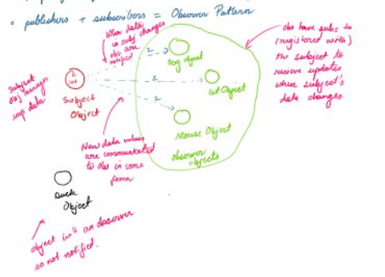
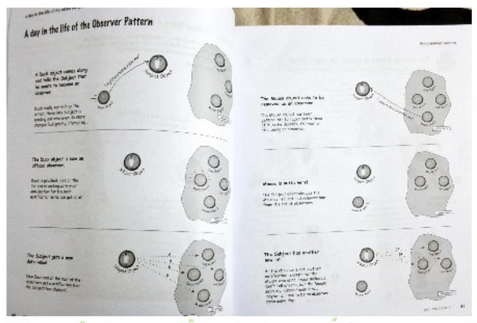
Observer Pattern

11 May 2025 11:06

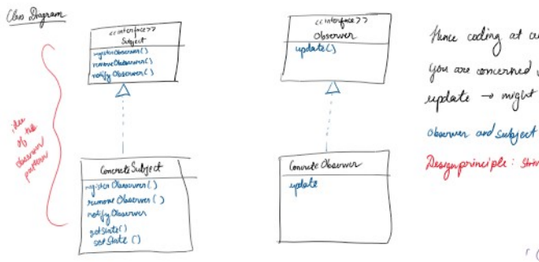
- keeps your objects in the know when something they care about happens.
- publishers + subscribers = Observer Pattern





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

state of one object changes → dependents are notified



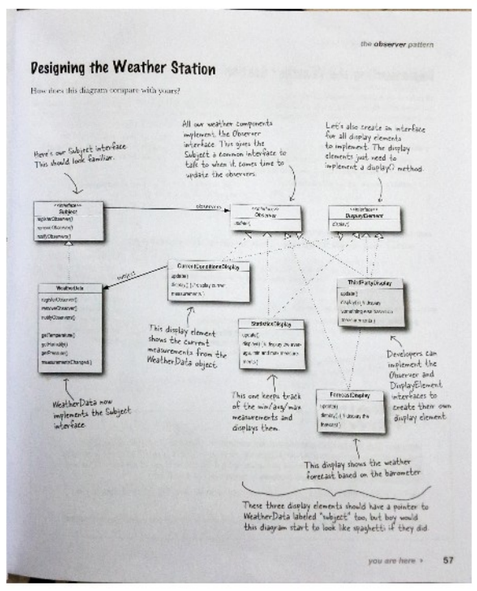
Tight coding at an interface level

You are concerned with interface function

update → might be diff for diff observers

Observer and Subject have knowledge at an interface level only. This an example of loose coupling

Design principle: Strive for loosely coupled designs bw objects that interact



"Push" Implementation

→ Subject sends info to all observers on change

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

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

public interface DisplayElement {
    public void display();
}

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 notifyObserver() {
        for (Observer obs : observers)
        {
            observer.update(temp, pressure, humidity);
        }
    }
    public void measurementChanged() {
        notifyObserver();
    }
    public void setMeasurements(float temp, float humidity, float pressure) {
        this.temp = temp
        this.humidity = humidity
        this.pressure = pressure
        measurementChanged();
    }
}

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temp;
    private float humidity;
    private WeatherData weatherData;

    public CurrentConditionsDisplay(WeatherData weatherData) {
        this.weatherData = weatherData
        weatherData.registerObserver(this);
    }

    public void display() {
        print em
    }

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

weatherData implements Subject

adds to register using weatherData object

when we have a new set of values of weatherData, Subject calls update and passes new values which are set to observer

New value → Subject calls update → values sent to all obs.

notifyObserver() calls update on observer

Setter is used to modify values

Pull Notifical

→ Observers can check on their own and Server can also call them on change.

"Pull" Implementation

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

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

public interface DisplayElement {
    public void display();
}

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 notifyObserver() {
        for (Observer obs : observers)
        {
            observer.update(temp, pressure, humidity);
        }
    }

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temp;
    private float humidity;
    private WeatherData weatherData;

    public CurrentConditionsDisplay(WeatherData weatherData) {
        this.weatherData = weatherData
        weatherData.registerObserver(this);
    }

    public void display() {
        print em
    }

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

weatherData implements Subject

adds to register using weatherData object

Imp ⁂

Let's define Pull notification. It is the ability for observer to ask for new values. Initially update method was dependent on new value i/p which was only with Subject.

- Now since we added getters of Object, observers can call update any time and we check if values have change
- The original functionality is intact, on change of values of subject, all observers notified

Push	Pull
• update has parameters ◦ update values are sent as in parameters	• update with parameters • uses getter to get values hence can be called multiple times.


case of coding to interface
and not implementation
}
public void measureandDump() {
notify Observers();
}
public void setMeasurements(float temp, float humidity, float pressure) {
this.temp = temp;
this.humidity = humidity
this.pressure = pressure
measurementsChanged();
}