**Observer Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Observer Design Pattern :**

The Observer Design Pattern is a behavioral design pattern that defines a one-to-many dependency between objects, so that when one object changes state, all its dependents (observers) are notified and updated automatically. This pattern is useful in scenarios where one object's state change should trigger changes in other objects, while keeping the objects loosely coupled.

In the Observer pattern, there are typically three main components:

1. Subject (Observable): This is the object that maintains a list of its dependents (observers), and notifies them of any changes in its state. The subject provides methods for attaching, detaching, and notifying observers.

2. Observer: This is the interface or abstract class defining the update method, which is called by the subject to notify the observer of changes. Observers register with the subject to receive updates.

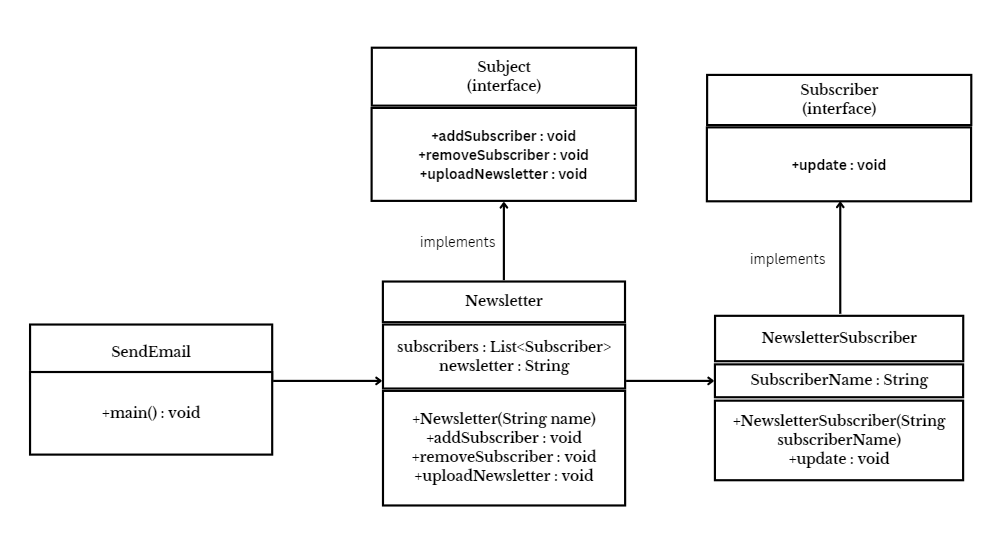
3. Concrete Subject: This is a concrete implementation of the subject interface. It maintains the state and sends notifications to observers when the state changes.

4. Concrete Observer: This is a concrete implementation of the observer interface. It defines the behavior to be executed in response to updates from the subject.

The Observer pattern promotes loose coupling between objects, as the subject doesn't need to know the details of its observers, it only knows they implement a certain interface. This makes it easy to add or remove observers without affecting the subject or other observers.

This pattern is commonly used in user interface frameworks (e.g., event handling in GUIs), distributed systems, and in many other scenarios where the state changes of one object need to be propagated to multiple other objects in a decoupled manner.

* **Program :** Implement observer design pattern for newsletter example.
* **UML Diagram :**



* **Code :**

import java.util.\*;

interface Subject

{

public void addSubscriber(Subscriber subscriber);

public void removeSubscriber(Subscriber subscriber);

public void uploadNewsletter(String newsletter);

}

interface Subscriber

{

public void update(String newsletter);

}

class Newsletter implements Subject

{

private List<Subscriber>subscribers = new ArrayList<>();

public void addSubscriber(Subscriber subscriber)

{

subscribers.add(subscriber);

}

public void removeSubscriber(Subscriber subscriber)

{

subscribers.remove(subscriber);

}

public void uploadNewsletter(String newsletter)

{

System.out.println("Uploading newsletter: "+newsletter);

for (Subscriber subscriber : subscribers)

{

subscriber.update(newsletter);

}

}

}

class NewsletterSubscriber implements Subscriber

{

private String subscriberName;

public NewsletterSubscriber(String subscriberName)

{

this.subscriberName = subscriberName;

}

public void update(String newsletter)

{

System.out.println(subscriberName+" recieved newsletter: "+newsletter);

}

}

class SendEmail

{

public static void main(String[] args)

{

Newsletter newsletter = new Newsletter();

NewsletterSubscriber s1 = new NewsletterSubscriber("Heet");

NewsletterSubscriber s2 = new NewsletterSubscriber("Devanshi");

NewsletterSubscriber s3 = new NewsletterSubscriber("Maitri");

NewsletterSubscriber s4 = new NewsletterSubscriber("Meet");

newsletter.addSubscriber(s1);

newsletter.addSubscriber(s2);

newsletter.addSubscriber(s3);

newsletter.addSubscriber(s4);

newsletter.uploadNewsletter("February Highlights");

newsletter.removeSubscriber(s1);

newsletter.uploadNewsletter("March Highlights");

}

}

* **Output:**

