Software Engineering - Design

Summer Term 2024
05 Design Patterns

Prof. Dr. Robert Lokaiczyk
Frankfurt University of Applied Sciences
Faculty of Computer Science and Engineering
robert.lokaiczyk@fb2.fra-uas.de

Agenda for this course

01) Introduction / Motivation

02) Architectures

03) Designing classes

04) Modular Design

05) Design patterns

06) Coding conventions / Refactoring

07) Cloud Software Engineering

08) CICD (Guest lecture)

09) Software testing

10) Project management

11) Exam preparation

Agenda for today

The roots of the pattern idea

The gang of four

Popular patterns:

- Iterator
- Observer
- Strategy
- (Abstract) Factory
- Facade

Inspiration for the pattern approach

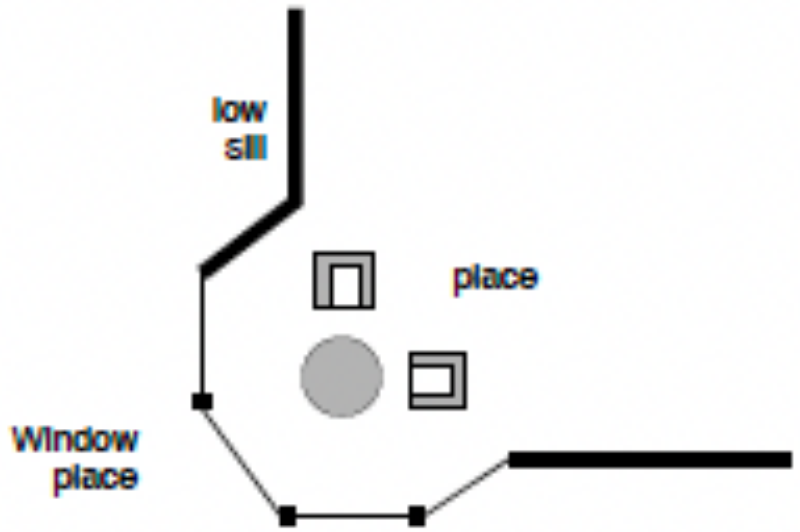
- **Christoper Alexander**: A Pattern language, The Timeless Way of Building (1977)
- Provides guidelines in building **city architecture**
- **Re-use and combine** patterns to assemble a solution

Source: Bryce Duffy/Corbis SABA

Example

Pattern Window Place

Source: Christoper Alexander: A Pattern language, The Timeless Way of Building (1977)



"Everybody loves **window seats**, bay windows, and big windows with **low sills and comfortable chairs** drawn up to them . . . A room which does not have a place like this seldom allows you to feel comfortable or perfectly at ease . . .

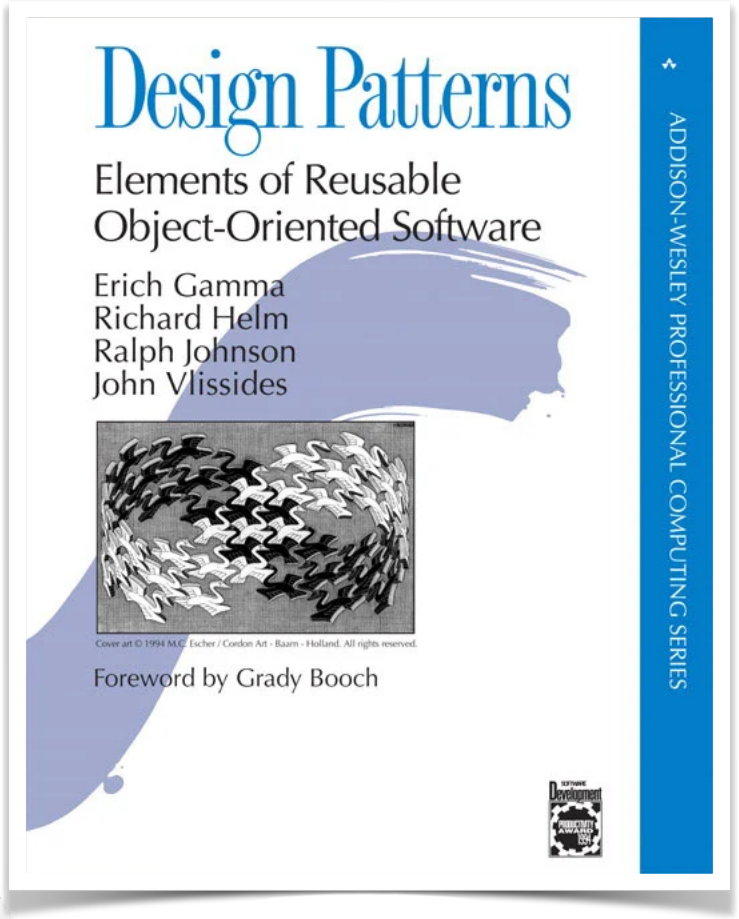
If the room contains no window which is a "place", a person in the room will be torn between two forces: He wants to sit down and be comfortable. He is drawn toward the light.

Obviously, if the comfortable places – those places in the room where you most want to sit – are away from the windows, there is no way of overcoming this conflict . . .

Therefore: In every room where you spend any length of time during the day, make at least one window into a 'window place'."

Alexander describes patterns in natural language using a consistent format which contains the patterns name, the given context, the forces that act and the solution.

The gang of four



- Design Patterns: elements of Reusable Object-Oriented Software (1994)
- > 40th edition
- Evergreen: Sold 500.000+ times
- „The design patterns are descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context.“

GoF patterns

Creational Design Patterns	Structural Design Patterns	Behavioural Design Patterns
• Singleton Pattern • Factory Pattern • Abstract Factory Pattern • Builder Pattern • Prototype Pattern	• Adapter Pattern • Composite Pattern • Proxy Pattern • Flyweight Pattern • Façade Pattern • Bridge Pattern • Decorator Pattern	• Template Method Pattern • Mediator Pattern • Chain of Responsibility Pattern • Observer Pattern • Strategy Pattern • Command Pattern • State Pattern • Visitor Pattern • Interpreter Pattern • Iterator Pattern • Memento Pattern

Creational patterns

deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. They aim to abstract the instantiation process, making the system independent of how its objects are created, composed, and represented.

- **Singleton**: Ensures a class has only one instance and provides a global point of access to that instance.
- **Factory Method**: Defines an interface for creating an object but allows subclasses to alter the type of objects that will be created.
- **Abstract Factory**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder**: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
- **Prototype**: Creates new objects by copying an existing object, known as a prototype, through cloning.

Structural patterns

focus on how classes and objects are composed to form larger structures. They deal with object composition or assembling objects into larger structures while keeping these structures flexible and efficient.

- **Adapter**: Allows incompatible interfaces to work together by wrapping an interface around an existing class.
- **Bridge**: Decouples an abstraction from its implementation so that the two can vary independently.
- **Composite**: Composes objects into tree structures to represent part-whole hierarchies. Clients can treat individual objects and compositions of objects uniformly.
- **Decorator**: Adds behaviour or responsibilities to objects dynamically without affecting the behaviour of other objects from the same class.
- **Facade**: Provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use.
- **Proxy**: Provides a placeholder for another object to control access to it.

Behavioural patterns

are concerned with the interaction and responsibility assignment between objects. They identify common communication patterns between objects and realise these patterns. These patterns help in defining how objects interact in a loosely coupled manner.

- **Observer**: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Strategy**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- **Command**: Encapsulates a request as an object, thereby allowing for parameterisation of clients with queues, requests, and operations.
- **Chain of Responsibility**: Passes a request along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
- **Iterator**: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterator pattern

- The Iterator pattern is a **behavioural design pattern** that provides a way to **access elements of a collection sequentially** without exposing its underlying representation.
- It **separates the responsibility** of accessing and traversing the collection **from the collection** itself.
- The pattern typically involves defining an Iterator interface with methods like **hasNext() and next()**, and concrete classes implementing this interface to iterate over different types of collections.
- This pattern allows developers to iterate over collections in a uniform manner. Allows **code reusability** and therefore **flexibility**.

Image: Adam Ijachi https://www.linkedin.com/pulse/iterator-pattern-adam-ijachi-7g2hf/

Iterator pattern

UML

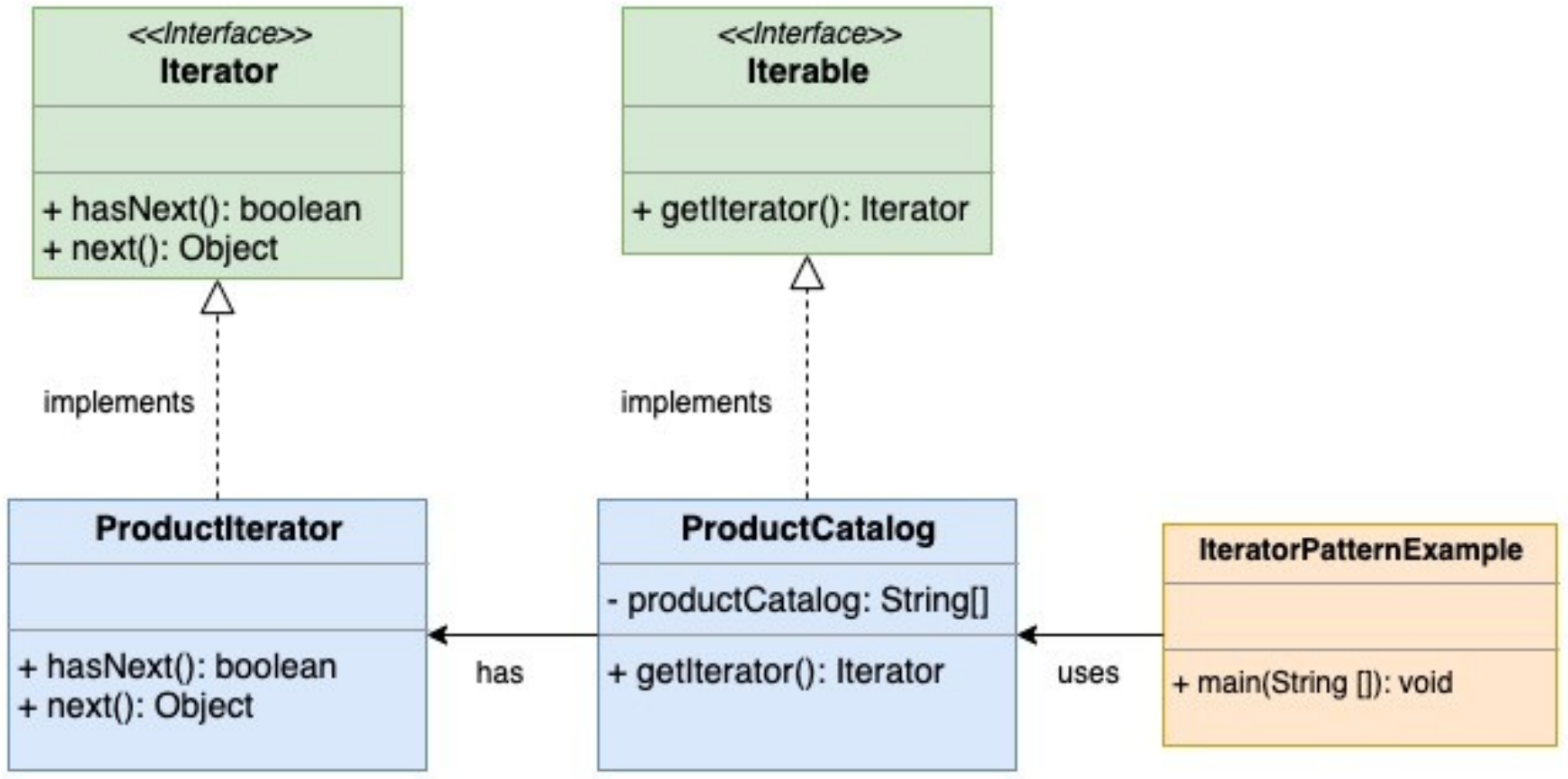


Image: Adam Ijachi https://www.linkedin.com/pulse/iterator-pattern-adam-ijachi-7g2hf/

Example

Base class

```java
// Define the Product class
class Product {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}
```

Example

```java
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

class ProductCatalog implements Iterable<Product> {
    private List<Product> products;

    public ProductCatalog() {
        products = new LinkedList<>();
    }

    // Method to add products to the catalog
    public void addProduct(String name, double price) {
        Product product = new Product(name, price);
        products.add(product);
    }

    // Implementing the iterator() method
    @Override
    public Iterator<Product> iterator() {
        return new ProductIterator();
    }

    // Inner class implementing Iterator<Product>
    private class ProductIterator implements Iterator<Product> {
        private int currentIndex = 0;

        @Override
        public boolean hasNext() {
            return currentIndex < products.size();
        }

        @Override
        public Product next() {
            return products.get(currentIndex++);
        }
    }
}
```

Example

```java
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Create a ProductCatalog
        ProductCatalog catalog = new ProductCatalog();
        catalog.addProduct("Laptop", 999.99);
        catalog.addProduct("Smartphone", 599.99);
        catalog.addProduct("Tablet", 399.99);

        // Iterate over the products using the Iterator
        System.out.println("Products in the catalog:");

        Iterator<Product> productIterator = catalog.iterator();
        while (productIterator.hasNext()) {
            Product product = productIterator.next();
            System.out.println("Name: " + product.getName() + ", Price: $" + product.getPrice());
        }
    }
}
```

Example

```
$ javac Main.java
$ java Main
Products in the catalog:
Name: Laptop, Price: $999.99
Name: Smartphone, Price: $599.99
Name: Tablet, Price: $399.99
$
```

Observer pattern

Pattern name	Observer
Description	**Separates the display of the state of an object** from the object itself and allows alternative displays to be provided. **When** the object **state changes**, all displays are **automatically notified** and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

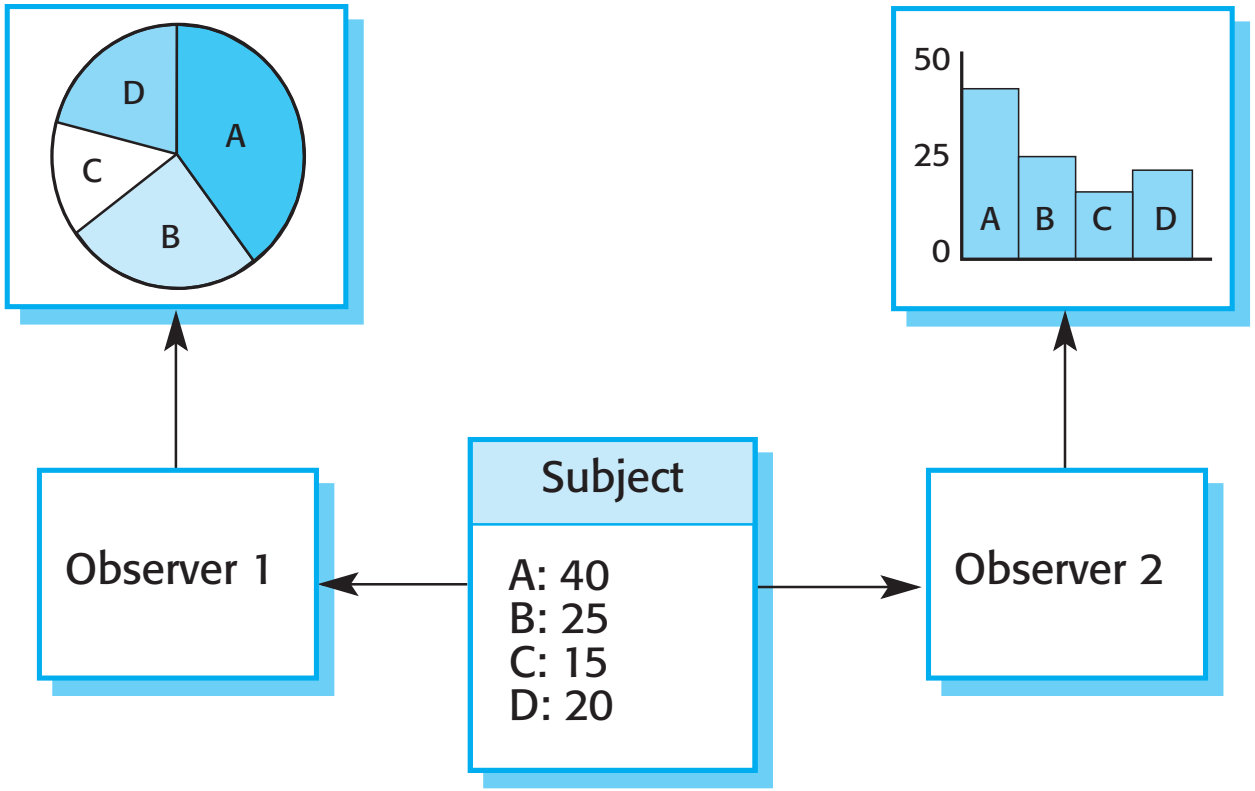
Source: Ian Sommerville - Software Engineering 2016

Observer pattern

Pattern name	Observer
Solution description	This involves two abstract objects, **Subject and Observer**, and two concrete objects, **ConcreteSubject and ConcreteObject**, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Source: Ian Sommerville - Software Engineering 2016

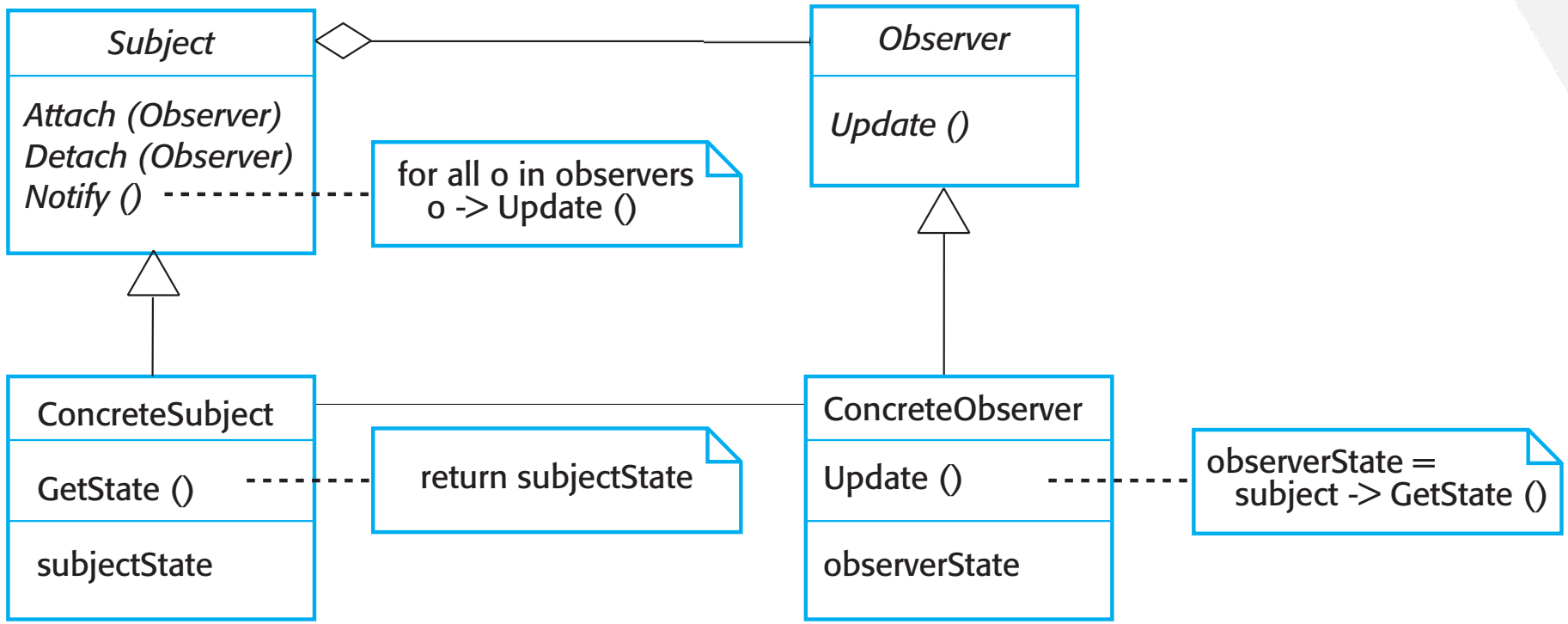
Example



Source: Ian Sommerville - Software Engineering 2016

Observer pattern

UML



Source: Ian Sommerville - Software Engineering 2016

Strategy pattern

- The Strategy pattern is a **behavioral design pattern** that enables an **algorithm**'s behavior to be **selected at runtime**.
- It involves defining a family of algorithms, encapsulating each one, and making them interchangeable.
- **Clients** can then **choose the appropriate algorithm** without modifying the client code

Strategy pattern

UML

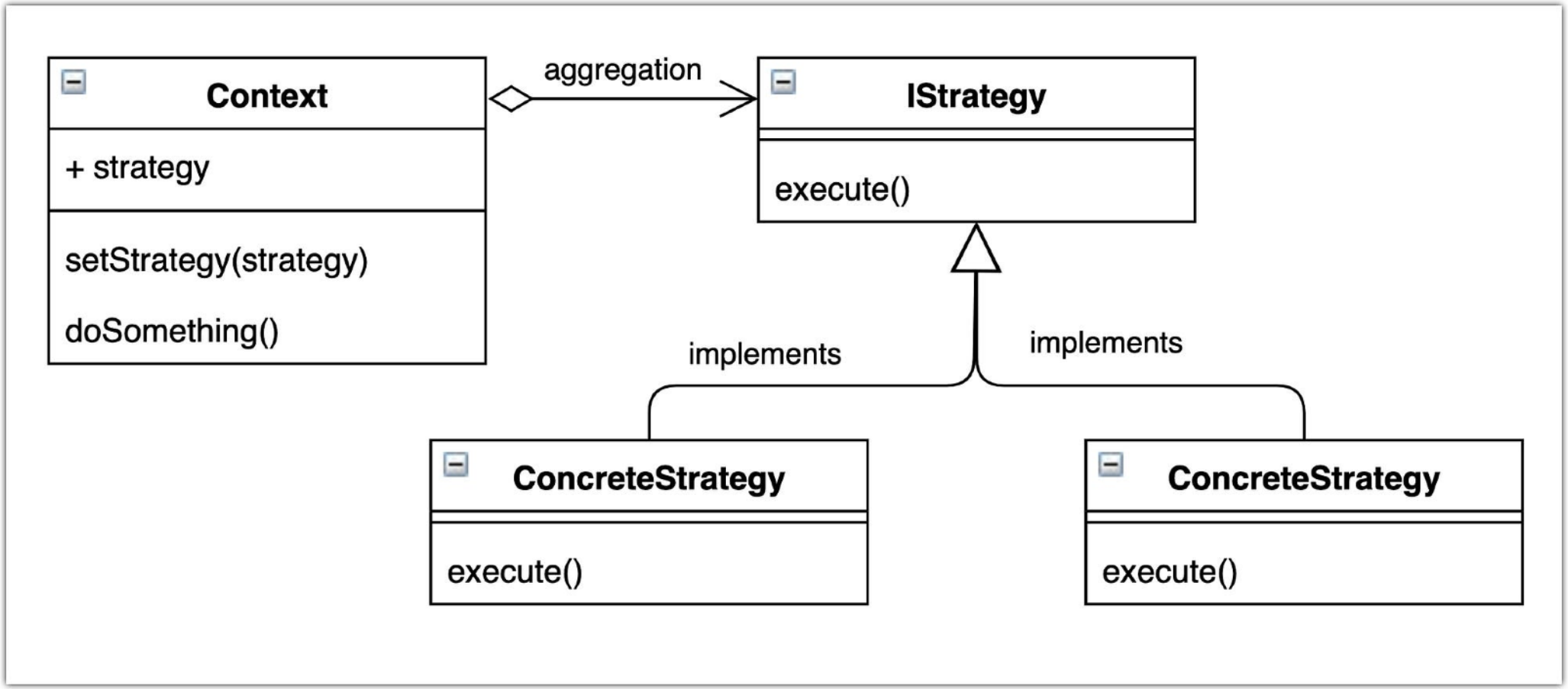


Image: Artem Diashkin https://medium.com/litslink/design-patterns-strategy-in-examples-eae7bf10a817

Strategy pattern

In code

```java
import java.util.Arrays;

// Strategy interface
interface SortingStrategy {
    void sort(int[] array);
}
```

Strategy pattern

In code

```java
// Concrete strategies
class BubbleSort implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        System.out.println("Sorting using Bubble Sort");
        // Bubble sort logic
        int n = array.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    // swap arr[j+1] and arr[i]
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }
}

```

Strategy pattern

In code

```java
class QuickSort implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        System.out.println("Sorting using Quick Sort");
        // Quick sort logic
        quickSort(array, 0, array.length - 1);
    }

    private void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pi = partition(array, low, high);

            quickSort(array, low, pi - 1);
            quickSort(array, pi + 1, high);
        }
    }

    private int partition(int[] array, int low, int high) {
        int pivot = array[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (array[j] < pivot) {
                i++;

                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;

        return i + 1;
    }
}
```

Strategy pattern

In code

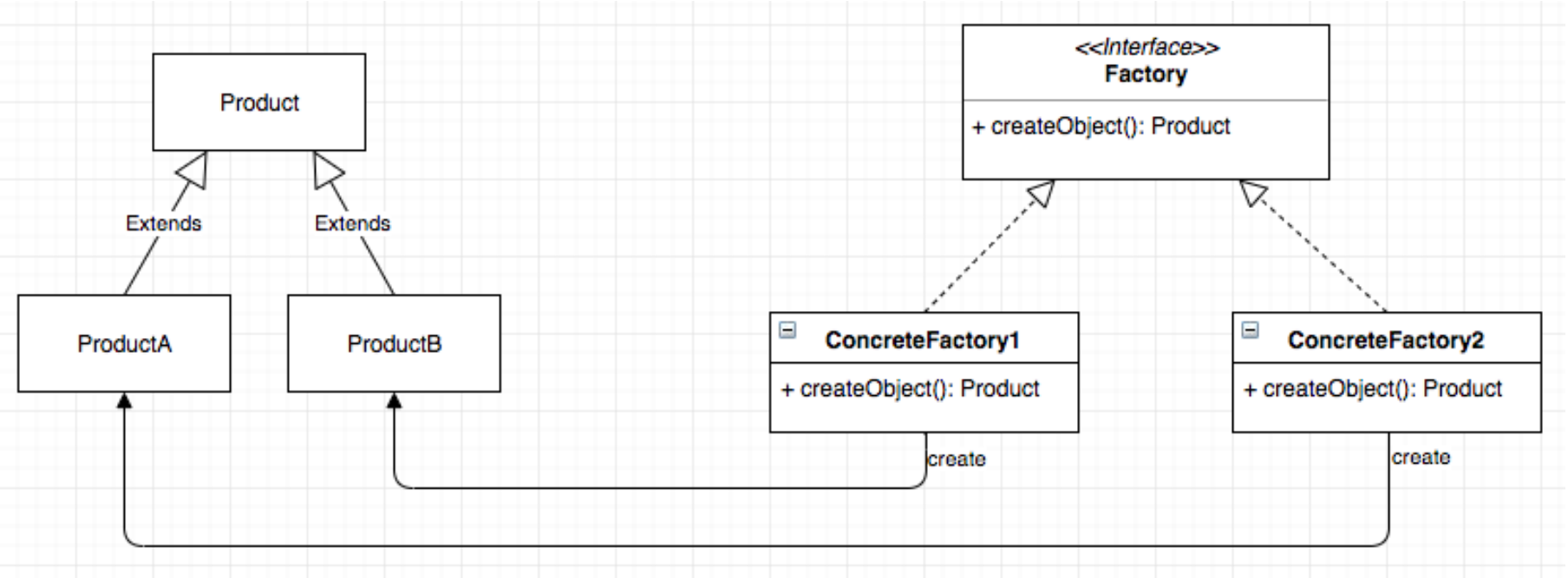
```
// Context
class SortingContext {
    private SortingStrategy sortingStrategy;

    public void setSortingStrategy(SortingStrategy sortingStrategy) {
        this.sortingStrategy = sortingStrategy;
    }

    public void performSort(int[] array) {
        sortingStrategy.sort(array);
        System.out.println("Sorted array: " + Arrays.toString(array));
    }
}
```

Strategy pattern

In code

```
80  // Client code
81  public class Main {
82      public static void main(String[] args) {
83          int[] array = {5, 2, 9, 1, 5, 6};
84
85          SortingContext sortingContext = new SortingContext();
86
87          // Set BubbleSort strategy
88          sortingContext.setSortingStrategy(new BubbleSort());
89          sortingContext.performSort(array.clone());
90
91          // Set QuickSort strategy
92          sortingContext.setSortingStrategy(new QuickSort());
93          sortingContext.performSort(array.clone());
94      }
95  }
```

Factory pattern



Source: https://heidyhe.github.io/factory-method/

Factory pattern

- The essence of the Factory pattern is to provide an **interface for creating objects**, allowing subclasses or implementing classes to **alter the type of objects** that will be created **without modifying the client code**.
- It promotes **loose coupling** by abstracting the process of object creation.
- The Factory pattern encapsulates the instantiation process and centralizes object creation logic.

Factory pattern

Client code

```java
public class Demo {
    private static Dialog dialog;

    public static void main(String[] args) {
        configure();
        runBusinessLogic();
    }

    /**
     * The concrete factory is usually chosen depending on configuration or
     * environment options.
     */
    static void configure() {
        if (System.getProperty("os.name").equals("Windows 10")) {
            dialog = new WindowsDialog();
        } else {
            dialog = new HtmlDialog();
        }
    }

    /**
     * All of the client code should work with factories and products through
     * abstract interfaces. This way it does not care which factory it works
     * with and what kind of product it returns.
     */
    static void runBusinessLogic() {
        dialog.renderWindow();
    }
}
```

https://refactoring.guru/design-patterns/factory-method/java/example

Example

```
/**
 * Base factory class. Note that "factory" is merely a role for the class. It
 * should have some core business logic which needs different products to be
 * created.
 */
public abstract class Dialog {

    public void renderWindow() {
        // ... other code ...

        Button okButton = createButton();
        okButton.render();
    }

    /**
     * Subclasses will override this method in order to create specific button
     * objects.
     */
    public abstract Button createButton();
}
```

Example

```java
/**
 * HTML Dialog will produce HTML buttons.
 */
public class HtmlDialog extends Dialog {

    @Override
    public Button createButton() {
        return new HtmlButton();
    }
}


public class WindowsDialog extends Dialog {

    @Override
    public Button createButton() {
        return new WindowsButton();
    }
}
```

Abstract factory pattern

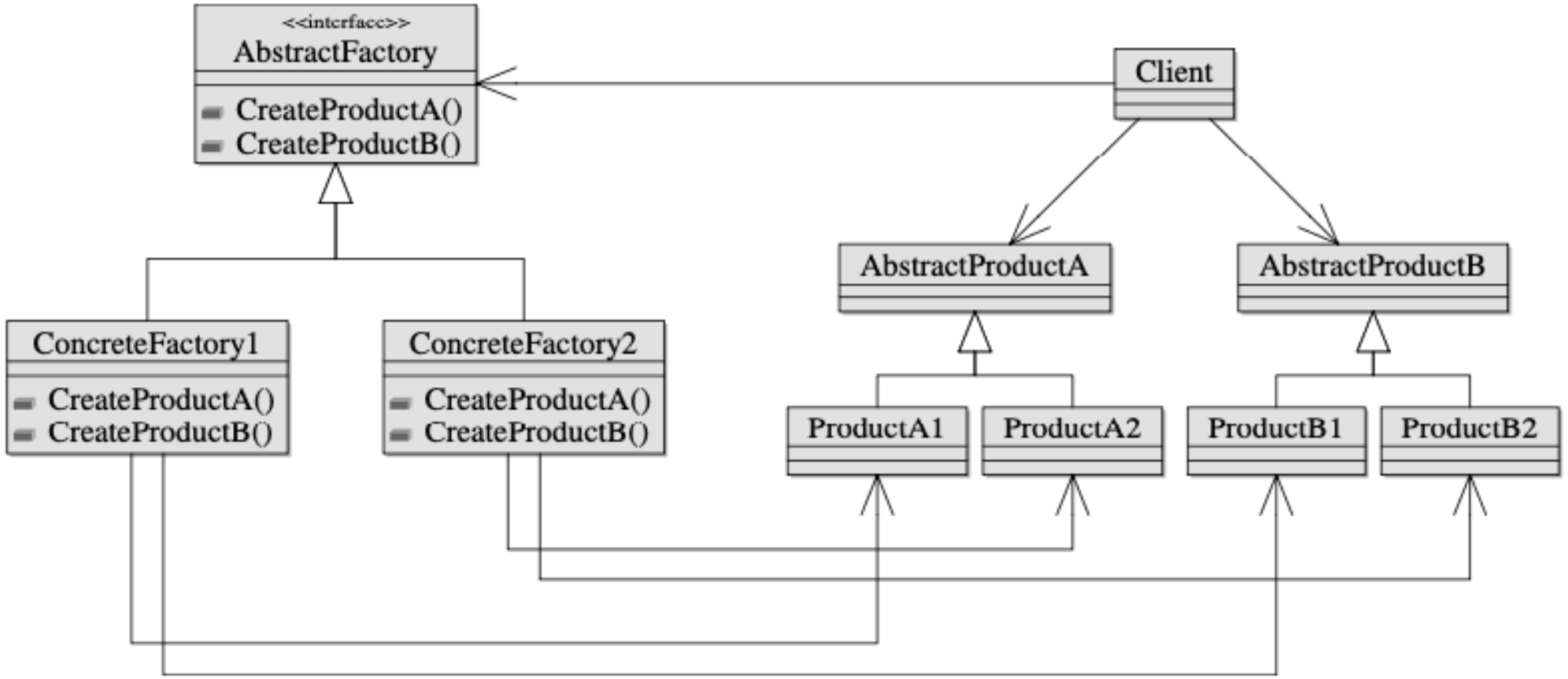
Intent:

Provide an interface for creating families of related or dependent objects without specifying their concrete class.

Abstract factory pattern

- **Motivation**:
  - Enable clients to create objects such as database connections or GUI-Widgets without making the client code dependent on any concrete class or implementation. Applicability:
- **Use** the Abstract Factory pattern **when**:
  - The client should be independent of how products are created.
  - The application should be configured with one of multiple families of products.
  - Objects need to be created as a set, in order to be compatible.
  - You want to provide a class library, i.e., a collection of classes and you want to reveal just their contracts and their relationships, not their implementations.

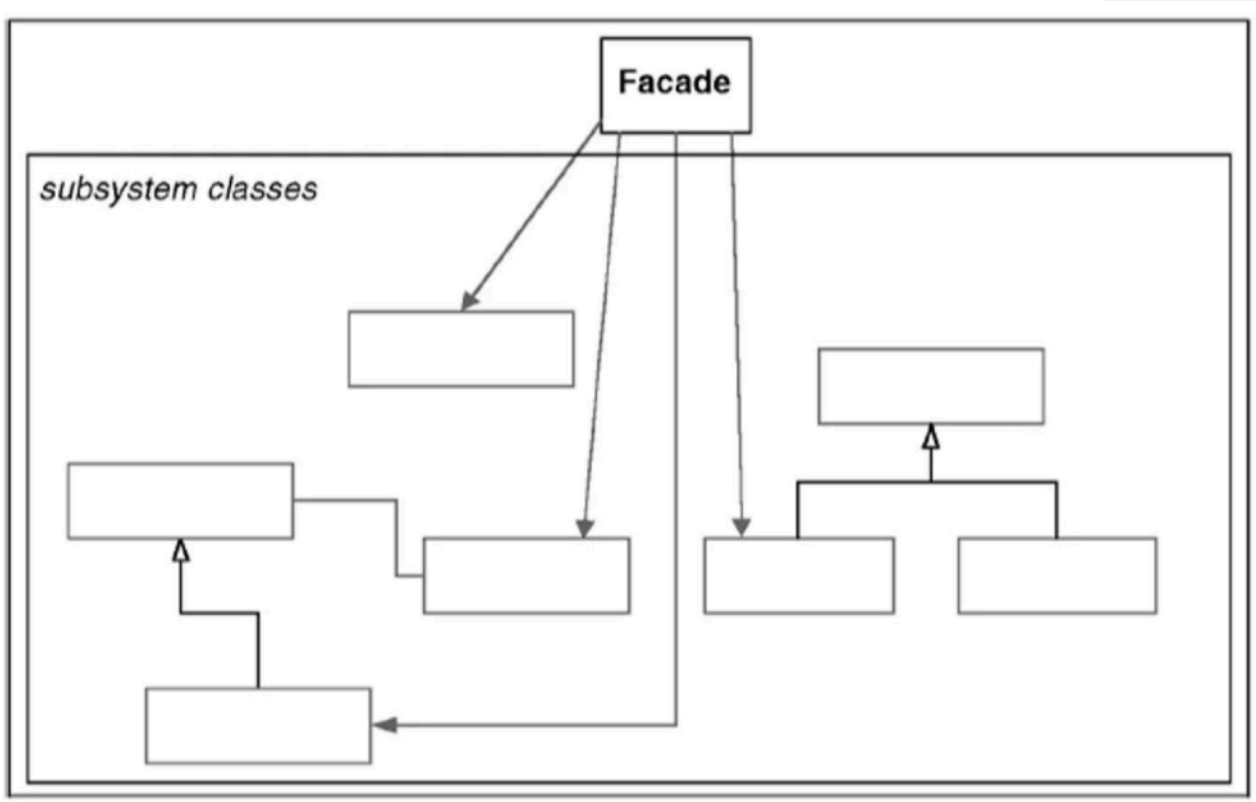
Abstract factory pattern



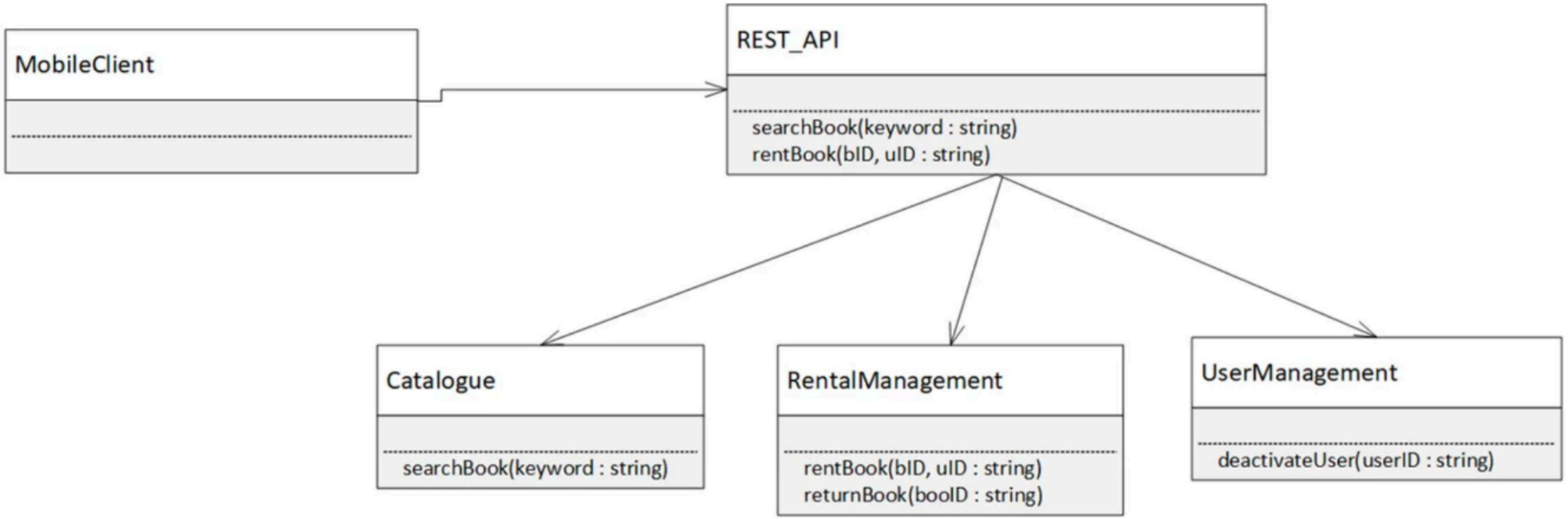
Facade pattern

Intent:

You want to simplify the use of an existing system. You need to define your own interface to use a subset of the system in a particular way.



Facade pattern



Source: Prof. Jung, SWED, 2022

Pattern pitfalls

- Patterns should not be applied mechanically
- A simple solution is preferred over one with a lot of patterns
- Inexperienced developers tend to use less patterns
- Experienced developers tend to use too many of them
- It takes practice and experience to apply pattern correctly

When to apply a pattern?

- To use patterns in your design, **you need to recognise** that any design problem you are facing may have an associated pattern that can be applied.
  - Tell several objects that the state of some other object has changed (Observer pattern).
  - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
  - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
  - Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

Source: Ian Sommerville - Software Engineering 2016

Final remarks

- **Summary**:
  - You understand OO design pattern
  - You can create UML diagrams and code for patterns
  - You recognise opportunities to apply patterns from code
- **Next week**: Idioms, Coding conventions and refactoring
- **Please read**: