# Weekly Homework 8

Yevgeniy Terentyev, Miriam Gaetano
Softwaretechnik

June 13, 2025

## 8-1: Concepts

### a) Briefly explain the most important difference or relationship between...

- **Interface and Signature**:
  - **Signature**: A method's description including its name, parameters, and return type. Example: `int add(int x, int y)`
  - **Interface**: A set of abstract method signatures that a class can implement. It defines **what** a class does, not **how**.
- **Class and Component**:
  - **Class**: Defines the structure and behavior of objects.
  - **Component**: A reusable and self-contained software module, often composed of multiple classes. It's an architectural unit.
- **Component and Module**:
  - **Module**: A logical unit of code organization, e.g., a `.java` file.
  - **Component**: Has a well-defined interface, can be independent and reusable, and usually implements more complex features.
- **Cohesion and Coupling**:
  - **Cohesion**: Measures how strongly the responsibilities inside a module are related.
  - **Coupling**: Measures the dependency between different modules.

## b) Compare Components, Design Patterns, and Architectural Styles

| Concept | What it represents | Example |
|---|---|---|
| **Component** | A modular and reusable part of a system, often with a well-defined interface.<br>It is a **real element** of the system that can apply design patterns. | `PaymentProcessor` |
| **Design Pattern** | An abstract solution to a recurring software design problem.<br>It is a **reusable conceptual solution**, typically independent of specific programming languages. | Singleton, Adapter |
| **Architectural Style / Architecture** | A general model for organizing a software system at a global level.<br>Provides a **macro-level view** of the system and its components and relationships. | MVC, Microservices |

## c) Research the Singleton Design Pattern

### 1. Explain the pattern: What problem does it solve and how?

Sometimes, an application needs to ensure that **only one object exists** to handle something central, such as a configuration file or a logging system. This is where the Singleton pattern comes in.

- **Singleton** is a creational design pattern that ensures a class has **only one instance**, while providing a **global point of access** to it. It does so by keeping a private static reference to the sole instance, which only the Singleton class itself can manage.
  The object is lazily initialized: it is created the first time it is requested.
- This requires **hiding the constructor** (making it `private`) and exposing a public method like `getInstance()` that returns the single instance.

### 2. Illustrate the pattern with three hypothetical use cases

1. **Global Logger:** Without Singleton, each class would log to a different file or logger instance. Using Singleton, all classes use the same logger: `Logger.getInstance().log(...)`.
2. **Configuration Manager:** `AppConfig.getInstance()` returns the single object containing settings loaded at application start (e.g., from files or environment variables). This ensures consistent behavior across the application.
3. **Database Connection:** `DBConnection.getInstance()` ensures a single active or shared connection, reducing overhead and ensuring thread-safe access.

**3. Pattern category: Where does it fit in the taxonomy?**

Singleton belongs to the category of **Creational Patterns**, specifically those that **control object creation**. It **centralizes and regulates access to a single instance** of a class, avoiding unnecessary object duplication. It is a special case of the Factory concept, where the "product" is always the same instance.

## d) Characterize and compare Proxy, Adapter, Facade, Bridge

**Learning Objective:** Understand and compare structural design patterns based on purpose, structure, and usage context.

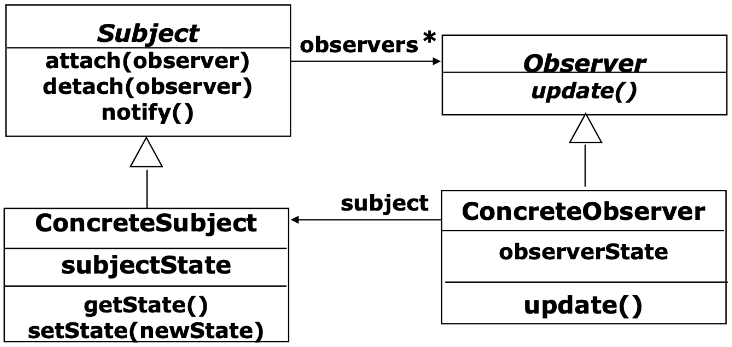
| Pattern | Purpose | When to Use | Structure | Practical Example | Main Motivation |
|---|---|---|---|---|---|
| **Proxy** | Provides a **substitute** to control access to a real object. | When you want to **control, protect, or delay** access to an object. | Client → Proxy → RealSubject | Lazy loading of images in a browser. | **Controlled access**, allowing behavior modification without changing the real object. |
| **Adapter** | Makes two **incompatible interfaces compatible**. | When you want to **reuse existing components** with a different interface. | Client → Adapter → Adaptee | Using an old API inside a new software system. | **Integration and compatibility** between interfaces. |
| **Facade** | Provides a **simplified interface** to a complex subsystem. | When you want to **hide internal complexity** from the user. | Client → Facade → Subsystems | Graphics library exposing a single public API. | **Simplification of interfaces**, ideal for public APIs or beginners. |
| **Bridge** | **Separates** abstraction from its implementation. | When **both abstraction and implementation may vary independently**. | Client → Abstraction → Implementation | GUI toolkit supporting multiple operating systems. | **Separation and flexibility**, designed **from the start**, unlike Adapter. |

# 8-2: Design Pattern for Your Software Idea

## a) Reflect on which of the patterns presented in the lesson (excluding Singleton and Adapter) you could apply in your software idea.

The **Observer pattern** is suitable for our project because it allows us to maintain a modular and reactive structure. The **Timer**, which is the heart of the app, represents the observed subject. Every time its state changes (start, pause, complete, reset), it notifies the various observers — such as the UI, the notification system, the cloud sync module, etc. This way, we can add new features (e.g. new types of notifications or logging) **without changing the Timer itself**, but only by registering new observers. This satisfies the principle of **low coupling and high cohesion**, as recommended in class.

## b) Represent the chosen pattern in its general form (using role names) in UML notation. Perform searches if necessary and cite sources.

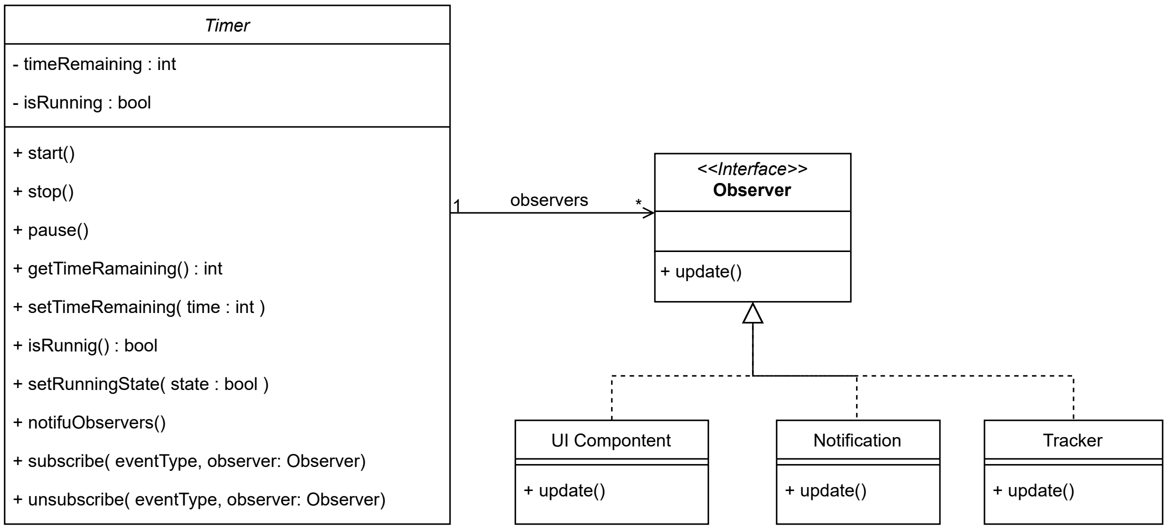


## c) Adapt the chosen pattern to your context and also represent the result with a UML diagram.

See the figure on the next page.

## d) Not all aspects of a pattern are easily expressible in UML. Implement in Java or pseudo code an idea of how the pattern might be used in your context. Create the necessary interfaces and classes with the relevant code, showing at least data structures and control (empty methods are not enough).

```
// Observer Interface
interface Observer {
```



```
    method update(data)
}

// Subject: Timer
class Timer {
    private timeRemaining
    private isRunning
    private observers: hashmap of event types and observers

    method start() {
        isRunning = true
        while (timeRemaining > 0 and isRunning) do
            wait(1 second) // Simulate countdown
            timeRemaining -= 1
            if (timeRemaining == 0) then
                isRunning = false
                notify("sessionEnded", this)

            notify("timeUpdated", this)
        end while
    }

    method notify(eventType, data) is
        foreach (listener in observers.of(eventType)) do
            listener.update(data)
        end foreach
    end method
    method subscribe(eventType, observer) is
        observers.add(eventType, observer)
    end method

    method unsubscribe(eventType, observer) is
        observers.remove(eventType, observer)
    end method

    // Concrete Observer: UI Component
class UIComponent implements Observer {
    method update(data) {
        // Update the UI with the new timer data
    }
}

// Concrete Observer: Notification Component
class NotificationComponent implements Observer {
    method update(data) {
        // Send a notification based on the timer data
    }
}

// Concrete Observer: Statistics Tracker
class StatisticsTracker implements Observer {
    method update(data) {
        // Log the timer data for statistics
    }
}

// Main Program
timer = new Timer()
ui = new UIComponent()
notification = new NotificationComponent()
statistics = new StatisticsTracker()

timer.subscribe("timeUpdated", ui)
timer.subscribe("sessionEnded", notification)
timer.subscribe("sessionEnded", statistics)

timer.setTimeRemaining(25 * 60) // Set for 25 minutes
timer.start() // Start the Pomodoro session
}
```

## 8-3

We engaged with the task