OBSERVER PATTERN

OLUWAPELUMI ADEWUYI

TYIONA ALFORD

AMARI HALL

Introduction

- The Observer Pattern is a behavioral design pattern in java and is one of the most commonly used design patterns in Java. It defines a **one-to-many** relationship between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Some quick history on the observer pattern is that it was first formally introduced in the 1994 book “Design Patterns: Elements of Reusable object-oriented software” by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The group is known as “the Gang of Four”

Definition

- The Observer Pattern allows objects (observers) to "subscribe" to another object (subject). When the subject changes, it notifies all its observers.

When to use it:

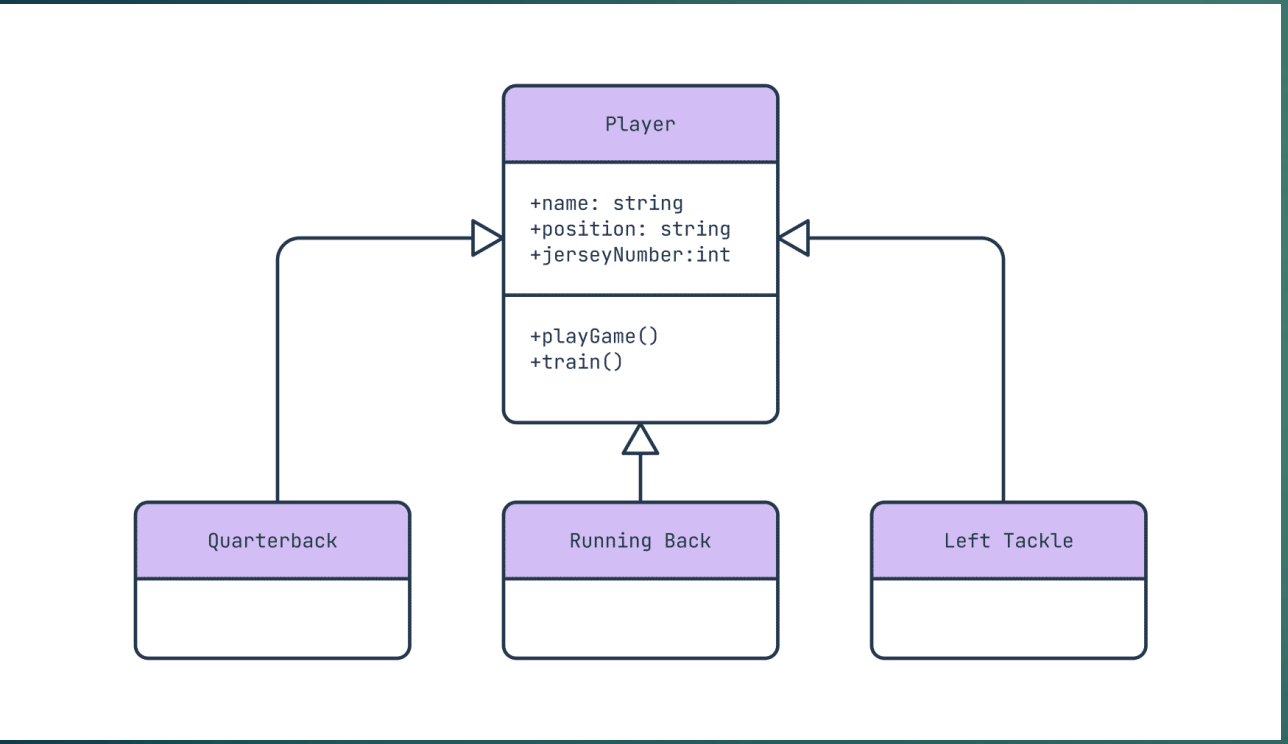
- When an object's state change should trigger updates in other objects.
- Especially useful in **event-driven systems**, **UI updates**, or **real-time notifications**.

Real-Life Analogy

Analogy: Pizza Order Tracker
Think of a pizza delivery app:

- You (Observer) subscribe to updates on your order.
- The pizza shop (Subject) updates the order status.
- You get notifications: "Order received", "Being prepared", "Out for delivery".
- This shows how the Observer is notified every time the Subject's state changes.

UML Diagram



Subject (interface):
defines register(), unregister(), notifyObservers( )
ConcreteSubject: maintains a list of observers
Observer (interface): defines update()
ConcreteObserver: implements update() to receive changes

SAMPLE JAVA CODE

```
interface Subject {
    void register(Observer o);
    void unregister(Observer o);
    void notifyObservers();
}
interface Observer {
    void update(String message);
class Order implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String status;

    public void register(Observer o) { observers.add(o); }
    public void unregister(Observer o) { observers.remove(o); }
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(status);
        }
    }
```

```
    public void setStatus(String s) {
        this.status = s;
        notifyObservers();
    }
}

class Customer implements Observer {
    private String name;
    public Customer(String name) { this.name = name; }
    public void update(String message) {
        System.out.println(name + " received update: " + message);
    }
}
```

Code Explanation

- Subject and Observer are interfaces for flexibility.
- Order (Subject) stores customers (Observers).
- When setStatus() is called, notifyObservers() loops through all customers and sends them the update.
- Demonstrates how you can add/remove customers without changing the main logic.

Pros & Cons

- ✅ **Pros:**
- Promotes loose coupling
- Dynamic relationships – observers can be added/removed at runtime
- Great for real-time data updates
- Scalability
- ❌ **Cons:**
- Can be hard to debug with many observers
- May cause performance issues if too many updates
- Risk of memory leaks if observers aren't unregistered properly
- Can Update unnecessarily as it reacts to every change in the subjects state

Use Cases

- **Java's java.util.Observer and Observable** (now deprecated, but a classic example)
- **MVC architecture** – Views observe Models
- **Event-driven systems** – e.g., GUI toolkits like JavaFX or Swing
- **Notification systems** –- stock apps, social media alerts
- **E-commerce systems** –- tracking price changes

Key Takeaways:

- Observer Pattern connects objects in a flexible way.
- It's great for event handling and reactive programming.
- Helps manage changes across dependent objects without tight coupling.