Observer Design Pattern

Observer Pattern Simplified 👀 | Real-World Examples + Best Practices 📚

Topic Tags:

LLD System Design

🐈‍⬛ Github Codes Link: https://github.com/aryan-0077/CWA-LowLevelDesignCode

Observer Design Pattern: How to Stay Updated Without Constantly Checking 📲🔔

Imagine you're watching your favorite YouTube channel. Every time they upload a new video, you get a notification. You don't have to keep checking the channel to see if there's something new. Instead, you get notified automatically when they post. This is exactly how the Observer Design Pattern works in software.

In programming, the Observer Pattern allows one object (the subject) to notify other objects (the observers) whenever there is a change in its state. This is great for systems where certain parts of your application need to stay updated in real-time but shouldn't be tightly coupled to each other.

Why Is It Called the Observer Pattern? 👀

The name Observer comes from the fact that some parts of the program (observers) are "watching" another part (subject) for changes. When something changes in the subject (for example, a new video is posted on YouTube), all the observers are notified. This keeps everything in sync without directly linking the two parts. So, the observers observe the subject for changes and react accordingly.

Solving the Problem Using the Traditional Method 🔧

Let's imagine a scenario where you have a YouTube channel, and you want to notify your subscribers every time a new video is uploaded. The traditional way to solve this problem could involve manually checking for updates each time.
Let's write some code for this traditional approach. Here, the YouTubeChannel is the subject, and YouTubeSubscriber is the observer.

Java

```java
import java.util.ArrayList;
import java.util.List;
class YouTubeChannel {
  private List<String> subscribers = new ArrayList<>();
  private String video;

  // Method to add a new subscriber
  public void addSubscriber(String subscriber) {
    subscribers.add(subscriber);
  }

  // Method to upload a new video
  public void uploadNewVideo(String video) {
    this.video = video;
    notifySubscribers(); // Notify all subscribers about the new vi
  }

  // Notify all subscribers
  public void notifySubscribers() {
    for (String subscriber : subscribers) {
      System.out.println(
          "Notifying " + subscriber + " about new video: " + video)
    }
  }
}

class YouTubeSubscriber {
  private String name;
```

```
29    public YouTubeSubscriber(String name) {
30      this.name = name;
31    }
32    public void subscribe(YouTubeChannel channel) {
33      channel.addSubscriber(name);
34    }
35    public void watchVideo(YouTubeChannel channel) {
36      System.out.println(name + " is watching the video: " + channel.
37    }
38  }
```

Why Is This Approach Not Ideal? 😕

• Manual Checking:
In this approach, we manually notify each subscriber every time a new video is uploaded. If there are hundreds of subscribers, this becomes cumbersome.

• Not Scalable:
Adding a new notification method (e.g., email, SMS) requires modifying the YouTubeChannel class, which leads to tight coupling and difficult maintenance.

• Hard to Extend:
If we wanted to add more observers (for example, send notifications through an app), we would have to touch the YouTubeChannel class, breaking the open/closed principle.

Interviewer's Questions: What's Wrong with This? 🤔

Now, an interviewer might ask:
• What happens if you have a lot of subscribers?
○ The code could get messy and slow down because you are doing everything manually.

• What if we need to add a new feature like sending notifications by email?
○ You would need to modify the YouTubeChannel class and update the logic everywhere, which increases complexity.

The Ugly Code: When Things Start to Break Down 🧩

As you can see, this approach doesn't scale well. With each new feature, we would need to keep adding new lines of code in the notifySubscribers() method. Here's how the ugly code might look when we add email notifications:

Java

```
public void notifySubscribers() {
  for (String subscriber : subscribers) {
    System.out.println(
        "Notifying " + subscriber + " about new video: " + video);
    // Add new feature: send an email notification
    sendEmail(subscriber);
  }
}

public void sendEmail(String subscriber) {
  System.out.println("Sending email to " + subscriber);
}
```

• Code Duplication:
Every time we add a new feature, like email notifications, we're repeating logic in the YouTubeChannel class.

• Hard to Maintain:
As we add more notification types (SMS, Push notifications), this class will get bloated and hard to maintain.

The Observer Design Pattern Explained in Detail 🧐

Now that we've introduced the Observer Design Pattern, let's break it down step by step to understand how it works, how it improves our solution, and how to use it with multiple users (observers) watching the same subject (YouTube channel). We will take you through interfaces, classes, and the driver code in simple terms, just like we are explaining to a friend!

1. The Observer Interface 📝

In the Observer Design Pattern, the Observer is the one that reacts to the changes (like a subscriber reacting to a new video). To make this pattern work, we create an interface called Subscriber. The job of this interface is to define what methods a subscriber (observer) should have. In our case, the update() method is the one we use to notify a subscriber when something happens (like a new video).

Java

```
public interface Subscriber {
  void update(String video); // This is the method the observer will
```

```
3 }
```

• update(String video):
This method is called when the YouTubeChannel uploads a new video. Each observer (subscriber) will implement this method to decide how they should react to the update (e.g., watching the video).

2. Concrete Observer Class 👀

Now, let's create a class for the YouTubeSubscriber, which implements the Subscriber interface. When a new video is uploaded, this class will print a message saying that the subscriber is watching the new video.

Java

```java
public class YouTubeSubscriber implements Subscriber {
  private String name; // Name of the subscriber

  public YouTubeSubscriber(String name) {
    this.name = name; // Initialize the subscriber with their name
  }

  @Override
  public void update(String video) {
    // When notified, this method will execute, and the subscriber wa
    // new video
    System.out.println(name + " is watching the video: " + video);
  }
}
```

Here, the YouTubeSubscriber class:
• Takes the name of the subscriber when created.
• Implements the update() method to react when a new video is uploaded.

Solving the problem in the Ugly Code - Observer Classes (Email, Push Notifications, etc)
Instead of having all logic inside the notifySubscribers() method, we define separate observer classes for each notification type:

Java

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

```
  public EmailSubscriber(String email) {
    this.email = email;
  }

  @Override
  public void update(String video) {
    System.out.println(
        "Sending email to " + email + ": New video uploaded: " + vi
  }
}

public class PushNotificationSubscriber implements Subscriber {
  private String userDevice;
  public PushNotificationSubscriber(String userDevice) {
    this.userDevice = userDevice;
  }

  @Override
  public void update(String video) {
    System.out.println("Sending push notification to " + userDevice
        + ": New video uploaded: " + video);
  }
}
```

Here, we have two different observers: one for email notifications and another for push notifications. Each observer is responsible for notifying the user in a way that suits them.

3. The Subject Interface 🗣️

The Subject is the one that changes. In our case, it's the YouTubeChannel (the channel posting videos). The YouTubeChannel needs to keep track of all its subscribers and notify them when something changes (like uploading a new video).

We create a YouTubeChannel interface to define the methods required for any channel:

Java

```
public interface YouTubeChannel {
  void addSubscriber(Subscriber subscriber); // Method to add a new s
  void removeSubscriber(Subscriber subscriber); // Method to remove a
  void notifySubscribers(); // Method to notify all subscribers
}
```

• addSubscriber(Subscriber subscriber): Adds a new subscriber to the channel.
• removeSubscriber(Subscriber subscriber): Removes a subscriber from the channel.
• notifySubscribers(): Notifies all the subscribed users about the new video.

4. Concrete Subject Class 🖥️

Now, we create the concrete class for the subject, which is the actual YouTubeChannel that manages the subscribers and uploads the videos. This class will keep track of all the subscribers and notify them when a new video is uploaded.

Java

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

public class YouTubeChannelImpl implements YouTubeChannel {
  private List<Subscriber> subscribers =
      new ArrayList<>(); // List of subscribers
  private String video; // The video that will be uploaded

  @Override
  public void addSubscriber(Subscriber subscriber) {
    subscribers.add(subscriber); // Add a subscriber to the channel
  }

  @Override
  public void removeSubscriber(Subscriber subscriber) {
    subscribers.remove(subscriber); // Remove a subscriber from the
  }

  @Override
  public void notifySubscribers() {
    // Notify all subscribers about the new video
    for (Subscriber subscriber : subscribers) {
      subscriber.update(video); // Call update() for each subscribe
    }
  }

  public void uploadNewVideo(String video) {
    this.video = video; // Set the video that is being uploaded
    notifySubscribers(); // Notify all subscribers about the new vi
```

```
30    }
31  }
```

In this class:
• We use a List<Subscriber> to store all subscribers who are interested in receiving updates.
• The uploadNewVideo() method is used to set the new video and call notifySubscribers() to alert all the observers about the new video.

5. Driver Code: Putting It All Together 🎬

Now, let's use all the classes we've created and run the program. We will have a YouTubeChannel, a couple of subscribers, and upload a new video. The subscribers will get updated and react to the new video.

Java

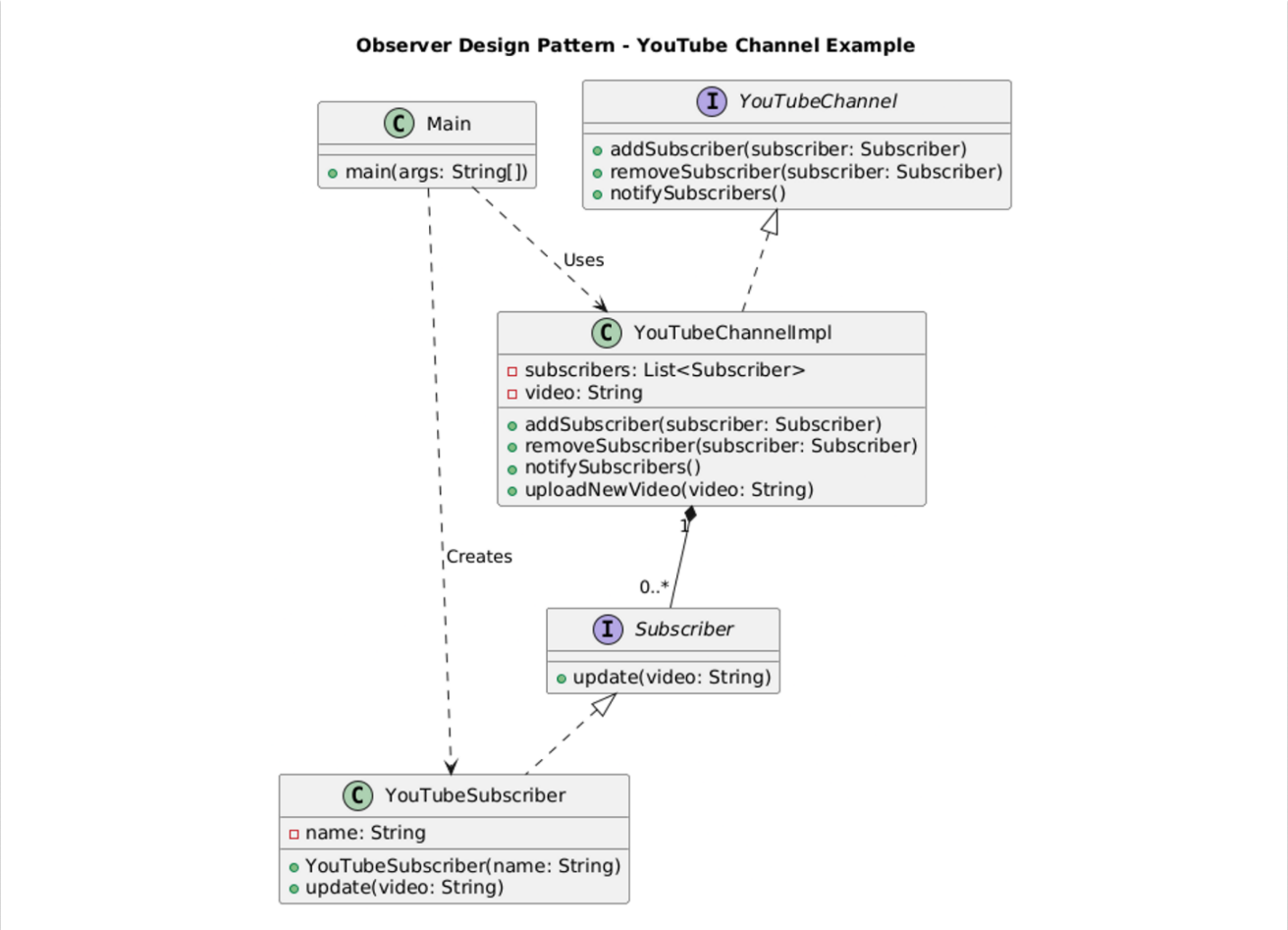
```java
public class Main {
  public static void main(String[] args) {
    // Create a YouTube channel
    YouTubeChannelImpl channel = new YouTubeChannelImpl();
    // Create subscribers
    YouTubeSubscriber alice = new YouTubeSubscriber("Alice");
    YouTubeSubscriber bob = new YouTubeSubscriber("Bob");
    // Subscribe to the channel
    channel.addSubscriber(alice);
    channel.addSubscriber(bob);
    // Upload a new video and notify subscribers
    channel.uploadNewVideo("Java Design Patterns Tutorial");
    // Output:
    // Alice is watching the video: Java Design Patterns Tutorial
    // Bob is watching the video: Java Design Patterns Tutorial
    // You can also remove a subscriber and upload another video
    channel.removeSubscriber(bob);
    channel.uploadNewVideo("Observer Pattern in Action");
    // Output:
    // Alice is watching the video: Observer Pattern in Action
  }
}
```

Here's what happens in the code:
1. Creating the Channel and Subscribers:
We create the YouTubeChannelImpl and YouTubeSubscriber objects.

2. Subscribing to the Channel:
Both Alice and Bob subscribe to the YouTubeChannelImpl.

3. Uploading a Video:
When a new video is uploaded via uploadNewVideo(), both subscribers are notified and will "watch" the video by executing their update() method.

4. Unsubscribing a Subscriber:
Bob unsubscribes, and when a new video is uploaded, only Alice is notified.



Explanation of the Diagram

1. Main Class:
○ Acts as the driver code.
○ Creates the YouTubeChannelImpl and YouTubeSubscriber instances.
○ Adds/removes subscribers and triggers video uploads.

2. Subscriber Interface:
○ Represents the Observer in the pattern.
○ Defines the update(video: String) method that all subscribers must implement.

3. YouTubeSubscriber Class:

○ Implements the Subscriber interface.
○ Reacts to video updates by displaying a message.

4. YouTubeChannel Interface:
○ Represents the Subject in the pattern.
○ Defines methods to add, remove, and notify subscribers.

5. YouTubeChannelImpl Class:
○ Implements YouTubeChannel.
○ Manages a list of subscribers and notifies them when a new video is uploaded.

6. Relationships:
○ Inheritance: YouTubeSubscriber and YouTubeChannelImpl implement their respective interfaces.
○ Aggregation: YouTubeChannelImpl maintains a list of Subscriber objects.
○ Usage: The Main class interacts with the system to demonstrate the pattern.
This demonstrates the Observer Pattern with a YouTube-like notification system.

Advantages of the Observer Pattern 🎉

1. Decoupling: The YouTubeChannel doesn't need to know what each observer does. It just notifies them about the update. 🎯

2. Scalability: Adding new types of observers (e.g., email, SMS) is as simple as implementing the Subscriber interface. 📈

3. Flexibility: Observers can join or leave at any time without affecting the YouTubeChannel. 🔄

4. Maintainability: The YouTubeChannel stays clean and simple, while the observers handle their own logic. 🔧

Real-Life Use Cases for the Observer Pattern 🌍

1. Social Media Notifications: When someone you follow posts something, you get a notification.
2. Stock Market Alerts: When stock prices change, you are notified.
3. Weather Apps: The app notifies you about weather changes.
4. Message Systems: When a new message arrives, all subscribers are notified.

Conclusion: Observer Pattern in Action 🏆

The Observer Design Pattern is an excellent way to implement notification systems in software. It helps keep things decoupled, modular, and scalable. Whether it's YouTube notifications, stock market updates, or weather changes, the Observer pattern is a simple

and efficient way to keep your system’s components updated without the headache of direct dependencies. 😊