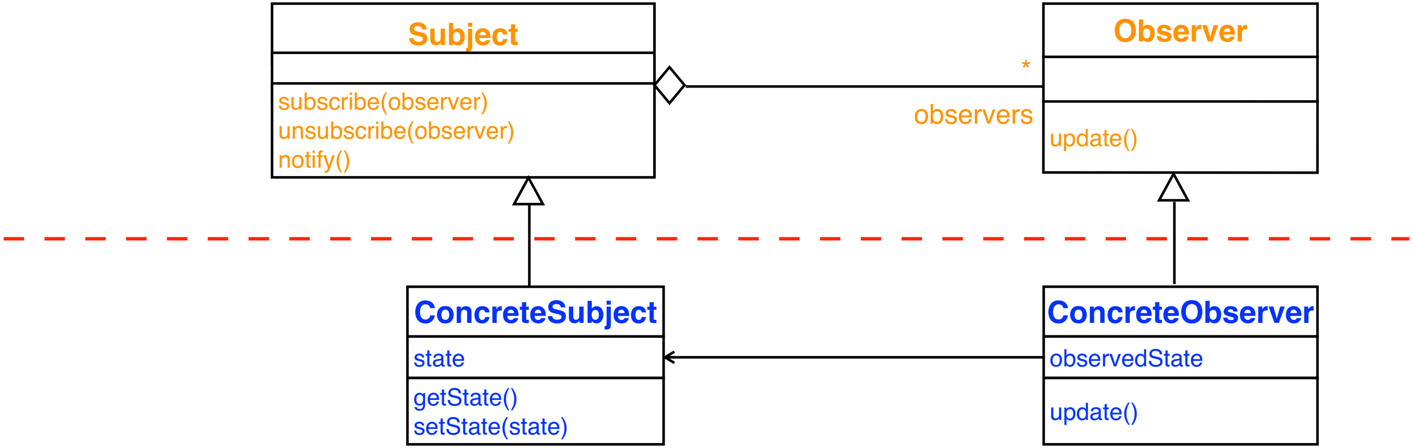
3. **Periodic pull:** an `Observer` periodically (e.g. every 5s) pulls the state of the `Subject` by calling `getState()`



Variant 1:
Notification + pull

Review: application domain vs solution domain objects

Requirements analysis (language of application domain)



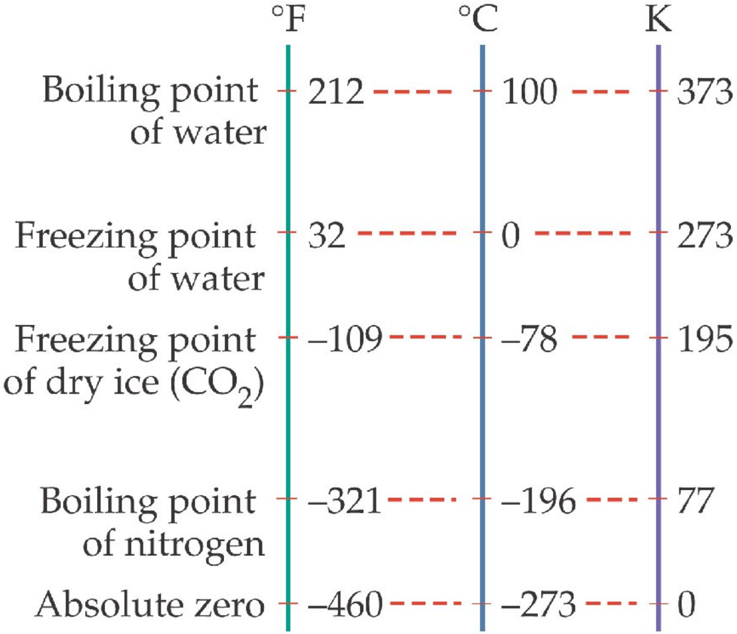
Object design (language of solution domain)

Exercise: observer pattern

Problem (stated in natural language): a temperature converter

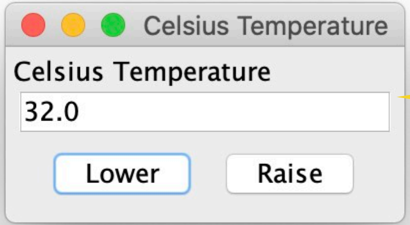
- We want an application with a graphical user interface
  - Display the temperature in **Fahrenheit** or **Celsius**
  - Convert from **Fahrenheit** to **Celsius** and vice versa
  - Allow the temperature to be raised or lowered
  - Allow to visualize the temperature with a gauge (like a thermometer)
  - Allow to change the temperature by moving the mouse across a slider
- Initial temperature value at the start up of the application: the temperature of the freezing point of water
- **Solution:** synchronize the views with the observer pattern

Temperature scales: Fahrenheit (F), Celsius (C), Kelvin (K)



	°F	°C	K
Boiling point of water	212	100	373
Freezing point of water	32	0	273
Freezing point of dry ice (CO_2)	–109	–78	195
Boiling point of nitrogen	–321	–196	77
Absolute zero	–460	–273	0

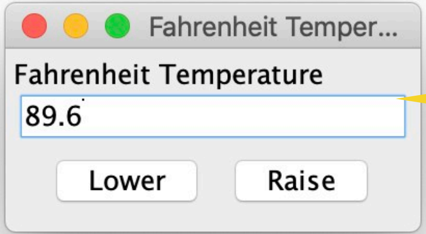
User interface design of the temperature converter



Celsius Temperature

32.0

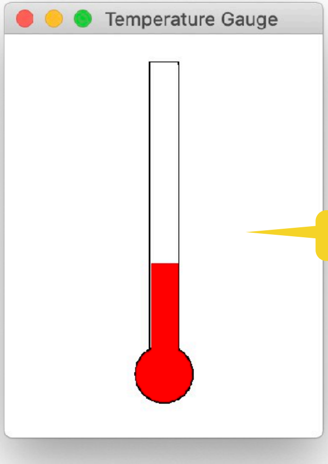
Text View: Celsius



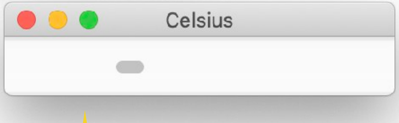
Fahrenheit Temperature

89.6

Text View: Fahrenheit

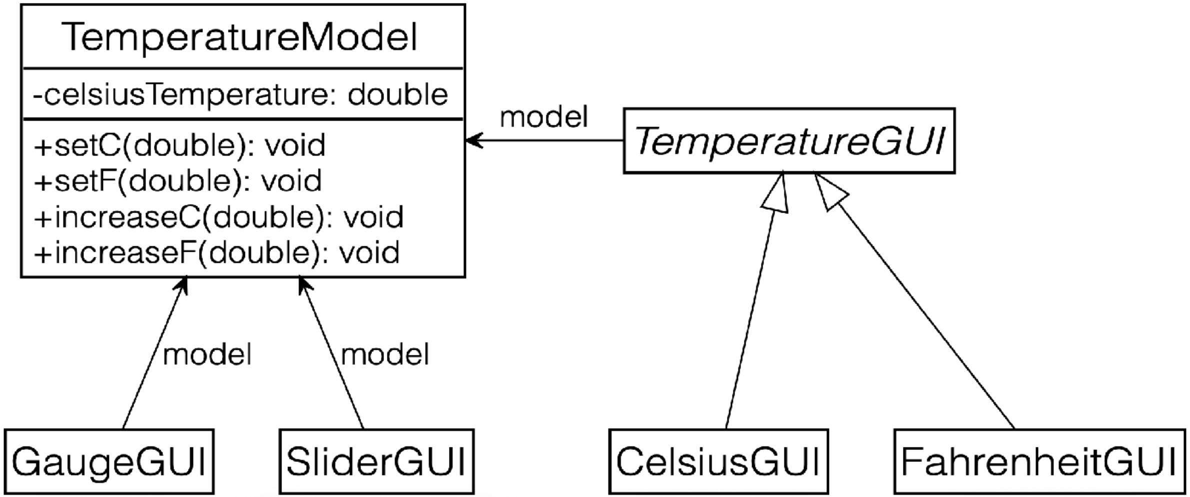


Graph View



Slider View

Existing model and views



TemperatureModel

-celsiusTemperature: double

+setC(double): void
+setF(double): void
+increaseC(double): void
+increaseF(double): void

model

TemperatureGUI

model

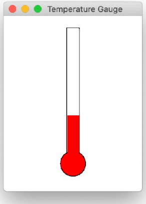
model

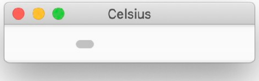
GaugeGUI

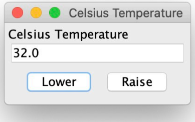
SliderGUI

CelsiusGUI

FahrenheitGUI



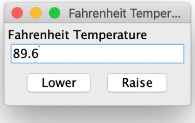




Celsius Temperature

Celsius Temperature

32.0



Fahrenheit Temper...

Fahrenheit Temperature

89.6

L07E03 Observer Pattern

Not started yet.

Start exercise

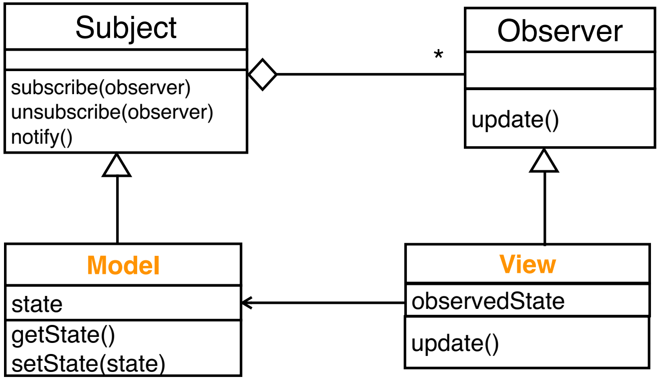
Medium

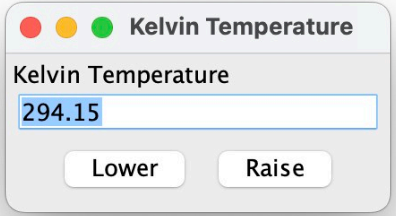
Due date: end of today

20 min

6 pts

- **Problem statement**
  - **Part 1:** Connect model and views using the observer pattern
  - **Part 2:** Add a new Kelvin view

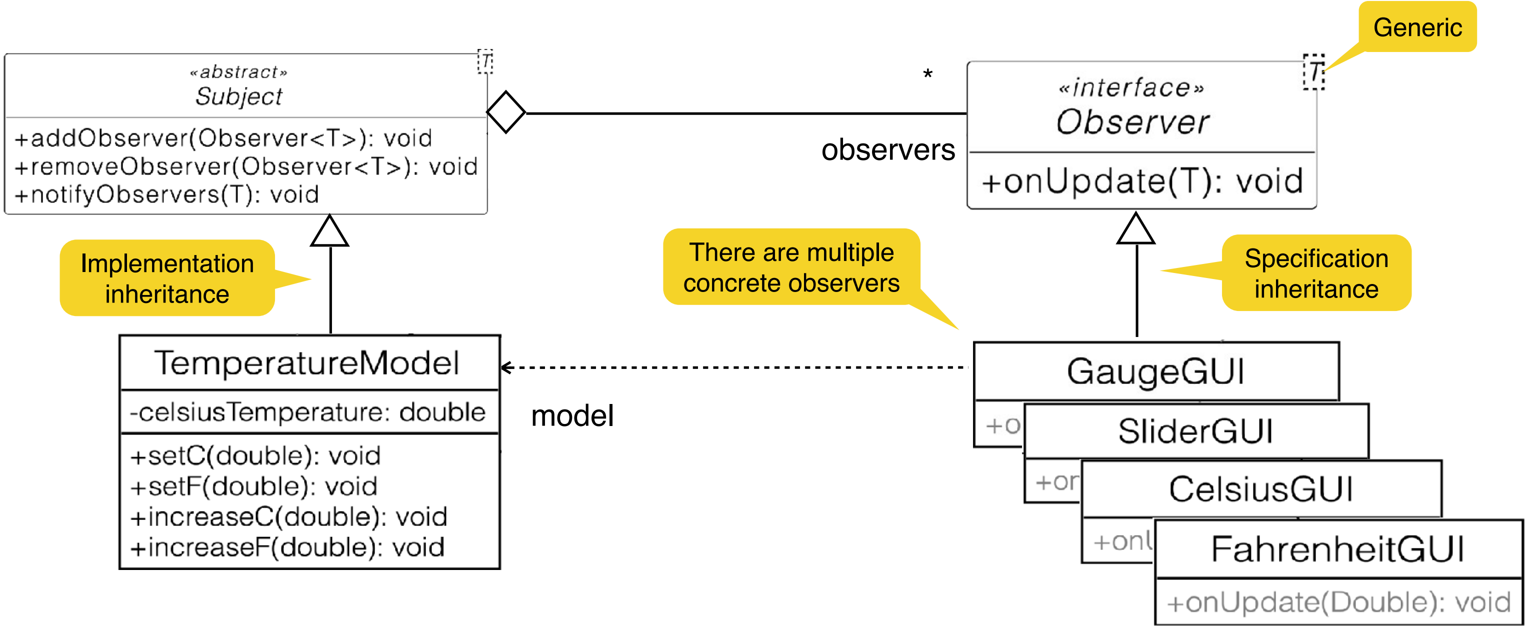




Kelvin Temperature

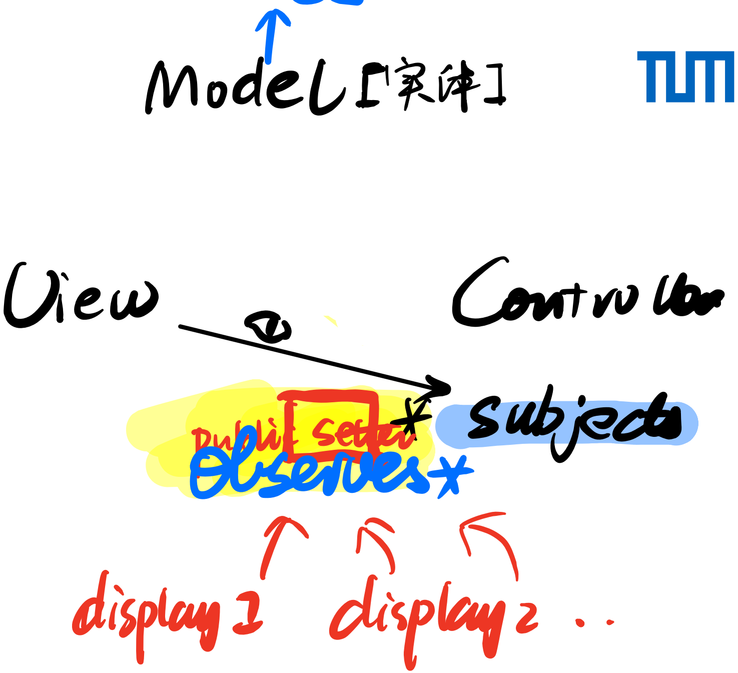
294.15

Hint: observer pattern in L07E03



Outline

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

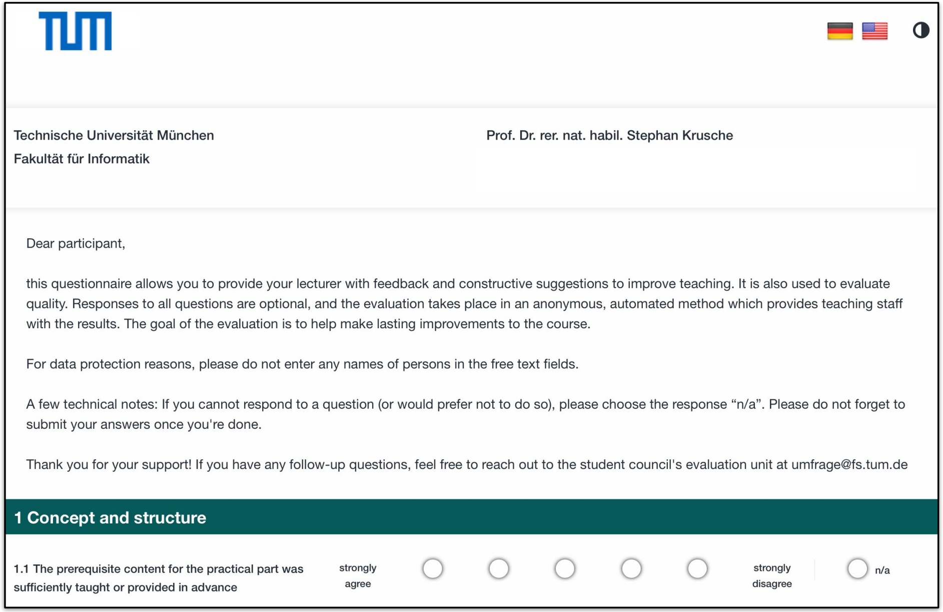


University course evaluation

- EIST with 2200 students in a hybrid setup is a real challenge!
- We put a lot of effort and passion into creating a great learning atmosphere and providing you with the latest concepts, tools, and workflows
- We hope you appreciate our effort 😊 and comment on issues, that we can improve in the future semesters
- **Your feedback is valuable to us and the university!**
- You should have received an email from the **Department Student Council** MPI (”Fachschaft”) to evaluate EIST
- **You now have 15 minutes to fill out the anonymous online survey**

University course evaluation (15 min)

- Find the email with a link to https://evasys.zv.tum.de/... for **INHN0006**
- Fill out the following form



Technische Universität München
Fakultät für Informatik

Prof. Dr. rer. nat. habil. Stephan Krusche

Dear participant,

this questionnaire allows you to provide your lecturer with feedback and constructive suggestions to improve teaching. It is also used to evaluate quality. Responses to all questions are optional, and the evaluation takes place in an anonymous, automated method which provides teaching staff with the results. The goal of the evaluation is to help make lasting improvements to the course.

For data protection reasons, please do not enter any names of persons in the free text fields.

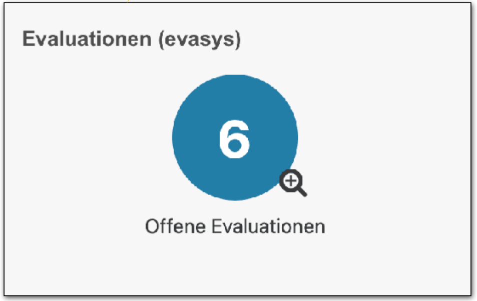
A few technical notes: If you cannot respond to a question (or would prefer not to do so), please choose the response “n/a”. Please do not forget to submit your answers once you're done.

Thank you for your support! If you have any follow-up questions, feel free to reach out to the student council's evaluation unit at umfrage@fs.tum.de

1 Concept and structure

1.1 The prerequisite content for the practical part was sufficiently taught or provided in advance — strongly agree ○ ○ ○ ○ ○ strongly disagree ○ n/a

Alternatively, you can find the survey on Moodle



Evaluationen (evasys)

6

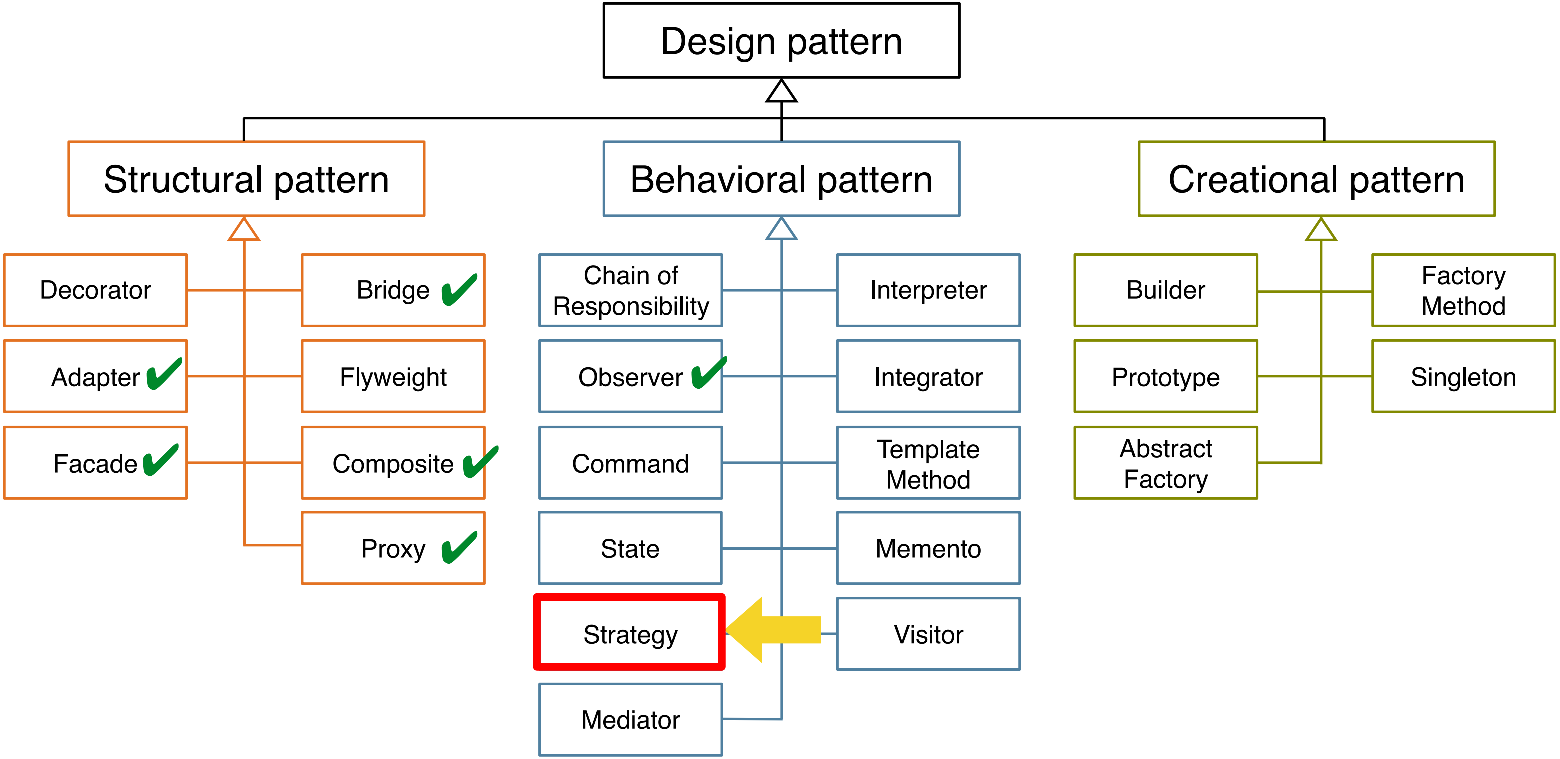
Offene Evaluationen

Outline

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

➡ **Strategy pattern**

Design patterns taxonomy

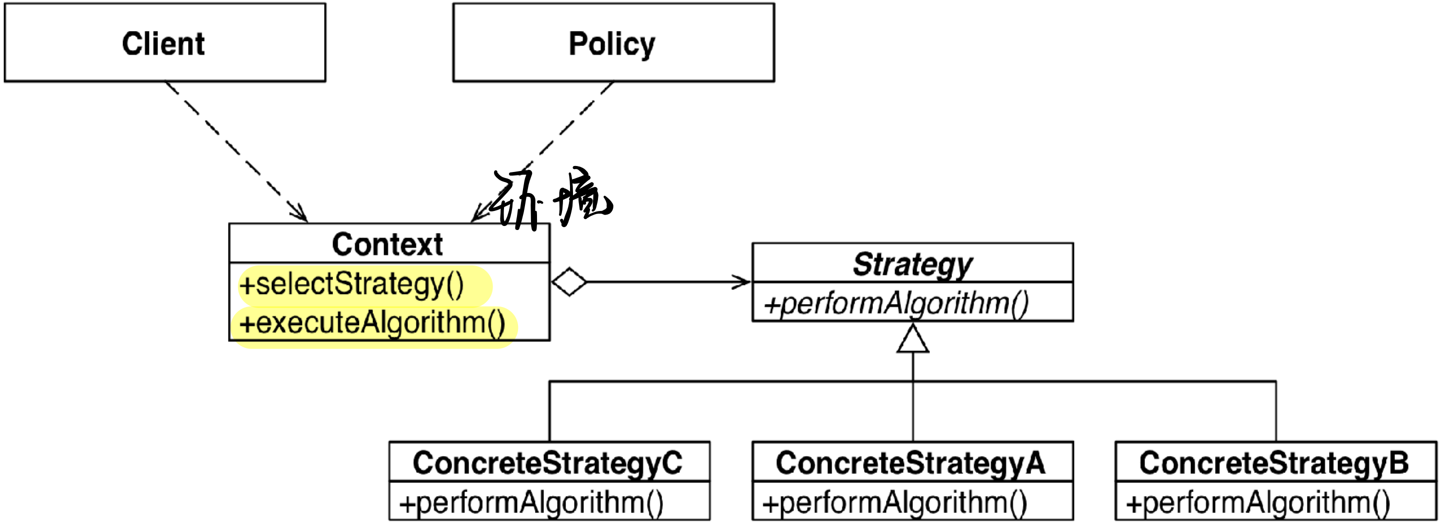


Strategy pattern

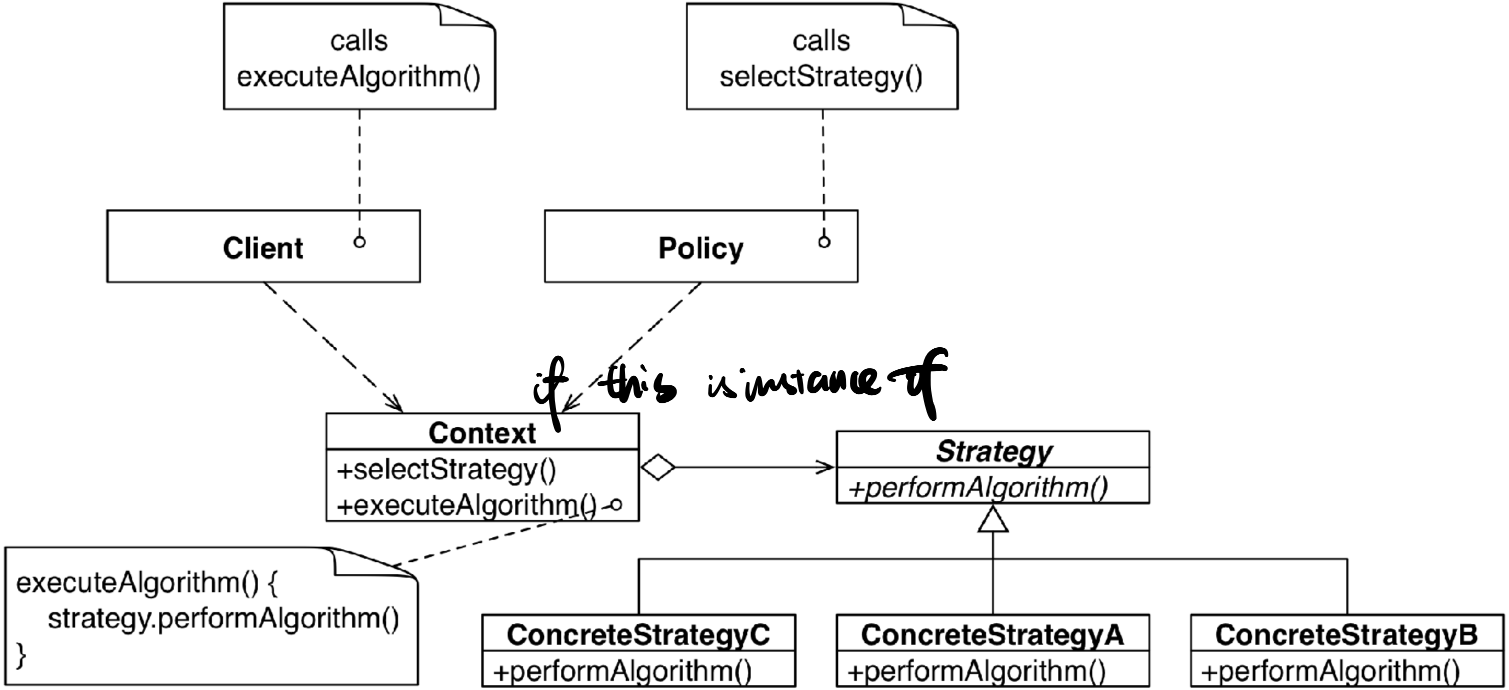
- **Problem:** different algorithms exist for a specific task
- Examples of specific tasks
  - Different ways to sort a list (bubble sort, merge sort, quick sort)
  - Different collision strategies for objects in video games
  - Different ways to parse tokens into an abstract syntax tree (bottom-up, top-down)
- If we need a new algorithm, we want to add it without changing the rest of the application or the other algorithms
- **Solution:** the strategy pattern allows to switch between different algorithms at run time based on the context and a policy

Strategy pattern: UML class diagram

The **`Policy`** decides which **`ConcreteStrategy`** is best in a given **`Context`**



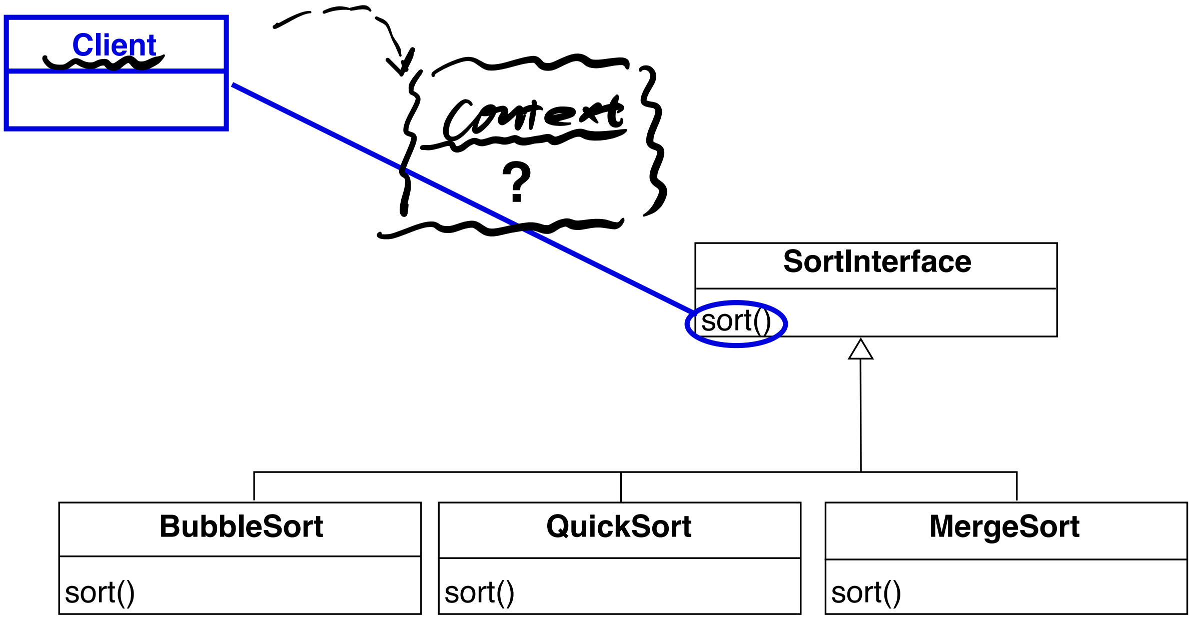
Strategy pattern: UML class diagram



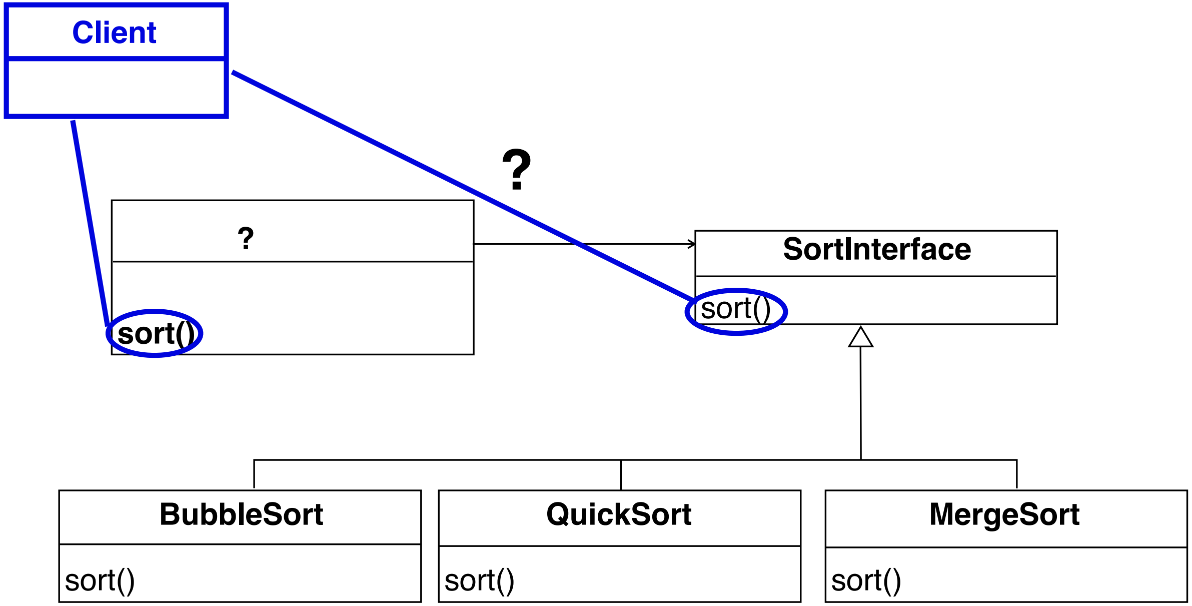
```
executeAlgorithm() {
   strategy.performAlgorithm()
}
```

if this isinstance of

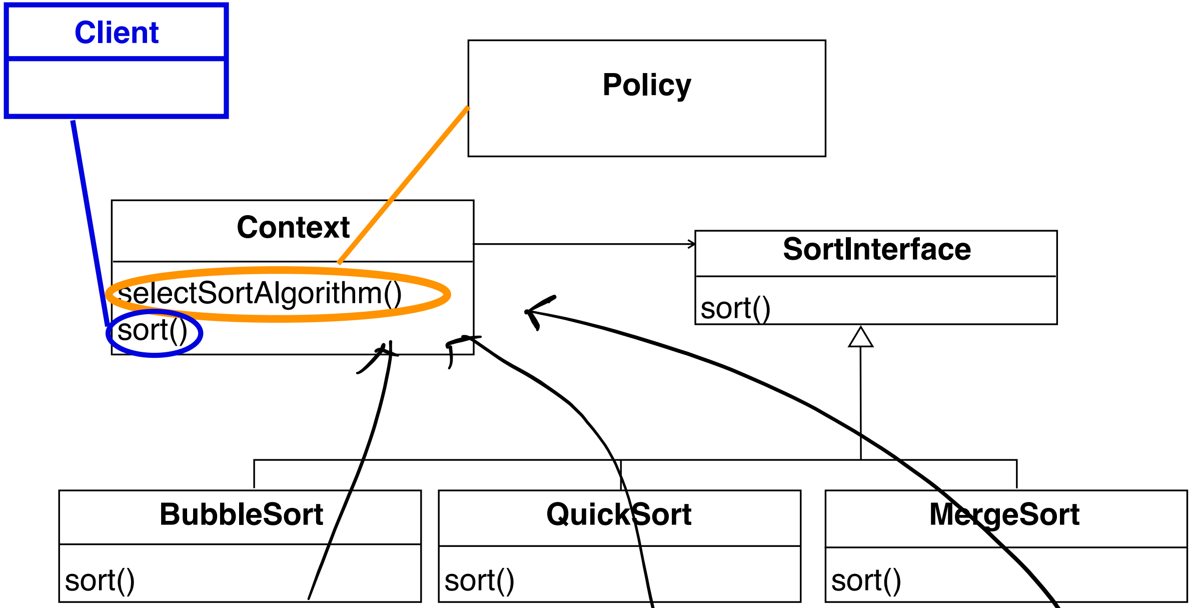
Example: using the strategy pattern to switch between different algorithms



Example: using the strategy pattern to switch between different algorithms



Example: using the strategy pattern to switch between different algorithms

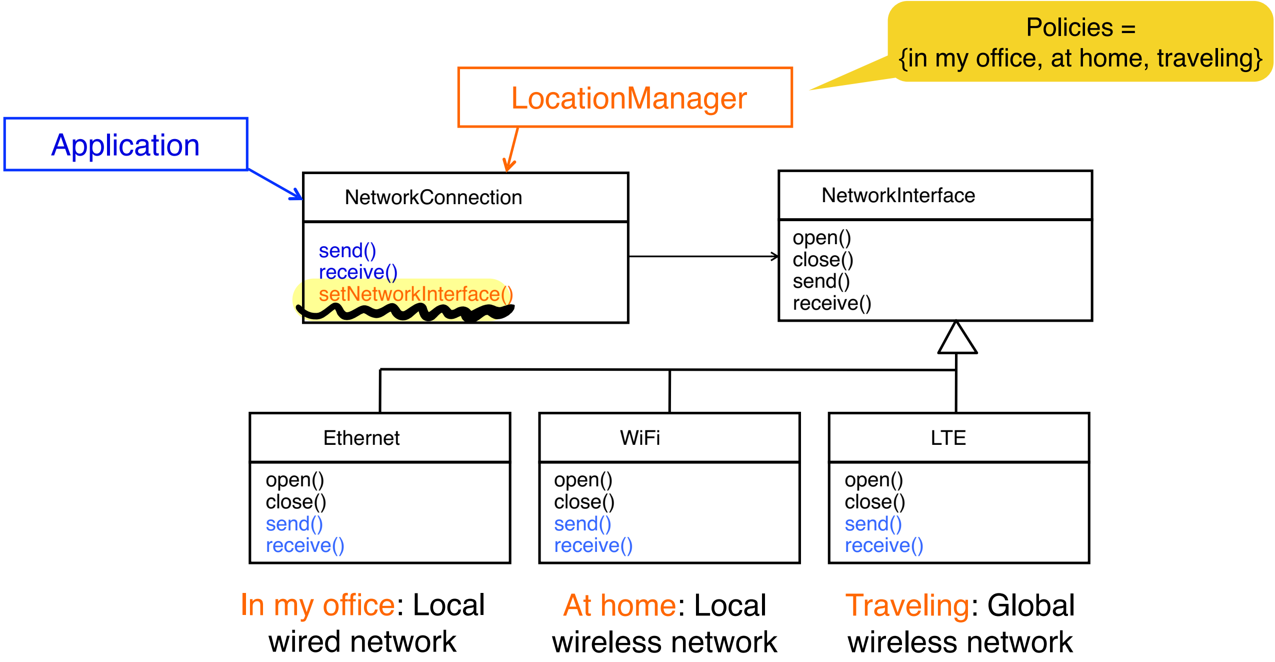


Simple development is important

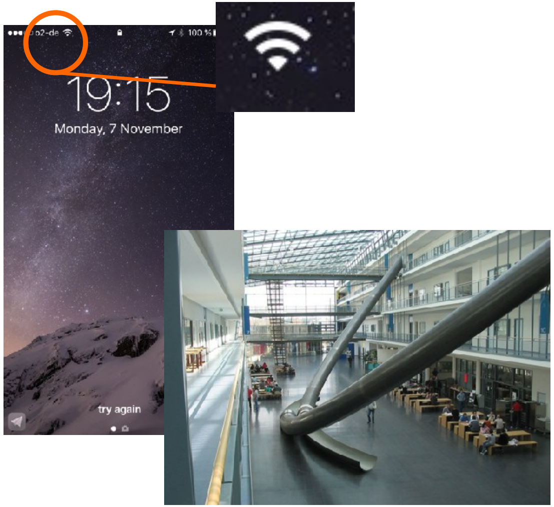
Fast execution time is important

Efficient use of memory is important

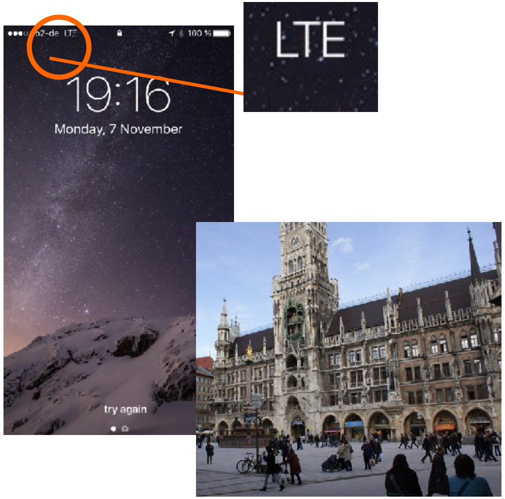
Supporting multiple implementations of a network connection



Another policy for network connections



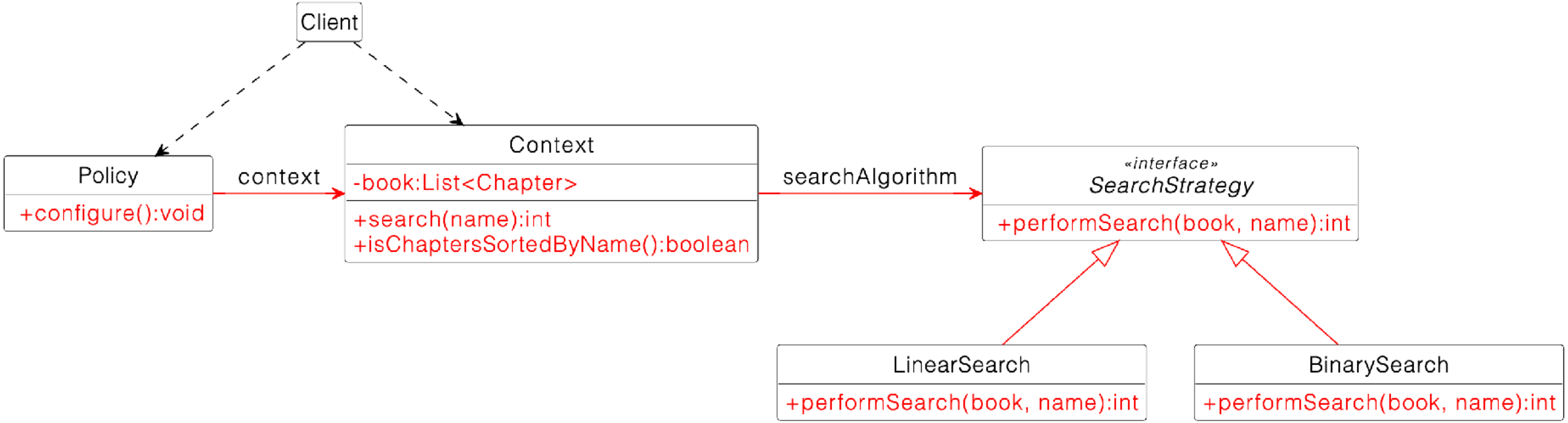
If WiFi available, use WiFi …



… otherwise, use mobile data

Homework **H07E01**: strategy pattern

- **Goal:** find an entry in a book with multiple chapters
- Problem statement
  - Implement **linear search** and **binary search** to search by chapter name
  - Apply the strategy pattern to choose which algorithm is used at runtime



Clues for the use of design patterns

- **Text:** “complex structure”, “must have variable depth and width”

  → **Composite pattern**

- **Text:** “must provide a policy independent from the mechanism”, “must allow to change algorithms at runtime”

  → **Strategy pattern**

- **Text:** “must be location transparent”

  → **Proxy pattern**

- **Text:** “states must be synchronized”, “many systems must be notified”

  → **Observer pattern** (part of the MVC architectural pattern)

Clues for the use of design patterns

- **Text:** “must interface with an existing object”

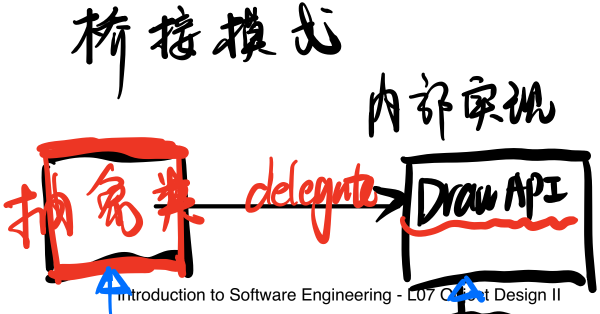
  → **Adapter pattern**

- **Text:** “must interface to several systems, some of them to be developed in the future”, “an early prototype must be demonstrated”, “must provide backward compatibility”

  → **Bridge pattern**

- **Text:** “must interface to an existing set of objects”, “must interface to an existing API”, “must interface to an existing service”

  → **Façade pattern**



构造器传入个
API的实现类，
放入super的

其他参数存于算法

画形状

画圆 画方

Homework

- **H07E01** Strategy Pattern (programming exercise)
- **H07E02** Model the Strategy Pattern (modeling exercise)
- **H07E03** MVC & Observer Pattern (text exercise)
- Read more about **design patterns** on https://sourcemaking.com (see Literature)

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

Summary

- Design patterns combine inheritance and delegation
- **Adapter pattern:** connects incompatible components and allows the reuse of existing components
- **Observer pattern:** maintains consistency across multiple observers: the basis for model view controller
- **Strategy pattern:** switches between multiple implementations of an algorithm at run time based on the context and a policy
- There are certain clues when to use which design pattern

Readings

- 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