Observer Design Pattern

The Observer Design Pattern is a behavioral design pattern that defines a one-to-many relationship between objects. When one object (the subject) changes its state, all its dependents (the observers) are notified and updated automatically.

Customer Requirement:

Imagine you are building a news agency application. The agency needs to send news updates to multiple subscribers. Whenever there is a new update, the system should automatically notify all subscribers, so they can receive the latest news.

Solution: Observer Pattern

The Observer Pattern is an ideal solution for this scenario. It allows the news agency (the subject) to maintain a list of subscribers (observers) who are interested in updates. When the news agency has new information, it notifies all subscribers, and they can receive the updates.

Components of the Observer Pattern:

1. Subject: Maintains a list of observers. Provides methods to add or remove observers. Notifies all observers when its state changes.

2. Observer: Defines an interface for receiving updates from the subject. Implements the update method, which is called when the subject's state changes.

3. Concrete Subject: Extends the subject and stores the actual state of interest to the observers. When the state changes, it notifies all registered observers.

4. Concrete Observer: Implements the Observer interface. Maintains a reference to a concrete subject. Implements the update method to keep its state consistent with the subject's state.

Example: News Agency and Subscribers

1. Subject Interface (NewsAgency):

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

public class NewsAgency {
    private List<Subscriber> subscribers = new ArrayList<>();

    public void addSubscriber(Subscriber subscriber) {
        subscribers.add(subscriber);
    }

    public void removeSubscriber(Subscriber subscriber) {
        subscribers.remove(subscriber);
    }

    public void notifySubscribers(String news) {
        for (Subscriber subscriber : subscribers) {
            subscriber.update(news);
        }
    }
}
```

2. Observer Interface (Subscriber):

```java
public interface Subscriber {
    void update(String news);
}
```

3. Concrete Subject (NewsAgencyImpl):

```java
public class NewsAgencyImpl extends NewsAgency {
    private String latestNews;

    public void setLatestNews(String news) {
        this.latestNews = news;
        notifySubscribers(news);
    }
}
```

4. Concrete Observer (EmailSubscriber, SMSSubscriber):

```java
public class EmailSubscriber implements Subscriber {
    private String name;

    public EmailSubscriber(String name) {
        this.name = name;
    }
    @Override
    public void update(String news) {
        System.out.println(name + " received news via Email: " + news);
    }
}

public class SMSSubscriber implements Subscriber {
    private String name;

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

    @Override
    public void update(String news) {
        System.out.println(name + " received news via SMS: " + news);
    }
}
```

5. Client Code:

```java
public class ObserverPatternDemo {
    public static void main(String[] args) {
        NewsAgencyImpl newsAgency = new NewsAgencyImpl();

        Subscriber emailSubscriber1 = new EmailSubscriber("Alice");
        Subscriber emailSubscriber2 = new EmailSubscriber("Bob");
        Subscriber smsSubscriber = new SMSSubscriber("Charlie");


        newsAgency.addSubscriber(emailSubscriber1);
        newsAgency.addSubscriber(emailSubscriber2);
        newsAgency.addSubscriber(smsSubscriber);


        newsAgency.setLatestNews("Breaking News: Observer Pattern in Action!");


        newsAgency.removeSubscriber(emailSubscriber1);


        newsAgency.setLatestNews("Update: Observer Pattern Still in Action!");
    }
}
```

How It Works:

- Subject (NewsAgency): The NewsAgency maintains a list of Subscriber objects that have registered interest in receiving updates.
- Concrete Subject (NewsAgencyImpl): When the NewsAgencyImpl receives new information, it updates its internal state and then calls notifySubscribers() to notify all subscribers.
- Observer (Subscriber): The Subscriber interface defines the update() method that all concrete observers must implement.
- Concrete Observers (EmailSubscriber, SMSSubscriber): These classes implement the Subscriber interface and define how they will handle the updates from the NewsAgency.

Why It Is a Behavioral Design Pattern:

The Observer Pattern is a behavioral design pattern because it focuses on the communication between objects. It defines the relationship between the subject and observers and how they interact. The pattern ensures that changes in one object (the subject) automatically propagate to all dependent objects (observers) without the subject needing to know the details of how those observers handle the update.

Benefits of the Observer Pattern:

1. Loose Coupling: The subject and observers are loosely coupled. The subject only knows that observers implement a specific interface, not how they handle the update.

2. Scalability: New observers can be added without modifying the subject. The subject can notify any number of observers.

3. Flexibility: The observer pattern allows dynamic relationships between objects. Observers can be added or removed at runtime.

When to Use the Observer Pattern:

- When an object needs to notify other objects without knowing who or how many objects need to be notified.
- When changes to one object require changing others, but you do not want the objects to be tightly coupled.
- When the number of objects that need to be notified may change dynamically.