Observer Design Pattern

Last Updated : 03 Jan, 2025

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

Table of Content

What is the Observer Design Pattern?

The Observer Design Pattern is a behavioral design pattern that defines a one-to-many dependency between objects. When one object (the subject) changes state, all its dependents (observers) are notified and updated automatically. It primarily deals with the interaction and communication between objects, specifically focusing on how objects behave in response to changes in the state of other objects.

> ***Note***: ***Subjects*** *are the objects that maintain and notify observers about changes in their state, while* ***Observers*** *are the entities that react to those changes.*

Below are some key points about observer design pattern:

- Defines how a group of objects (observers) interact based on changes in the state of a subject.
- Observers react to changes in the subject's state.
- The subject doesn't need to know the specific classes of its observers, allowing for flexibility.
- Observers can be easily added or removed without affecting the subject.

Real-world analogy of the Observer Design Pattern

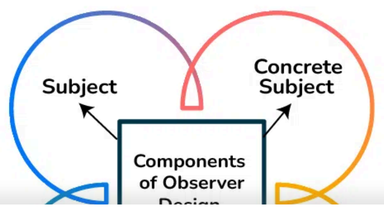
Let's understand the observer design pattern through a real-world example:

*magine a scenario where a **weather station** is observed by various **smart devices**. The weather station maintains a list of registered devices. Weather Station will update all the devices whenver there is change in the weather.*

- Each of the devices are concrete observers and each have their ways to interpret and display the information.
- The Observer Design Pattern provides a flexible and scalable system where adding new devices or weather stations doesn't disrupt the overall communication, providing real-time and location-specific weather updates to users.

Components of Observer Design Pattern

Below are the main components of Observer Design Pattern:



- **Subject:**
    - The subject maintains a list of observers (subscribers or listeners).
    - It Provides methods to register and unregister observers dynamically and defines a method to notify observers of changes in its state.
- **Observer:**
    - Observer defines an interface with an update method that concrete observers must implement and ensures a common or consistent way for concrete observers to receive updates from the subject.
- **ConcreteSubject:**
    - ConcreteSubjects are specific implementations of the subject. They hold the actual state or data that observers want to track. When this state changes, concrete subjects notify their observers.
    - For instance, if a weather station is the subject, specific weather stations in different locations would be concrete subjects.
- **ConcreteObserver:**
    - Concrete Observer implements the observer interface. They register with a concrete subject and react when notified of a state change.
    - When the subject's state changes, the concrete observer's `update()` method is invoked, allowing it to take appropriate actions.
    - For example, a weather app on your smartphone is a concrete observer that reacts to changes from a weather station.

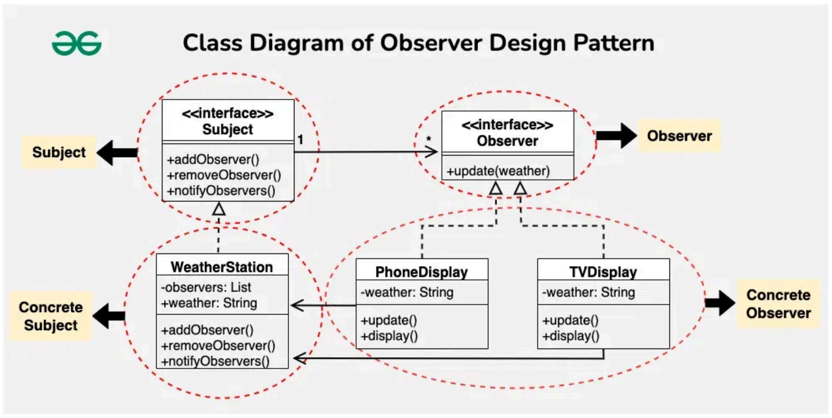
Consider a scenario where you have a weather monitoring system. Different parts of your application need to be updated when the weather conditions change.

Challenges or difficulties while implementing this system without Observer Design Pattern

- Components interested in weather updates would need direct references to the weather monitoring system, leading to **tight coupling**.
- Adding or removing components that react to weather changes requires modifying the core weather monitoring system code, making it hard to maintain.

How Observer Pattern helps to solve above challenges?

The Observer Pattern facilitates the decoupling of the weather monitoring system from the components that are interested in weather updates (via interfaces). Every element can sign up as an observer, and observers are informed when the weather conditions change. The weather monitoring system is thus unaffected by the addition or removal of components.



Below is the code of above problem statement using Observer

- The “`Subject"` interface outlines the operations a subject (like “`WeatherStation"`) should support.
- `"addObserver"` and “`removeObserver"` are for managing the list of observers.
- `"notifyObservers"` is for informing observers about changes.

```
public interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
```

2. Observer

- The “`Observer"` interface defines a contract for objects that want to be notified about changes in the subject (“`WeatherStation"` in this case).
- It includes a method “`update"` that concrete observers must implement to receive and handle updates.

```
public interface Observer {
    void update(String weather);
}
```

3. ConcreteSubject(WeatherStation)

- `"WeatherStation"` is the concrete subject implementing the “`Subject"` interface.
- It maintains a list of observers (“`observers"`) and provides methods to

- "setWeather" method updates the weather and notifies observers of the change.

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

public class WeatherStation implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String weather;

    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(weather);
        }
    }

    public void setWeather(String newWeather) {
        this.weather = newWeather;
        notifyObservers();
    }
}
```

4. ConcreteObserver(PhoneDisplay)

- "PhoneDisplay" is a concrete observer implementing the "Observer" interface.
- It has a private field **weather** to store the latest weather.
- The "update" method sets the new weather and calls the "display"

```
public class PhoneDisplay implements Observer {
    private String weather;

    @Override
    public void update(String weather) {
        this.weather = weather;
        display();
    }

    private void display() {
        System.out.println("Phone Display: Weather updated - "
+ weather);
    }
}
```

5. ConcreteObserver(TVDisplay)

- **"TVDisplay"** is another concrete observer similar to “**PhoneDisplay"**.
- It also implements the “**Observer"** interface, with a similar structure to “**PhoneDisplay"**.

```
class TVDisplay implements Observer {
    private String weather;

    @Override
    public void update(String weather) {
        this.weather = weather;
        display();
    }

    private void display() {
        System.out.println("TV Display: Weather updated - " +
weather);
    }
}
```

- Two observers (“`PhoneDisplay`" and “`TVDisplay`") are registered with the weather station using “`addObserver`".
- The “`setWeather`" method simulates a weather change to “Sunny,” triggering the “`update`" method in both observers.
- The output shows how both concrete observers display the updated weather information.

```
public class WeatherApp {
    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();

        Observer phoneDisplay = new PhoneDisplay();
        Observer tvDisplay = new TVDisplay();

        weatherStation.addObserver(phoneDisplay);
        weatherStation.addObserver(tvDisplay);

        // Simulating weather change
        weatherStation.setWeather("Sunny");

        // Output:
        // Phone Display: Weather updated - Sunny
        // TV Display: Weather updated - Sunny
    }
}
```

Complete code for the above example

Below is the complete code for the above example:

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

// Observer Interface
interface Observer {
```

```java
interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// ConcreteSubject Class
class WeatherStation implements Subject {
    private List<Observer> observers = new
ArrayList<>();
    private String weather;

    @Override
    public void addObserver(Observer observer)
{
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer
observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(weather);
        }
    }

    public void setWeather(String newWeather) {
        this.weather = newWeather;
        notifyObservers();
    }
}

```

```java
    @Override
    public void update(String weather) {
        this.weather = weather;
        display();
    }

    private void display() {
        System.out.println("Phone Display: Weather updated - " + weather);
    }
}

// ConcreteObserver Class
class TVDisplay implements Observer {
    private String weather;

    @Override
    public void update(String weather) {
        this.weather = weather;
        display();
    }

    private void display() {
        System.out.println("TV Display: Weather updated - " + weather);
    }
}

// Usage Class
public class WeatherApp {
    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();

        Observer phoneDisplay = new PhoneDisplay();
        Observer tvDisplay = new TVDisplay();

        // Simulating weather change
        weatherStation.setWeather("Sunny");

        // Output:
        // Phone Display: Weather updated - Sunny
        // TV Display: Weather updated - Sunny
    }
}
```

```
Phone Display: Weather updated - Sunny
TV Display: Weather updated - Sunny
```

When to use the Observer Design Pattern?

Below is when to use observer design pattern:

- When you need one object to notify multiple others about changes.
- When you want to keep objects loosely connected, so they don’t rely on each other’s details.
- When you want observers to automatically respond to changes in the subject’s state.
- When you want to easily add or remove observers without changing the

When not to use the Observer Design Pattern?

Below is when not to use observer design pattern:

- When the relationships between objects are simple and don’t require notifications.
- When performance is a concern, as many observers can lead to overhead during updates.
- When the subject and observers are tightly coupled, as it defeats the purpose of decoupling.
- When number of observers is fixed and won’t change over time.
- When the order of notifications is crucial, as observers may be notified in an unpredictable sequence.

Next Article

State Design Pattern

Similar Reads

Software Design Patterns Tutorial

Software design patterns are important tools developers, providing proven solutions to common problems encountered during software development. This article will act as tutorial to help you…

9 min read

Complete Guide to Design Patterns

Design patterns help in addressing the recurring issues in software design and provide a shared

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. Christopher Alexander says, "Each pattern describes a problem which occurs over and over...

9 min read

1. Creational Design Patterns

Creational Design Patterns

Creational Design Patterns focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed, and...

4 min read

Types of Creational Patterns

2. Structural Design Patterns

Structural Design Patterns

Structural Design Patterns are solutions in software design that focus on how classes and objects are organized to form larger, functional structures. These patterns help developers simplify relationships...

7 min read

Types of Structural Patterns

3. Behvioural Design Patterns

Company

About Us
Legal
Privacy Policy
In Media
Contact Us
Advertise with us
GFG Corporate Solution
Placement Training Program
GeeksforGeeks Community

Languages

Python
Java
C++
PHP
GoLang
SQL
R Language
Android Tutorial
Tutorials Archive

DSA

Data Structures
Algorithms
DSA for Beginners
Basic DSA Problems
DSA Roadmap
Top 100 DSA Interview Problems
DSA Roadmap by Sandeep Jain
All Cheat Sheets

Data Science & ML

Data Science With Python
Data Science For Beginner
Machine Learning
ML Maths
Data Visualisation
Pandas
NumPy
NLP
Deep Learning

Web Technologies

HTML
CSS
JavaScript
TypeScript
ReactJS

Python Tutorial

Python Programming Examples
Python Projects
Python Tkinter
Web Scraping
OpenCV Tutorial

Operating Systems
Computer Network
Database Management System
Software Engineering
Digital Logic Design
Engineering Maths
Software Development
Software Testing

Git
Linux
AWS
Docker
Kubernetes
Azure
GCP
DevOps Roadmap

System Design
High Level Design
Low Level Design
UML Diagrams
Interview Guide
Design Patterns
OOAD
System Design Bootcamp
Interview Questions

Inteview Preparation
Competitive Programming
Top DS or Algo for CP
Company-Wise Recruitment Process
Company-Wise Preparation
Aptitude Preparation
Puzzles

School Subjects
Mathematics
Physics
Chemistry
Biology
Social Science
English Grammar
Commerce
World GK

GeeksforGeeks Videos
DSA
Python
Java
C++
Web Development
Data Science
CS Subjects