Observer Pattern

514770-1
Fall 2021
9/27/2021
Kyoung Shin Park
Computer Engineering
Dankook University

Observer Pattern

- "Define a **one-to-many dependency** between objects so that when one object **changes state**, all its dependent are **notified** and **updated automatically**."
- Subject represents the core (or independent) abstraction. Observer represents the variable (or dependent or optional or user interface) abstraction.
- This pattern is also referred to as **Publish-Subscribe Pattern** or **Dependent Pattern**.
- Let you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Observer Pattern

- java.util.Observer/java.util.Observable (deprecated since Java9)
- All implementations of java.util.EventListener (practically all over Swing)

Observer Pattern

- Push Notification Service
  - News feed
    - It is cumbersome to visit news sites to check for news everyday
    - Rather, it is convenient to apply for a subscription service, and notify you of new news
    - Mailing service like **Newneek**
  - Hotel wake-up call service
    - You may have difficulty in getting up in the morning without an alarm or a wake-up call.
    - It is desirable to receive a wake-up call or set an alarm at a specific time.

Observer Pattern

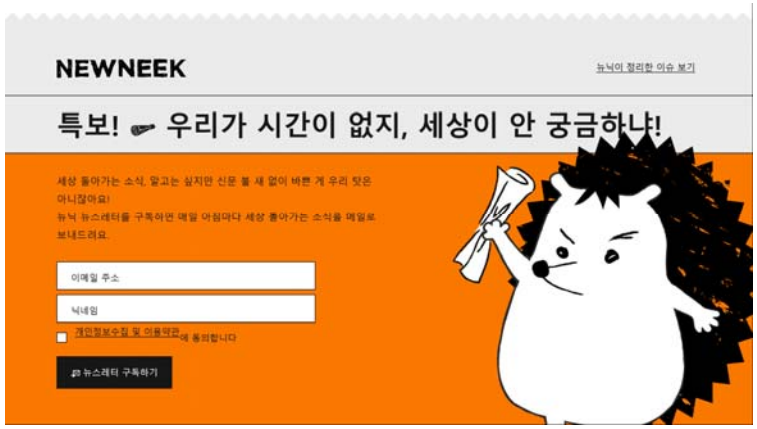
- GUI-based programming
  - UI Event handler
    - Continuous polling (where the button state is checked every frame) is a waste to see if the button is clicked.
    - When a **button click event occurs**, it is efficient to **push** the event to the program and **handle** it.
  - Multiple views of the same data
    - Suppose you have a **map** and a **table view** that contained the **same data**. Whenever the data changes, you want those changes to reflect immediately in the map and the table view without your intervention.
    - This is a recurring problem in GUI-based programming, so you want a design solution that can be re-used whenever the context arises.

Observer Pattern

- The observer pattern procedure
  - Differentiate between the core (or independent) functionality and the optional (or dependent) functionality
  - Model the independent functionality with the **Subject** abstraction
  - Model the dependent functionality with **Observer** hierarchy.
  - Observers **register** themselves with the Subject for changes in the data.
  - The Subject **notifies** to all registered Observers whenever the data changes.
  - The observers then **update**.

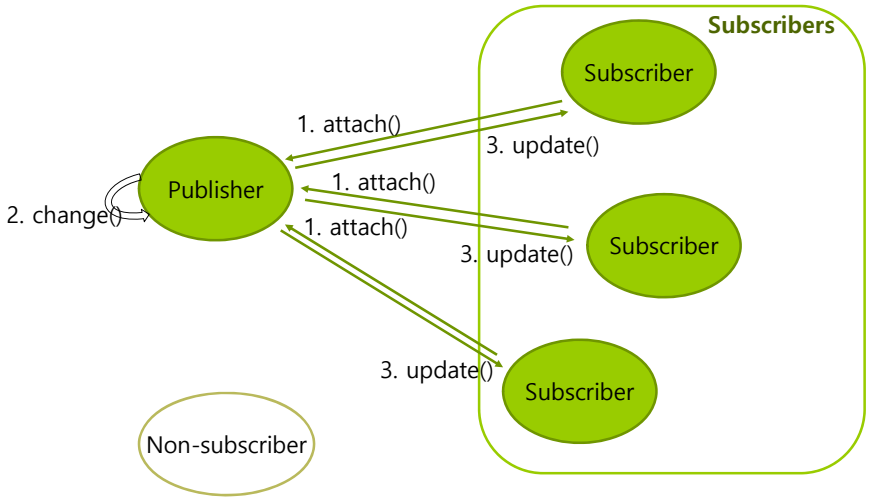
Observer Pattern

- Newsletter Publisher + Subscriber = Observer Pattern
  - Newsletter publisher is Subject (often called Publisher)
  - Subscriber is Observer (often called Subscriber)



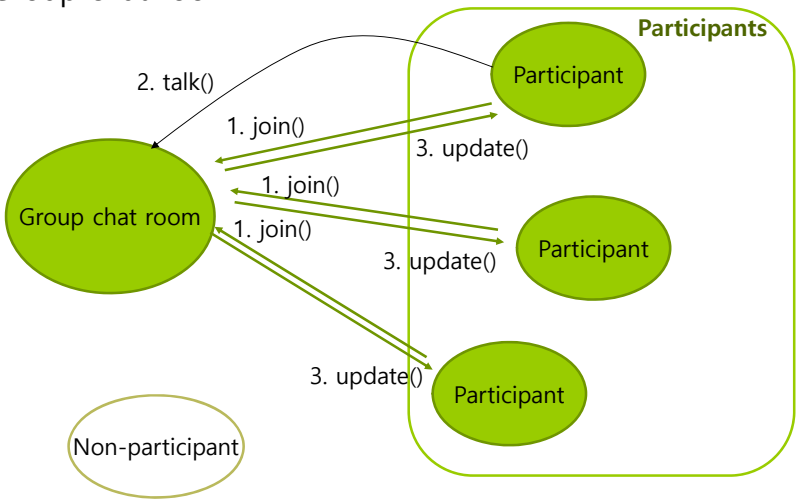
Observer Pattern

- Newsletter



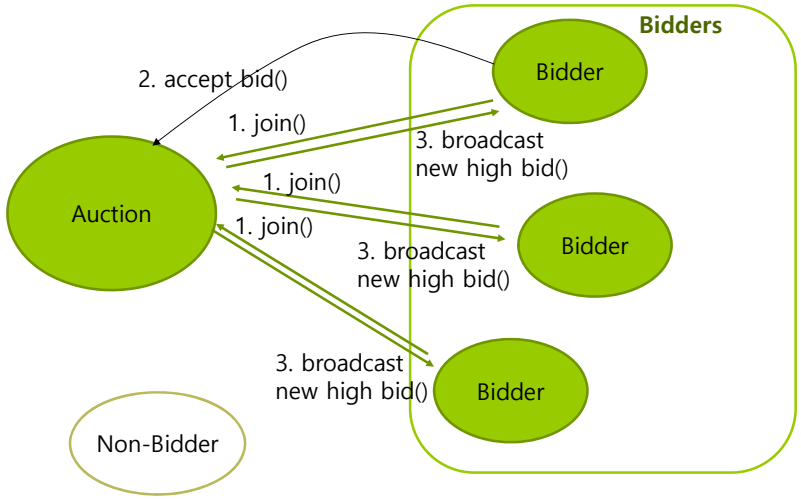
Observer Pattern

- Group chat room



Observer Pattern

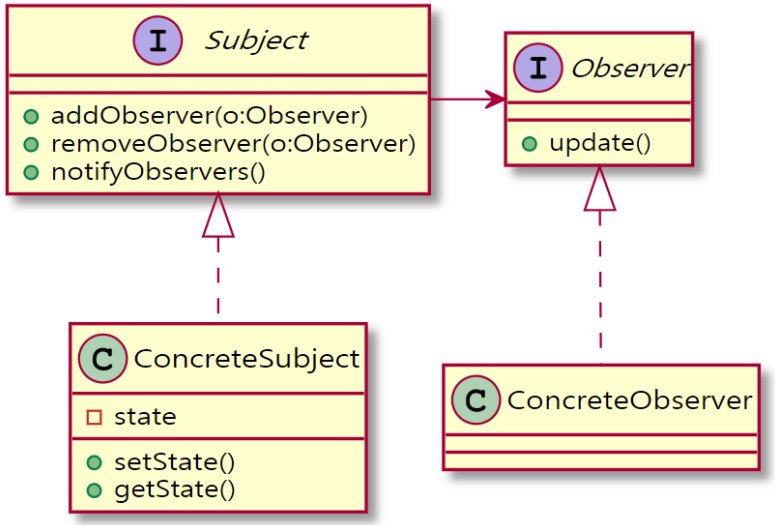
- Auction



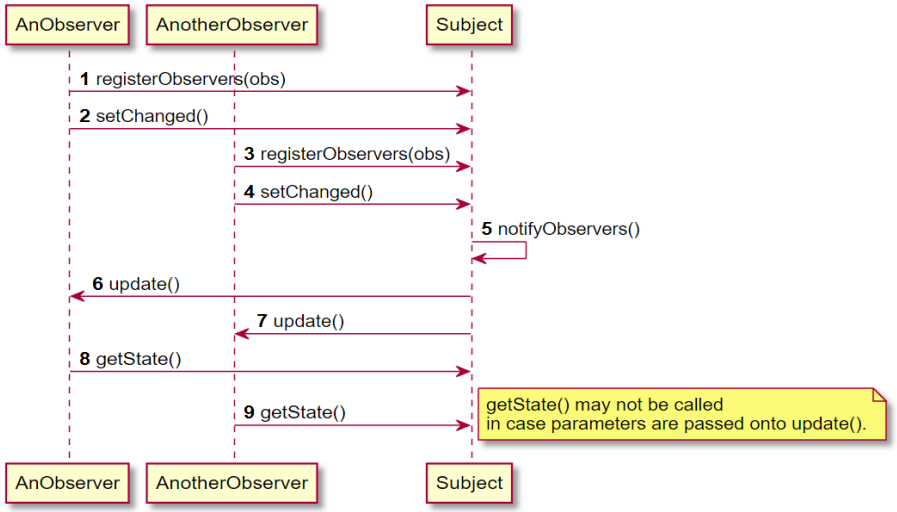
Observer Pattern

	Description
Pattern	Observer
Problem	Need to update information in 1:n dependency relationship
Solution	Register observers, notify them when information changes, so that they can automatically update.
Result	**Loose coupling**, Scalability, Dependency Inversion Principle

Observer Pattern



Observer Pattern

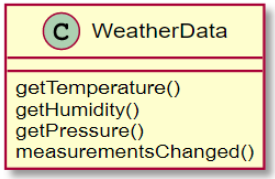


Weather Monitoring (HFDP Ch. 2)

- Weather Monitoring (HFDP Ch. 2)
  - Construction of Weather-O-Rama Inc's next generation Internet-based Weather Monitoring Station
    - WeatherStation : equipment that collects meteorological information
    - WeatherData object : object that tracks data coming from a weather station
    - 3 Different Displays:
      - Current conditions (temperature, humidity, and pressure)
      - Weather statistics (average, min, max weather data)
      - Simple Forecasts (weather forecast)

Weather Monitoring (HFDP Ch. 2)

- The information provider continuously measures and collects temperature, humidity, and pressure.
- WeatherData Class



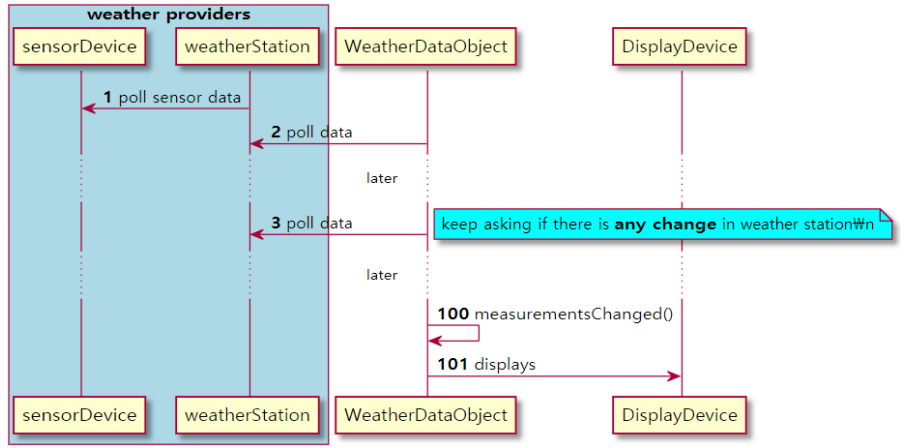
getTemperature(), getHumidity(), getPressure() methods return the **most recent** weather measurements

- The **measurementsChanged()** method is called any time **new** weather measurement **data** is available.
- We need to implement *3 display elements* that use the weather data: *a current condition display, a statistics display, and a forecast display.*

Weather Monitoring (HFDP Ch. 2)

```
// WeatherData.java
public void measurementsChanged() {
    // grab the most recent measurements
    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();
    // update the displays
    currentConditionsDisplay.update(temp,
                                    humidity, pressure);
    statisticsDisplay.update(temp, humidity,
                                     pressure);
    forecastDisplay.update(temp, humidity,
                                    pressure);
}
```

Weather Monitoring (HFDP Ch. 2)

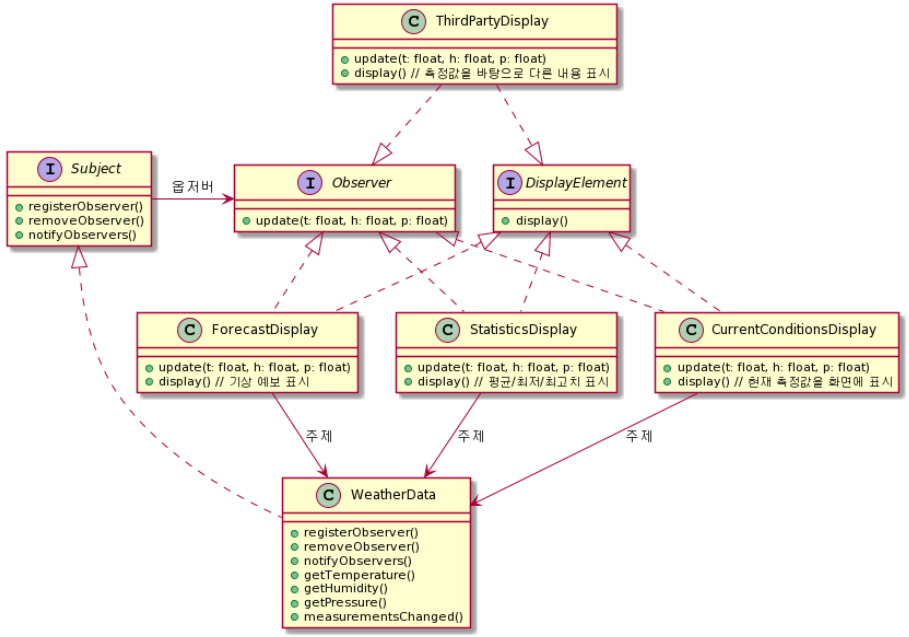


Weather Monitoring (HFDP Ch. 2)

- Problem
  - The **measurementsChanged() method uses concrete objects** such as `currentConditionsDisplay`, `statisticsDisplay`, `forecastDisplay`, and hence we need to modify this method in order to add new display or remove an existing display.
  - **Area of change**, we need to **encapsulate** this.

Loose Coupling

- When two objects are **loosely coupled**, they can interact but have very little knowledge of each other.
- The observer pattern provides an object design where subjects and observers are loosely coupled.
  - The only thing Subject knows about Observer is that it implements a specific interface (you don't need to know what the actual class that implements Observer is)
  - We can add or remove new observers at any time (even while execution)
  - We never need to modify the subject to add new types of observers.
  - We can **reuse subjects and observers independently** of each other.



Weather Monitoring (HFDP Ch. 2)

- Using non-generic ArrayList

```
// Subject.java
public interface Subject {
      void registerObserver(Observer o);
      void removeObserver(Observer o);
      void notifyObservers();
}
```

```
// Observer.java
public interface Observer {
      void update(float temperature,
                  float humidity,
                  float pressure);
}
```

```
// WeatherData.java
import java.util.*;

public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;
    public WeatherData() {
        observers = new ArrayList();
    }
    public void registerObserver(Observer o) {
        observers.add(o);
    }
    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }
```

Weather Monitoring (HFDP Ch. 2)

```
    public void notifyObservers() {
        for (Observer observer : this.observers) {
            observer.update(temperature, humidity,
pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature,
                      float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
```

Weather Monitoring (HFDP Ch. 2)

```
    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

Weather Monitoring (HFDP Ch. 2)

```
// DisplayElement.java
public interface DisplayElement {
      void display();
}
```

```
// CurrentConditionsDisplay.java
public class CurrentConditionsDisplay implements
                                Observer, DisplayElement {
      private float temperature;
      private float humidity;
      private Subject weatherData;
      public CurrentConditionsDisplay(
                                    Subject weatherData) {
            this.weatherData = weatherData;
            weatherData.registerObserver(this);
      }

      public void update(float temperature,
                  float humidity,
                  float pressure) {
            this.temperature = temperature;
            this.humidity = humidity;
            display();
      }

      public void display() {
            System.out.println("Current conditions: "
           + temperature + "F degrees and " + humidity
           + "% humidity");
      }
}
```

Weather Monitoring (HFDP Ch. 2)

```
// WeatherStation.java
import java.util.*;

public class WeatherStation {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();
        CurrentConditionsDisplay currentDisplay
            = new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay
                  = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay
                  = new ForecastDisplay(weatherData);
        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

Weather Monitoring (HFDP Ch. 2)

- Using generics ArrayList

```
package headfirst.observer.weather;

import java.util.*;

public class WeatherData implements Subject {
    private ArrayList<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<Observer>();
    }
    public void registerObserver(Observer o) {
        observers.add(o);
    }
```

Weather Monitoring (HFDP Ch. 2)

```
    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) observers.remove(i);
    }
    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = observers.get(i);
            observer.update(temperature, humidity,
pressure);
        }
        /*for (Observer observer : this.observers) {
      observer.update(temperature, humidity, pressure);
        }*/
        /* for (Iterator<Observer> it =
         this.observers.iterator(); it.hasNext();) {
            Observer observer = it.next();
      observer.update(temperature, humidity, pressure);
        }*/
    }
```

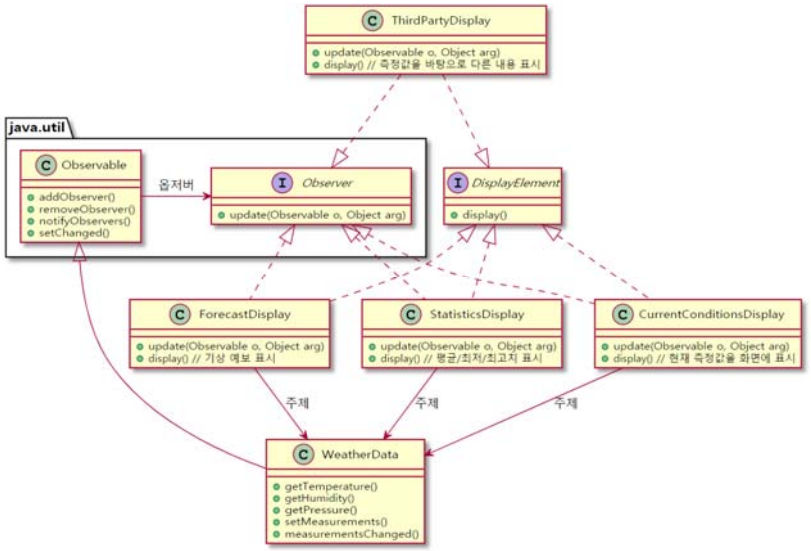
Weather Monitoring (HFDP Ch. 2)

```
    public void measurementsChanged() {
        notifyObservers();
    }
    public void setMeasurements(float temperature,
                        float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
    public float getTemperature() {
        return temperature;
    }
    public float getHumidity() {
        return humidity;
    }
    public float getPressure() {
        return pressure;
    }
}
```

Weather Monitoring (HFDP Ch. 2) – Java

- Use java.util.Observable/java.util.Observer
  - Warning: Deprecated (since Java 9)
  - How an object becomes an Observer
    - Call the Observable's **addObserver()** method after implementing the Observer interface
  - How to push notifications in Observables
    - First, call the **setChanged()** method to notify that the object's state has changed.
    - Secondly, call the **notifyObservers()** or **notifyObservers(Object arg)** method to notify the observers
  - How the Observer gets informed
    - Implement the **update(Observable o, Object arg)** method

Weather Monitoring (HFDP Ch. 2) – Java



```
import java.util.*;

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() { }
    public void measurementsChanged() {
        setChanged(); // state changed
        notifyObservers(this);
    }
    public void setMeasurements(float temperature,
                    float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
```

Weather Monitoring (HFDP Ch. 2) – Java

```
    public float getTemperature() {
        return temperature;
    }
    public float getHumidity() {
        return humidity;
    }
    public float getPressure() {
        return pressure;
    }
}
```

Weather Monitoring (HFDP Ch. 2) – Java

```
// CurrentConditionsDisplay.java
public class CurrentConditionsDisplay implements
                            Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Observable weatherData;

    public CurrentConditionsDisplay(
                            Observable weatherData) {
        this.weatherData = weatherData;
        this.weatherData.addObserver(this);
    }
```

Weather Monitoring (HFDP Ch. 2) – Java

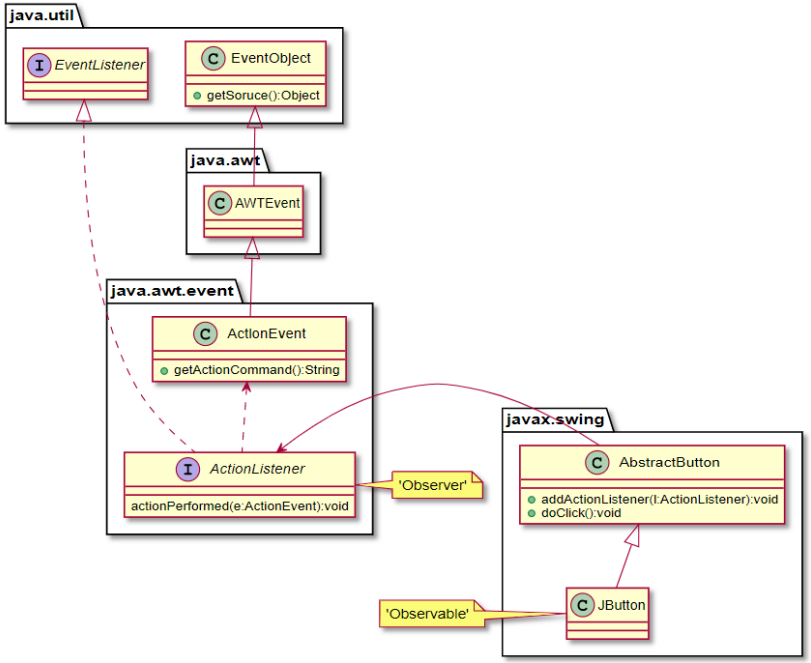
```
    public void update(Observable o, Object arg) {
        if (arg instanceof WeatherData) {
            WeatherData wd = (WeatherData) arg;
            this.temperature = wd.getTemperature();
            this.humidity = wd.getHumidity();
            display();
        }
    }

    public void display() {
        System.out.println("Current conditions: "
            + temperature + "F degrees and " + humidity
            + "% humidity");
    }
}
```

Weather Monitoring (HFDP Ch. 2) – Java

- Problem with java.util.Observable/java.util.Observer
  - java.util.Observable is a class, not an interface
  - That is, classes that should be inherited from other classes cannot be inherited from Observables.
  - The setChanged() method is protected. It is not a problem because it can be used only if it is inherited from Observable anyway, but it violates the design principle of using composition rather than inheritance.
  - Deprecated since Java 9 (not recommend to use it)

Java Swing ActionListener

- Swing JButton is Observable (Subject)
- addActionListener() method in AbstractButton, which is the parent class of Jbutton.
  - Swing's event listener is an Observer
- When a JButton event occurs, it calls actionPerformed() method of listener registered in JButton.



Java Swing ActionListener

```
public class SwingObserverFrame extends JFrame {
  public static void main(String[] args) {
    new SwingObserverFrame();
  }

  public SwingObserverFrame() {
    JButton button = new JButton("정말 해도 될까?");
    button.addActionListener(new AngelListener());
    button.addActionListener(new DevilListener());
    button.addActionListener(event ->
        System.out.println("할지 말지 고민이네~"));
    this.getContentPane().add(
                        BorderLayout.CENTER, button);
    this.setSize(200, 200);
    this.setVisible(true);
  }
```

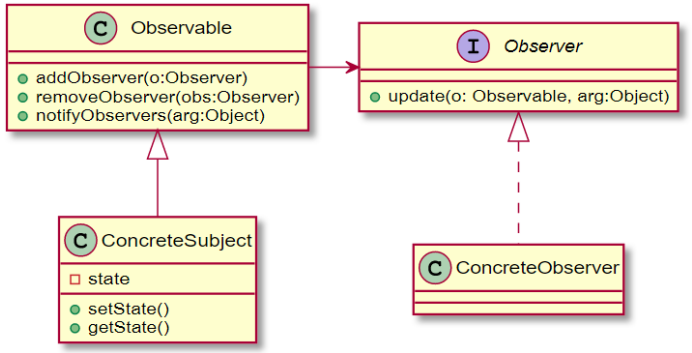
Java Swing ActionListener

```
class AngelListener implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    System.out.println("안돼. 분명 나중에 후회할거야");
  }
}

class DevilListener implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    System.out.println("당연하지. 그냥 저질러 버려!");
  }
}
```

Observer Pattern

- Design
  - **Interface Segregation Principle (ISP)**
  - Concrete class inheritance
  - Client inherits after using abstract class rather than concrete class
  - **Subject**
    - Notify when status changes (**notify**)
    - Pre-register observers to be notified (**register**)
    - Also known as **Observable** or **Publisher**
  - **Observer**
    - Update new information when the status of the information provider changes (**update**)
    - Also known as **Subscriber**

Observer Pattern



Observer Pattern