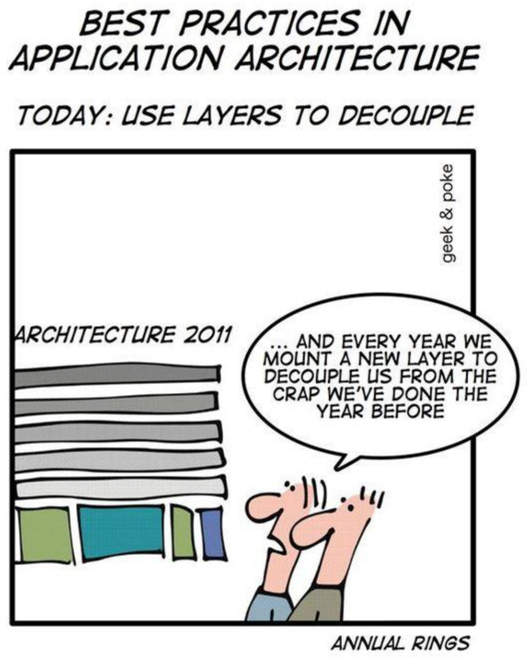
University
Mohammed VI
Polytechnic


BEST PRACTICES IN APPLICATION ARCHITECTURE
TODAY: USE LAYERS TO DECOUPLE
geek & poke
ARCHITECTURE 2011
... AND EVERY YEAR WE MOUNT A NEW LAYER TO DECOUPLE US FROM THE CRAP WE'VE DONE THE YEAR BEFORE
ANNUAL RINGS

Design Patterns: Part2

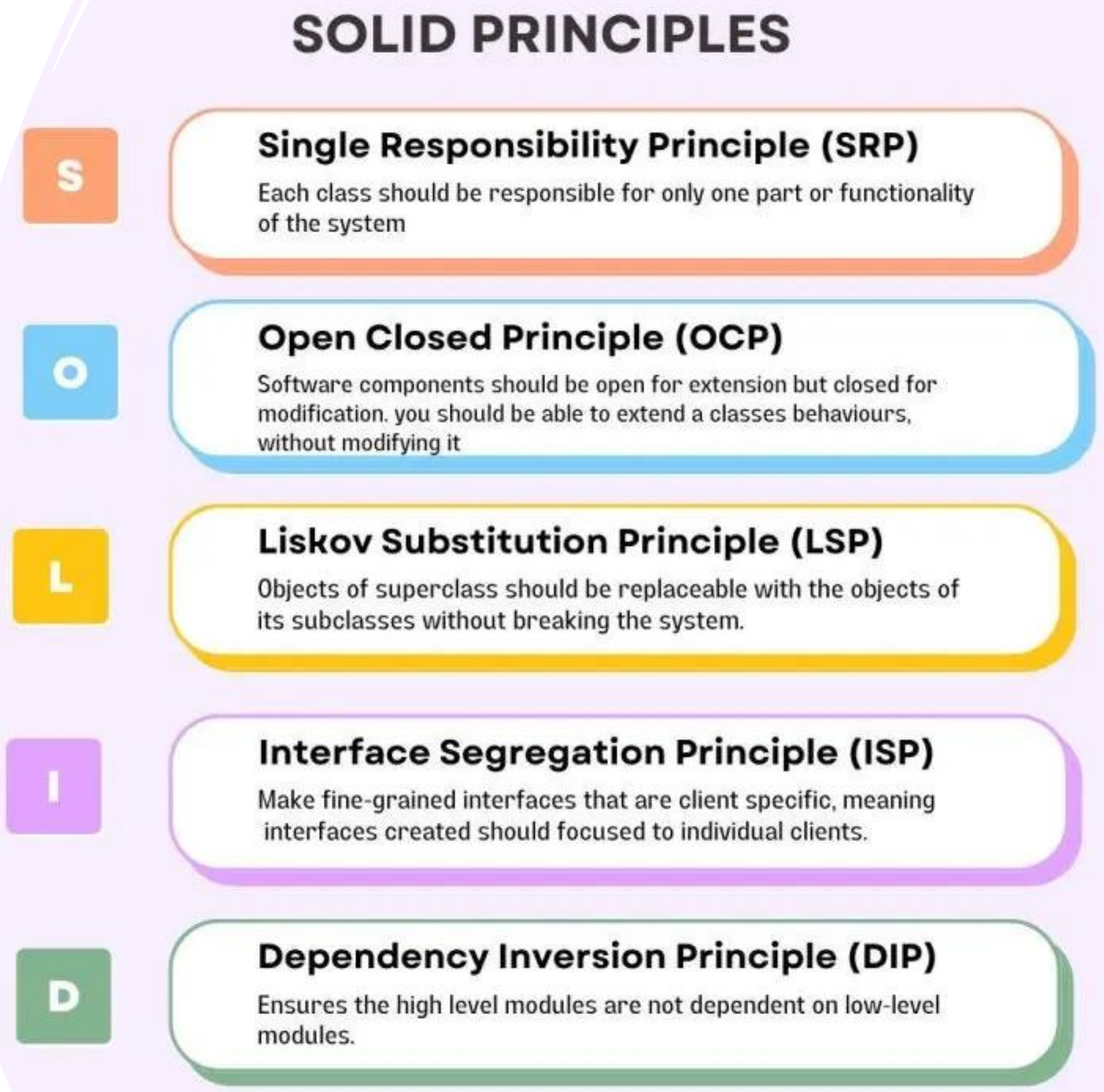
Pr. Imane Fouad

Pattern Language

Christopher Alexander says,

"Each **pattern** describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

What makes a good software



SOLID PRINCIPLES

S — **Single Responsibility Principle (SRP)**
Each class should be responsible for only one part or functionality of the system

O — **Open Closed Principle (OCP)**
Software components should be open for extension but closed for modification. you should be able to extend a classes behaviours, without modifying it

L — **Liskov Substitution Principle (LSP)**
Objects of superclass should be replaceable with the objects of its subclasses without breaking the system.

I — **Interface Segregation Principle (ISP)**
Make fine-grained interfaces that are client specific, meaning interfaces created should focused to individual clients.

D — **Dependency Inversion Principle (DIP)**
Ensures the high level modules are not dependent on low-level modules.

The Gang of For

- Defines a catalog of different design patterns.

- Three different types:
  - **Creational Patterns** – Deal with **object creation**, making it easier and more flexible.
  - **Structural Patterns** – Concerned with how classes and objects are composed to form larger structures
  - **Behavioral Patterns** – Deal with **object interaction and responsibility**, improving communication between objects.

Classification of GoF patterns

Creational	**Structural**	**Behavioral**
Factory Method	Adapter	Interpreter
Abstract Factory	Bridge	Template Method
Builder	Composite	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Flyweight	Iterator
	Facade	Mediator
	Proxy	Memento
		Observer
		State
		Strategy
		Visitor

Classification of GoF patterns

Creational	**Structural**	**Behavioral**
Factory Method	**Adapter**	Interpreter
Abstract Factory	Bridge	Template Method
Builder	**Composite**	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Flyweight	Iterator
	Facade	Mediator
	Proxy	Memento
		Observer
		State
		Strategy
		Visitor

Structural Design Patterns

Structural Design Patterns are solutions in software design that focus on how classes and objects are **organized** to form larger, functional structures

Purpose:

- Make systems **flexible** by defining clear relationships between objects.
- **Reuse existing code** without modifying it.
- Reduce **tight coupling** between parts of the system.

Adapter

Adapter

Intent

Convert the **interface** of a **class** into **another interface clients expect**. Adapter **lets classes work together** that couldn't otherwise because of **incompatible interfaces**.

Adapter Pattern : Real-World Example

- **File formats:** Convert CSV, JSON, XML to a common format for the application.
- **Language converters:** Chinese ↔ English, English ↔ Hindi.
- **API integration:** Adapt old APIs to new ones without rewriting existing code

Adapter Pattern: Overview

Also called Wrapper: converts interface of a class into another interface clients expect

Main participants: Target, Adaptee, Adapter, Client

Use when you want to reuse existing code with a different interface

Main Participants of the Adapter Pattern

1. Client

The object that requires functionality through the Target interface
Works only with the Target interface

2. Target

Defines the interface expected by the Client

3. Adaptee

Existing class with useful functionality
Has an incompatible interface (e.g., SpecificRequest())

4. Adapter

Converts/bridges method calls between Client and Adaptee

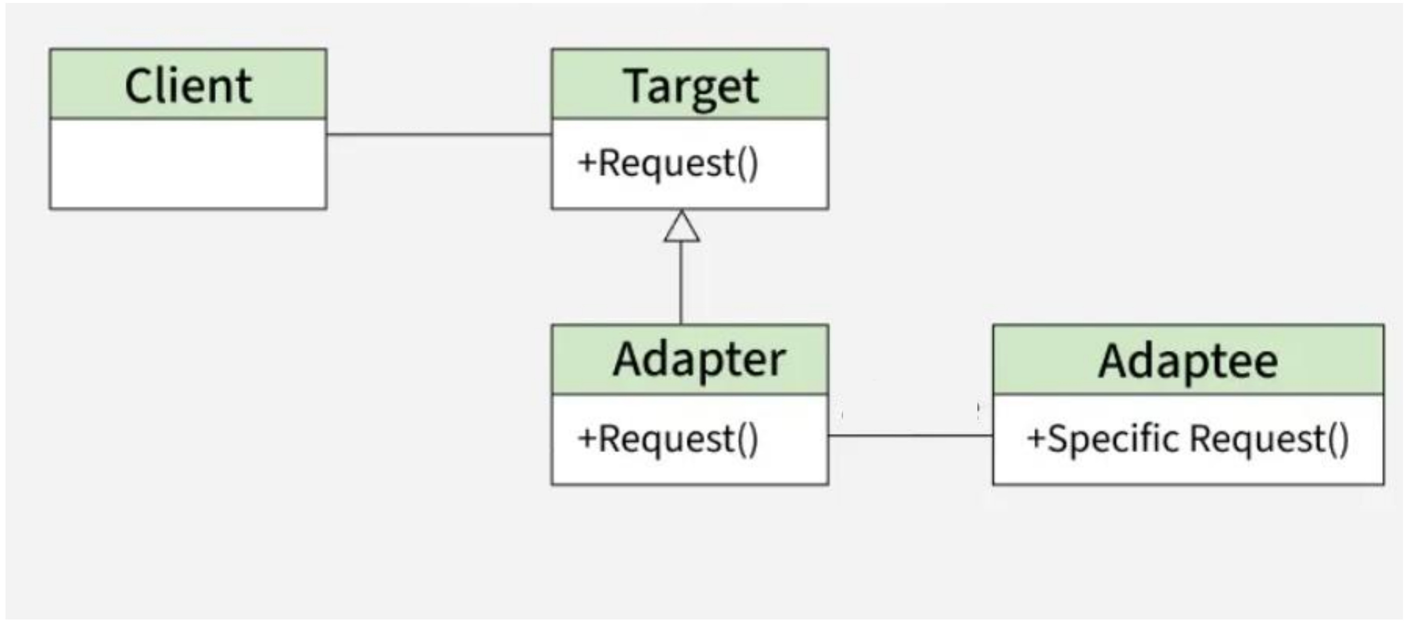
Adapter - Structure

Client collaborates with objects conforming to the Target interface.
Target defines the interface that Client uses.

Adaptee defines an existing interface that needs adapting.

Adapter adapts the interface of Adaptee to the Target interface.

Adapter - Structure



Target defines the interface that Client uses.

Client collaborates with objects conforming to the Target interface.

Adaptee defines an existing interface that needs adapting.

Adapter adapts the interface of Adaptee to the Target interface.

Step-by-step Implementation

1. **Identify the Target Interface:** what the client expects.

2. **Identify the Adaptee Class:** the existing class you want to reuse.

3. **Create the Adapter:** implement the Target interface and wrap the Adaptee.

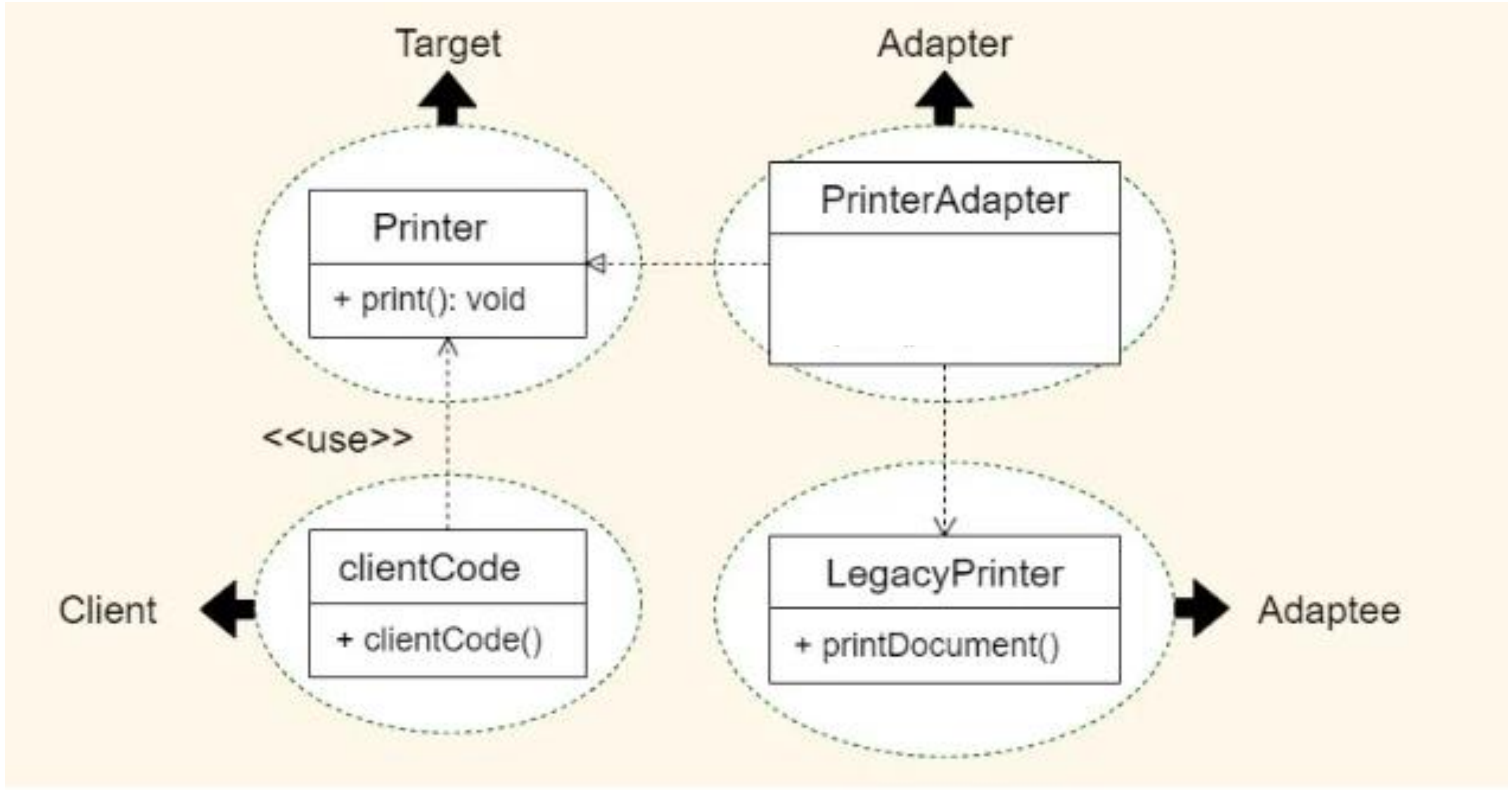
4. **Translate Calls:** Adapter methods convert Target calls into Adaptee calls.

Adapter Pattern – Problem Scenario

Scenario:

Let's consider a scenario where we have an existing system that uses a LegacyPrinter class with a method named **printDocument()** which we want to adapt into a new system that expects a Printer interface with a method named **print()**. We'll use the **Adapter design pattern** to make these two interfaces **compatible**.

Adapter Pattern – Problem Scenario



Adapter Pattern – Implementation

```
// Target interface – what the client expects
public interface Printer {
    void print();
}

// Adaptee – existing class with incompatible interface
public class LegacyPrinter {
    public void printDocument() {
        System.out.println("Legacy Printer is printing a document.");
    }
}
```

Adapter Pattern – Implementation

```
// Adapter – makes OldPrinter compatible with ModernPrinter
class PrinterAdapter implements Printer {
    private LegacyPrinter legacyPrinter;

    public PrinterAdapter() {
        this.legacyPrinter = new LegacyPrinter();
    }

    @Override
    public void print() {
        legacyPrinter.printDocument();
    }
}
```

Adapter Pattern – Implementation

```
public class Client {
    public static void main(String[] args) {
        // Create the Adapter (which wraps the legacy printer)
        PrinterAdapter printer = new PrinterAdapter();

        // Client calls the Target interface method
        printer.print(); // Internally, this calls LegacyPrinter.printDocument()

        // Output:
        // OldPrinter printing document...
    }
}
```

Why do we need Adapter Design Pattern?

- Enables communication between incompatible systems.
- Reuses existing code or libraries without rewriting.
- Simplifies integration of new components, keeping the system flexible.

Adapter

Benefits

- **Adaptees** added/used **without changes** to **existing code**.

Liabilities

- The overall **complexity** of the code **increases** because we add new classes and a level of indirection.

Composite

Composite

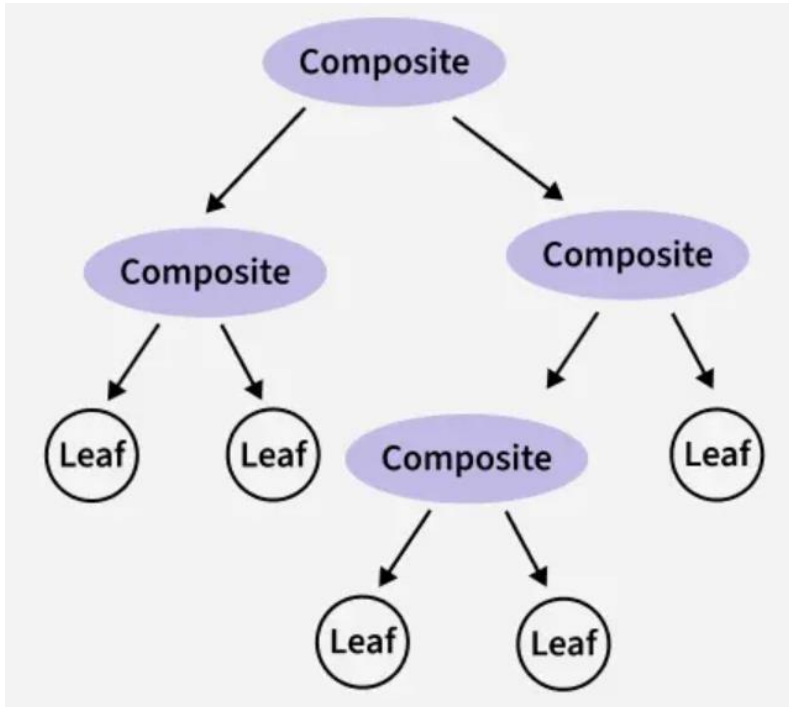
Intent

Compose objects into **structures** to represent **part-whole hierarchies**. Composite lets **clients treat** individual **objects** and **compositions** of objects **uniformly**.

Composite

Intent

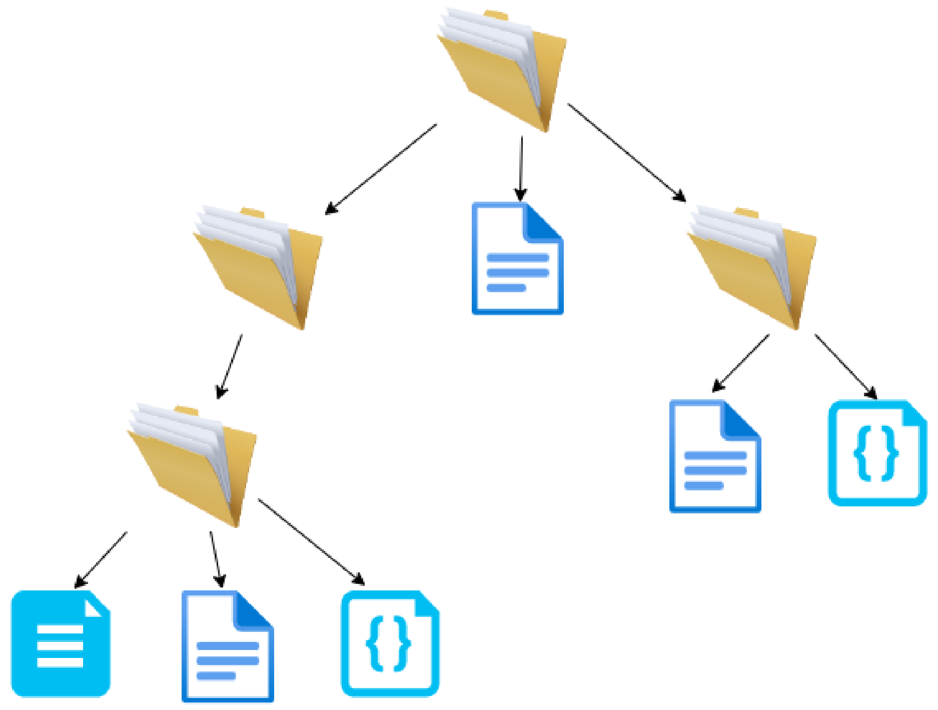
Compose objects into **structures** to represent **part-whole hierarchies**. Composite lets **clients treat** individual **objects** and **compositions** of objects **uniformly**.



Composite Pattern :Real-World Examples

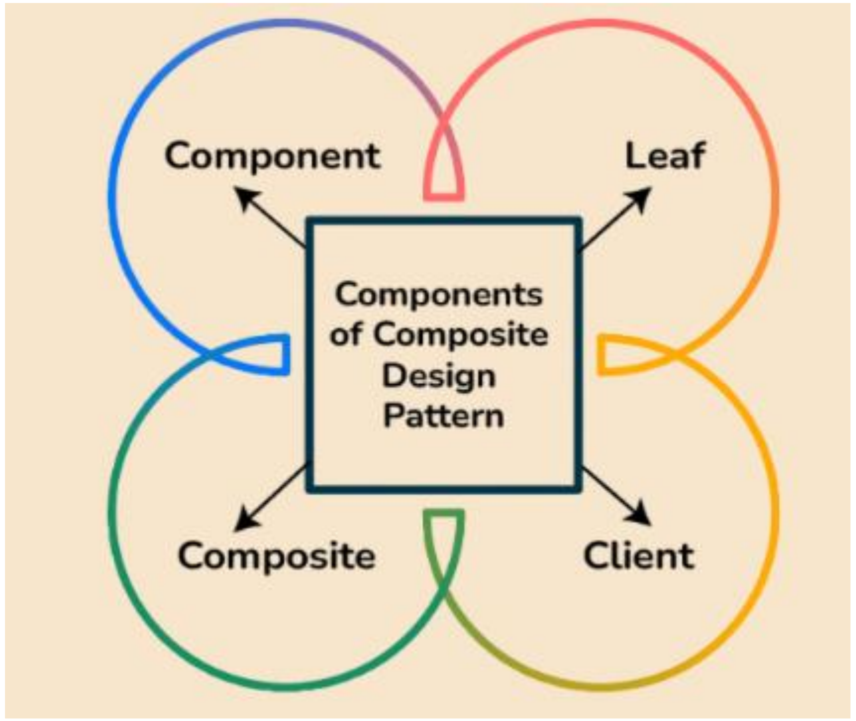
- Files and folders form a hierarchy.
- A folder can contain files or other folders.

- **Goal**: Both files and folders can be treated uniformly by the client.



Composite Pattern : Main Participants

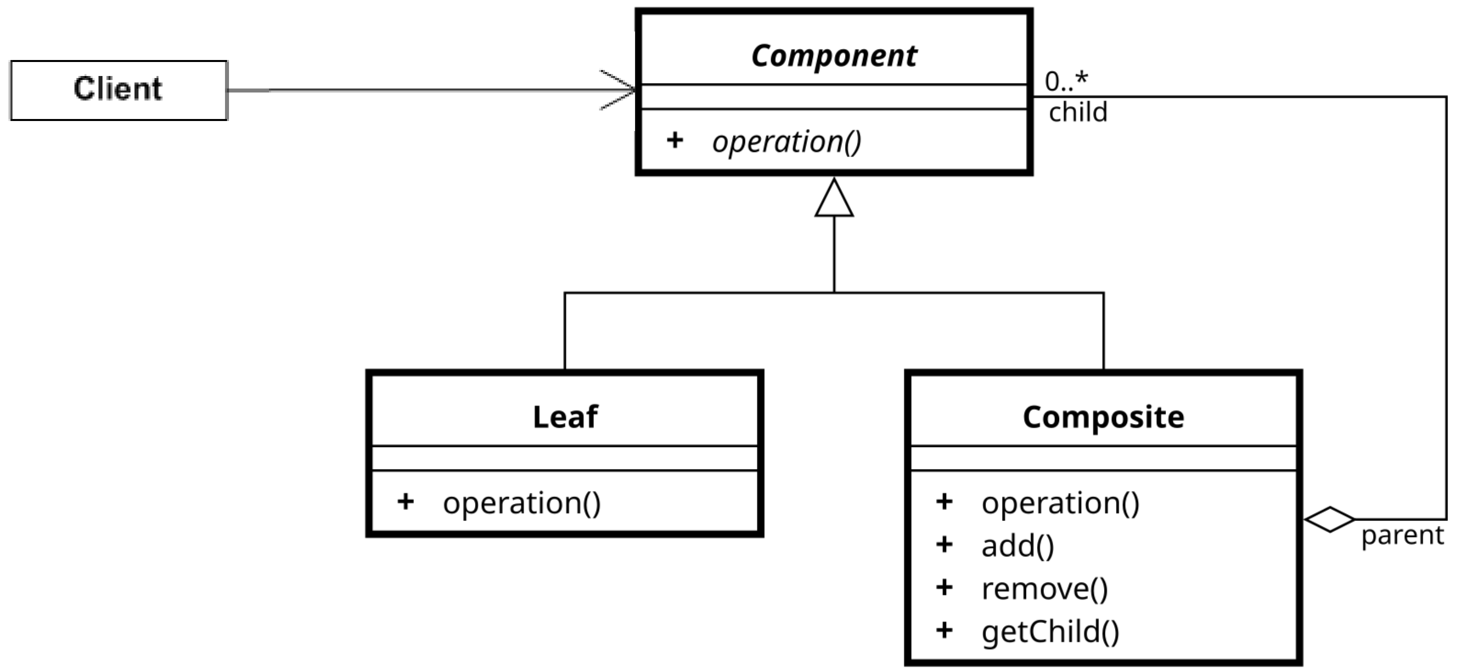
- **Component:** Common interface for leaves and composites.
- **Leaf:** Simple object that performs operations directly.
- **Composite:** Complex object containing children(leaves or other composites) and delegate operations to them
- **Client:** Uses Component interface; treats leaves and composites uniformly.



Composite Pattern : Structure

- **Component:** Common interface for leaves and composites.
- **Leaf:** Simple object that performs operations directly.
- **Composite:** Complex object containing children(leaves or other composites) and delegate operations to them
- **Client:** Uses Component interface; treats leaves and composites uniformly.

Composite Pattern : Structure



- **Component:** Common interface for leaves and composites.
- **Leaf:** Simple object that performs operations directly.
- **Composite:** Complex object containing children(leaves or other composites) and delegate operations to them
- **Client:** Uses Component interface, treats leaves and composites uniformly.

Step-by-step Implementation

1. **Define Component:** common interface for leaves and composites.
2. **Create Leaf:** implements the interface for simple objects.
3. **Create Composite:** holds children, delegates operations, allows add/remove.
4. **Client:** works with all objects uniformly through the Component interface.

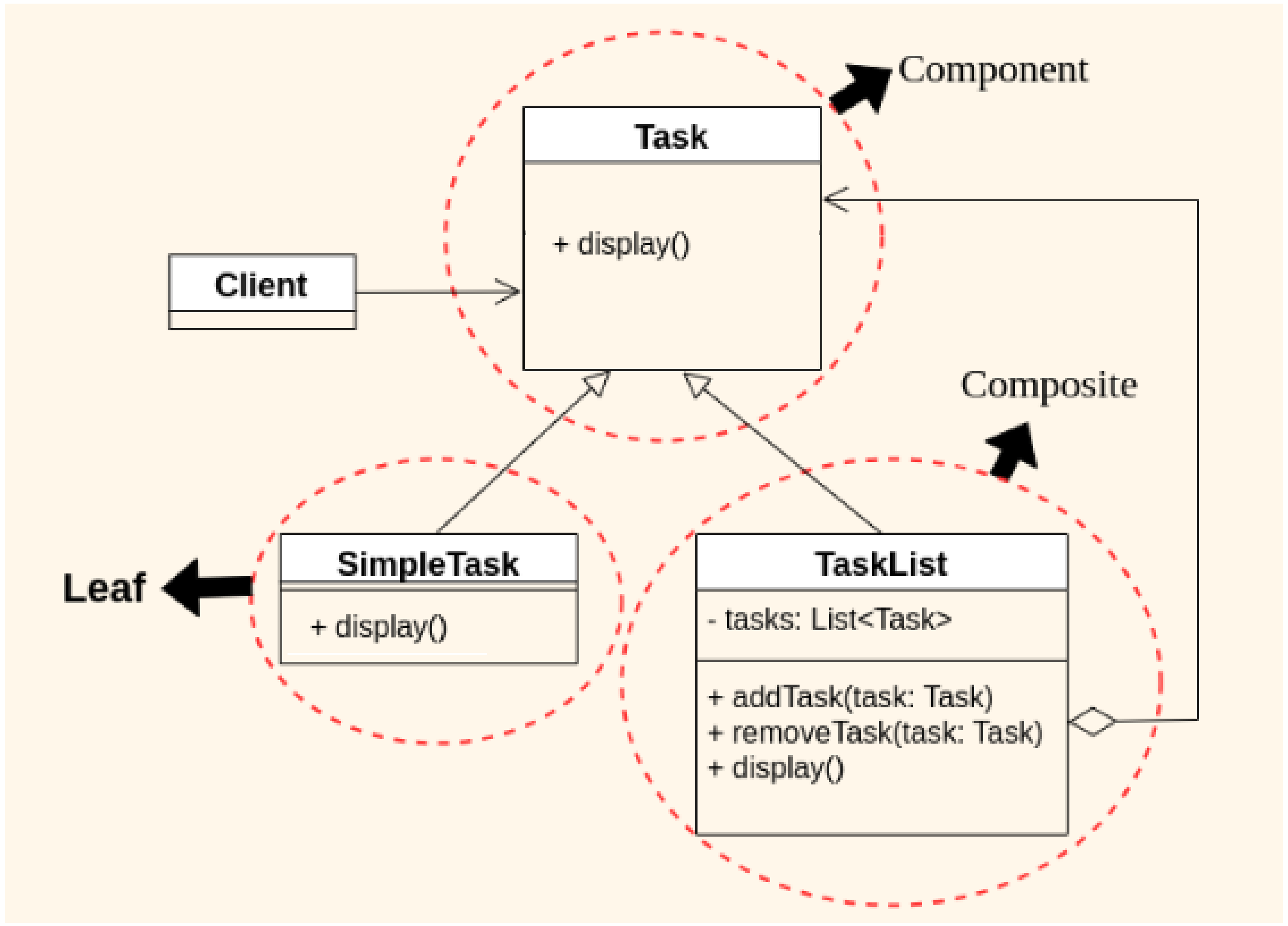
Composite Pattern – Problem Scenario

Scenario:

Imagine you are building a project management system where tasks can be either simple tasks or a collection of tasks (subtasks) forming a larger task.

- Both types of tasks must be treated uniformly by the system.
- We also want the ability to **display all tasks**, whether they are simple or made of many subtasks.

Composite Pattern – Implementation

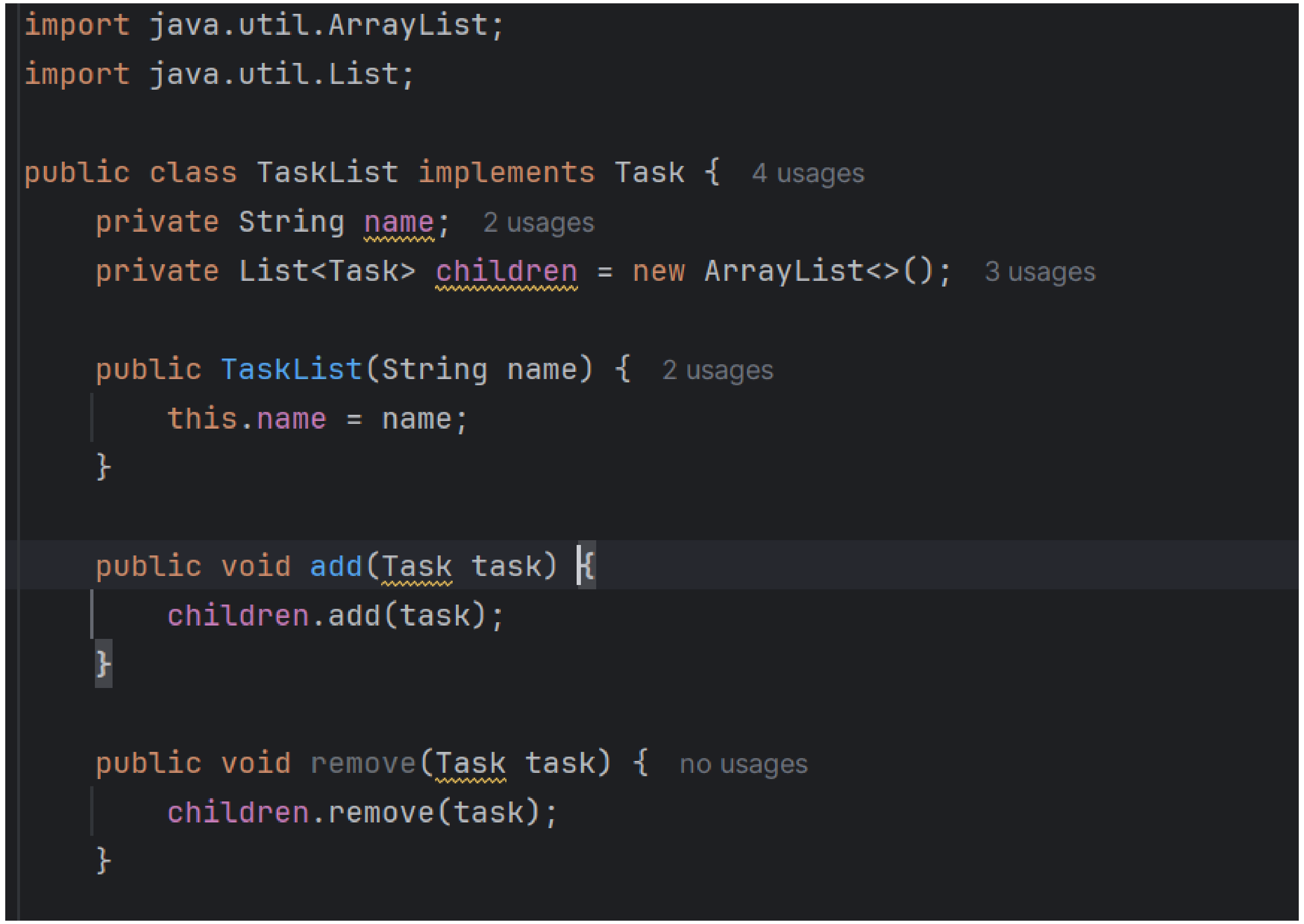


Composite Pattern – Implementation

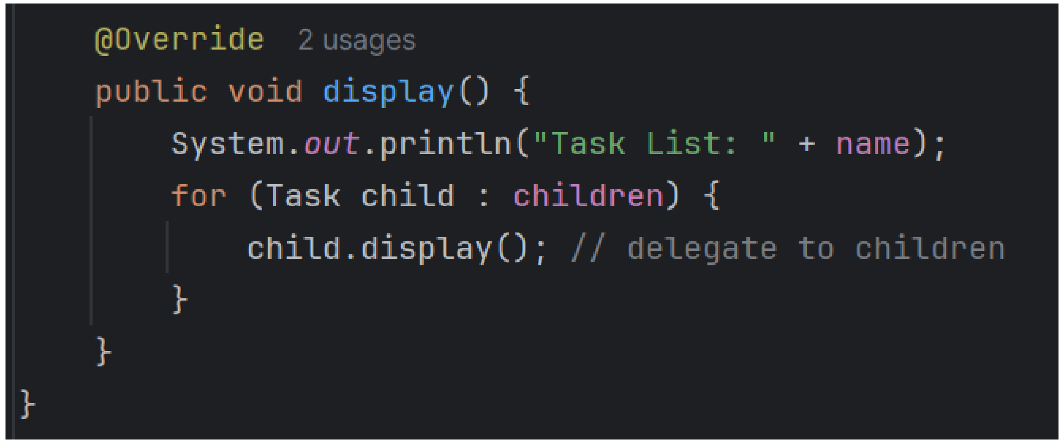
```
// Component: Common interface for all
tasks
public interface Task {
    void display(); // display the task
name
}
```

```
// Leaf
public class SimpleTask implements Task {
    private String name;

    public SimpleTask(String name) {
        this.name = name;
    }

    @Override
    public void display() {
        System.out.println("Task: " + name);
    }
}
```

Composite Pattern – Implementation

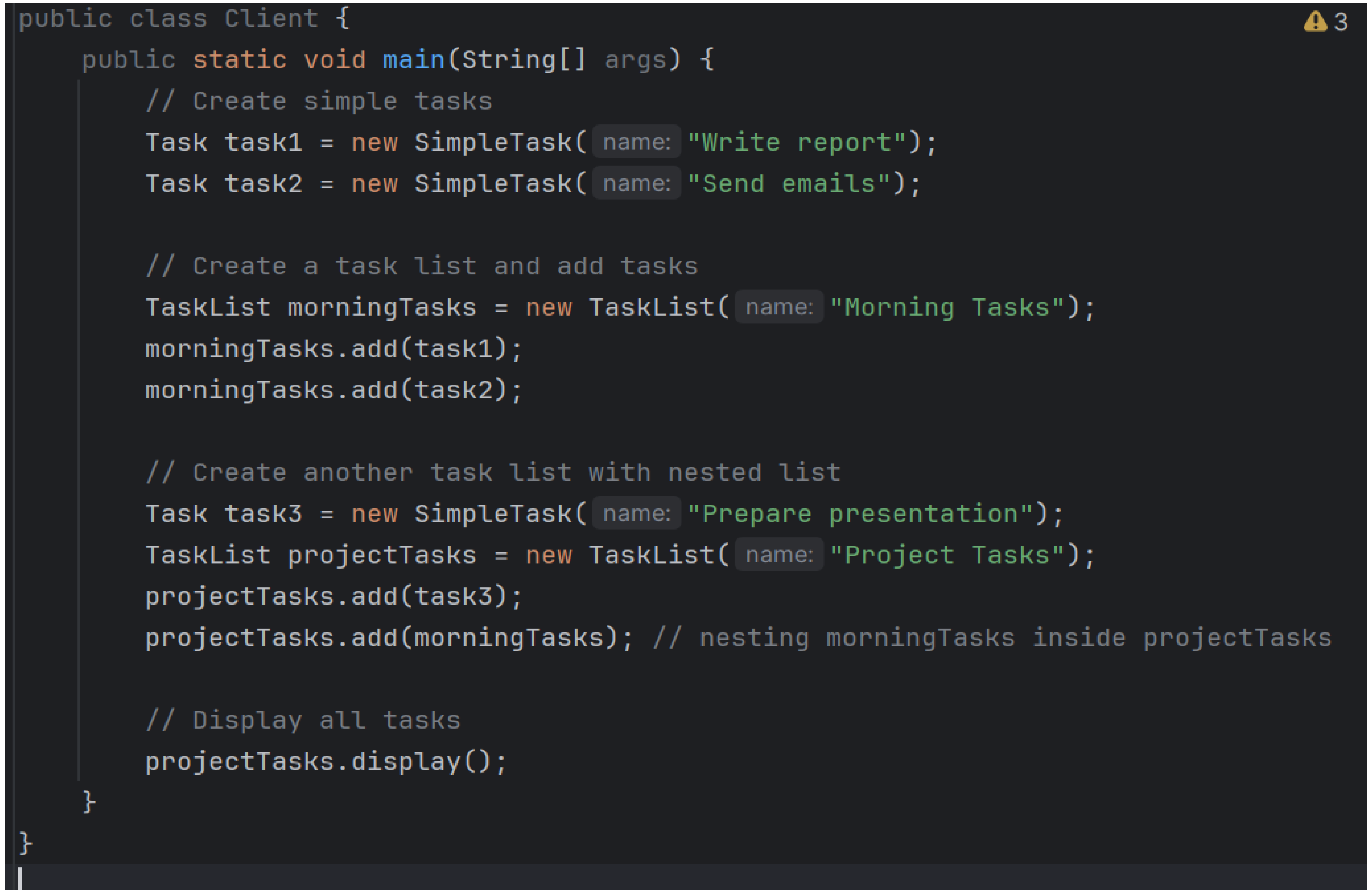


```
import java.util.ArrayList;
import java.util.List;

public class TaskList implements Task {  4 usages
    private String name;  2 usages
    private List<Task> children = new ArrayList<>();  3 usages

    public TaskList(String name) {  2 usages
        this.name = name;
    }

    public void add(Task task) {
        children.add(task);
    }

    public void remove(Task task) {  no usages
        children.remove(task);
    }
```



```
    @Override  2 usages
    public void display() {
        System.out.println("Task List: " + name);
        for (Task child : children) {
            child.display(); // delegate to children
        }
    }
}
```

Composite Pattern – Implementation



```
public class Client {                                                        ⚠3
    public static void main(String[] args) {
        // Create simple tasks
        Task task1 = new SimpleTask( name: "Write report");
        Task task2 = new SimpleTask( name: "Send emails");

        // Create a task list and add tasks
        TaskList morningTasks = new TaskList( name: "Morning Tasks");
        morningTasks.add(task1);
        morningTasks.add(task2);

        // Create another task list with nested list
        Task task3 = new SimpleTask( name: "Prepare presentation");
        TaskList projectTasks = new TaskList( name: "Project Tasks");
        projectTasks.add(task3);
        projectTasks.add(morningTasks); // nesting morningTasks inside projectTasks

        // Display all tasks
        projectTasks.display();
    }
}
```

Output:

```
Task List: Project Tasks
Task: Prepare presentation
Task List: Morning Tasks
Task: Write report
Task: Send emails
```

Why do we need Composite Design Pattern?

- **Uniform Interface** – Treats individual and composite objects the same, simplifying client code.
- **Hierarchical Structures** – Ideal for tree-like part-whole relationships.
- **Flexibility & Scalability** – Making structures easy to extend or modify.
- **Common Operations** – Defines shared operations at the component level, reducing duplication.
- **Client Simplification** – Provides a unified way to handle complex structures efficiently.

Composite

Benefits

- Makes the **clients simple**. Clients can treat composite structures and individual objects uniformly.
- Makes it **easier** to **add new kinds of components**. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code.

Liabilities

- Can make your **design too general**. The disadvantage of making it easy to add new components is that it makes it **harder to restrict the components** of a **composite.**

Classification of GoF patterns

Creational	Structural	**Behavioral**
Factory Method	Adapter	Interpreter
Abstract Factory	Bridge	Template Method
Builder	Composite	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Flyweight	Iterator
	Facade	Mediator
	Proxy	Memento
		Observer
		State
		Strategy
		Visitor

Behavioral Design Patterns

Patterns that define **how objects interact and communicate**.

Purpose:

- Improve **communication** between objects.
- Clarify responsibilities and roles.
- Reduce **tight coupling** between interacting components.

Observer

Observer

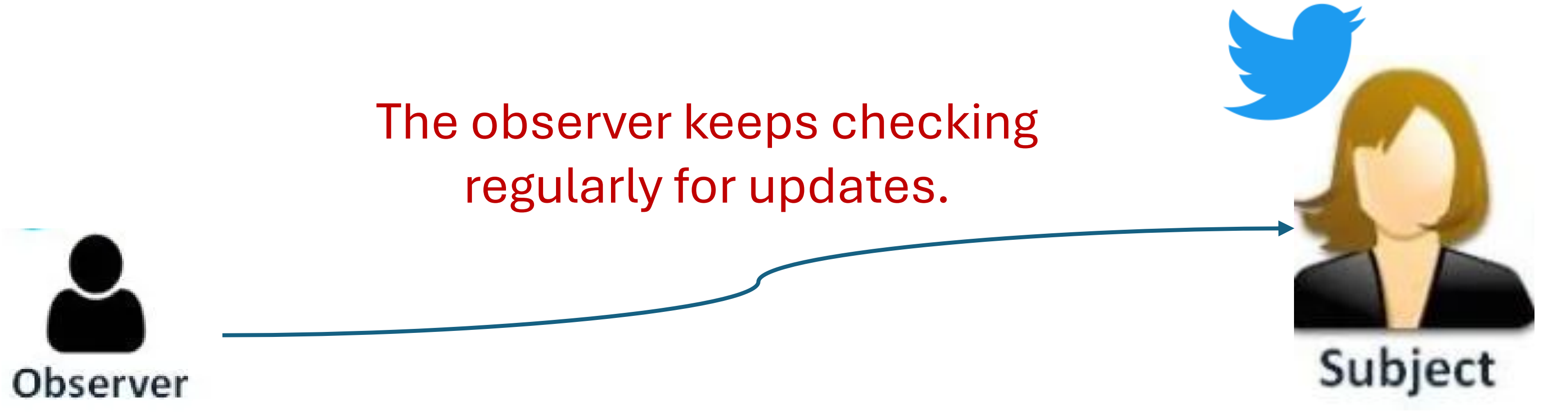
Intent

Define a **one-to-many association** between objects so that when one object **changes state**, all its **dependents** are **notified** and updated automatically.

Observer - Motivation

Observer - Motivation

Polling

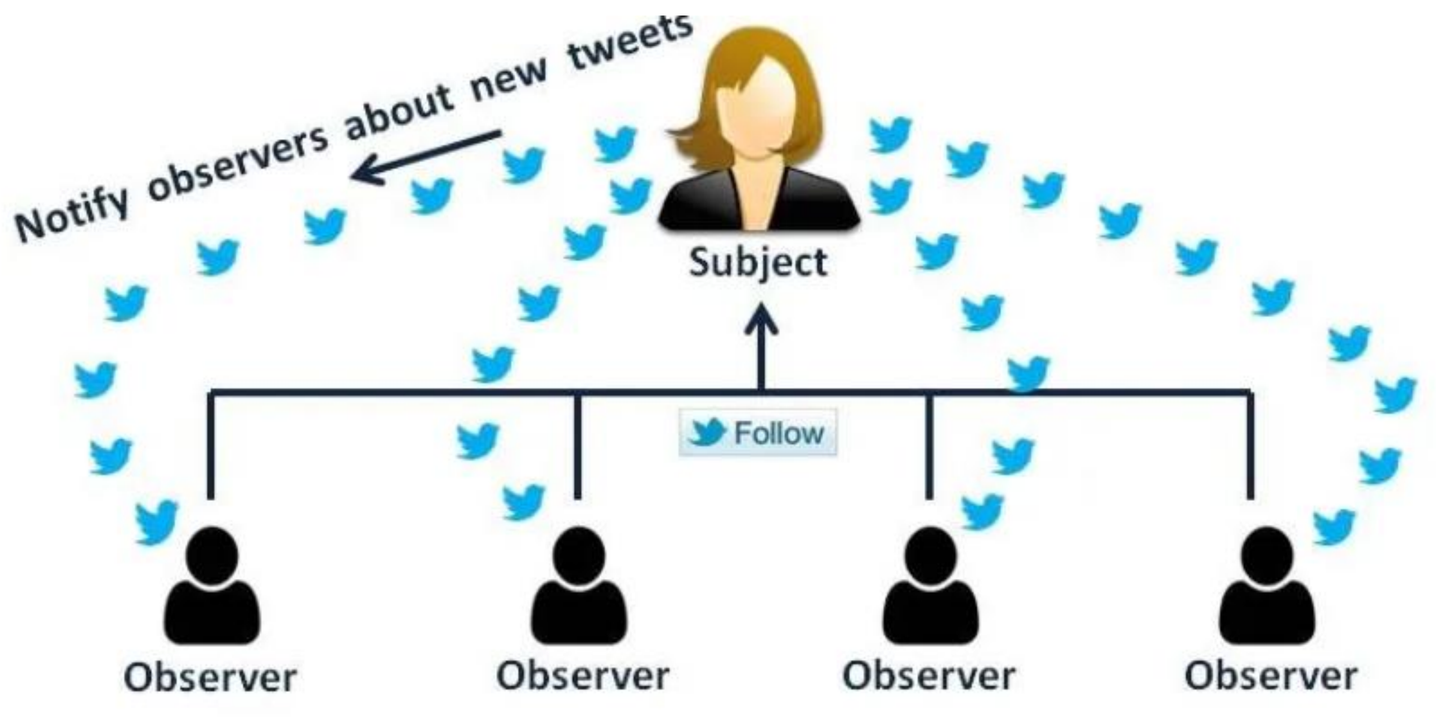


(-) Scalability Issues: With many observers constantly polling, the subject becomes overloaded and resources are wasted.

(-) Redundant Updates: Observers poll even when nothing has changed, causing wasted effort and risking outdated information between polls.

Observer - Motivation

Pushing



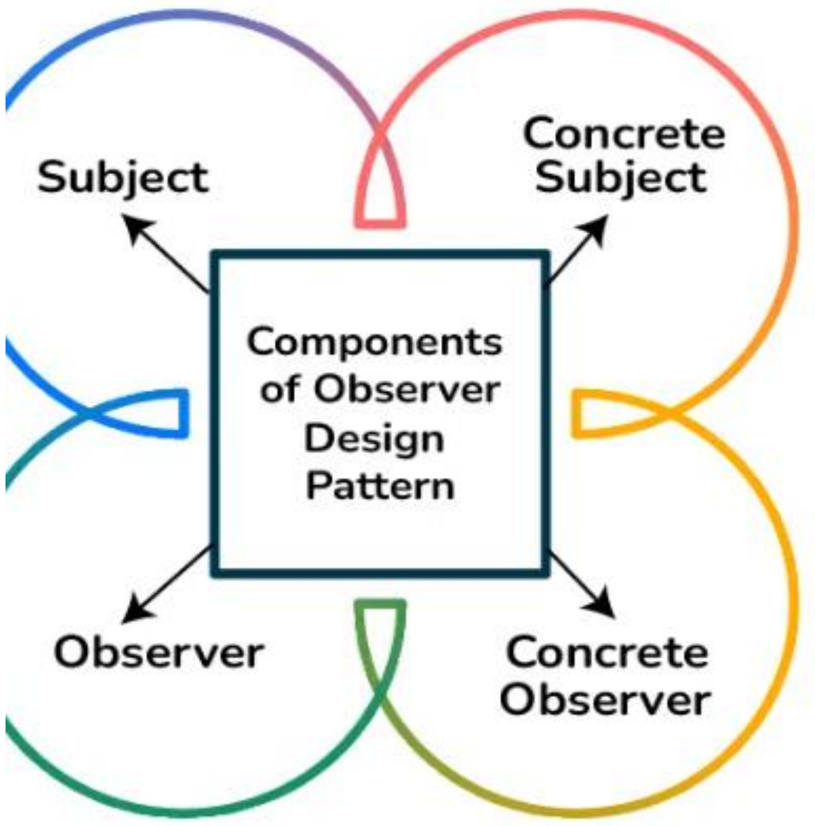
Publisher + Subscribers = Observer Pattern

Real Life Use Of Observer Design Pattern

- **Social Media Notifications** – Users get updates instantly when someone they follow posts.
- **Stock Market Apps** – Investors receive real-time changes when stock prices update.
- **GUI Event Listeners** – UI elements react to clicks, typing, or other user actions.
- **Weather Monitoring Systems** – Devices auto-refresh when central weather data changes.

Components of Observer Design Pattern

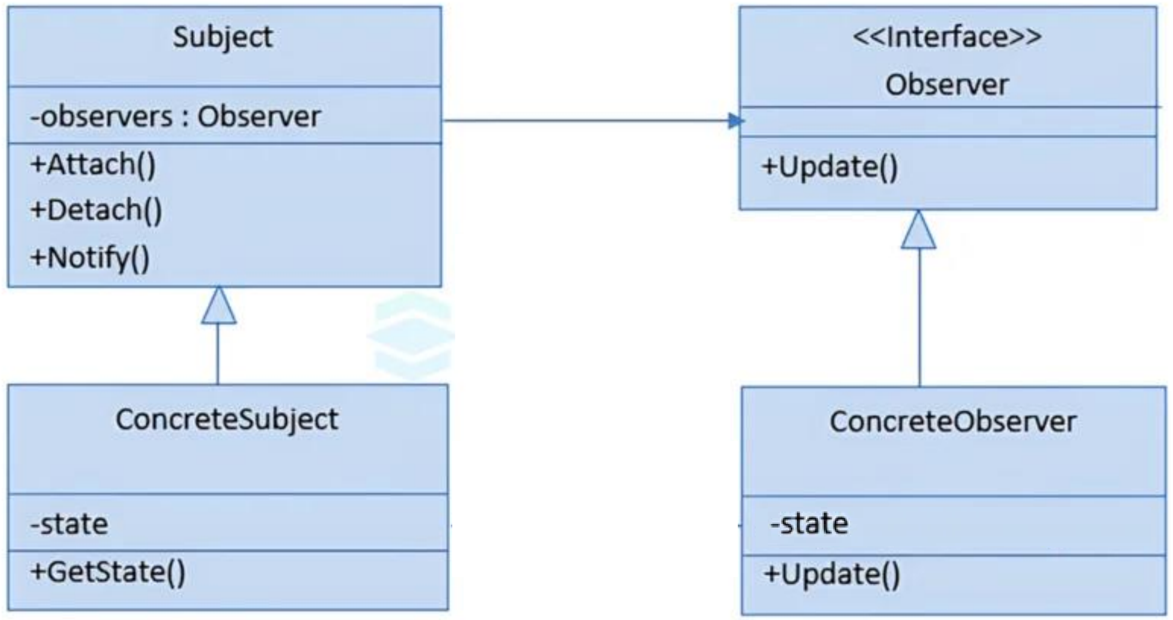
- **Subject** knows its observers. **Any number of Observer objects** may observe a Subject object. Subject Provides an **interface** for **attach** and **detach** Observer objects.
- **Observer** defines an **updating interface** for objects that should be notified of changes in a subject.
- **ConcreteSubject** stores state of interest to ConcreteObserver objects. Sends a notification to its observers when its state changes.
- **ConcreteObserver** Implements the Observer updating interface to keep its state consistent with the subject's



Observer Pattern: Structure

- **Subject** knows its observers. **Any number of Observer objects** may observe a Subject object. Subject Provides an **interface** for **attach** and **detach** Observer objects.
- **Observer** defines an **updating interface** for objects that should be notified of changes in a subject.
- **ConcreteSubject** stores state of interest to ConcreteObserver objects. Sends a notification to its observers when its state changes.
- **ConcreteObserver** Implements the Observer updating interface to keep its state consistent with the subject's

Observer Pattern: Structure



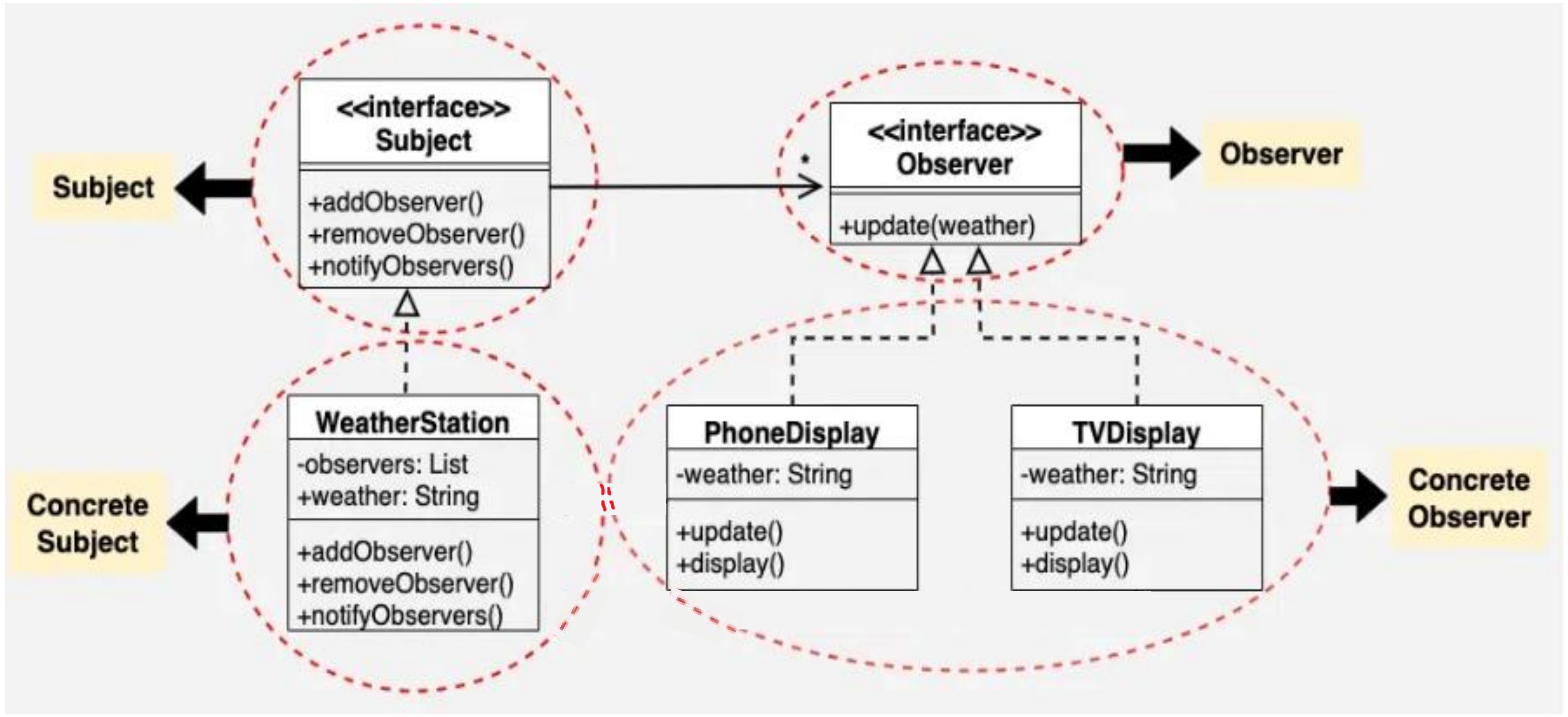
- **Subject** knows its observers. **Any number of Observer objects** may observe a Subject object. Subject Provides an **interface** for **attach** and **detach** Observer objects.
- **Observer** defines an **updating interface** for objects that should be notified of changes in a subject.
- **ConcreteSubject** stores state of interest to ConcreteObserver objects. Sends a notification to its observers when its state changes.
- **ConcreteObserver** maintains a reference to a ConcreteSubject object. Stores state that should stay consistent with the subject's. Implements the Observer updating interface to keep its state consistent with the subject's

Observer Pattern: Problem Scenario

Consider a scenario where you have a weather monitoring system. Different parts of your application need to be updated when the weather conditions change.

- The Weather Station (Subject) maintains weather data.
- Devices (Observers) like mobile apps, and TVs display the latest weather.
- Whenever the weather changes, all registered devices are automatically notified and updated.

Observer Pattern: Problem Scenario

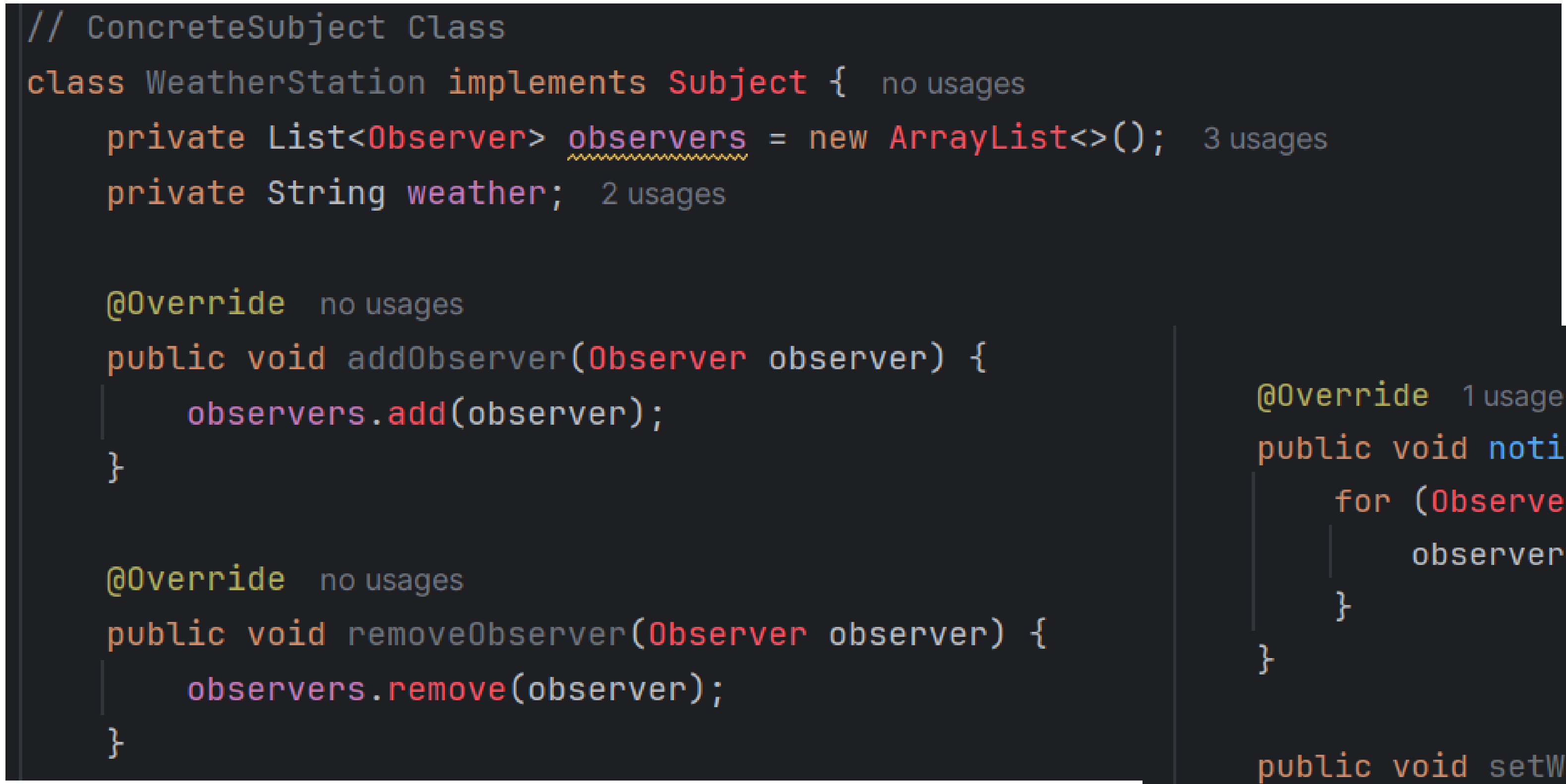


Observer Pattern: Implementation

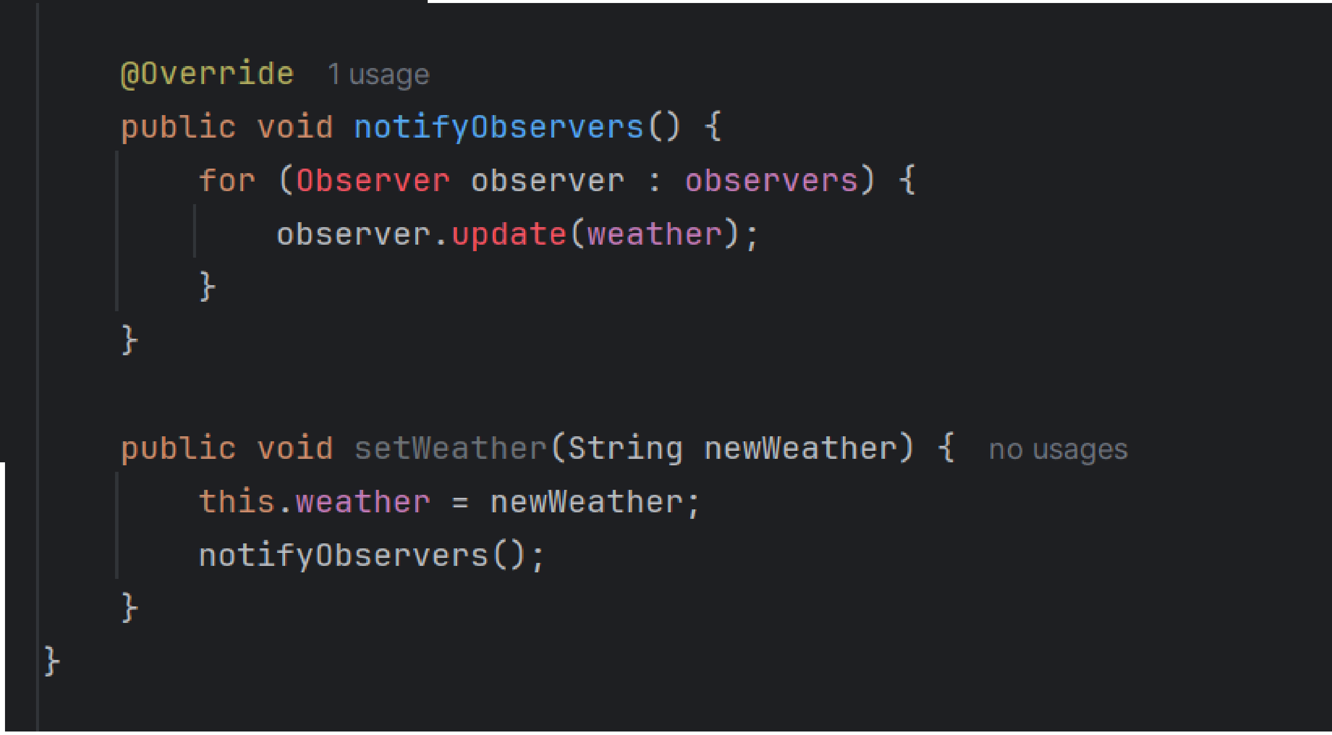
```
// Subject interface
public interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
```

```
// Observer interface
interface Observer {
    void update(String weather);
}
```

Observer Pattern: Implementation

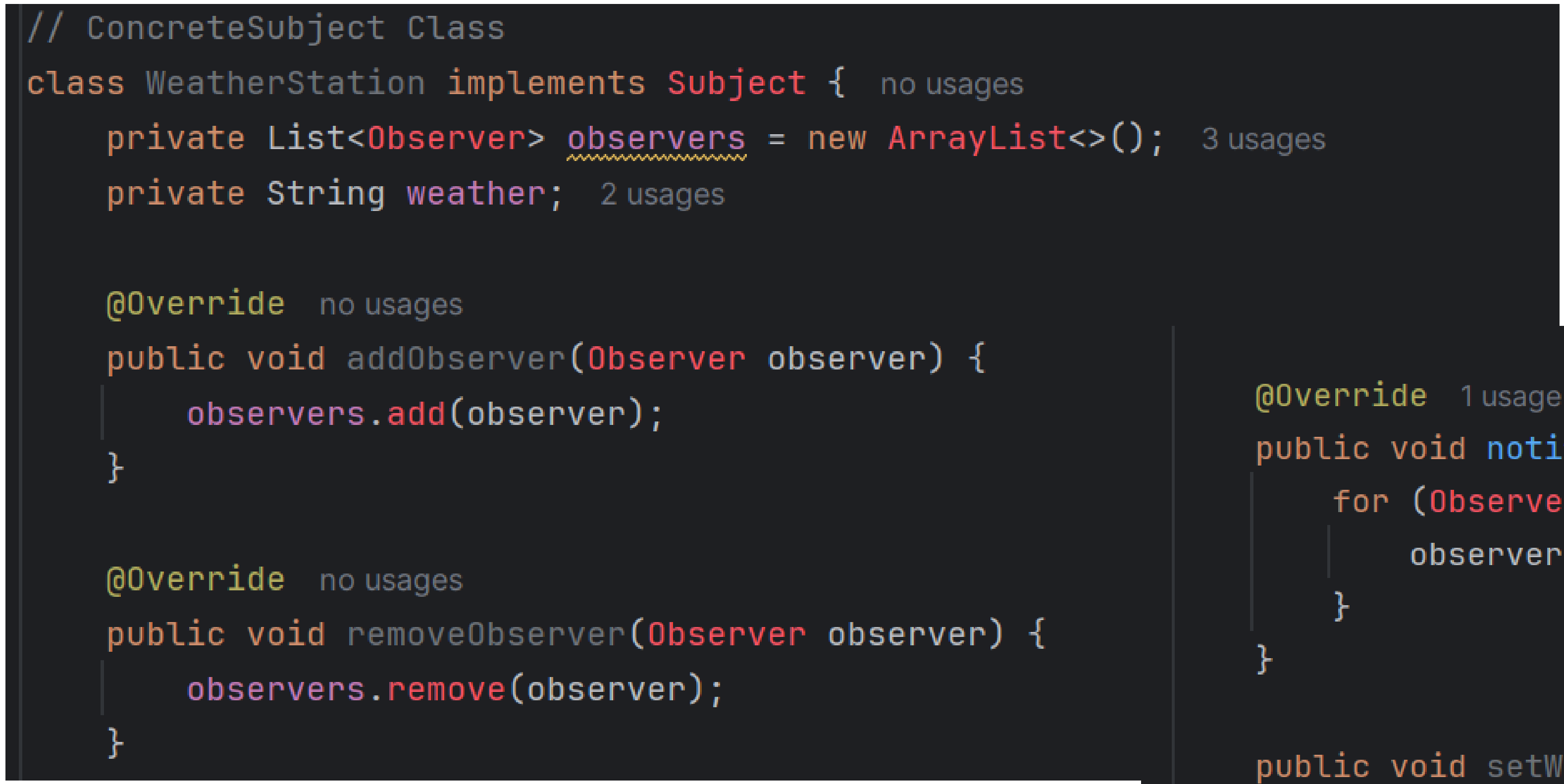


```
// ConcreteSubject Class
class WeatherStation implements Subject {  no usages
    private List<Observer> observers = new ArrayList<>();  3 usages
    private String weather;  2 usages

    @Override  no usages
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override  no usages
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
```

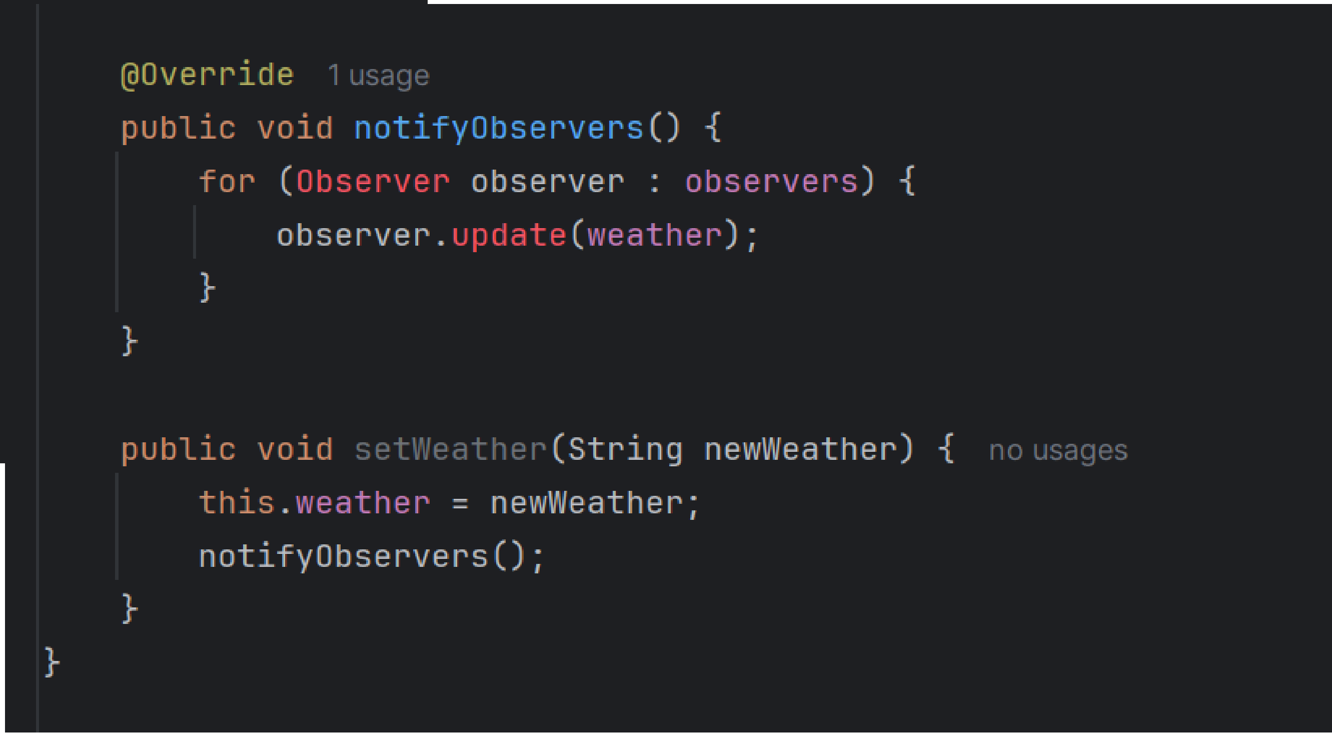


```
    @Override  1 usage
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(weather);
        }
    }

    public void setWeather(String newWeather) {  no usages
        this.weather = newWeather;
        notifyObservers();
    }
}
```

Observer Pattern: Implementation

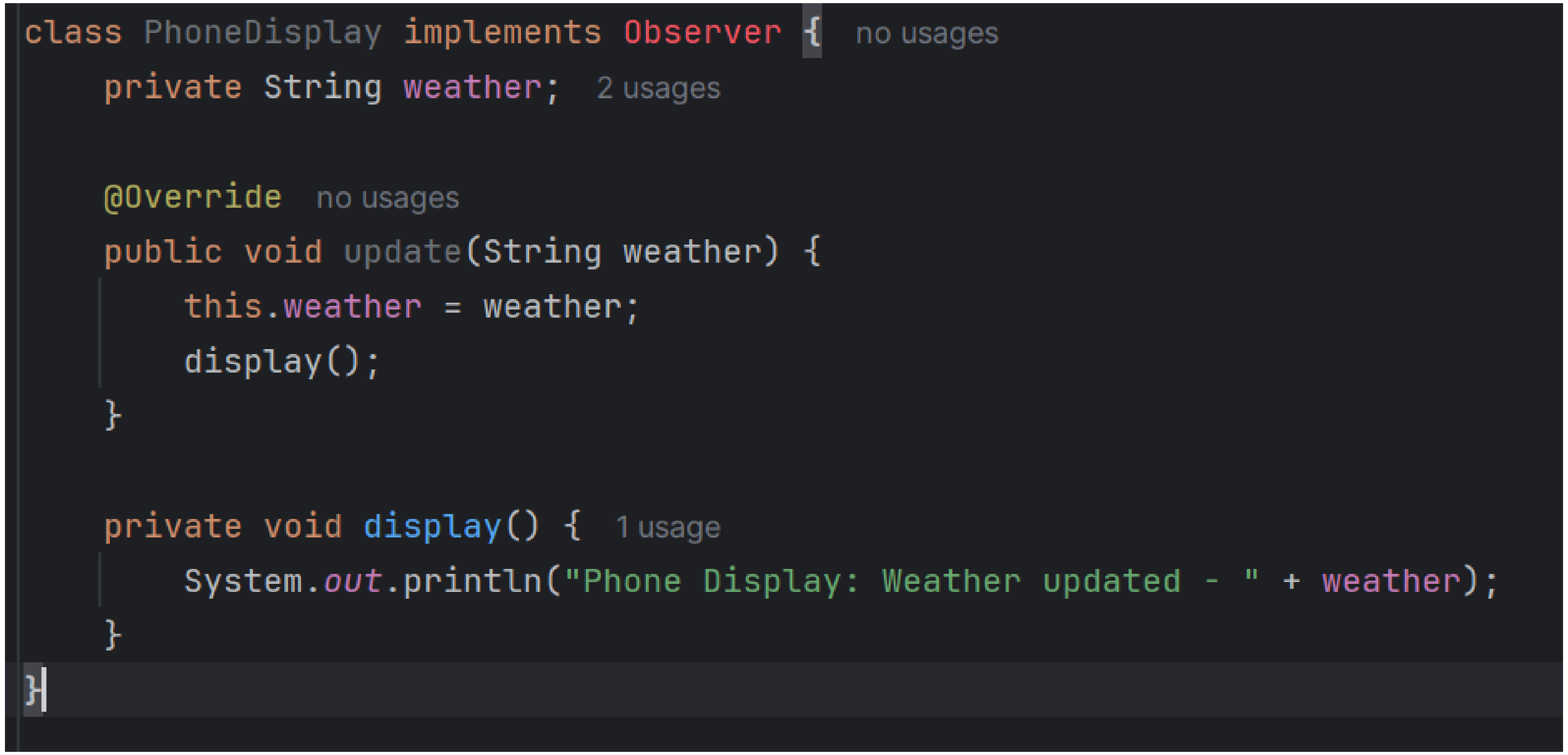


```
// ConcreteSubject Class
class WeatherStation implements Subject {  no usages
    private List<Observer> observers = new ArrayList<>();  3 usages
    private String weather;  2 usages

    @Override  no usages
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override  no usages
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
```



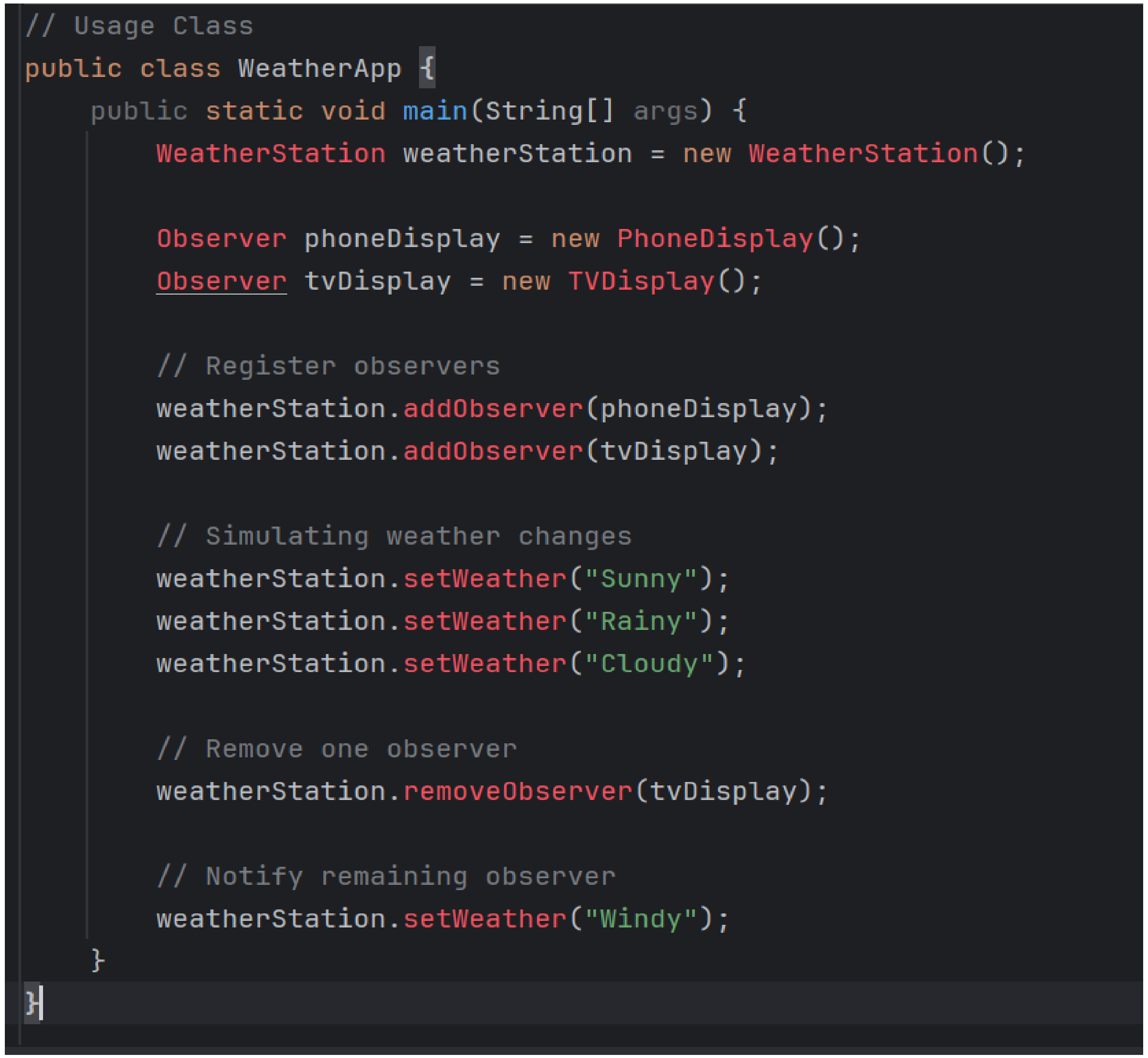
```
    @Override  1 usage
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(weather);
        }
    }

    public void setWeather(String newWeather) {  no usages
        this.weather = newWeather;
        notifyObservers();
    }
}
```

Observer Pattern: Implementation



```
class PhoneDisplay implements Observer {  no usages
    private String weather;  2 usages

    @Override  no usages
    public void update(String weather) {
        this.weather = weather;
        display();
    }

    private void display() {  1 usage
        System.out.println("Phone Display: Weather updated - " + weather);
    }
}
```

Observer Pattern: Implementation



```java
// Usage Class
public class WeatherApp {
    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();

        Observer phoneDisplay = new PhoneDisplay();
        Observer tvDisplay = new TVDisplay();

        // Register observers
        weatherStation.addObserver(phoneDisplay);
        weatherStation.addObserver(tvDisplay);

        // Simulating weather changes
        weatherStation.setWeather("Sunny");
        weatherStation.setWeather("Rainy");
        weatherStation.setWeather("Cloudy");

        // Remove one observer
        weatherStation.removeObserver(tvDisplay);

        // Notify remaining observer
        weatherStation.setWeather("Windy");
    }
}
```

Output:

```
Phone Display: Weather updated - Sunny
TV Display: Weather updated - Sunny
Phone Display: Weather updated - Rainy
TV Display: Weather updated - Rainy
Phone Display: Weather updated - Cloudy
...
```

Why do we need Observer Design Pattern?

Loose Coupling: Subjects don’t need to know details about observers.

Dynamic Relationships: Observers can be added or removed.

Scalability: Works well when multiple objects depend on the same subject.

Reusability: Observers and subjects can be reused independently.

Automatic Synchronization: Any state change in the subject is propagated to all observers.

Flexibility: Supports many-to-many relationships (multiple subjects and multiple observers).

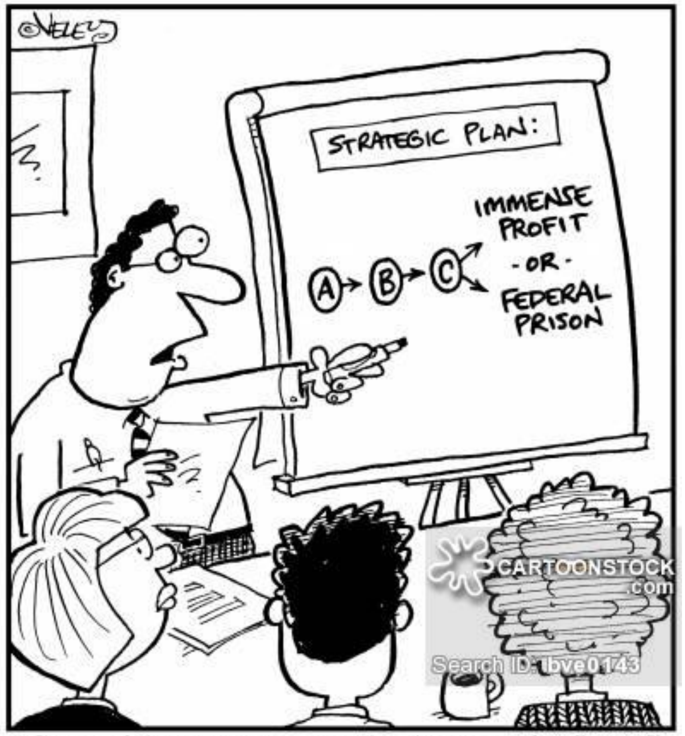
Observer

Benefits

- The **coupling** between **subjects** and **observers** is abstract and minimal.
- We can introduce **new Observer classes without** having to **change** the **Subject class**.
- We can establish **relations** between objects at **runtime**.

Liabilities

- **Unexpected/unwanted notifications.**



"Stay with me now, people, because in Step C, things get a bit delicate."

Strategy

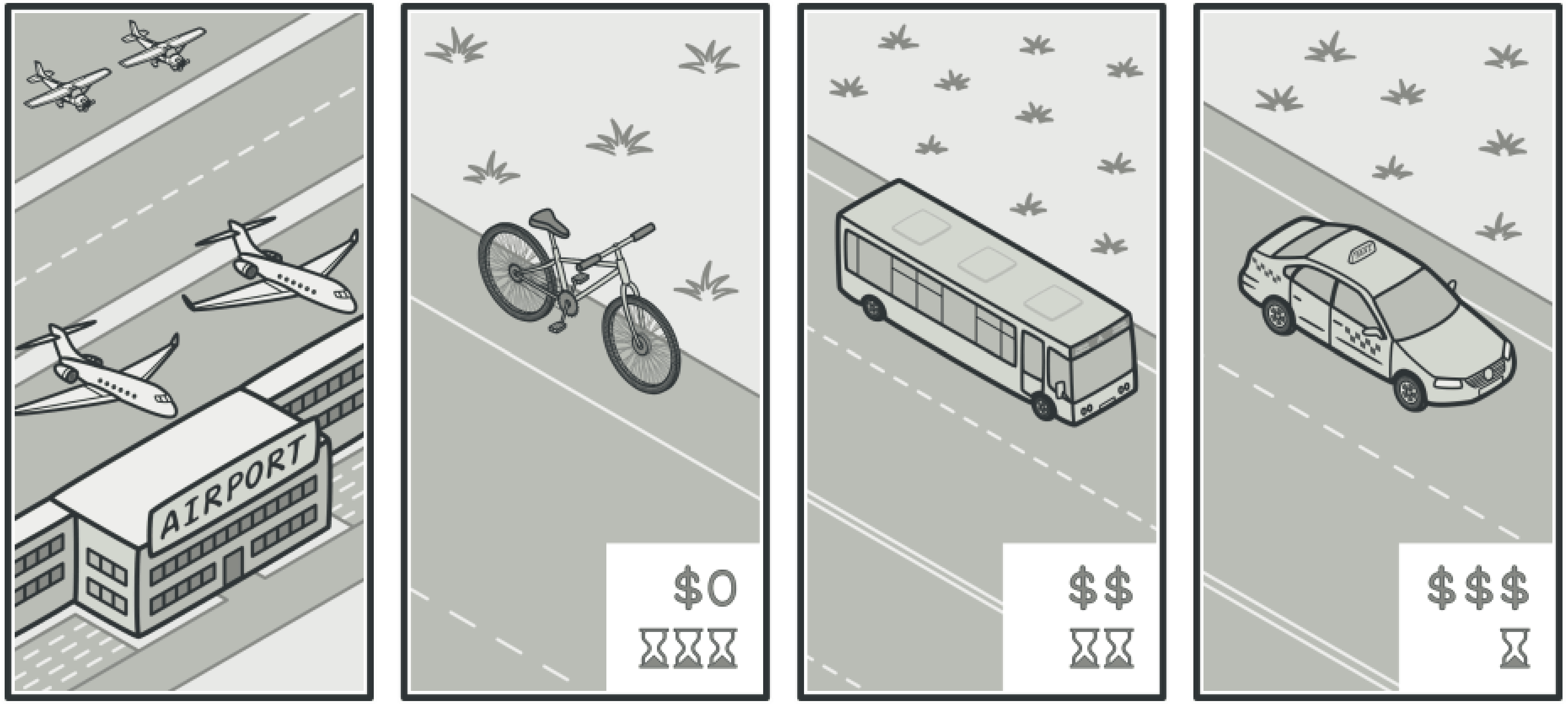
Strategy

Intent

Define a **family of algorithms**, **encapsulate each one**, and make them **interchangeable**.

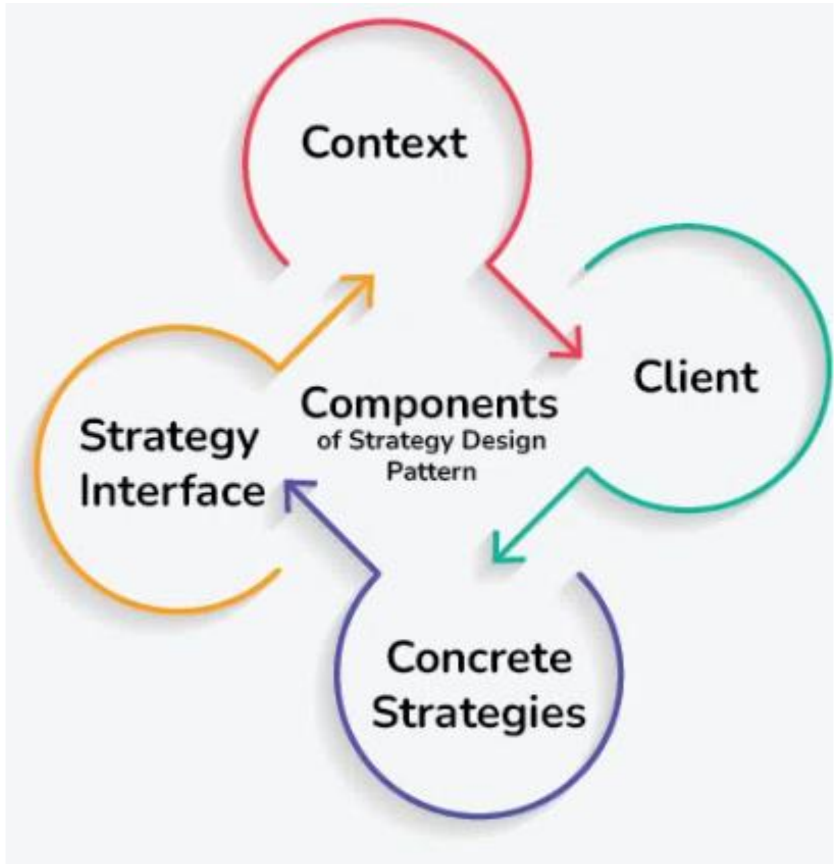
Strategy **lets the algorithm vary independently** from **clients** that use it.

Strategy Pattern :Real-World Examples



Components of the Strategy Design Pattern

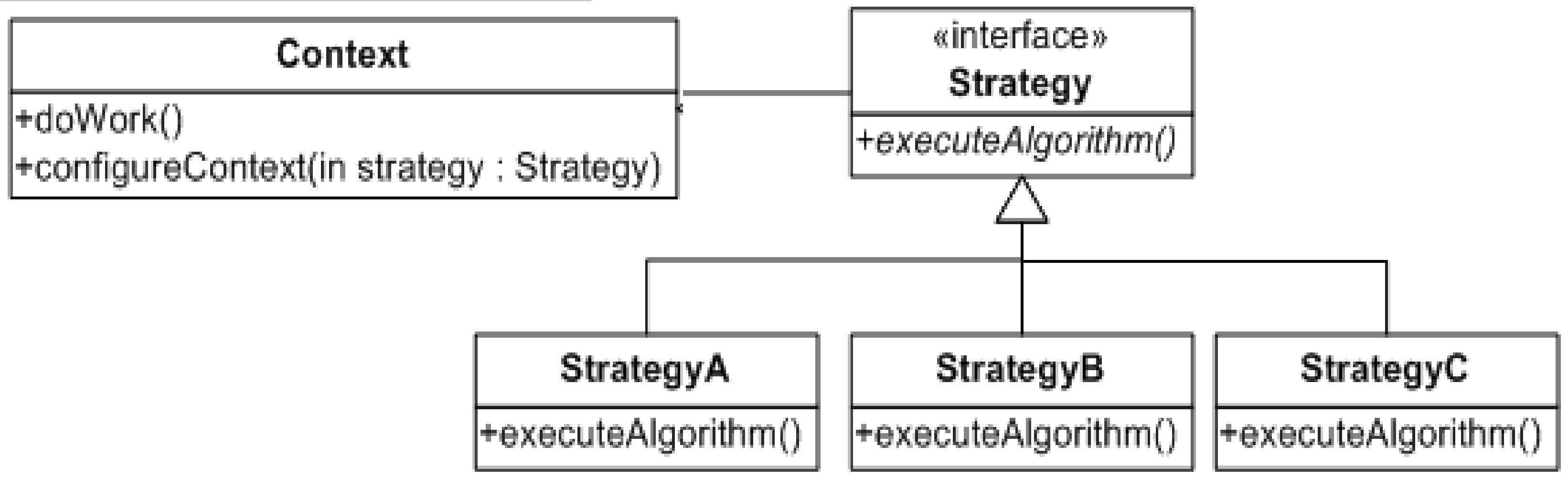
- **Strategy Interface:** Defines common methods for all strategies, ensuring interchangeability.
- **Concrete Strategies:** Implement the interface with specific algorithms; can be swapped based on needs.
- **Context:** Holds a strategy reference, delegates tasks, allows swapping strategies transparently.
- **Client:** Chooses and configures the strategy, passes it to the Context for flexible execution.



Strategy Pattern: Structure

Strategy Interface: Defines common methods for all strategies, ensuring interchangeability.
Concrete Strategies: Implement the interface with specific algorithms; can be swapped based on needs.
Context: Holds a strategy reference, delegates tasks, allows swapping strategies transparently.
Client: Chooses and configures the strategy, passes it to the Context for flexible execution.

Strategy Pattern: Structure



Strategy Interface: Defines common methods for all strategies, ensuring interchangeability.
Concrete Strategies: Implement the interface with specific algorithms; can be swapped based on needs.
Context: Holds a strategy reference, delegates tasks, allows swapping strategies transparently.
Client: Chooses and configures the strategy, passes it to the Context for flexible execution.

Step-by-step Implementation

Client -> Context: The client selects and configures a suitable strategy, then passes it to the context to execute the task.

Context -> Strategy: The context holds the strategy reference and delegates execution to it via the common interface.

Strategy -> Context: The strategy executes its algorithm, returns results, or performs necessary actions for the context to use.

Strategy Interface: Defines a contract ensuring all strategies are interchangeable.

Decoupling: Context remains unaware of strategy details, enabling flexibility and easy substitution.

Strategy Pattern: Problem Scenario

In our travel example, a Navigator allows the client to choose a travel strategy, such as walking, taking a car, or riding a bicycle. The client sets the strategy, and the Navigator executes the route using the chosen method

Strategy

Benefits

- We can **add new strategies without having to change Context**.
- We can **avoid complex conditionals** that realize the alternative algorithms in **Context**.

Liabilities

- Specifying a **common interface** for different algorithms **may not be easy**.
- Strategies increase the **n**
- **umber of objects** in an application.