2. THE OBSERVER PATTERN

Keeping your objects in the know

1) THE WEATHER MONITORING APPLICATION OVERVIEW :-

① Let's take a look at two aspects :

→ what Weather-O-Rama is giving us

→ what we are going to need to build or extend.

② The system has three components:

① the weather station

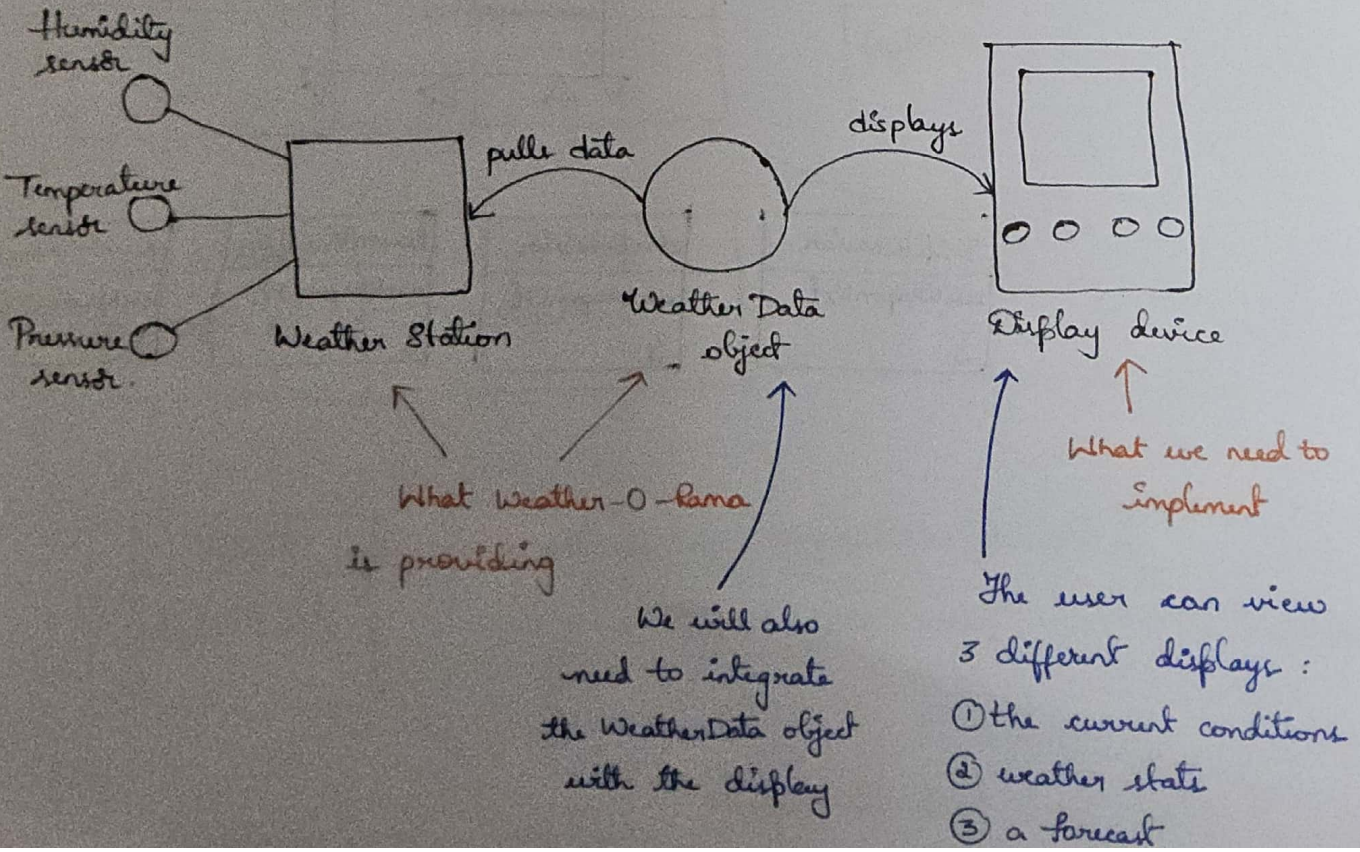
- the physical device that acquires the actual weather data.

② the WeatherData object

- that tracks the data coming from the Weather station and updates the displays.

③ the display

- that shows users the current weather conditions.



Scanned with CamScanner

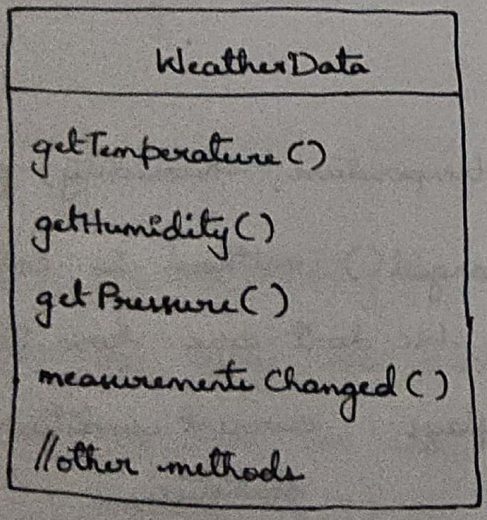
WeatherData object already knows how to get updated data from Weather station.

We'll need to adapt the WeatherData object so that it knows how to update the display.

Our job: create an app that uses the Weather Data object to update three displays: current conditions, weather stats, and a forecast.

2) UNPACKING THE WEATHERDATA CLASS:-

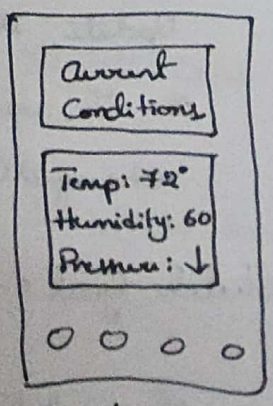
WeatherData class already exists and looks like follows:




```
/*
 * This method gets called
 * whenever the weather
 * measurements have been
 * updated
 *
*/
public void measurementsChanged(){
    //code
}
```

WeatherData.java

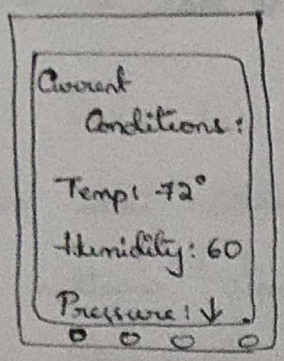
As every time measurements change, & measurementsChanged() method gets called, we need to alter this method so that it updates three displays: current conditions, weather stats, and forecast.



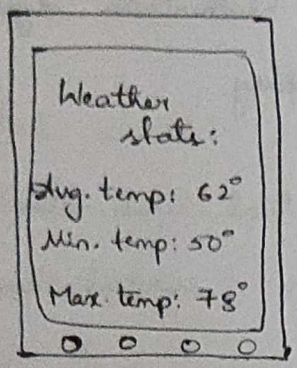
Display device

Scanned with CamScanner

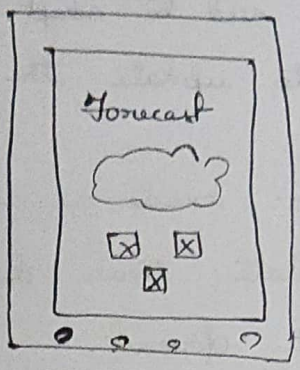
3) OUR GOAL :-



Display One



Display Two



Display Three.

We know we have to implement a display and WeatherData object have to be used to update that display each time it has new values, or, in other words, each time the measurements Changed() method is called.

What we are trying to achieve are :

* WeatherData has getters : temperature, humidity, pressure.
* WeatherData: measurements Changed() method is called when weather measurements change. We don't care how it gets called.
* Need to impliment 3 displays : current conditions,
  statistics,
  forecast

  Update them whenever measurements change.
* To update them, we'll add code to the measurements Changed() method.

Stretch Goal :-

* Expandability :- Allow more displays to add/remove in the future.

Scanned with CamScanner

4) TAKING A FIRST, MISGUIDED IMPLEMENTATION OF THE WEATHER STATION

```
public class WeatherData {

    public void measurementsChanged() {
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();
```

get recent measurements by calling getters

update each display by calling its update method

```
        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }
}
```

5) WHAT'S WRONG WITH OUR IMPLEMENTATION ANYWAY?

1. We are coding to concrete implementations, not interfaces.
2. For every new display we'll need to alter this code.
3. We have no way to add (or remove) display elements at runtime.
4. We haven't encapsulated the part that changes.
5. ~~We are violating encapsulation of the WeatherData class.~~

Let's look at observer pattern and see how it helps here.

Scanned with CamScanner

6) MEET THE OBSERVER PATTERN:-

How newspaper & magazine subscriptions work:-

1) A newspaper publisher goes into business and begins publishing newspapers.

2) You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.

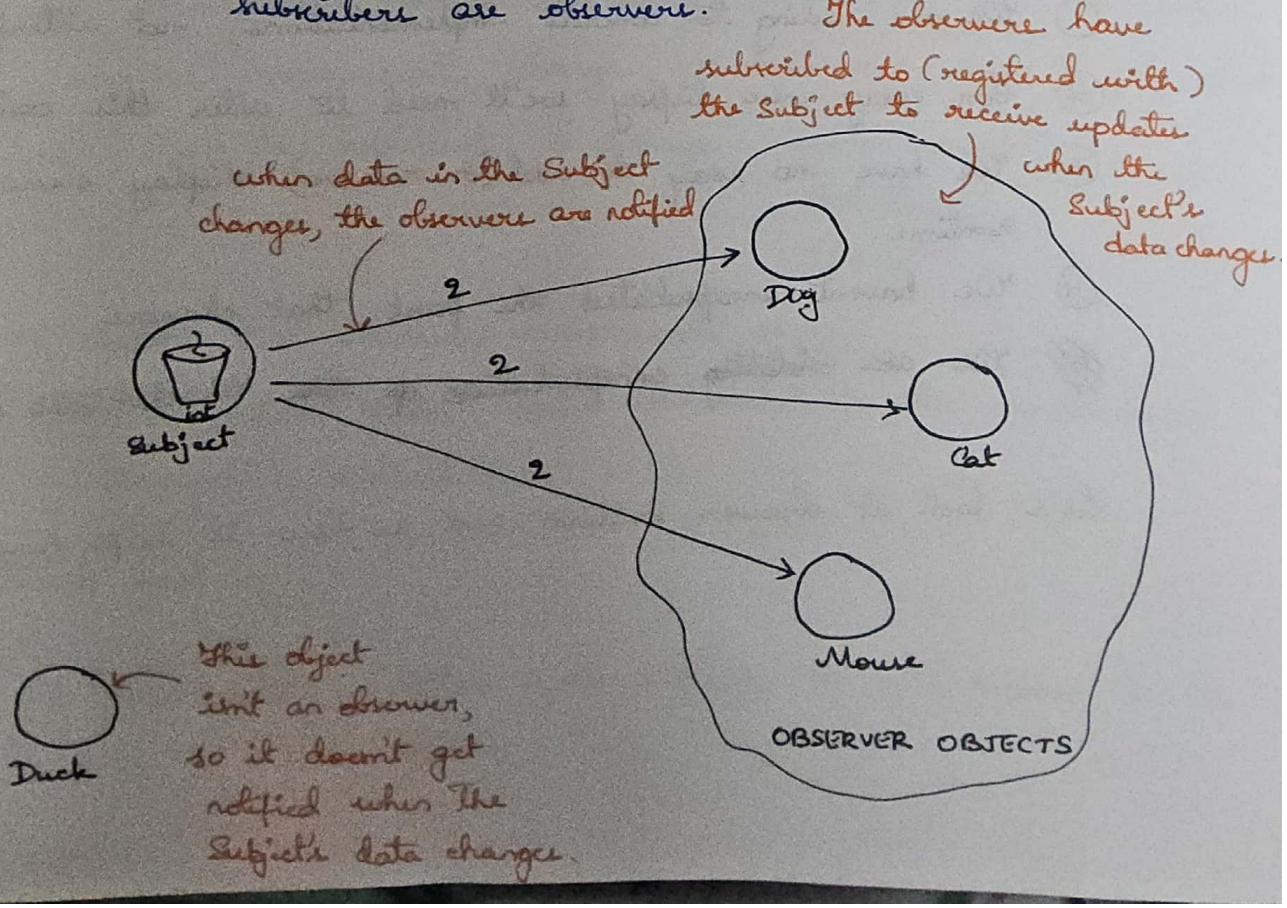
3) You unsubscribe when you don't want papers anymore, and they stop being delivered.

4) While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.

7) PUBLISHERS + SUBSCRIBERS = OBSERVER PATTERN:-

We call publisher a subject

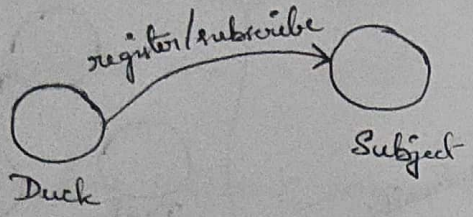
subscribers are observers.

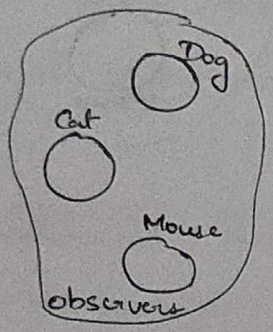


Scanned with CamScanner

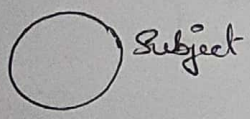
8) A DAY IN THE LIFE OF THE OBSERVER PATTERN

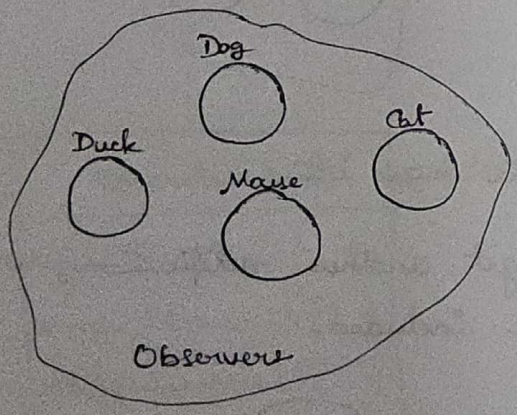
A Duck object comes along and tells the Subject that he wants to become an observer.



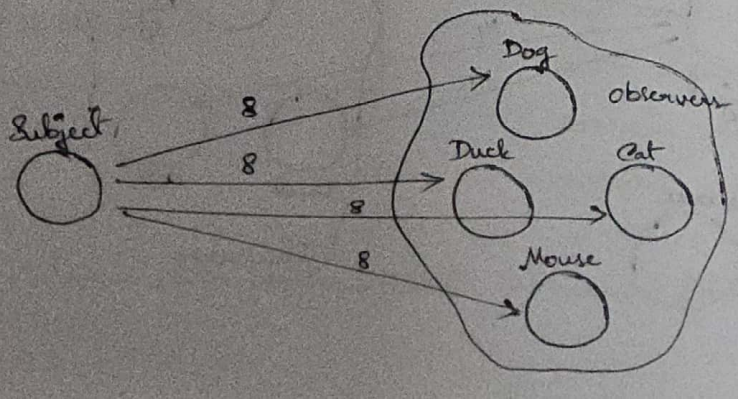


The Duck object is now an official observer.



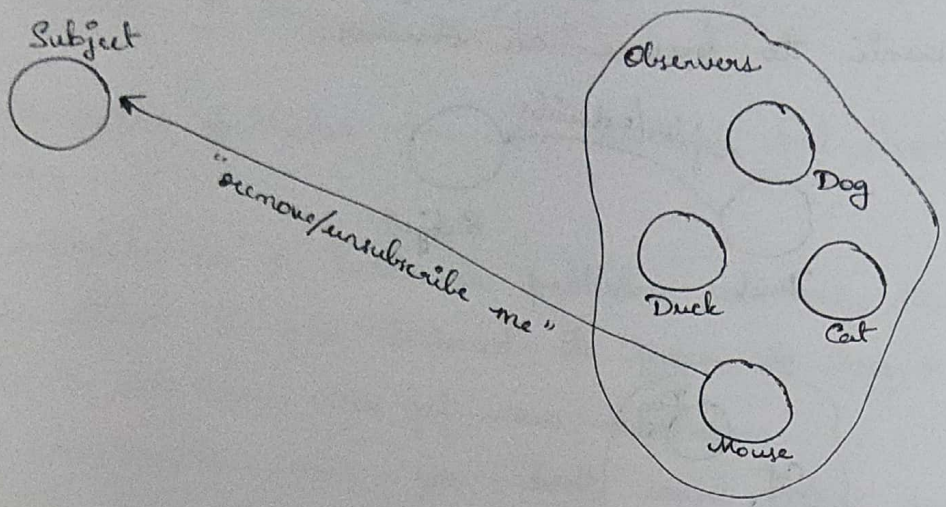


The Subject gets a new data value!

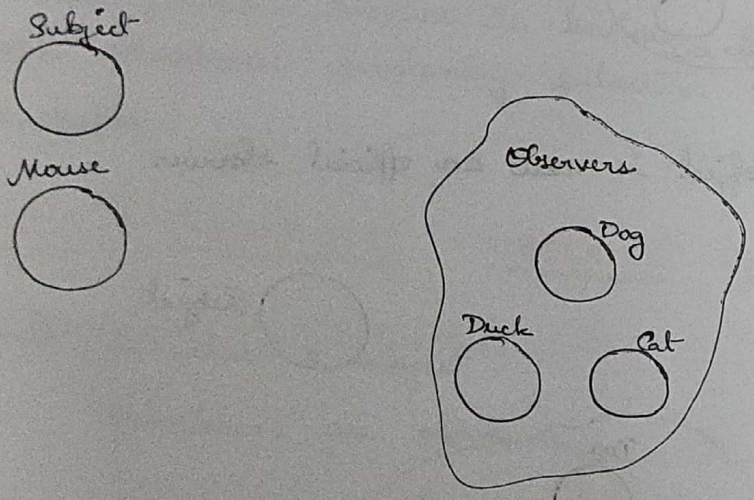


Scanned with CamScanner

The Mouse object asks to be removed as an observer.

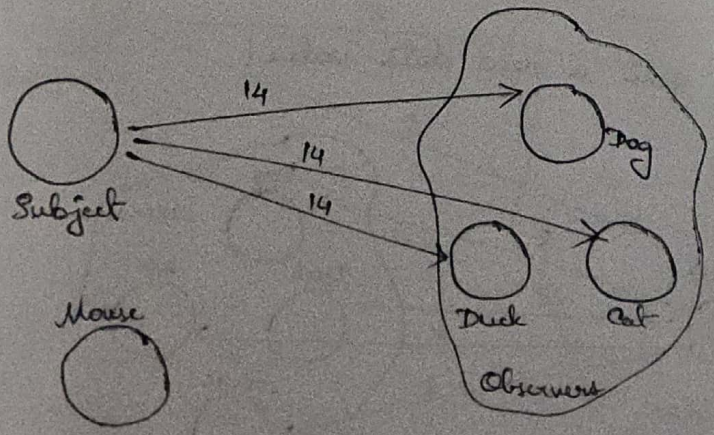


Mouse is outta here!



The Subject has another new int.

All the observers get another notification, except for the Mouse who is no longer included.



Scanned with CamScanner

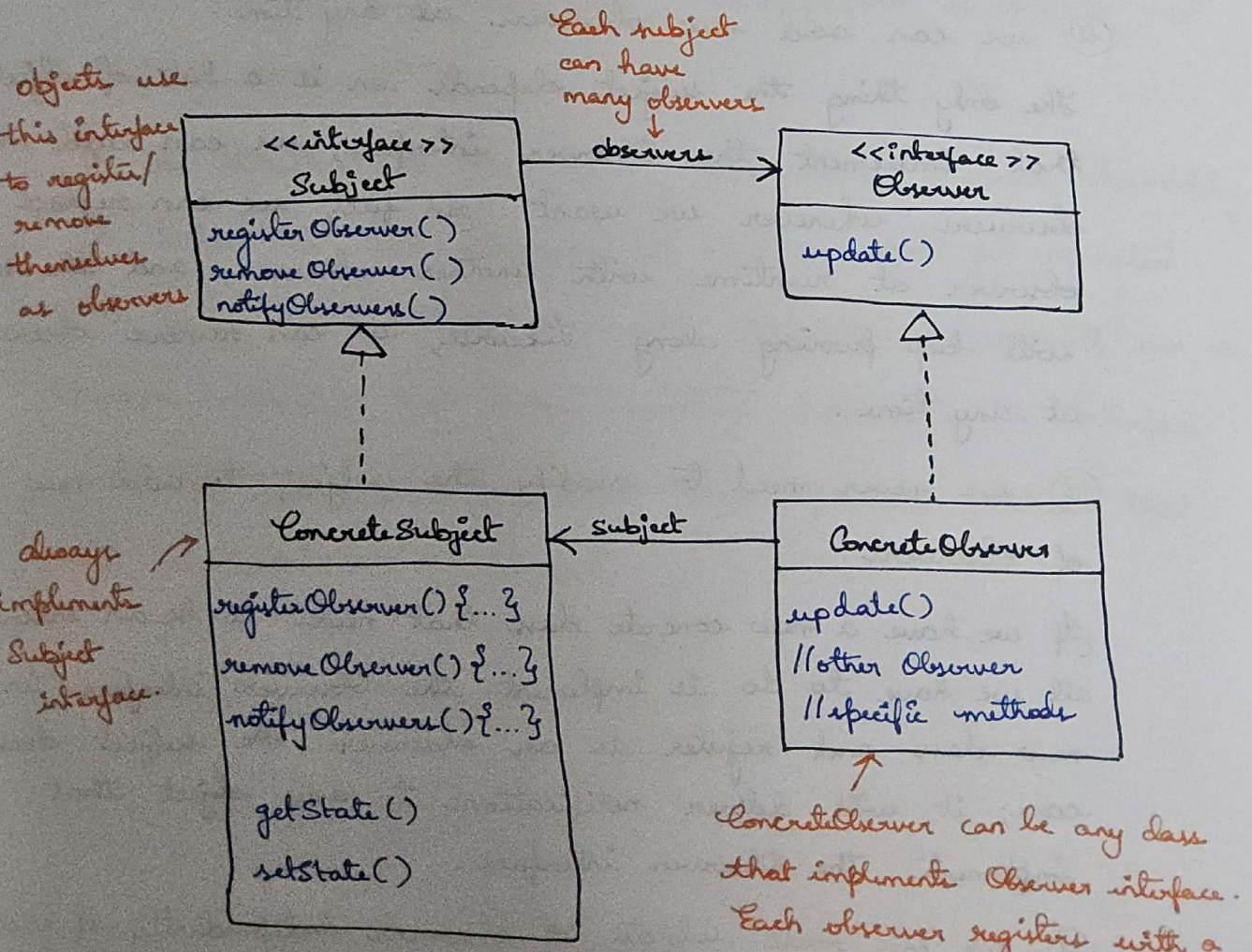
9) THE OBSERVER PATTERN DEFINED

Definition :-

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

There are few different ways to implement the Observer Pattern, but most revolve around a class design that includes Subject and Observer interfaces.

10) THE OBSERVER PATTERN : THE CLASS DIAGRAM



Publish-Subscribe pattern and Observer pattern are not same.

Scanned with CamScanner

10) THE POWER OF LOOSE COUPLING :-

* When two objects are loosely coupled, they can interact, but they typically have very little knowledge of each other.

* Loosely coupled designs give us a lot of flexibility.

* Observer Pattern is a great example of loose coupling.

All the ways the pattern achieves loose coupling :-

(1) the only thing the subject knows about an observer is that it implements a certain interface (the Observer interface). It doesn't need to know the concrete class of the observer.

(2) we can add new observers at any time.
The only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. In fact, we can replace any observer at runtime with another observer and the subject will keep proving along. Likewise, we can remove observers at any time.

(3) we never need to modify the subject to add new types of observers.
If we have a new concrete class that needs to be an observer, all we have to do is implement the Observer interface in this new class and register as an observer. The subject doesn't care; it will deliver notifications to any object that implements the Observer interface.

(4) We can reuse subjects or observers independently of each other since they aren't tightly coupled, we can reuse subject or observer.

(5) Changes to either the subject or an observer will not affect the other.

Scanned with CamScanner

Design Principle: Strive for loosely coupled designs between objects that interact

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

II) DESIGNING THE WEATHER STATION :-

WeatherData – Subject

All Displays – Observers.

* For one-to-many relationship, WeatherData is one and all displays are many

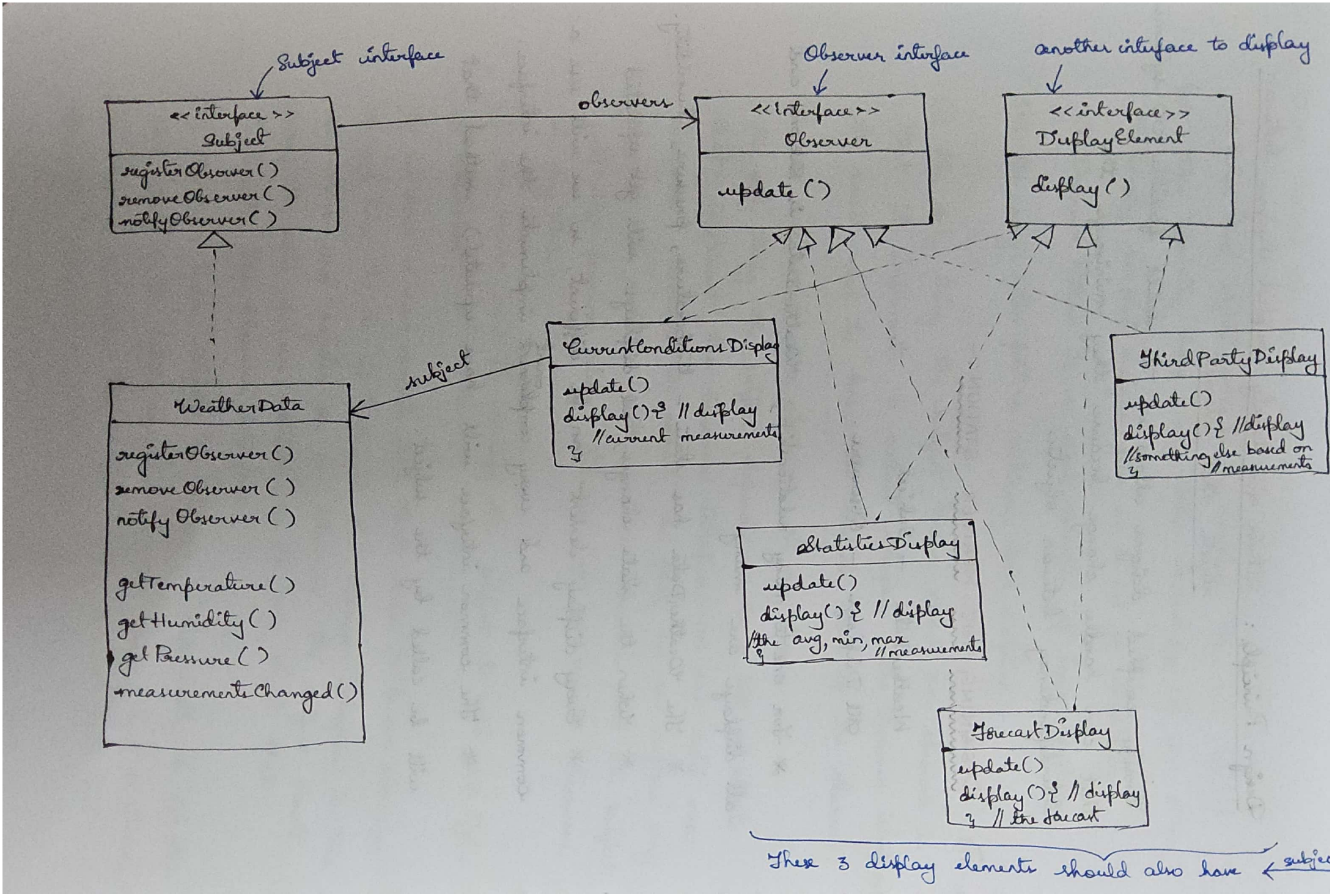
* The WeatherData has state – temperature, pressure, humidity.

* When the state changes, all displays will get updated

* Every display element can be different, so we will use a common interface and every component implements the interface.

* The common interface will have update() method that will be called by the subject.

Scanned with CamScanner



Scanned with CamScanner

12) IMPLEMENTING THE WEATHER STATION

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

← the observer to be registered / removed.

← to notify all observers when the Subject's state has changed.

```
public interface Observer {
    public void update(float temp, float humidity,
                       float pressure);
}
```

← to be implemented by all observers & all observers implement update method.

```
public interface DisplayElement {
    public void display();
}
```

← when the display element needs to be displayed, this method is called.

13) IMPLEMENTING THE SUBJECT INTERFACE IN WEATHER DATA

Previous implementation :-

```
public class WeatherData {

    //Getters

    public void measurementsChanged() {
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();
        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(..., ..., ...);
    }
}
```

Scanned with CamScanner

New implementation :-

```
public class WeatherData implements Subject {
    private List<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);
    }
    public void removeObserver(Observer o) {
        observers.remove(o);
    }
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }
    public void measurementsChanged() {
        notifyObservers();
    }
    public void setMeasurements(float temp, float humidity,
                                float pressure) {
        this.temperature = temp;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
}
```

WeatherData now implements Subject interface

when observer registers, we add it to the end of the observers list.

when observer wants to unregister, we remove it off the list.

we notify observers by calling update() method

we notify observers when we get updated measurements.

we are going to use this method to test our display elements

Scanned with CamScanner

14) NOW, LET'S BUILD THOSE DISPLAY ELEMENTS :-

```
public class CurrentConditionsDisplay implements Observer,
                                                 DisplayElement {

    private float temperature;
    private float humidity;
    private WeatherData weatherData;

    public CurrentConditionsDisplay (WeatherData 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");
    }

}
```

object necessary to register observer.

when update() is called, we save temp & humidity, then call display to display data.

* Is update() best place to call display()?

There is much better way to design the way the data gets displayed (MVC pattern)

* If not using it after constructor, why store WeatherData object in the variable?

In future, to un-register observer, it would be handy to have a reference to the subject.

Scanned with CamScanner

15) POWER UP THE WEATHER STATION

```
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, 30.64f);
        weatherData.setMeasurements (82, 70, 29.2f);
        weatherData.setMeasurements (78, 90, 29.2f);
    }
}
```

16) LOOKING FOR THE OBSERVER PATTERN IN THE WILD.

Some of applications of observer pattern in Java (JDK) are Java Beans and Swing libraries.

The Swing library :-

→ Swing is Java's GUI toolkit for user interfaces.

→ If you look up JButton's super class (one of the components of that toolkit), AbstractButton, it has a lot of add/remove listener methods.

→ These methods allow you to add and remove observers (in Swing, listeners) - to listen for various types of events that occur on the Swing component.

Scanned with CamScanner

→ For instance, an ActionListener lets you "listen in" on any types of actions that might occur on a button, like a button press.

17) A LITTLE LIFE-CHANGING APPLICATION

↳ A button - "Should I do it?"

↳ When you click on that button, the listeners (observers) get to answer the question in any way they want.

↳ Two such listeners called "AngelListener" and "DevilListener".

↳ Devil answer :- "Come on, do it!"

↳ Angel answer :- "Don't do it, you might regret it!"

18) CODING THE LIFE-CHANGING APPLICATION

Create a JButton object
add it to a JFrame.
set up our listeners.

```
public class SwingObserverExample {
    JFrame frame;
    public static void main (String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }
    public void go() {
        frame = new JFrame();

        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());

        // Set frame properties here
    }
```

Scanned with CamScanner

```
    class AngelListener implements ActionListener {
        public void actionPerformed (ActionEvent event) {
            System.out.println ("Don't do it, you might
                                    regret it!");
        }
    }
    class DevilListener implements ActionListener {
        public void actionPerformed (ActionEvent event) {
            System.out.println ("Come on, do it!");
        }
    }
}
```

Using Lambda Expression :-

```
public class SwingObserverExample {
    JFrame frame;
    public static void main (String[] args) {
        SwingObserverExample example = new SwingObserver
                                            Example();
        example.go();
    }
    public void go() {
        frame = new JFrame();
        JButton button = new JButton ("Should I do it?");
        button.addActionListener (event -> System.out.println (
            "Don't do it, you might regret it!"));
        button.addActionListener (event -> System.out.println (
            "Come on, do it!"));
    }
}
```

Scanned with CamScanner

Java's Observer and Observable classes :-

Observable class (the Subject) provided methods to add, delete and notify observers.

Observer interface provided an interface with update() method.

These classes were deprecated in java 9.

JavaBean's built-in support for Observer :-

JavaBeans offers built-in support through PropertyChangeEvents that are generated when a Bean changes a particular kind of property, and sends events to PropertyChangeListeners. There are also related publisher/subscriber components in the Flow API for handling asynchronous streams.

19) PULLING THE VALUES WHEN OBSERVER NEEDS

Right now, when the Subject's data changes, we push the new values for temperature, humidity, and pressure to the Observers, by passing that data in the call to update().

We now set up in such a way that, when an Observer is notified of a change, it calls getter methods on the Subject to pull the values it needs.

* For the Subject to send notifications...

① modify notifyObservers() method in WeatherData to call the method update() in the Observers with no arguments.

```
public void notifyObservers() {
    for(Observer observer: observers) {
        observer.update();
    }
}
```

Scanned with CamScanner

* For an Observer to receive notifications :-

① Modify Observer interface update() method with no arguments.

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

② Modify concrete Observer - change signature of update() and get weather data from Subject using WeatherData's getter methods

```
public void update() {
        this.temperature = weatherData.getTemperature();
        this.humidity = weatherData.getHumidity();
        display();
}
```

Note: check the comment "pull change" in GitHub repo to verify the changes.

OO Principles :-

↳ Encapsulate what varies.

↳ Favor composition over inheritance

↳ Program to interfaces, not implementations

↳ Strive for loosely coupled designs between objects that interact.

↓

current principle.

OO Patterns :-

Observer - defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Scanned with CamScanner

Important points :-

→ The Observer pattern defines a one-to-many relationship between objects.

→ Subjects update Observers using a common interface.

→ Observers of any concrete type can participate in the pattern as long as they implement the Observer interface.

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

→ You can push or pull data from the Subject when using the pattern

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

→ You'll also find the pattern in many other places, including RxJava, JavaBeans, and RMI, as well as in other language frameworks, like Cocoa, Swift and JavaScript events.

→ The Observer Pattern is related to the Publish/Subscribe Pattern, which is for more complex situations with multiple Subjects, and/or multiple message types.

→ The Observer Pattern is a commonly used pattern, and we'll see it again when we learn about Model-View-Controller.

How Observer Pattern makes use of each principle ?

① Identify the aspects of your application that vary and separate them from what stays the same.

The thing that varies in Observer Pattern is the state of the Subject and the number and types of Observers. With this pattern, you can vary the objects that are dependent on the state of the Subject, without having to change that Subject.

Scanned with CamScanner

② Program to an interface, not an implementation.

↳ Both the Subject and Observers use interfaces.

↳ The Subject keeps track of objects implementing the Observer interface, while the Observers register with, and get notified by, the Subject interface. This keeps things loosely coupled.

③ Favor composition over inheritance.

↳ The Observer Pattern uses composition to compose any number of Observers with their Subject.

↳ These relationships aren't setup by inheritance but setup at runtime by composition.

Scanned with CamScanner