Heaven's Light Is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

LAB REPORT

Course Code: CSE 3206

Course Title: Software Engineering Sessional

Experiment Name: Observer pattern and state pattern.

Submitted By

Taiyebah Tazkiyah – 2003142

Tahmid Sudad – 2003143

Submitted To

Farzana Parvin

Lecturer

Design Pattern: Observer

Summary:

The Observer Design Pattern is a behavioral pattern where a subject maintains a list of observers and notifies them of state changes, enabling a one-to-many relationship. It decouples subjects from observers, making it ideal for systems like event handling, real-time updates, and publish-subscribe mechanisms.

Codes:

Observer Interface:

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

ConcreteObserver Class:

```
public class ConcreteObserver implements Observer {
private String name;

    public ConcreteObserver(String name) {
this.name = name;
```

```
    }

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

Subject Interface:

```
public interface Subject {
    void registerObserver(Observer observer);
void unregisterObserver(Observer observer);     void
notifyObservers(String message); }
```

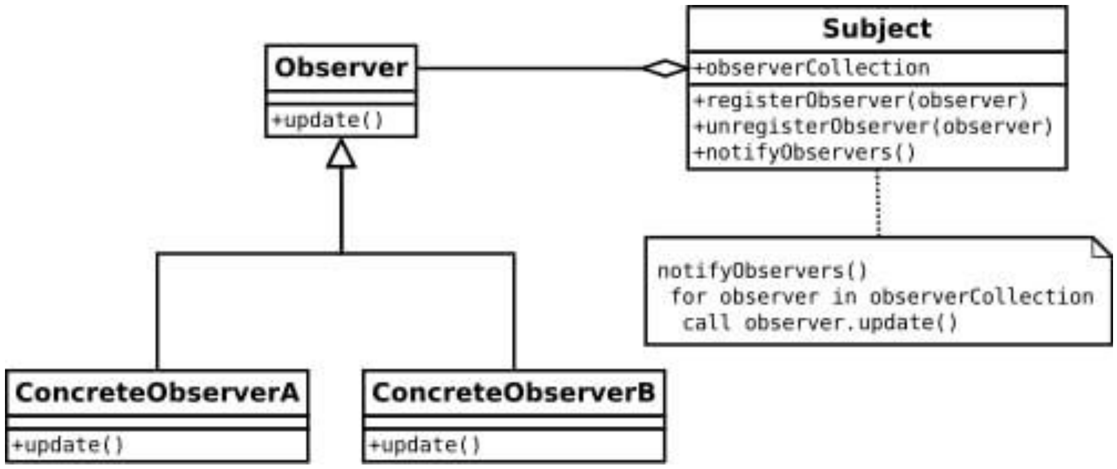
ConcreteSubject Class:

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

public class ConcreteSubject implements Subject {
    private List<Observer> observers = new ArrayList<>();
    @Override
    public void registerObserver(Observer observer) {
observers.add(observer);
    }

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

    @Override
    public void notifyObservers(String message) {
for (Observer observer : observers) {
            observer.update(message);
        }
    }
    public void changeState(String message) {
        System.out.println("Subject  state  changed:  "  +
message);
        notifyObservers(message);
    }

    // New method to get the list of observers
public List<Observer> getObservers() {
        return Collections.unmodifiableList(observers);
    }
}
```

ObserverPatternDemo class (includes main function):

```
import java.util.Iterator;

public class ObserverPatternDemo {
public ObserverPatternDemo() {
   }

   public static void main(String[] var0) {
      ConcreteSubject var1 = new ConcreteSubject();
      ConcreteSubject var2 = new ConcreteSubject();
      ConcreteObserver var3 = new ConcreteObserver("Observer 1");
      ConcreteObserver var4 = new ConcreteObserver("Observer 2");
ConcreteObserver var5 = new ConcreteObserver("Observer 3");
var1.registerObserver(var3);       var1.registerObserver(var4);
var2.registerObserver(var5);       var1.changeState("State 1");
var2.changeState("State 10");
      System.out.println("All registered observers of subject:");
      Iterator var6 = var1.getObservers().iterator();



      Observer var7;
while(var6.hasNext()) {
         var7 = (Observer)var6.next();
         System.out.println(var7);
      }

      System.out.println("All registered observers of subject2:");
      var6 = var2.getObservers().iterator();

      while(var6.hasNext()) {
         var7 = (Observer)var6.next();
         System.out.println(var7);
      }

   }
}
```

UML Diagram:



Problem: Design a **Weather Monitoring System** where multiple parts of an application, such as a mobile app, website dashboard, and smart devices, need to be automatically notified whenever there is a change in weather conditions like temperature, humidity, or pressure.

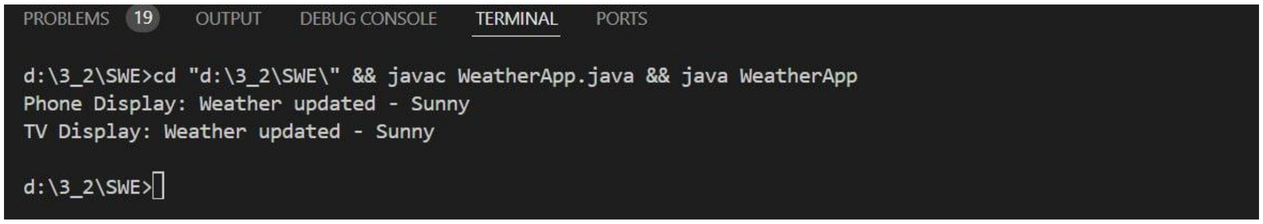
Code:

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


// Observer Interface
interface Observer {
   void update(String weather);
}


// Subject Interface
interface Subject {
   void addObserver(Observer observer);
```

```java
    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();
    }
}

// ConcreteObserver Class
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);
    }
}

// ConcreteObserver Class
```

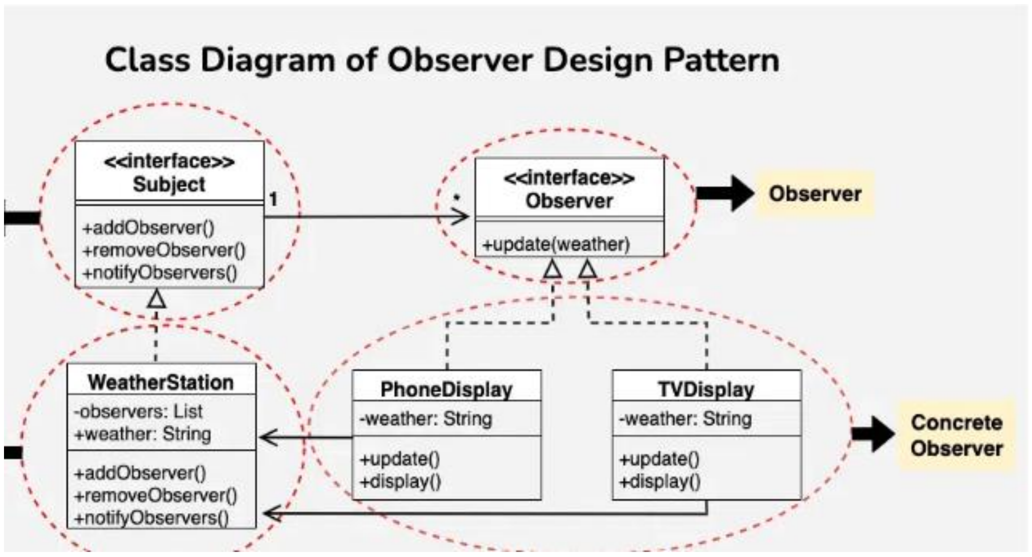
```
class TVDisplay implements Observer {
    private String weather;


    @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();


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

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

Output:



```
d:\3_2\SWE>cd "d:\3_2\SWE\" && javac WeatherApp.java && java WeatherApp
Phone Display: Weather updated - Sunny
TV Display: Weather updated - Sunny

d:\3_2\SWE>
```

UML Diagram:



State behavioral design pattern

Objective: The State Design Pattern is a behavioral design pattern in software engineering that allows an object to change its behavior dynamically based on its internal state. By encapsulating state-specific behaviors into separate classes, this pattern promotes open-closed principle, making it easier to extend and maintain the code. It is particularly useful in scenarios where an object's behavior is highly dependent on its current state, such as finite state machines, UI components, or workflows.

Example: Mobile Power State Management.

Code:

```
// State interface interface MobileState {     void

pressPowerButton(MobileContext context);

}


// Concrete State: Mobile is ON class

MobileOnState implements MobileState {

   @Override                              public       void

pressPowerButton(MobileContext          context)          {

System.out.println("Turning         mobile          OFF...");

context.setState(new MobileOffState());

   }

}


// Concrete State: Mobile is OFF

class MobileOffState implements MobileState {

   @Override                              public       void

pressPowerButton(MobileContext          context)          {
```

```
System.out.println("Turning             mobile             ON...");
context.setState(new MobileOnState());
   }
}

// Context class MobileContext {
private MobileState currentState;

   public MobileContext() {        currentState = new
MobileOffState(); // Default state
   }

   public void setState(MobileState state) {
currentState = state;
   }

   public void pressPowerButton() {
currentState.pressPowerButton(this);
   }
}

// Main class to demonstrate public
class Main {     public static void
main(String[] args) {
       MobileContext mobile = new MobileContext();

       // Pressing the power button multiple times
mobile.pressPowerButton(); // Turning mobile ON...
```

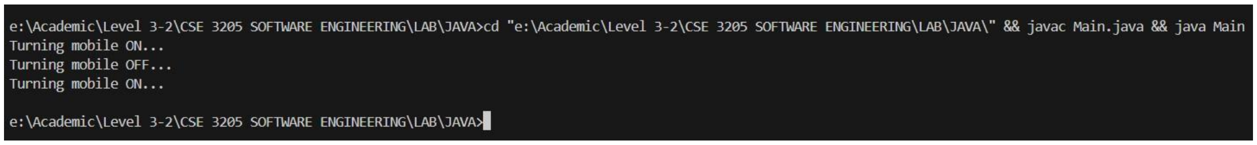
mobile.pressPowerButton(); // Turning mobile OFF...

mobile.pressPowerButton(); // Turning mobile ON...

    }

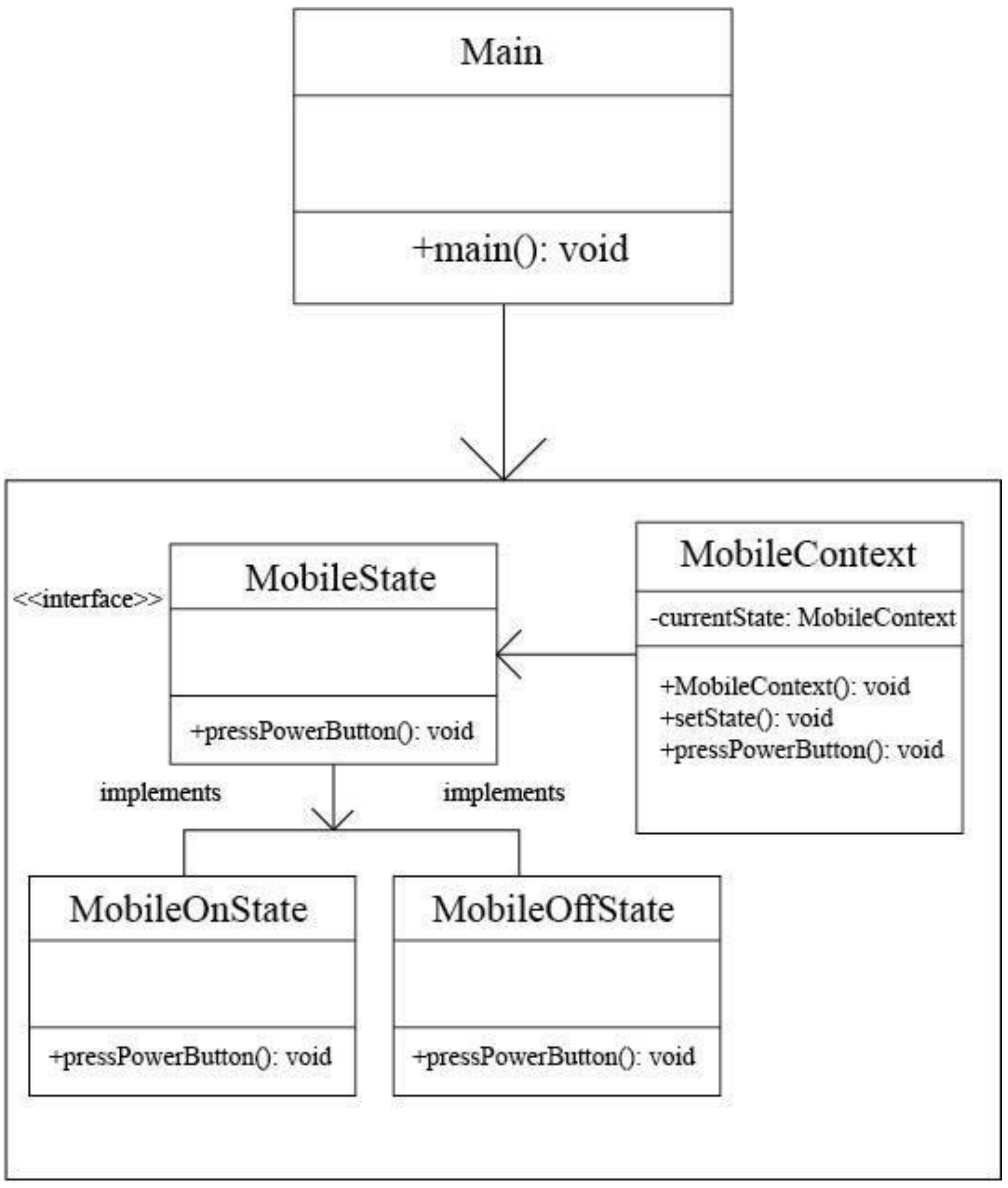
}

Output:



```
e:\Academic\Level 3-2\CSE 3205 SOFTWARE ENGINEERING\LAB\JAVA>cd "e:\Academic\Level 3-2\CSE 3205 SOFTWARE ENGINEERING\LAB\JAVA\" && javac Main.java && java Main
Turning mobile ON...
Turning mobile OFF...
Turning mobile ON...

e:\Academic\Level 3-2\CSE 3205 SOFTWARE ENGINEERING\LAB\JAVA>
```

UML Diagram:


Main
+main(): void
<<interface>>
MobileState
+pressPowerButton(): void
MobileContext
-currentState: MobileContext
+MobileContext(): void
+setState(): void
+pressPowerButton(): void
implements
implements
MobileOnState
+pressPowerButton(): void
MobileOffState
+pressPowerButton(): void