COMP2511
Observer Pattern

Prepared by
Dr. Ashesh Mahidadia
Observer Pattern

These lecture notes use material from the wikipedia page at: https://en.wikipedia.org/wiki/Observer_pattern

and

the reference book “Head First Design Patterns”.
Observer Pattern

• The **Observer Pattern** is used to implement distributed **event handling** systems, in "event driven" programming.

• In the observer pattern
  
  • an object, called the **subject** (or **observable** or **publisher** ), maintains a list of its dependents, called **observers** (or **subscribers**), and
  
  • **notifies** the **observers automatically** of any state **changes** in the **subject**, usually by calling one of their methods.

• Many programming languages support the observer pattern, Graphical User Interface libraries use the observer pattern extensively.
Observer Pattern

• The Observer Pattern defines a **one-to-many** dependency between objects so that when one object (*subject*) changes state, all of its dependents (*observers*) are notified and updated automatically.

• The aim should be to,
  
  • define a one-to-many dependency between objects **without** making the objects **tightly coupled**.
  
  • **automatically** notify/update an **open-ended** number of *observers* (dependent objects) when the *subject* changes state
  
  • be able to **dynamically** add and remove *observers*
Observer Pattern: Possible Solution

• Define *Subject* and *Observer interfaces*, such that when a subject changes state, all registered observers are notified and updated automatically.

• The responsibility of,

  • a *subject* is to maintain a list of observers and to notify them of state changes by calling their `update()` operation.

  • *observers* is to register (and unregister) themselves on a subject (to get notified of state changes) and to update their state when they are notified.

• This makes subject and observers **loosely coupled**.

• Observers can be **added** and **removed** independently **at run-time**.

• This notification-registration interaction is also known as **publish-subscribe**.
Java Observer and Observable : Deprecated

The following java library classes have been deprecated in Java 9 because the model implemented was quite limited.

- `java.util.Observer` and
- `java.util.Observable`

Limitations

- *Observable* is a class, not an interface!
- Observable protects crucial methods, the `setChanged()` method is protected.
- we can’t call `setChanged()` unless we subclass *Observable*! Inheritance is must, bad design 😞
- we can’t add on the *Observable* behavior to an existing class that already extends another superclass.
- there isn’t an *Observable* interface, for a proper custom implementation
Multiple Observers and Subjects

Observers / Subscribers / Listeners

Observables / Subjects / Publishers
Observer Pattern: Possible Solution

```
ArrayList<Observer> listObservers = new ArrayList<Observer>();

public void notifyObservers() {
    for (Observer obs : listObservers) {
        obs.update(this);
    }
}
```

Read the example code discussed/developed in the lectures, and also provided for this week.
Passing data: Push or Pull

The *Subject* needs to pass (change) data while notifying a change to an *Observer*. Two possible options,

**Push data**
- *Subject* passes the changed data to its observers, for example:
  \[
  \text{update}(\text{data1}, \text{data2}, \ldots)
  \]
- All *observers* must implement the above update method.

**Pull data**
- *Subject* passes reference to itself to its observers, and the observers need to get (pull) the required data from the subject, for example:
  \[
  \text{update}(\text{this})
  \]
- Subject needs to provide the required access methods for its observers.
  For example,  
  ```java
  public double getTemperature()
  ```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}

public class Thermometer implements Subject {
    ArrayList<Observer> listObservers = new ArrayList<Observer>();
    double temperatureC = 0.0;

    @Override
    public void registerObserver(Observer o) {
        if (!listObservers.contains(o)) { listObservers.add(o); }
    }

    @Override
    public void removeObserver(Observer o) {
        listObservers.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer obs : listObservers) {
            obs.update(this);
        }
    }

    public double getTemperatureC() {
        return temperatureC;
    }

    public void setTemperatureC(double temperatureC) {
        this.temperatureC = temperatureC;
        notifyObservers();
    }
}

Read the example code discussed/developed in the lectures, and also provided for this week

Notify Observers after every update
Read the example code discussed/developed in the lectures, and also provided for this week.

```java
public interface Observer {
    public void update(Subject obj);
}

public class DisplayUSA implements Observer {
    Subject subject;
    double temperatureC = 0.0;
    double humidity = 0.0;

    @Override
    public void update(Subject obj) {
        if (obj instanceof Thermometer) {
            update((Thermometer) obj);
        } else if (obj instanceof Hygrometer) {
            update((Hygrometer) obj);
        }
    }

    public void update(Thermometer obj) {
        this.temperatureC = obj.getTemperatureC();
        display();
    }

    public void update(Hygrometer obj) {
        this.humidity = obj.getHumidity();
        display();
    }

    public void display() {
        System.out.printf("From DisplayUSA: Temperature is %.2f F, " + "Humidity is %.2f\n", convertToF(), humidity);
    }

    public double convertToF() {
        return (temperatureC * (9.0/5.0) + 32);
    }
}
```
public class Test1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Thermometer thermo = new Thermometer();
        Observer usaDisplay = new DisplayUSA();
        thermo.registerObserver(usaDisplay);

        Observer ausDisplay = new DisplayAustralia();
        thermo.registerObserver(ausDisplay);

        System.out.println("\n----------------- thermo.setTemperatureC(30) ----------------- ");
        thermo.setTemperatureC(30);
        System.out.println("\n----------------- thermo.setTemperatureC(12) ----------------- ");
        thermo.setTemperatureC(12);

        Hygrometer hyg = new Hygrometer();
        hyg.registerObserver(usaDisplay);

        System.out.println("\n----------------- hyg.setHumidity(77) ----------------- ");
        hyg.setHumidity(77);
        System.out.println("\n----------------- hyg.setHumidity(96) ----------------- ");
        hyg.setHumidity(96);
        System.out.println("\n----------------- thermo.setTemperatureC(35) ----------------- ");
        thermo.setTemperatureC(35);

        thermo.removeObserver(usaDisplay);
        System.out.println("\n----------------- thermo.removeObserver(usaDisplay) ----------------- ");

        System.out.println("\n----------------- thermo.setTemperatureC(41) ----------------- ");
        thermo.setTemperatureC(41);
        System.out.println("\n----------------- thermo.setTemperatureC(41) ----------------- ");
    }
}
Demos ...

- Live Demos ...

- Make sure you **properly understand the demo example code** available for this week.
Observer Pattern: Example

The above image is from https://www.oodesign.com/observer-pattern.html
Observer Pattern: UI Example
Summary

Advantages:

• Avoids tight coupling between Subject and its Observers.
• This allows the Subject and its Observers to be at different levels of abstractions in a system.
• Loosely coupled objects are easier to maintain and reuse.
• Allows dynamic registration and deregistration.

Be careful:

• A change in the subject may result in a chain of updates to its observers and in turn their dependent objects – resulting in a complex update behaviour.
• Need to properly manage such dependencies.
The Observer Pattern defines a one-to-many relationship between objects.

Subjects, or as we also know them, Observables, update Observers using a common interface.

Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement the Observer interface.

You can push or pull data from the Observable when using the pattern (pull is considered more “correct”).

Don’t depend on a specific order of notification for your Observers.

Java has several implementations of the Observer Pattern, including the general purpose java.util.Observable.

Watch out for issues with the java.util.Observable implementation.

Don’t be afraid to create your own Observable implementation if needed.

Swing makes heavy use of the Observer Pattern, as do many GUI frameworks.

You’ll also find the pattern in many other places, including JavaBeans and RMI.

From the reference book: “Head First Design Pattern”