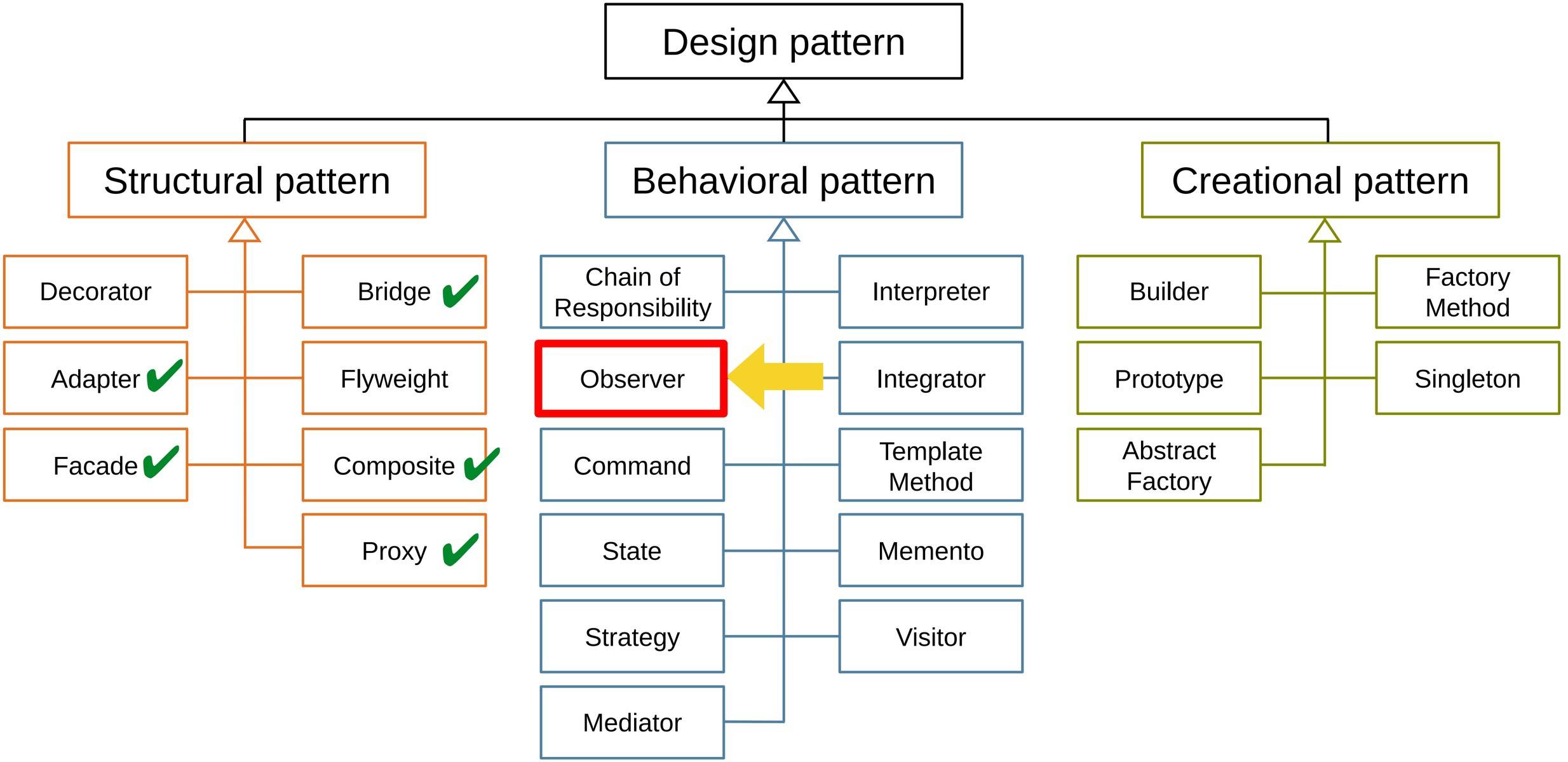
Outline

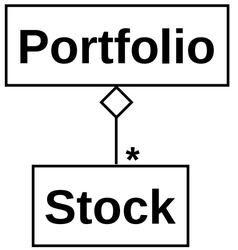
- Adapter pattern
- **Observer pattern**
- Winners of the Bumpers competition
- University course evaluation
- Strategy pattern

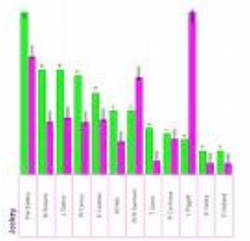
Design patterns taxonomy

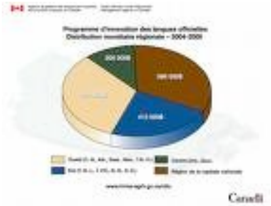


Observer pattern

- **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



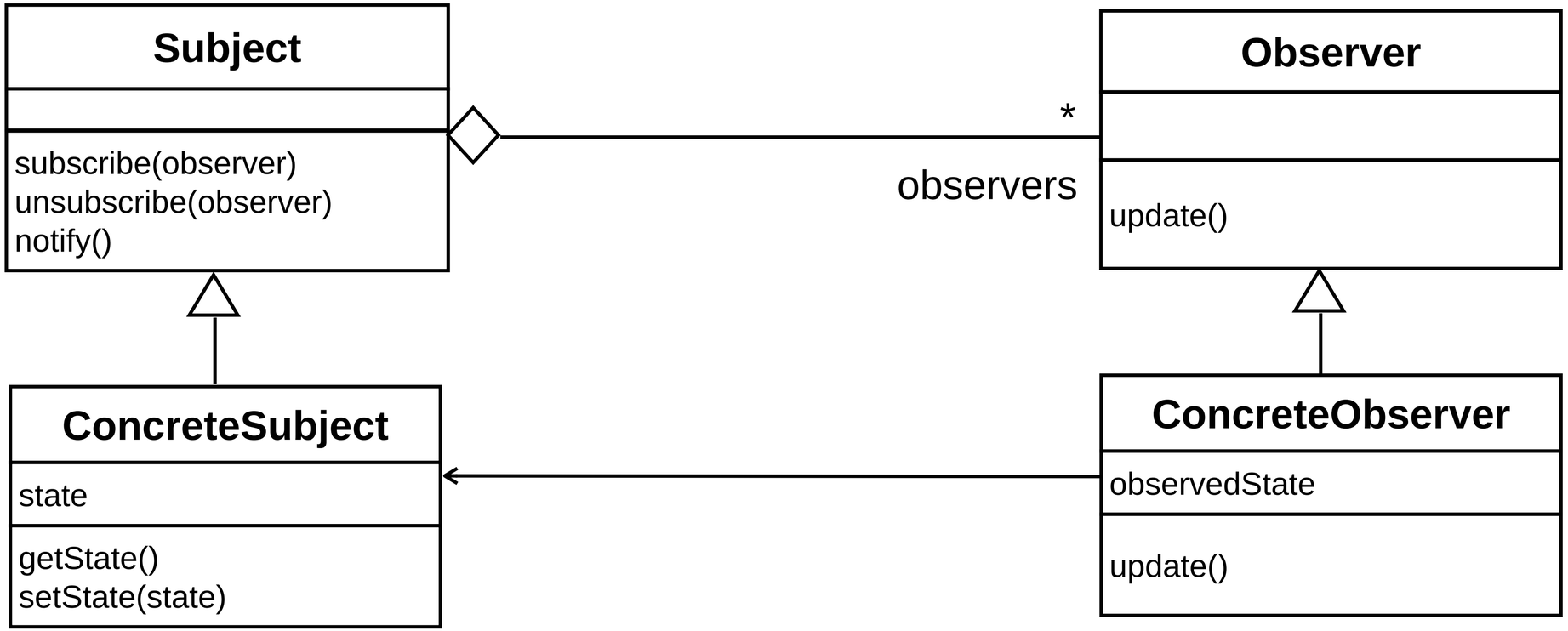




Observer pattern

- **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

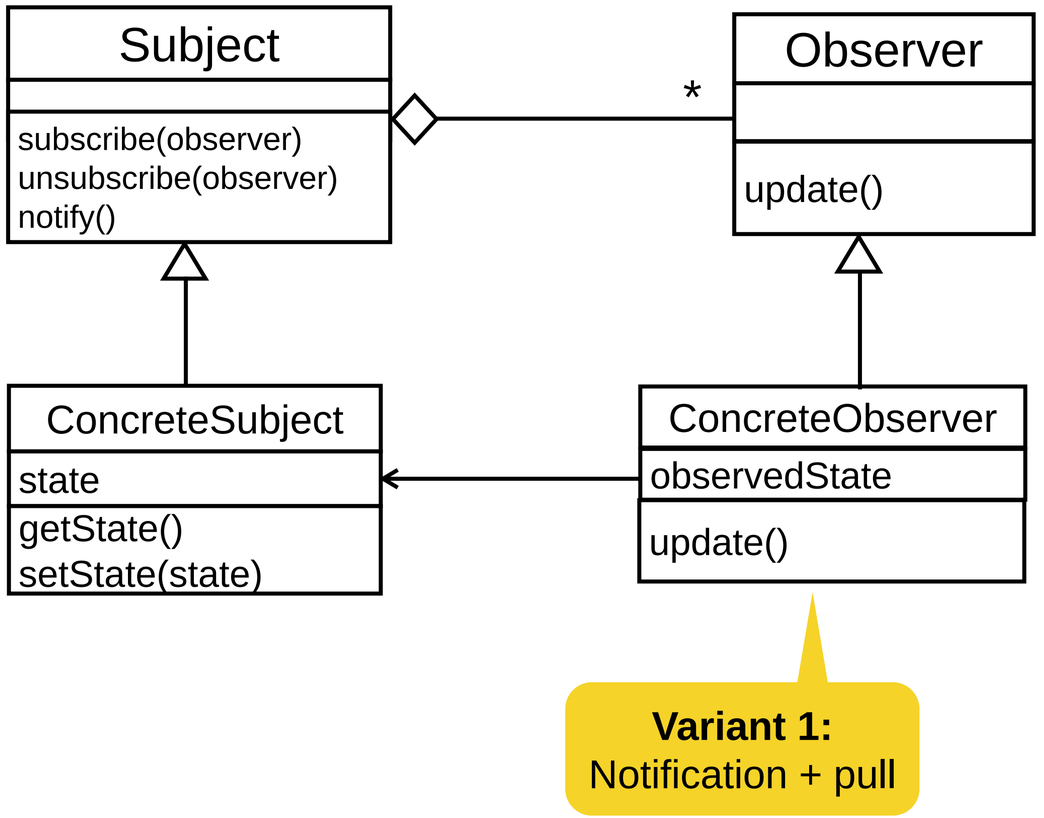
1. **Notification + pull:** every time the state of the `Subject` changes, `notify()` is called which calls **`update()`** in each `Observer`
   An observer can decide whether to pull the state of the `Subject` by calling `getState()`

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

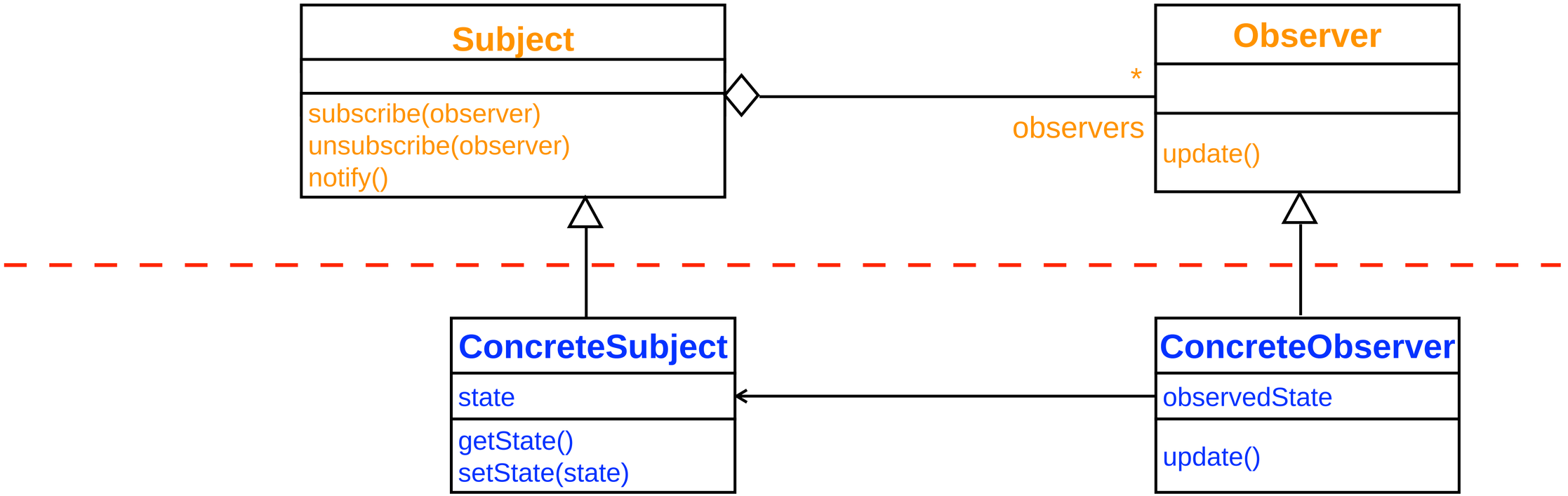
Used in the **push notification variant** of the MVC architectural style

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



Review: application domain vs solution domain objects

Requirements analysis (language of application domain)



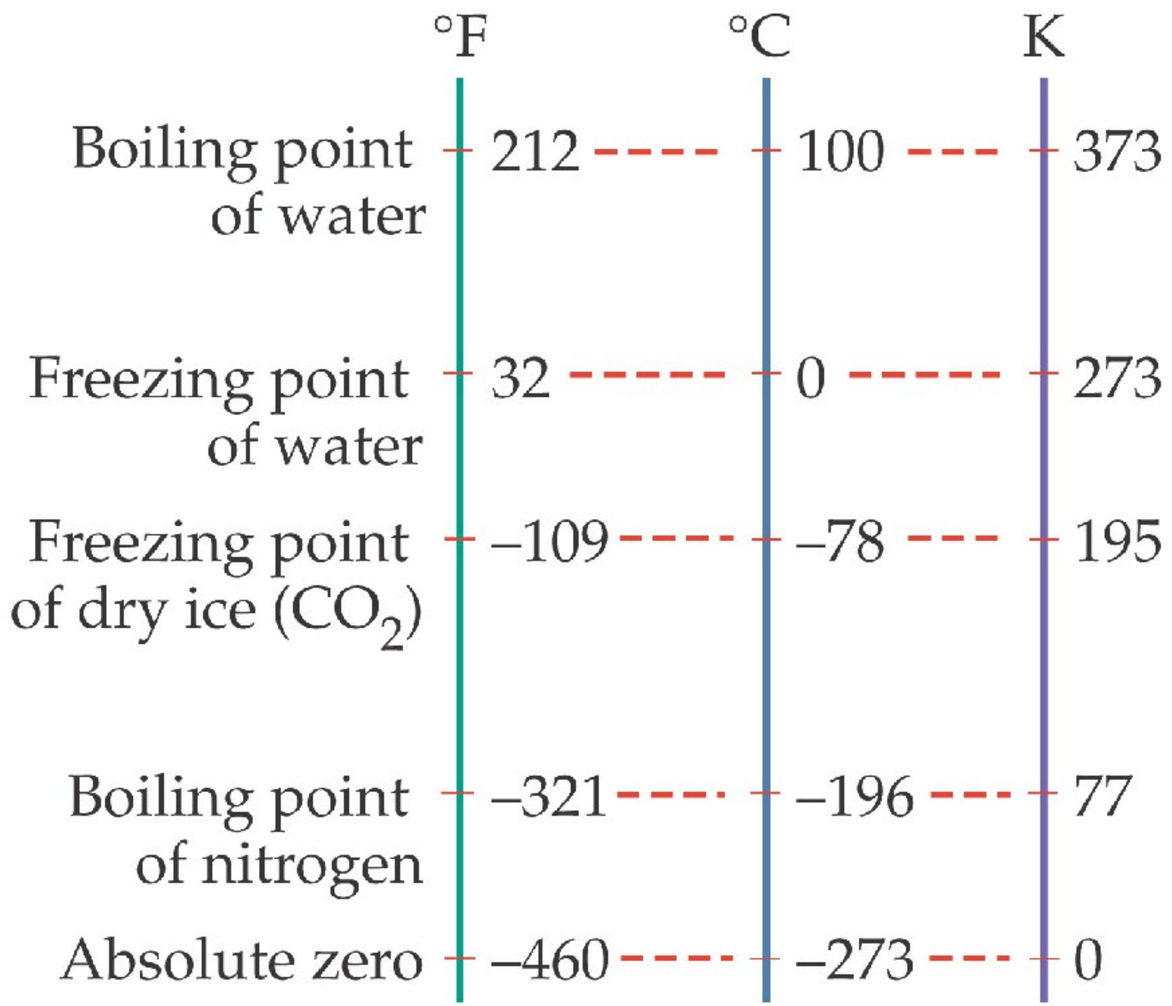
Object design (language of solution domain)

Exercise: observer pattern

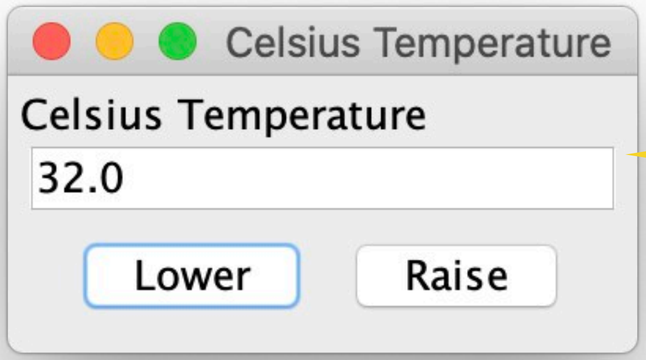
Problem (stated in natural language): a temperature converter

- 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

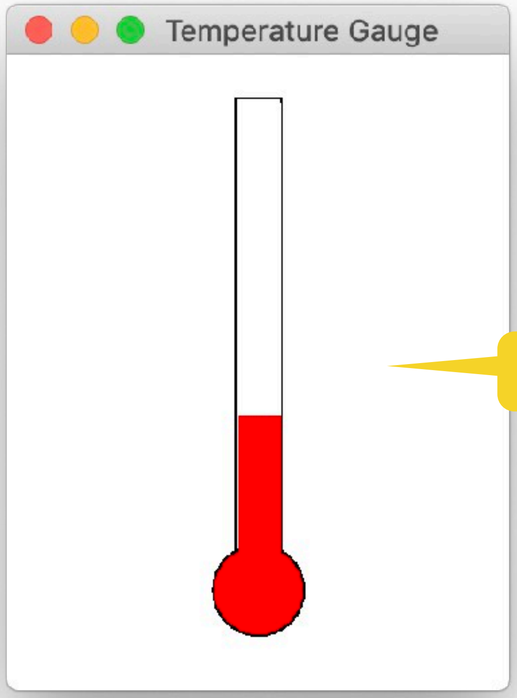
Temperature scales: Fahrenheit (F), Celsius (C), Kelvin (K)



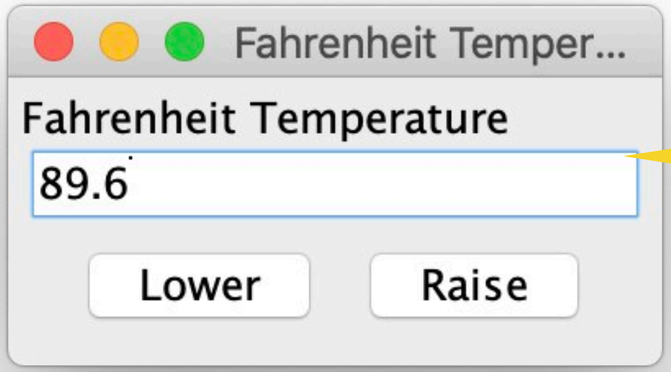
User interface design of the temperature converter



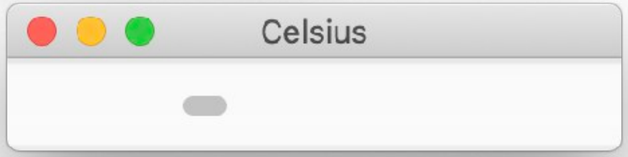
Text View: Celsius



Graph View

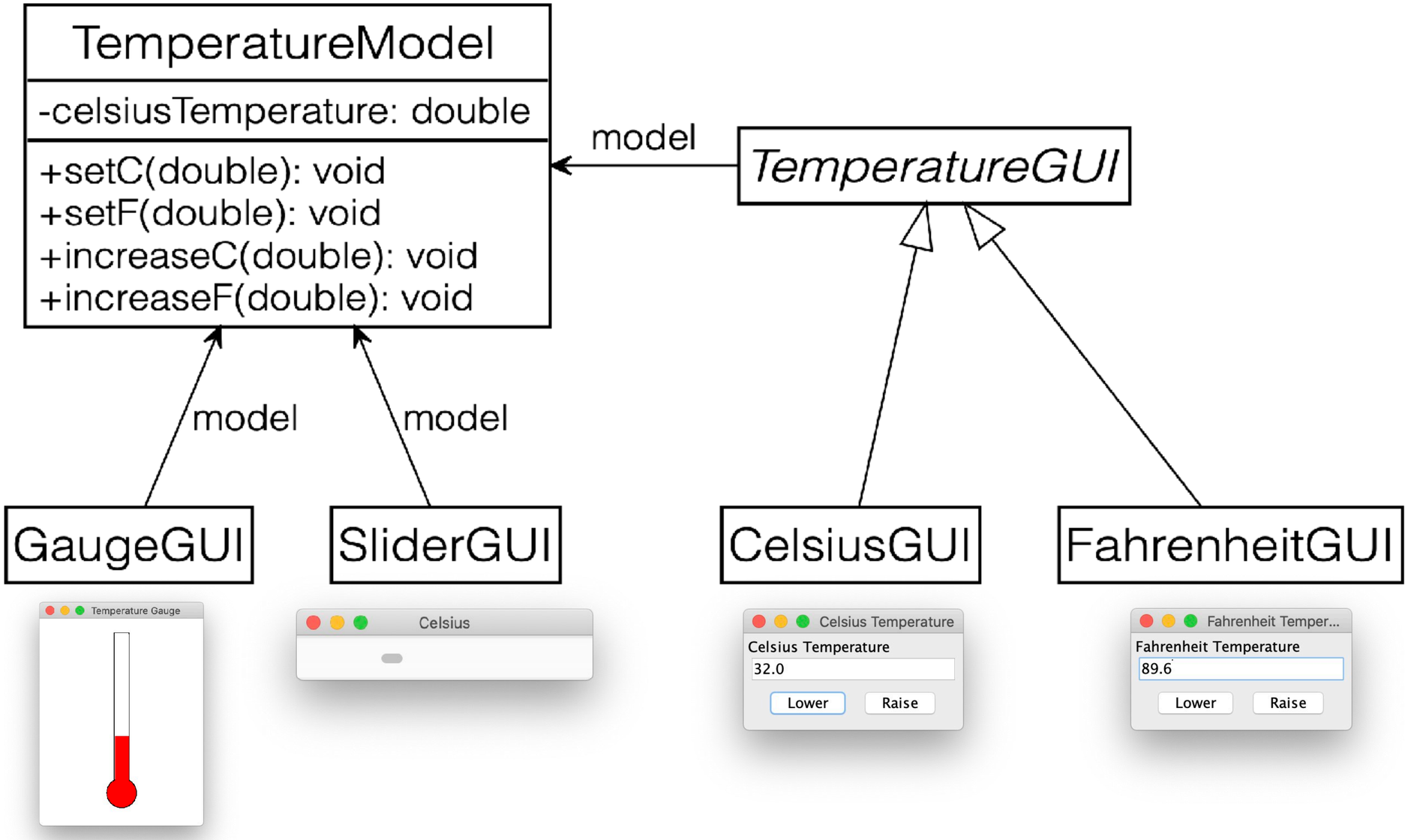


Text View: Fahrenheit



Slider View

Existing model and views



L07E03 Observer Pattern

Not started yet.

Start exercise

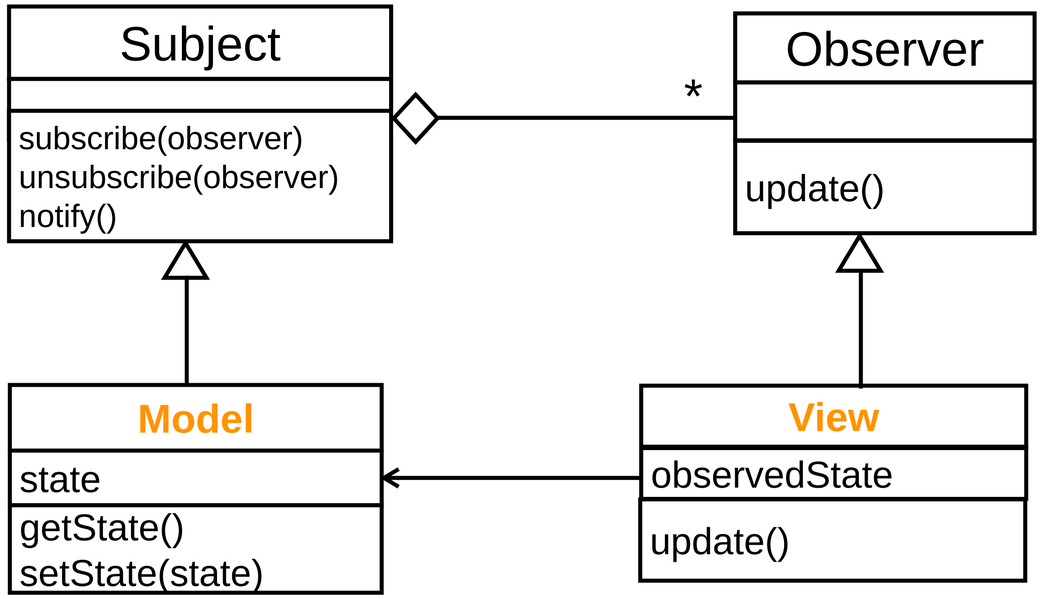
Medium

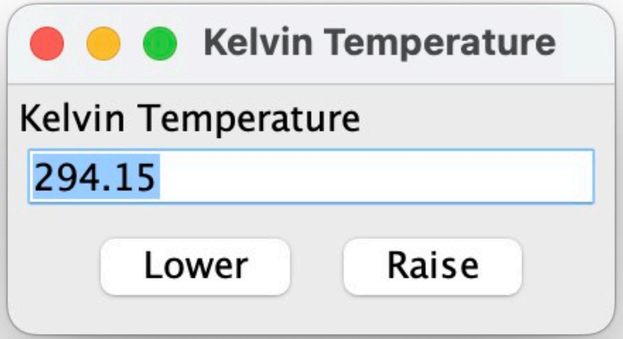
Due date: end of today

20 min

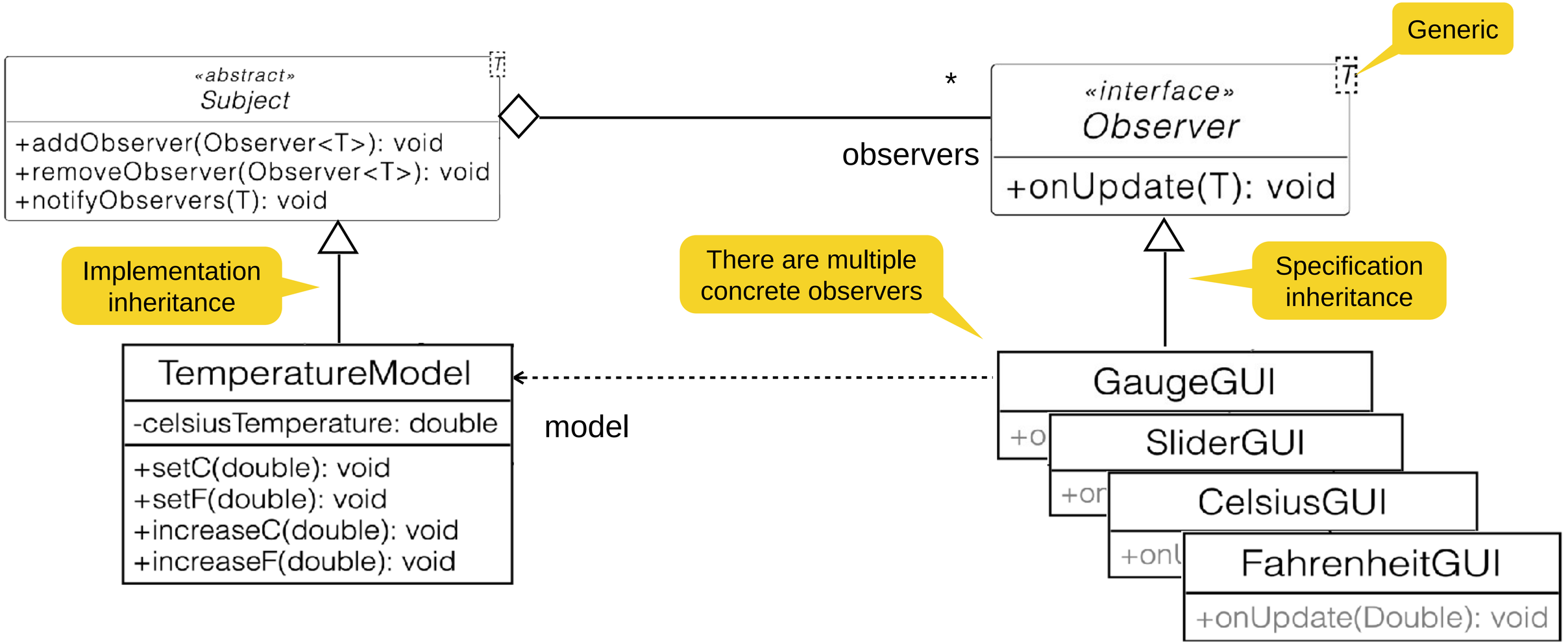
6 pts

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





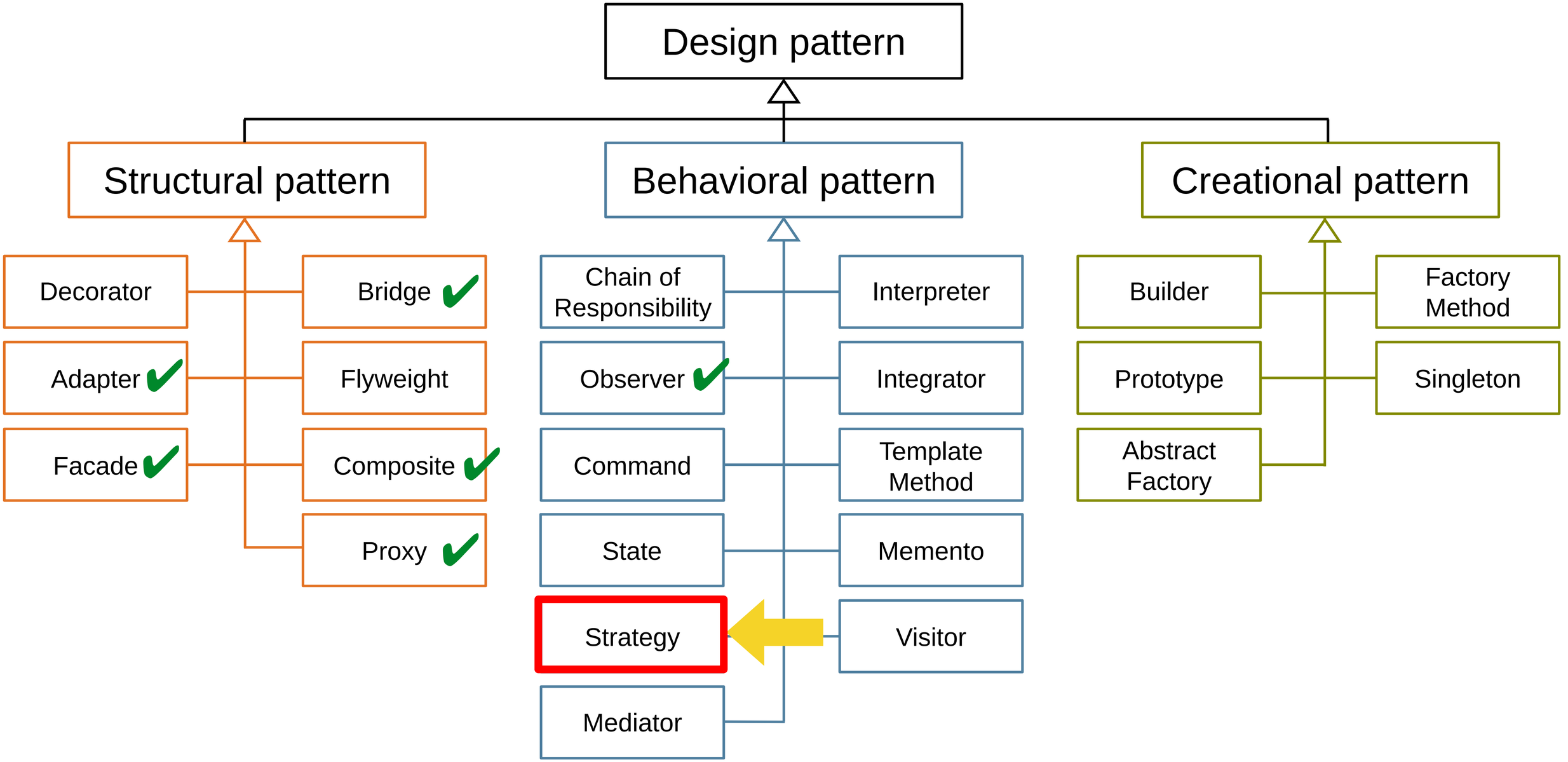
Hint: observer pattern in L07E03



Outline

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

Design patterns taxonomy

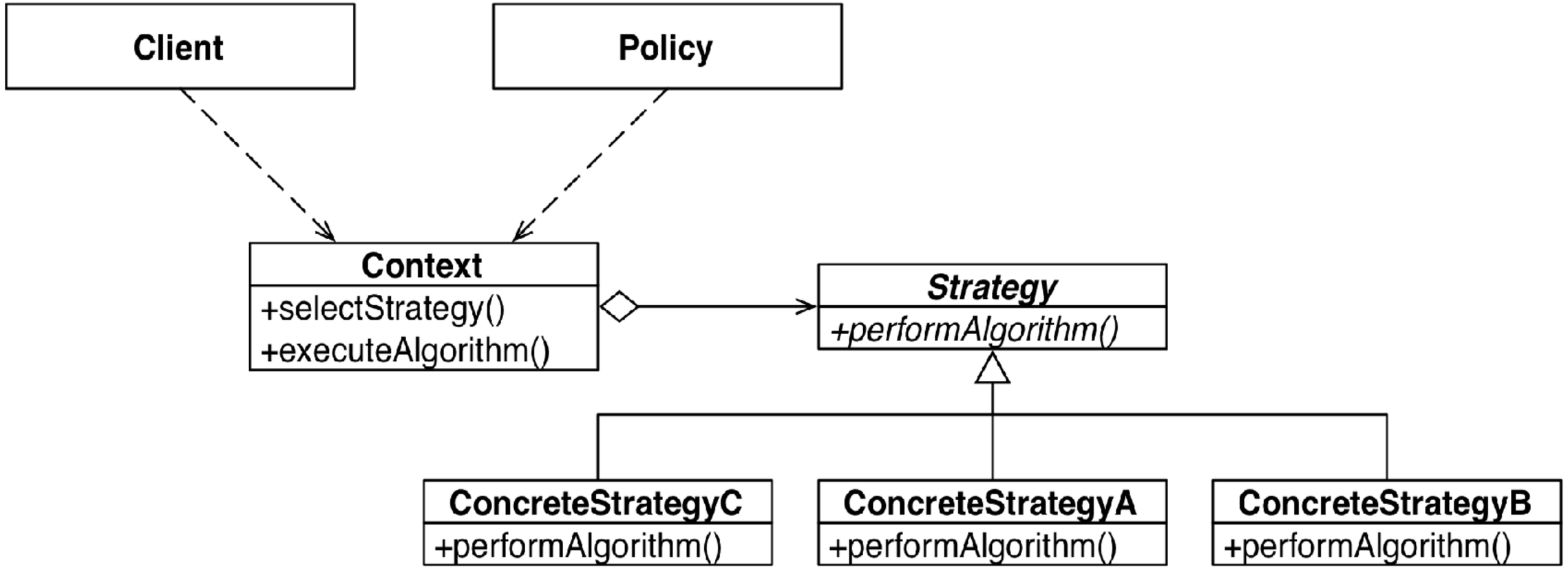


Strategy pattern

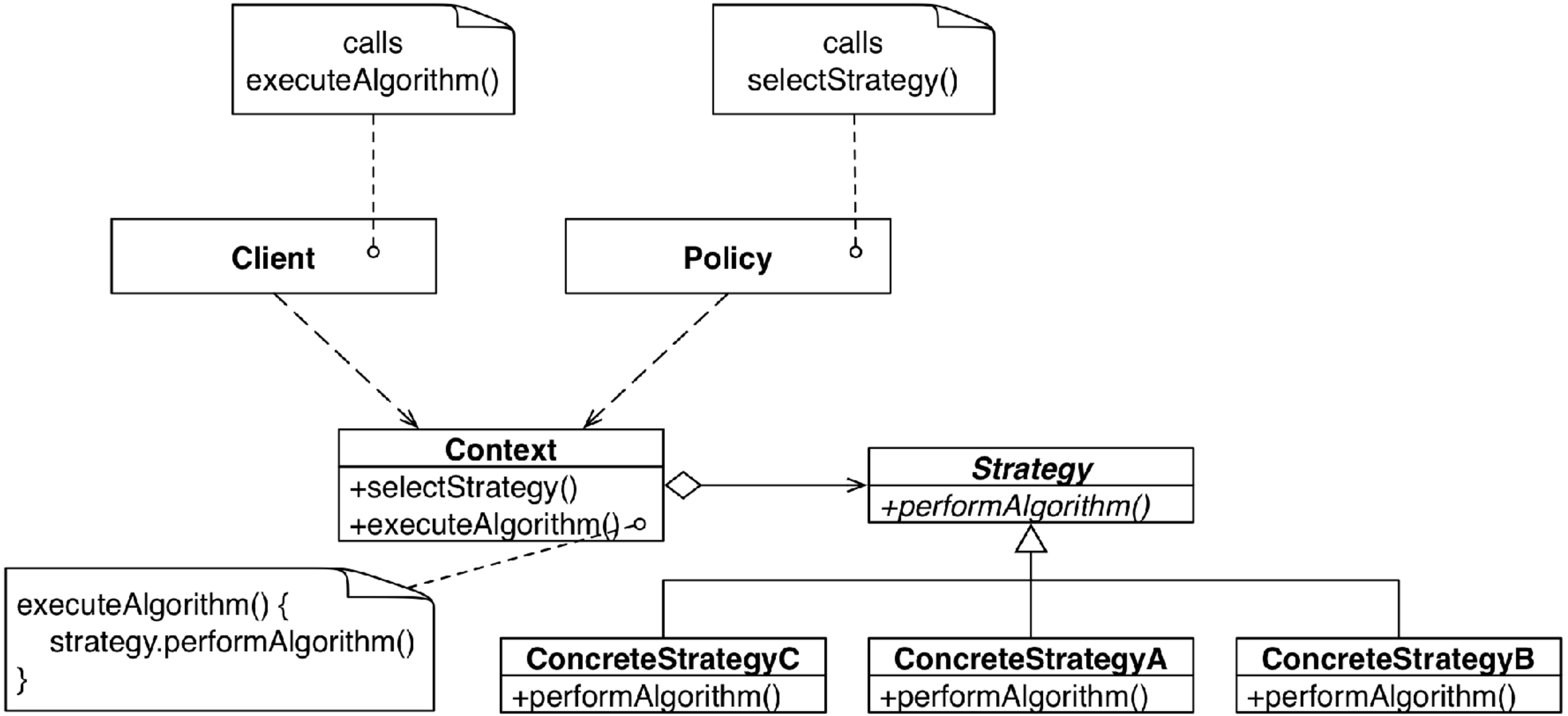
- **Problem:** different algorithms exist for a specific task
- Examples of specific tasks
  - Different ways to sort a list (bubble sort, merge sort, quick sort)
  - Different collision strategies for objects in video games
  - Different ways to parse tokens into an abstract syntax tree (bottom-up, 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

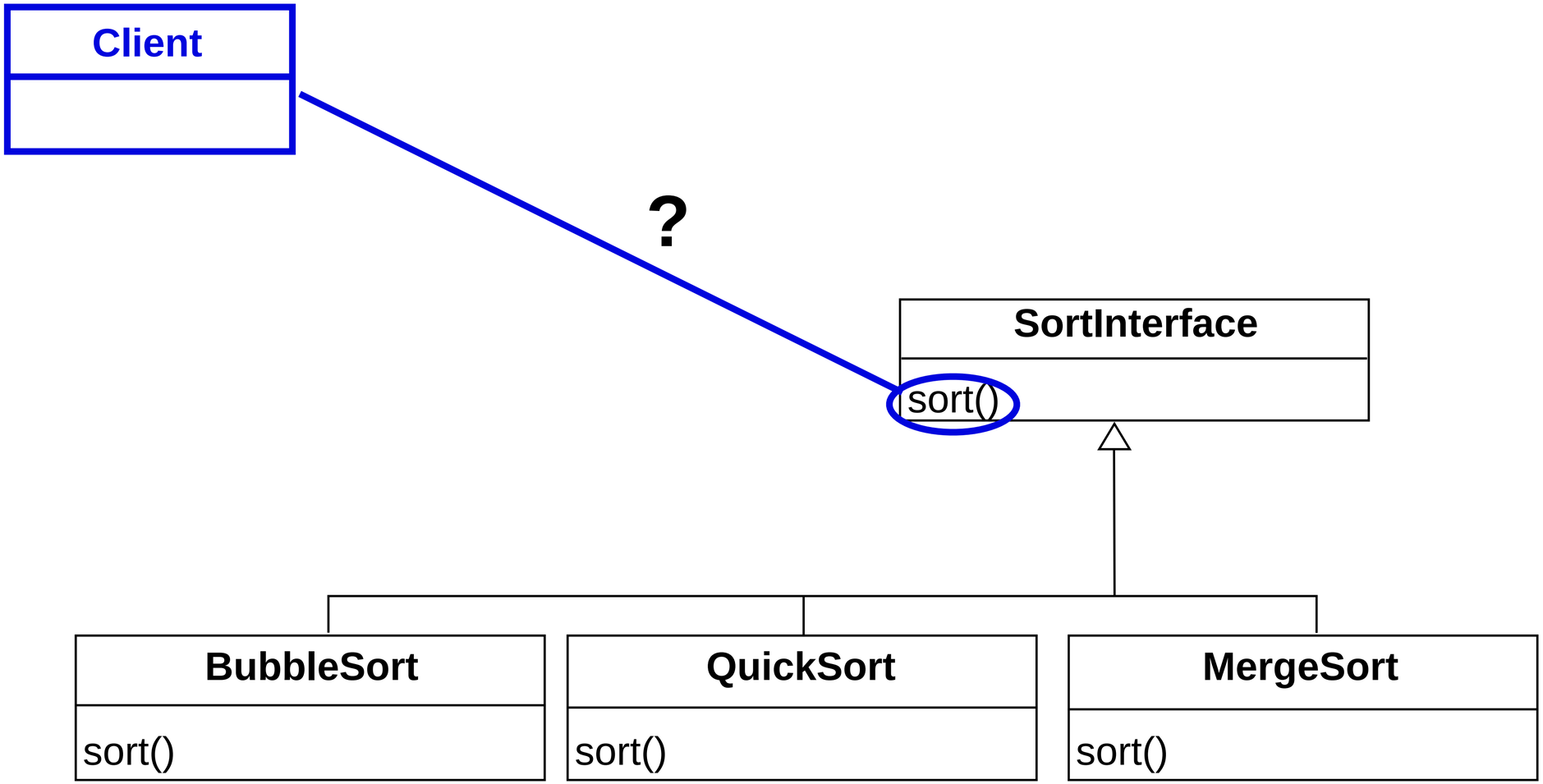
The **`Policy`** decides which **`ConcreteStrategy`** is best in a given **`Context`**



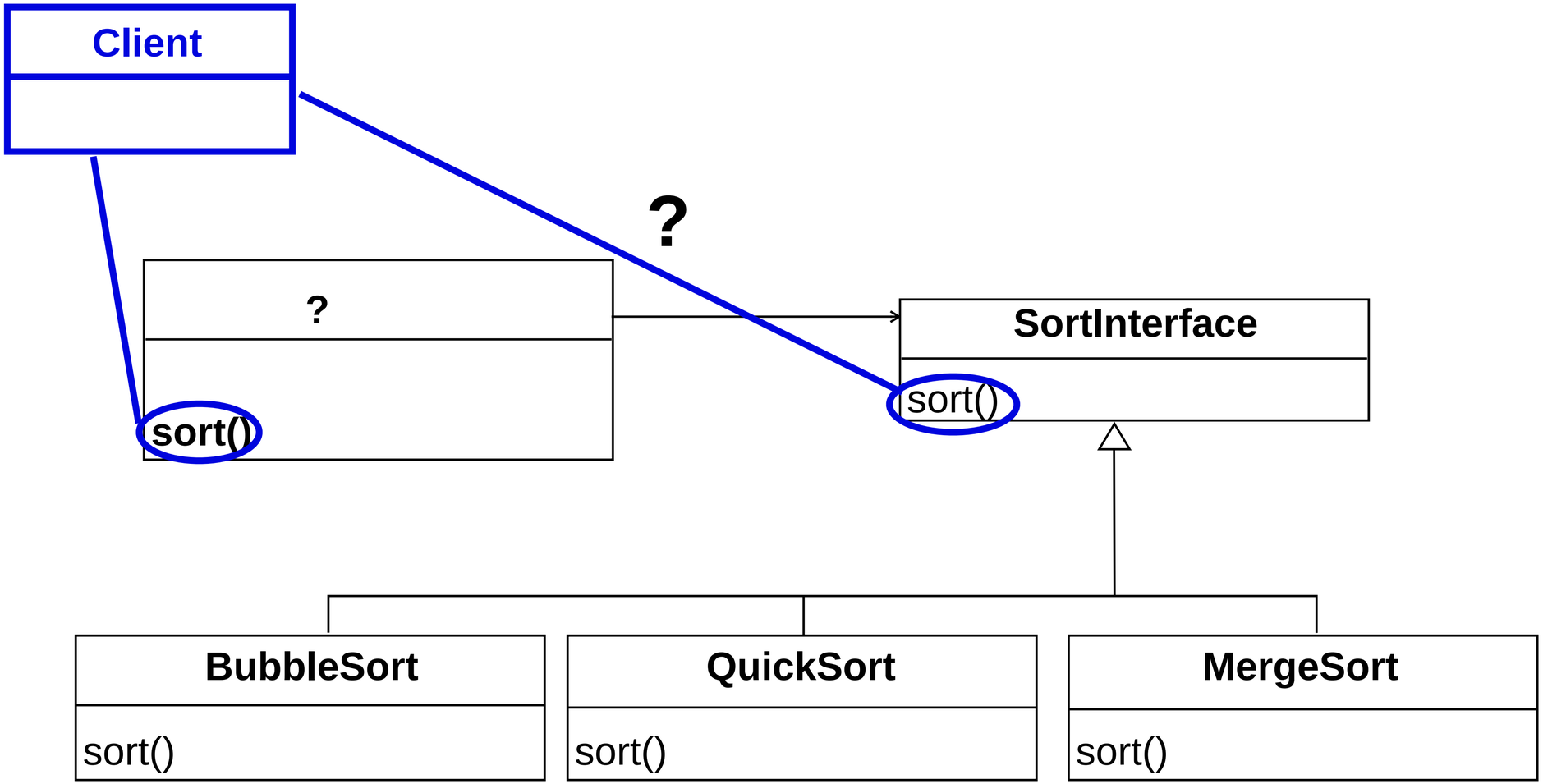
Strategy pattern: UML class diagram



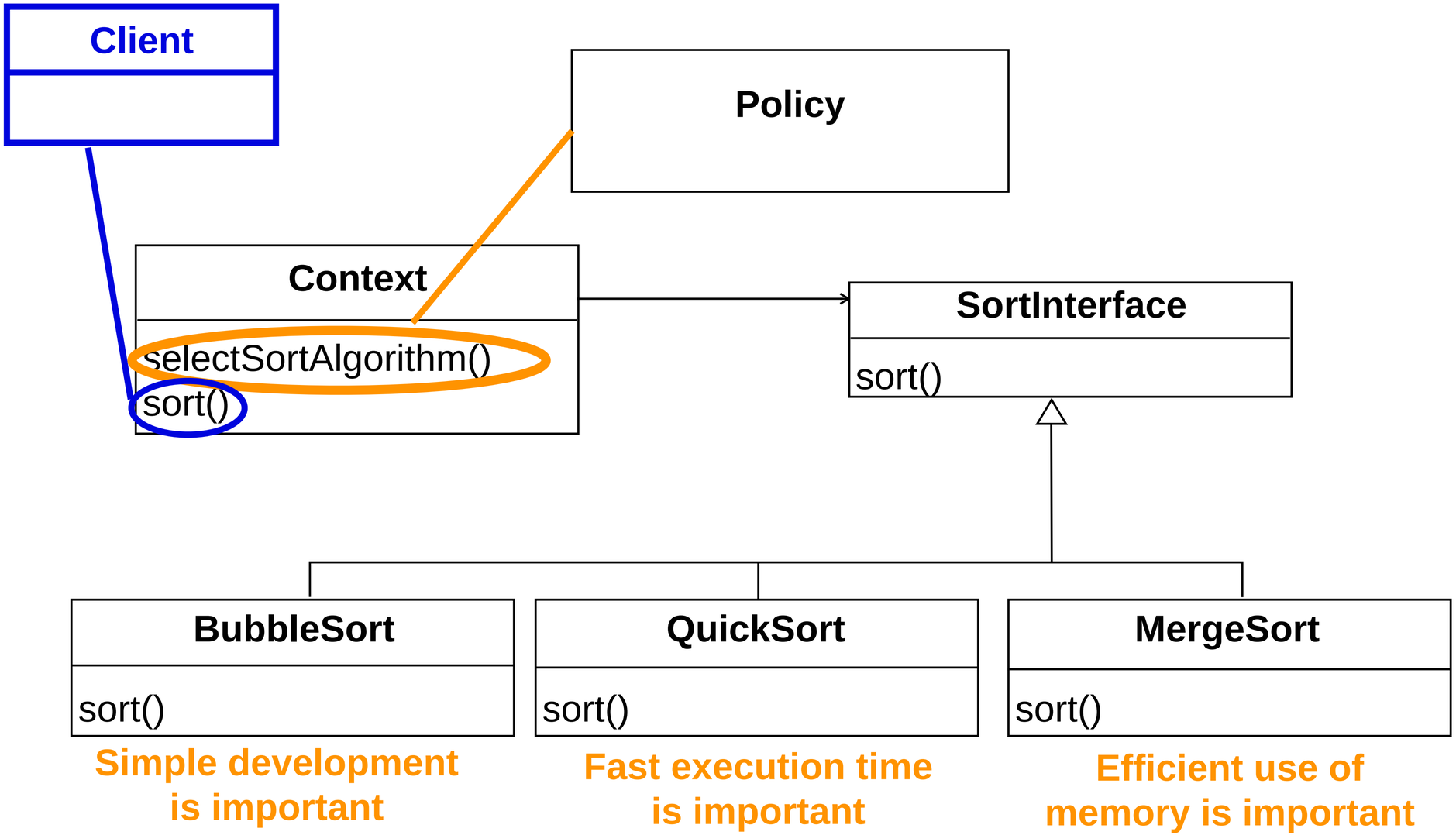
Example: using the strategy pattern to switch between different algorithms



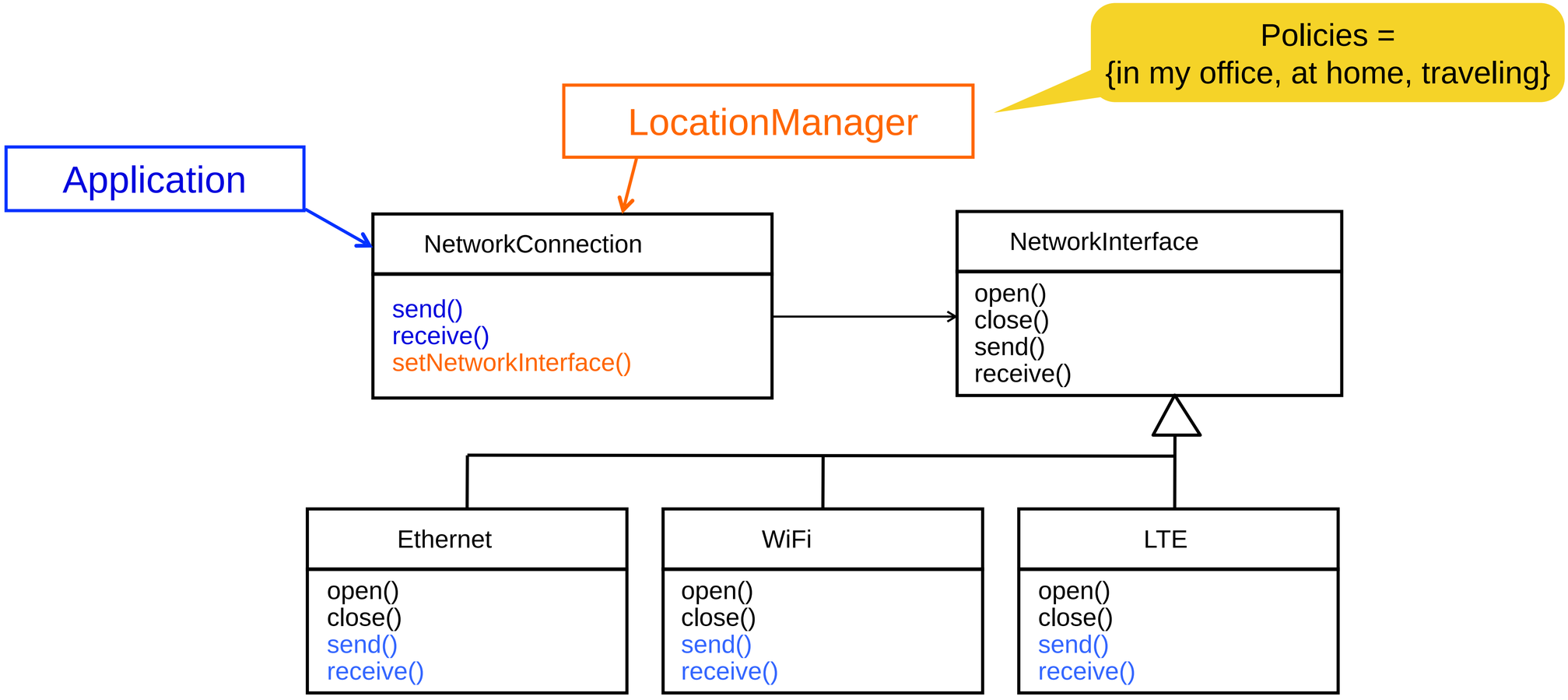
Example: using the strategy pattern to switch between different algorithms



Example: using the strategy pattern to switch between different algorithms



Supporting multiple implementations of a network connection

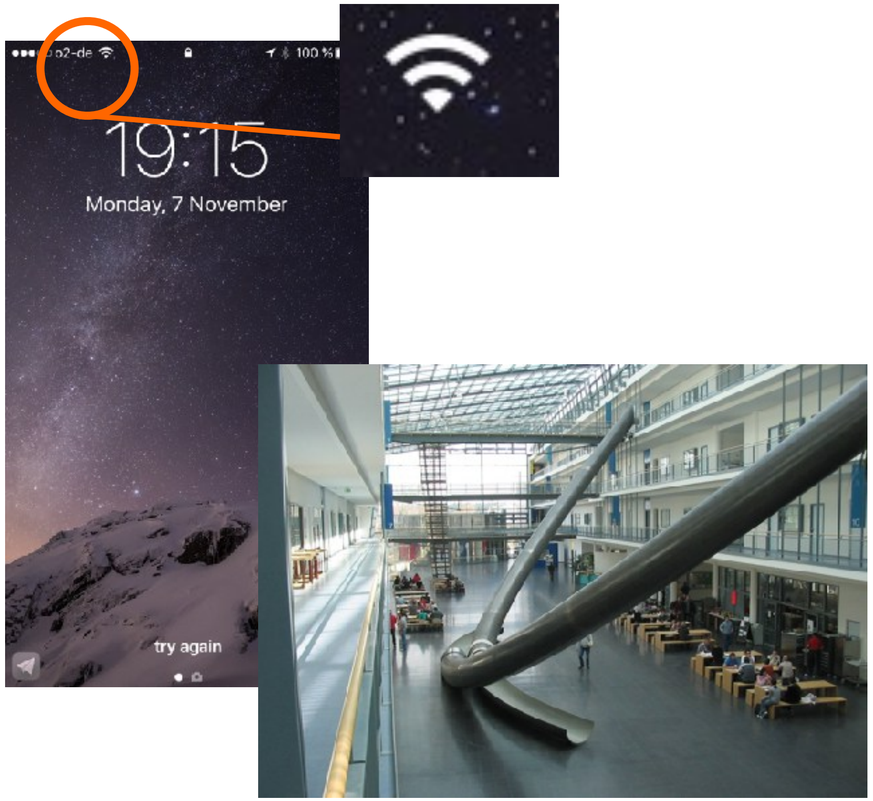


In my office: Local wired network

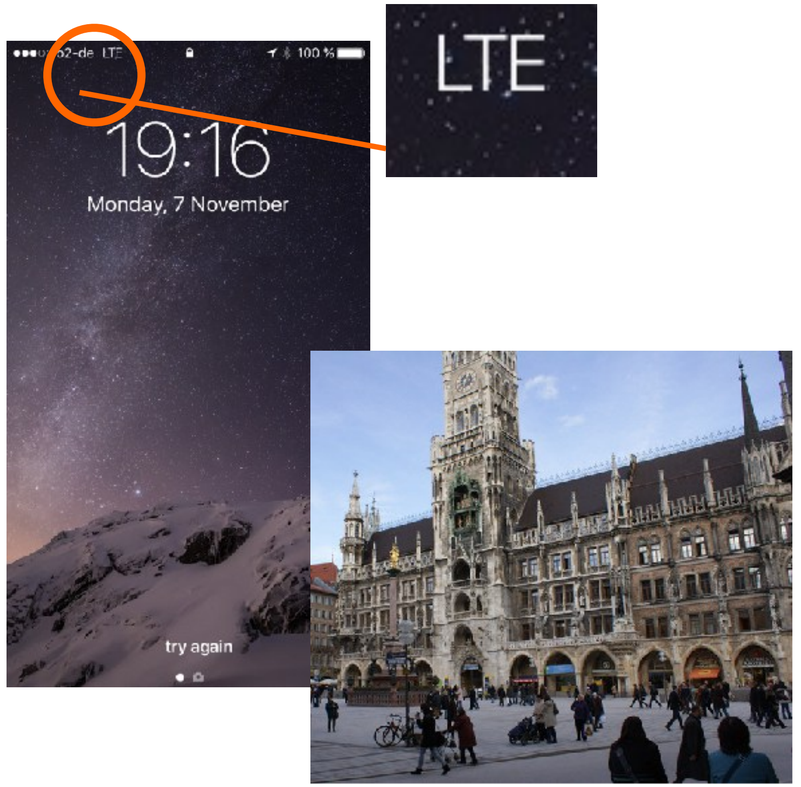
At home: Local wireless network

Traveling: Global wireless network

Another policy for network connections



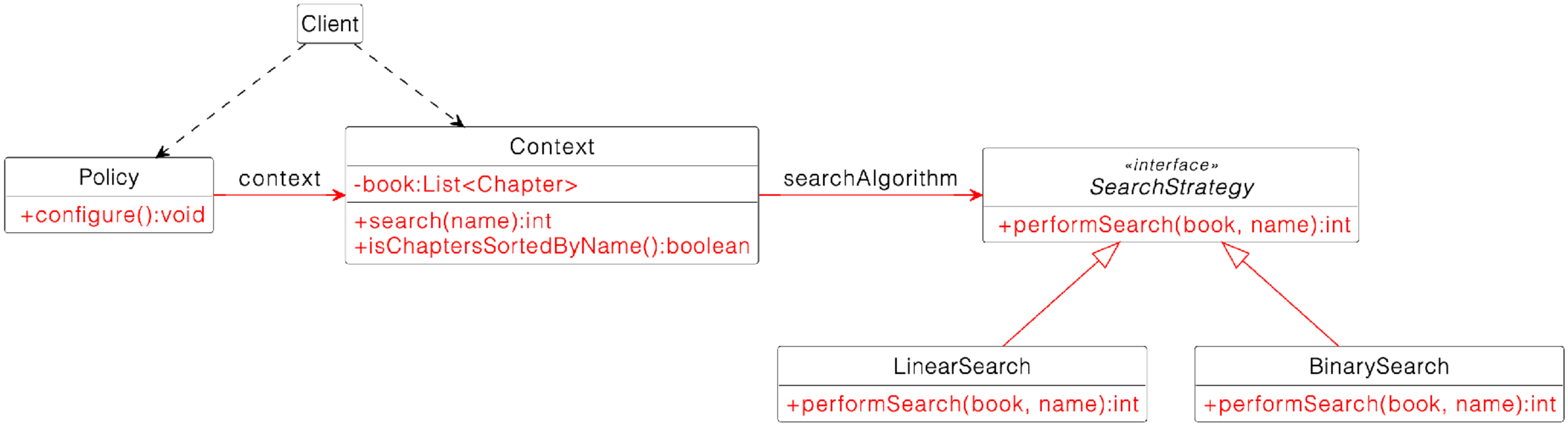
If WiFi available, use WiFi …



… 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**

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

Outline

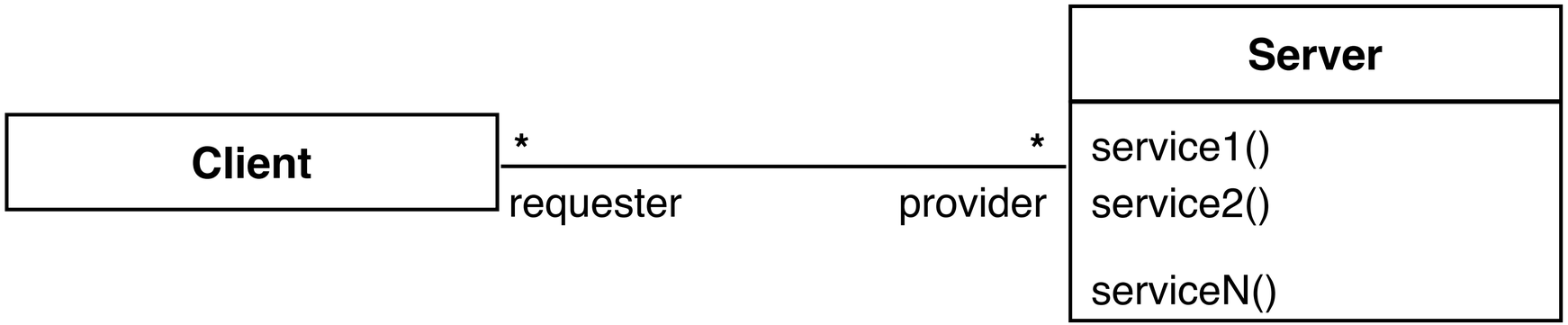
- Overview of system design
- Design goals
- Hints for system design
- Subsystem decomposition
- Façade pattern
- Architectural styles
  - Layered architecture
  - **Client server architecture**
  - REST architectural style
- UML component diagrams

Client server architecture

- Often used in the design of database systems
  - **Client**: user application
  - **Server**: database access and manipulation
  - **Client** requests a service from the **server**
- Functions performed by the client
  - Input by the user (customized user interface)
  - Sanity checks of input data
- Functions performed by the server
  - Centralized data management
  - Provision of data integrity and database consistency
  - Provision of database security

Client server architectural style

- One or more servers provide services to clients
- Each client calls a service offered by the server
  - **Server** performs service and returns result to client
  - **Client** knows interface of the server
  - **Server** does not know the interface of the client
- Response is typically immediate (i.e. less than a few seconds)
- End users interact only with the client

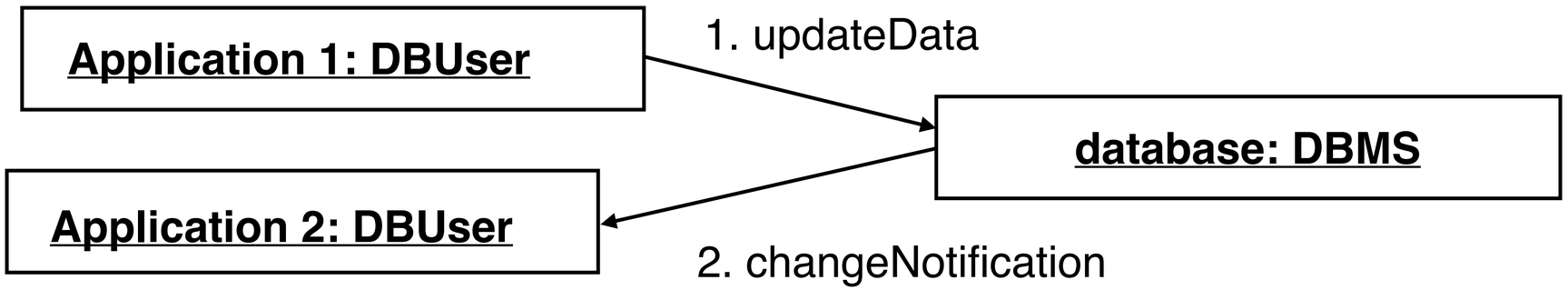


Design goals for client server architectures

Portability	Server runs on many operating systems and many networking environments
High performance	Client optimized for interactive display-intensive tasks Server optimized for CPU-intensive operations
Scalability	The server can handle large amounts of clients
Flexibility	The user interface of the client supports a variety of end-devices (phone, laptop, smart watch)
Reliability	Server should be able to handle client and communication problems

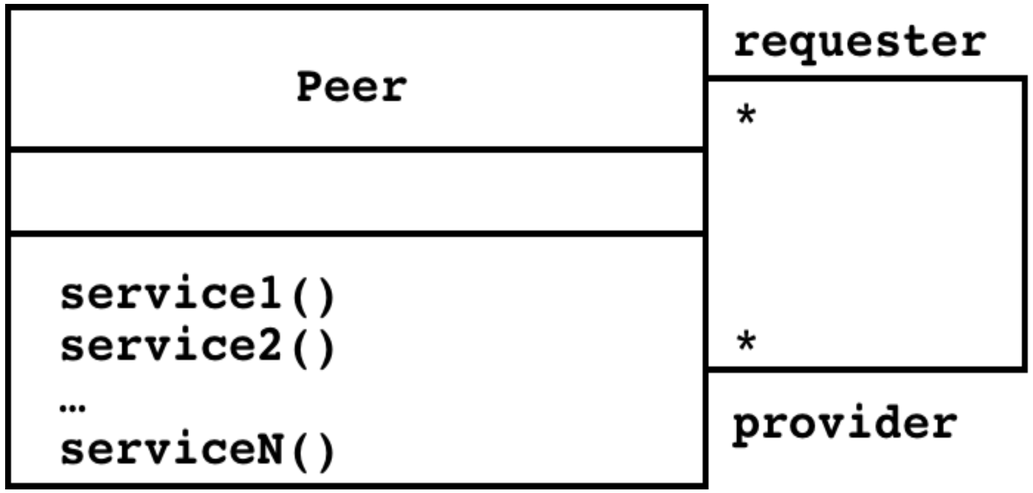
Problems with the client server architectural style

- Client server systems use a request-response protocol
- Peer to peer communication is often needed
- **Example:** a database must process queries from application 1 and should be able to send notifications to application 2 when data in the database has changed



Peer to peer architectural style

- Generalization of the client server architectural style
  - Clients can be servers and servers can be clients
- Introduction of a new abstraction: **Peer**



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

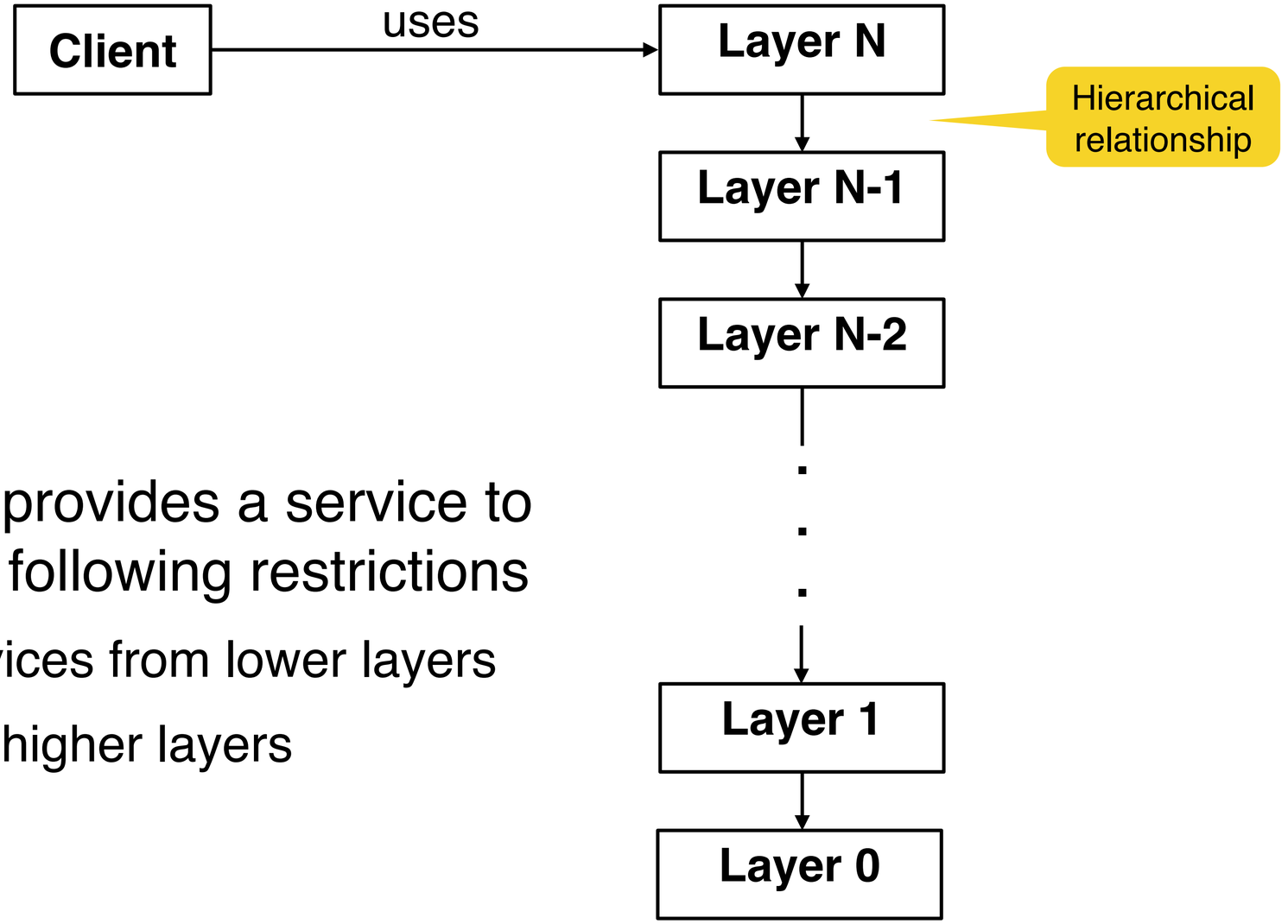
Architectural style vs. architecture

- **Subsystem decomposition:** identification of subsystems, services, and their relationships to each other

- **Architectural style:** a pattern for a subsystem decomposition

- **Software architecture:** instance of an architectural style

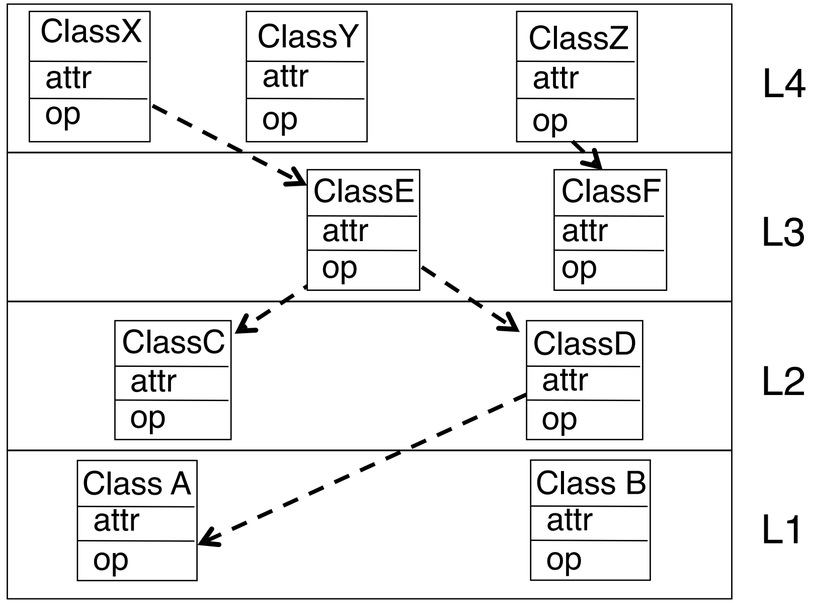
Layered architectural style



- A **layer** is a subsystem that provides a service to another subsystem with the following restrictions
  - A layer only depends on services from lower layers
  - A layer has no knowledge of higher layers

Closed architecture (opaque layering)

A **layered architecture is closed**, if each layer can only call operations from the layer directly below (also called “direct addressing”)

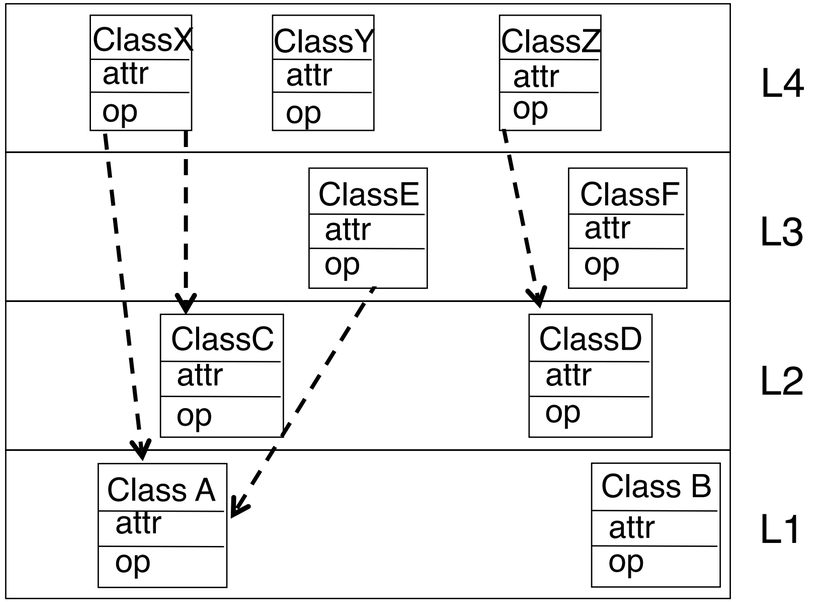


Design goals: maintainability, flexibility, portability

more portable ➡ **low coupling** 😄, but potentially a **bottleneck**

Open architecture (transparent layering)

A **layered architecture is open** if a layer can call operations from any layer below (also called “indirect addressing”)



Design goals: high performance, real-time operations support

more **efficient** ➡ **high coupling** 😞

3 layered architectural style

- Often used for the development of web applications
- **Example**
  1) The **web browser** implements the user interface
  2) The **web server** serves requests from the web browser
  3) The **database** manages and provides access to the persistent data

Layer vs. tier

- 3 layered architectural style: an **architectural style** where an application consists of 3 hierarchically ordered layers
- 3 tier architecture: a **software architecture** where the 3 layers are allocated on 3 separate hardware nodes
- **Note:** Layer is a type (e.g. class, subsystem) and tier is an instance (e.g. object, hardware node)
- In practice, the terms layer and tier are often used interchangeably (when blurring the distinction between type and instance is admissible)

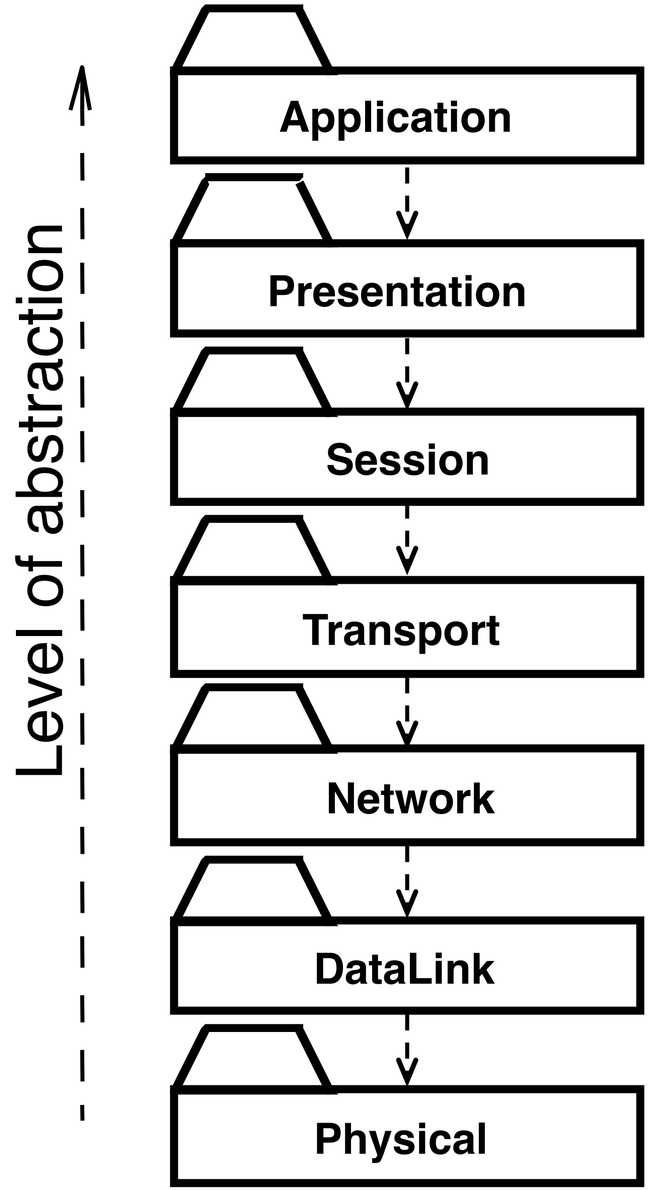
4 layered architectural style

- Hierarchically ordered layers
- **Example**
  1) A **web browser** provides the user interface
  2) A **web server** serves static HTML requests
  3) An **application server** provides session management (for example the contents of an electronic shopping cart) and processes dynamic HTML requests
  4) A **database** manages and provides access to the persistent data
     - Usually a relational database management system (RDBMS)

➡ If these layers reside on different hardware nodes, then it is a 4 tier architecture

7 layered architectural style

- ISO’s OSI Reference Model
  - ISO = International Standard Organization
  - OSI = Open System Interconnection
- The reference model defines 7 layers and communication protocols between the layers

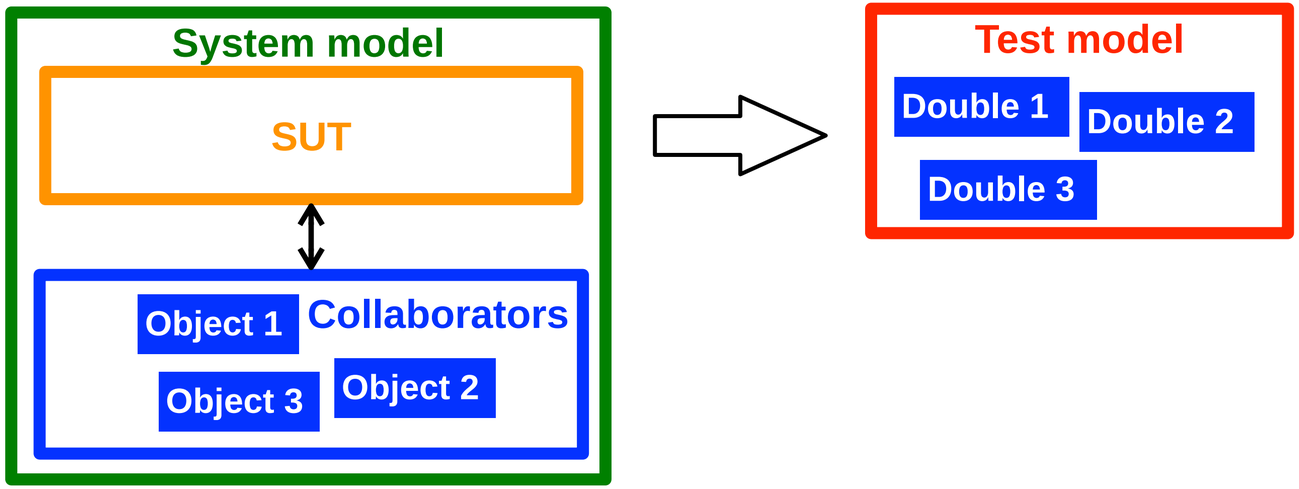


Outline

- Terminology
- Unit testing
- Integration testing
- System testing
- Model based testing
- **Object oriented testing**

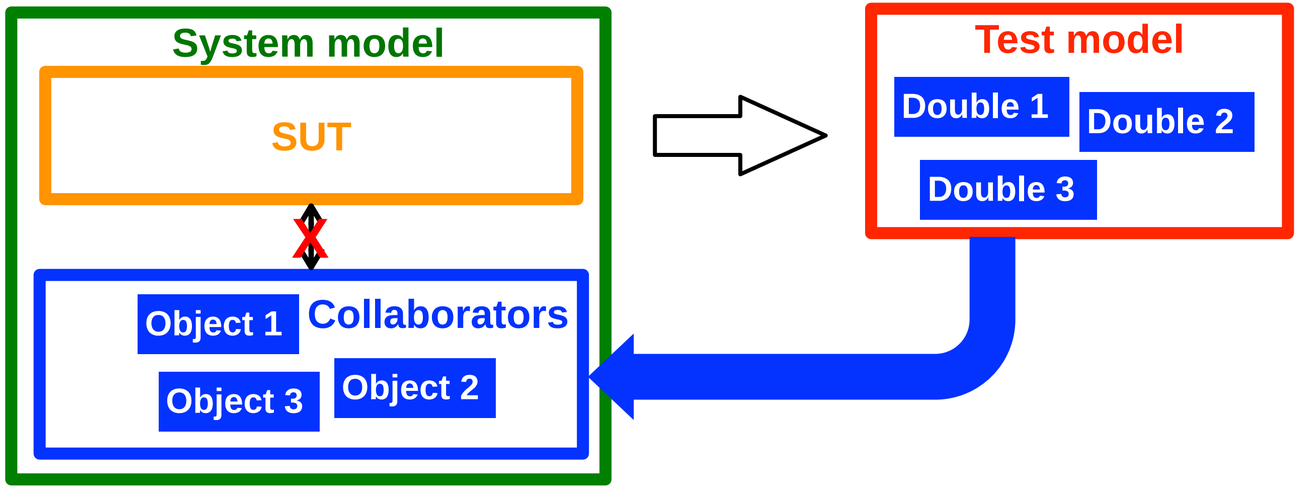
Object oriented test modeling

- Start with the **system model**
- The system contains the **SUT** (system under test)
- The **SUT** does not exist in isolation, it interacts with other participating objects in the system model that are not yet implemented: **collaborators**
- The **test model** is derived from the **SUT**
- To be able to interact with **collaborators**, we add objects to the **test model**
- These are called **test doubles**



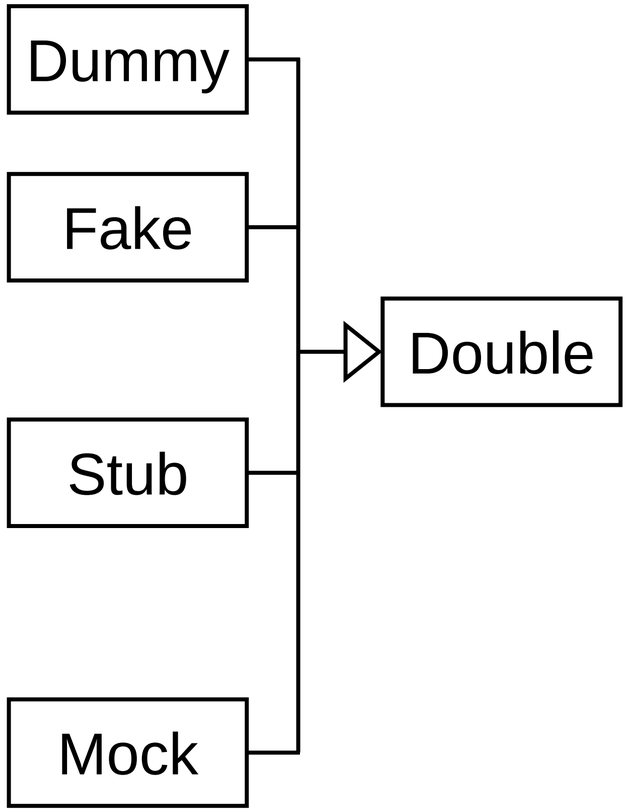
Object oriented test modeling

- Start with the **system model**
- The system contains the **SUT** (system under test)
- The **SUT** does not exist in isolation, it interacts with other participating objects in the system model that are not yet implemented: **collaborators**
- The **test model** is derived from the **SUT**
- To be able to interact with **collaborators**, we add objects to the **test model**
- These are called **test doubles** (substitutes for the **collaborators** during testing)



Taxonomy of test doubles

- **Dummy:** often used to fill parameter lists, passed around but never actually used
- **Fake:** a working implementation that contains a “shortcut” which makes it not suitable for production code
  - **Example:** a database stored in memory instead of on a disk
- **Stub:** provides canned answers (e.g. always the same) to calls made during the test
  - **Example:** random number generator that always return 3.14
- **Mock:** mimic the behavior of the real object and know how to deal with a specific sequence of calls they are expected to receive



Good design is crucial when using mock objects: the real object (subsystem) must be specified with an interface (façade) and a class for the implementation

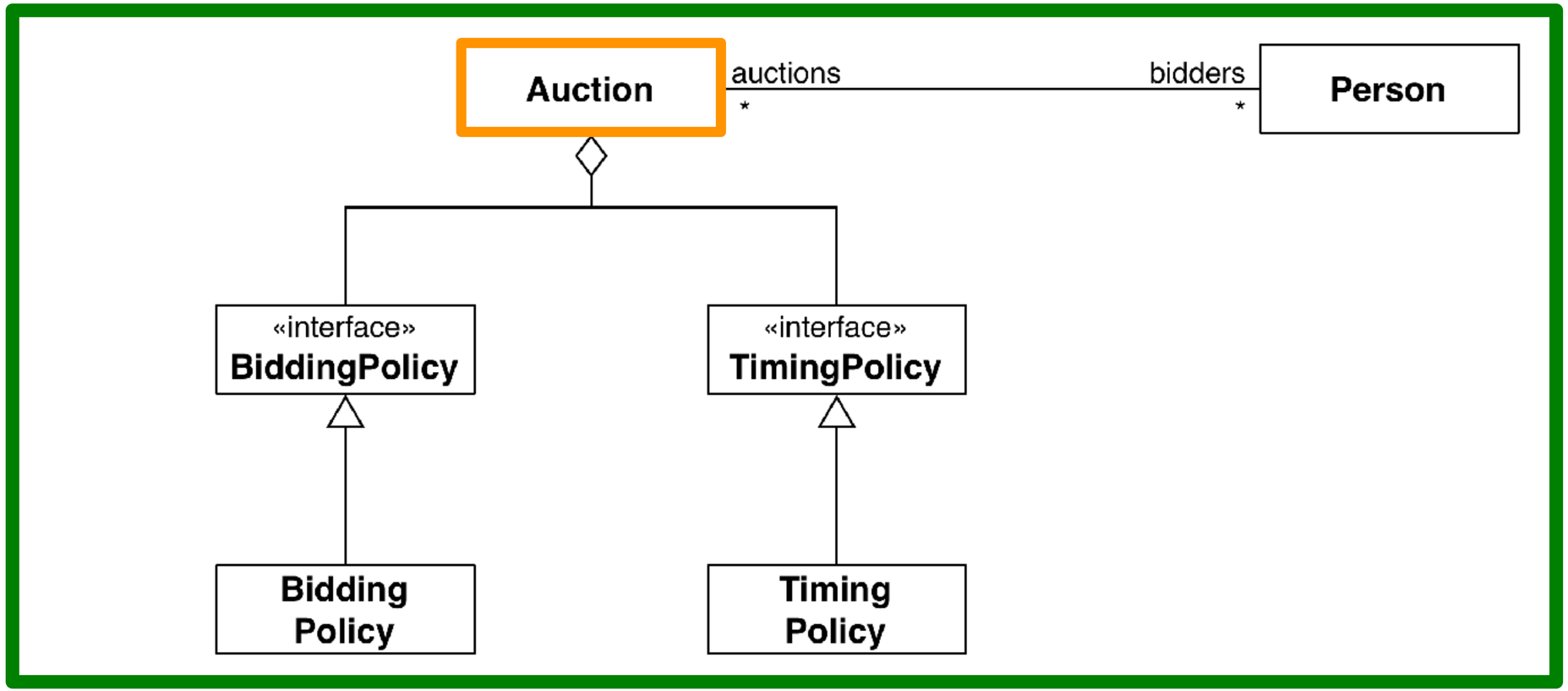
Testing patterns

- Mock object pattern
- Test driven development
- Reflection test pattern
- Four stage testing pattern

Want to learn more?
WS Course: Patterns in Software Engineering

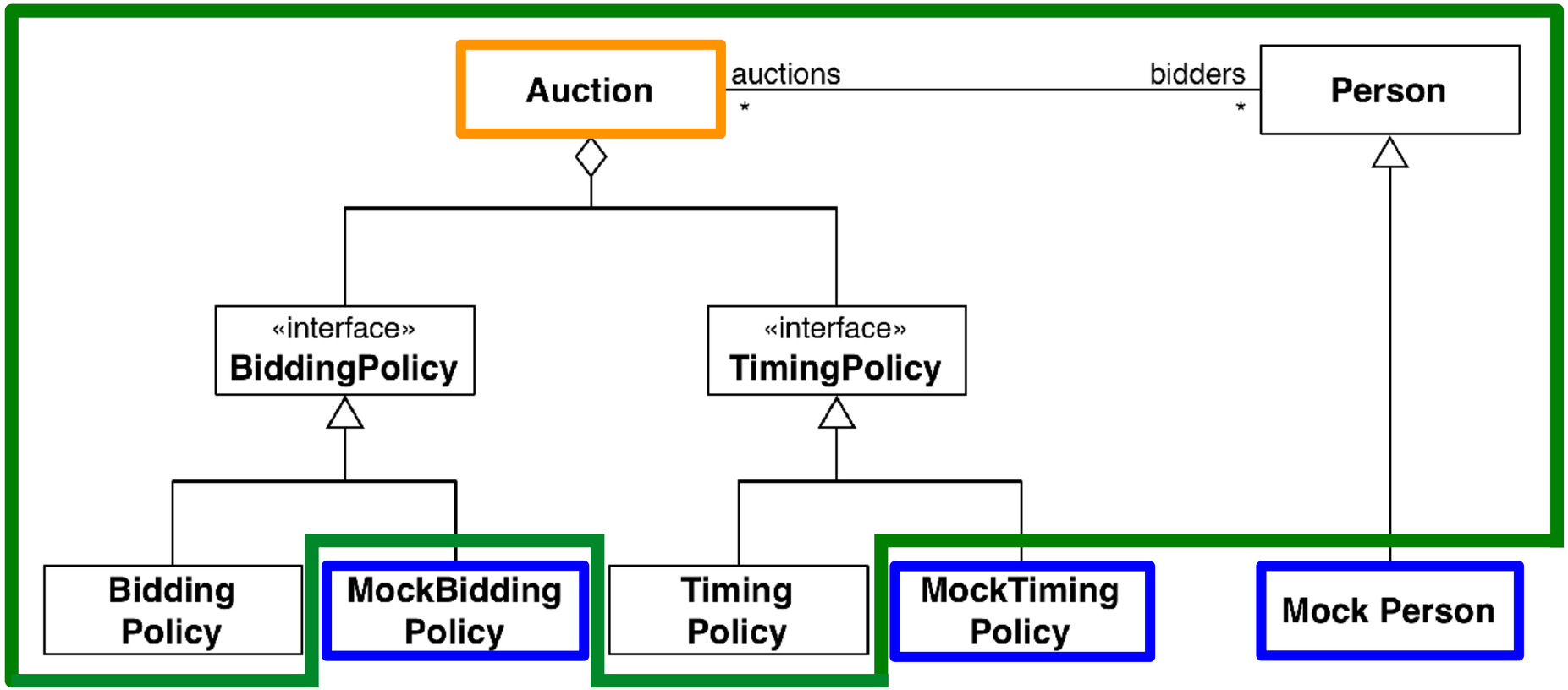
Motivation for mock objects

- There is a **system model** for an auction system with 2 types of policies
- We want to unit test **`Auction`**, which is the **SUT**

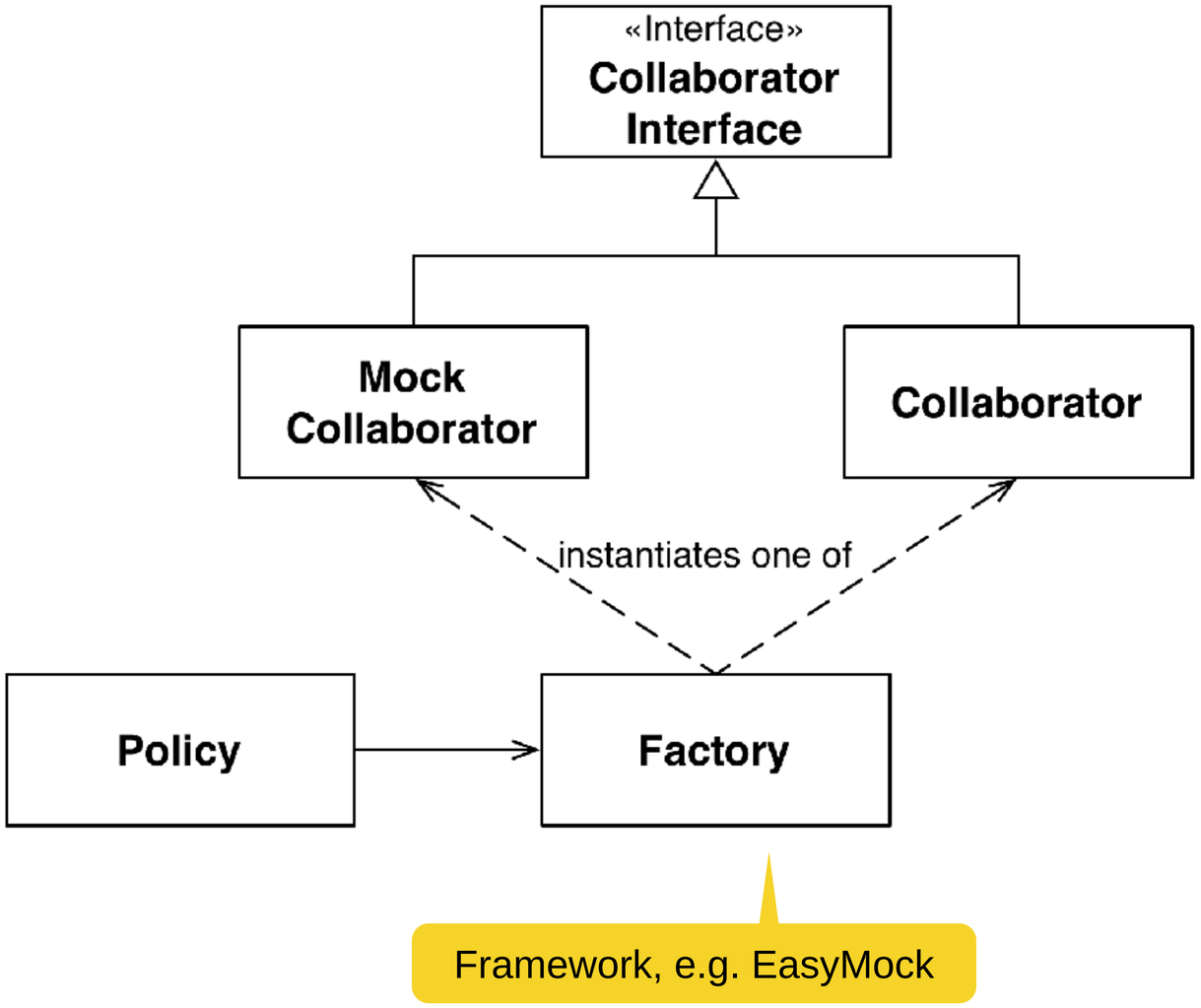


Motivation for mock objects

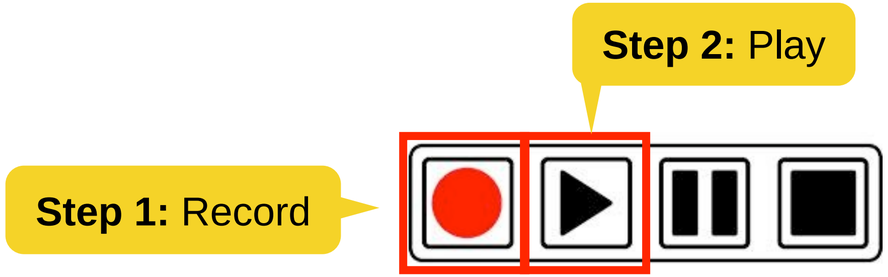
- There is a **system model** for an auction system with 2 types of policies
- We want to unit test **Auction**, which is the **SUT**
- The mock object test pattern is based on the idea to replace the interaction with the collaborators in the system model, that is **`Person`**, **`BiddingPolicy`** and **`TimingPolicy`**, by **mock objects**
- These mock objects are created at startup time



Mock object pattern



- A **mock object** replaces the behavior of a real object called the collaborator and returns hard-coded values
- A mock object can be created at startup time with the **factory pattern**
- Mock objects can be used for testing the state of individual objects and the interaction between objects
- The use of mock objects is based on the **record play metaphor**



Record play metaphor

Assume you want to perform a musical, which requires an orchestra and a choir. Most of the time the orchestra will not be available (too expensive), when the choir practices. But the choir needs to be accompanied by the music played by the orchestra when rehearsing the musical:

Rehearsal of a musical

Choir

During rehearsal the choir must sing in sync with the music on the mock (tape)

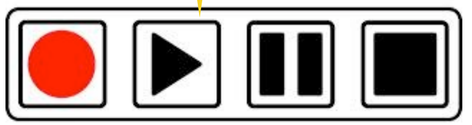
Orchestra

The tape is the mock

Tape

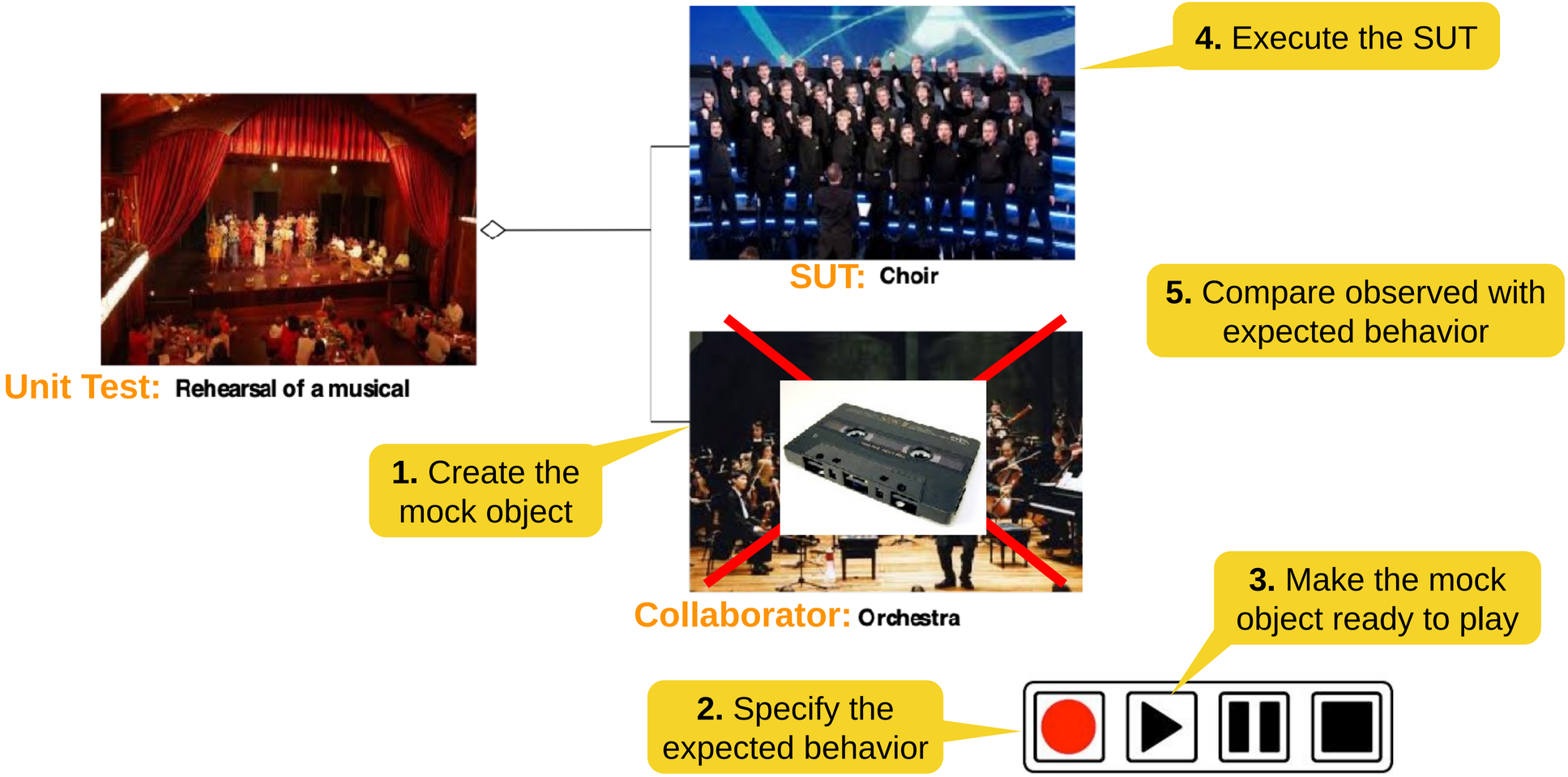
During rehearsal the music on the mock is played

The orchestra records the music onto the mock



Record play metaphor for mock objects

Mock objects are proxy collaborators in tests where the real collaborators are not available



EasyMock

- Open source testing framework for Java
- Uses annotations for test subjects (=**SUT**) and mocks

```
@TestSubject
private ClassUnderTest classUnderTest = new ClassUnderTest();

@Mock
private Collaborator mock;
```

- Specification of the behavior

```
expect(mock.invoke(parameter)).andReturn(42);
```

- Make the mock ready to play

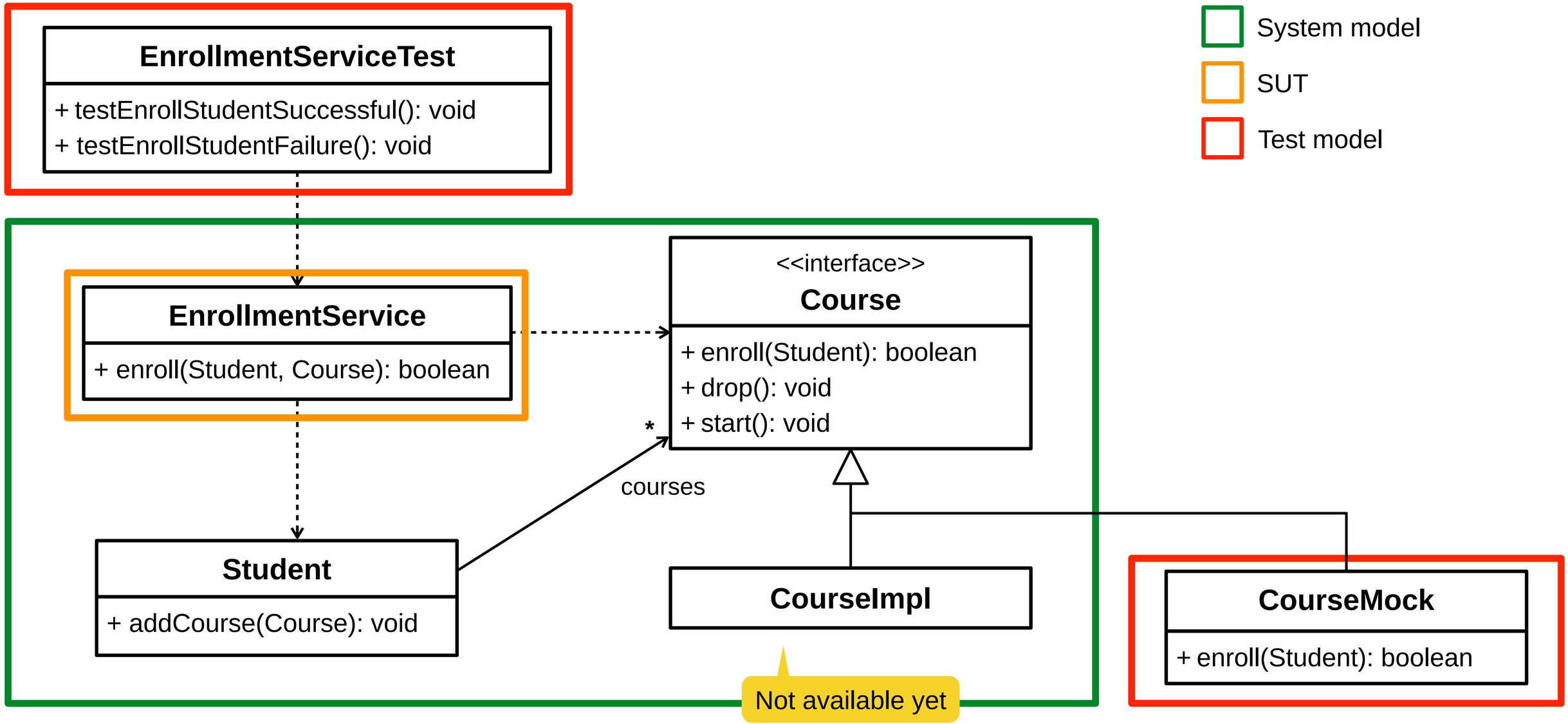
```
replay(mock);
```

- Make sure the mock has actually been called in the test (additional assertion)

```
verify(mock);
```

- Documentation: http://easymock.org/user-guide.html

Example: university app with a mock object



L08E03 Mock Object Pattern

Not started yet.

Start exercise

Easy

Due date: end of today

10 min

6 pts

- **Problem statement**
  - Apply the mock object pattern using EasyMock
  - Implement **`testEnrollStudentSuccessful()`**
  - Optional challenge (2 bonus points): implement **`testEnrollStudentFailure()`**

Example solution: unit test for enrolling students with EasyMock

```
@ExtendWith(EasyMockExtension.class)
class EnrollmentServiceTest {

    @TestSubject
    private EnrollmentService enrollmentService = new EnrollmentService();

    @Mock
    private Course courseMock;

    @Test
    void testEnrollStudentSuccessful() {

        Student student = new Student();
        int expectedSize = student.getCourses().size() + 1;
        expect(courseMock.enroll(student)).andReturn(true);

        replay(courseMock);

        enrollmentService.enroll(student, courseMock);

        assertEquals(expectedSize, student.getCourses().size());

        verify(courseMock);
    }
}
```

1. Instantiate the SUT

2. Create the mock object

3. Specify the expected behavior

4. Make the mock object ready to play

5. Execute the SUT

6. Validate observed against expected behavior

7. Verify that **`enroll()`** was invoked on **`courseMock`** once

Best practices for testing (1)

- Use prefixes **`actual*`** and **`expected*`**
- Use fixed data instead of randomized data
- Write small and specific tests: do **not** extend tests to “just one more tiny thing”
- Insert test data right in the test method
- Favor composition over inheritance
- Dumb tests are great: compare the output with hard coded values: do not reuse or rewrite production logic

Source: https://phauer.com/2019/modern-best-practices-testing-java

Best practices for testing (2)

- Focus on testing a complete functional requirement (vertical slide)
- Use constructor injection if possible
- Separate asynchronous execution and actual logic
- Use parameterized tests
- Mock remote services (e.g. using Spring MockMvc)

Source: https://phauer.com/2019/modern-best-practices-testing-java

Homework

- **H08E01** Unit Tests (programming exercise)
- **H08E02** Mock Object Pattern (programming exercise)
- Read more about the **mock object pattern** on https://martinfowler.com/articles/mocksArentStubs.html
- Read more about **EasyMock** on http://easymock.org/user-guide.html
- Read more about **best practices for testing** on https://phauer.com/2019/modern-best-practices-testing-java

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

Summary

- Testing is difficult, but many rules and heuristics are available
- Unit testing with JUnit
  - Assertions
  - Annotations
- Integration testing
  - Horizontal vs. vertical testing
- System testing
  - Fuzzing
- Object oriented testing
  - Mock object pattern
  - EasyMock

http://www.youtube.com/watch?v=bzBkSDb07iA

Literature

- Kent Beck, Erich Gamma, Junit Cookbook http://junit.sourceforge.net/doc/cookbook/cookbook.htm
- JUnit 5: https://junit.org/junit5/
- Martin Fowler, Mocks are not Stubs: http://martinfowler.com/articles/mocksArentStubs.html
- Brown & Tapolcsanyi: Mock Object Patterns. In Proceedings of the 10th Conference on Pattern Languages of Programs, 2003. http://hillside.net/plop/plop2003/papers.html
- Herman Bruyninckx, Embedded Control Systems Design, WikiBook, Learning from Failure: http://en.wikibooks.org/wiki/Embedded_Control_Systems_Design/Learning_from_failure
- Joanne Lim, An Engineering Disaster: Therac-25
- http://www.bowdoin.edu/~allen/courses/cs260/readings/therac.pdf
- Peter G. Neumann, Computer-Related Risks, Addison-Wesley, ACM Press, 384 pages, 1995
- Philipp Hauer: Modern Best Practices for Testing in Java, https://phauer.com/2019/modern-best-practices-testing-java
- EasyMock: http://easymock.org/user-guide.html