Design Pattern: Observer

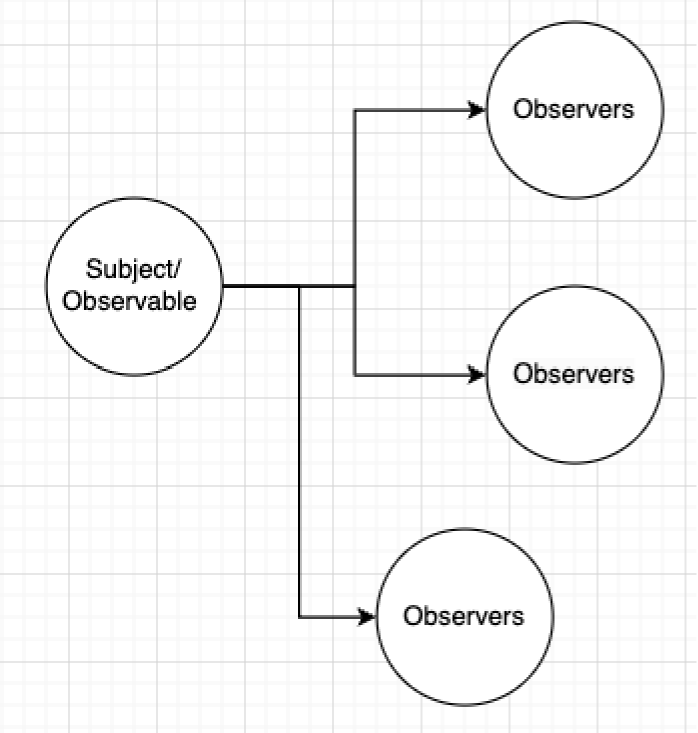
This is a solution to the problem of having a class that needs to be notified when something changes state. This is a behavior pattern.

This is a common problem in GUIs, where the screen needs to be updated when something else happens. This switches from the typical polling approach to a pushing approach.

What polling does is that it checks the state of the object at a certain interval. This is not efficient, and it is not a good solution. This also means that if the state changes every 5 seconds and you poll every 10 seconds, you will miss the change. Now imagine that you have 1000 objects that you need to poll, this will be a very inefficient solution and very resource intensive.

Pushing is when the object that contains the state notifies the object that needs to be updated. All of the things observing the object, need to be subscribed so they get notified and updated.This is a much more efficient solution.

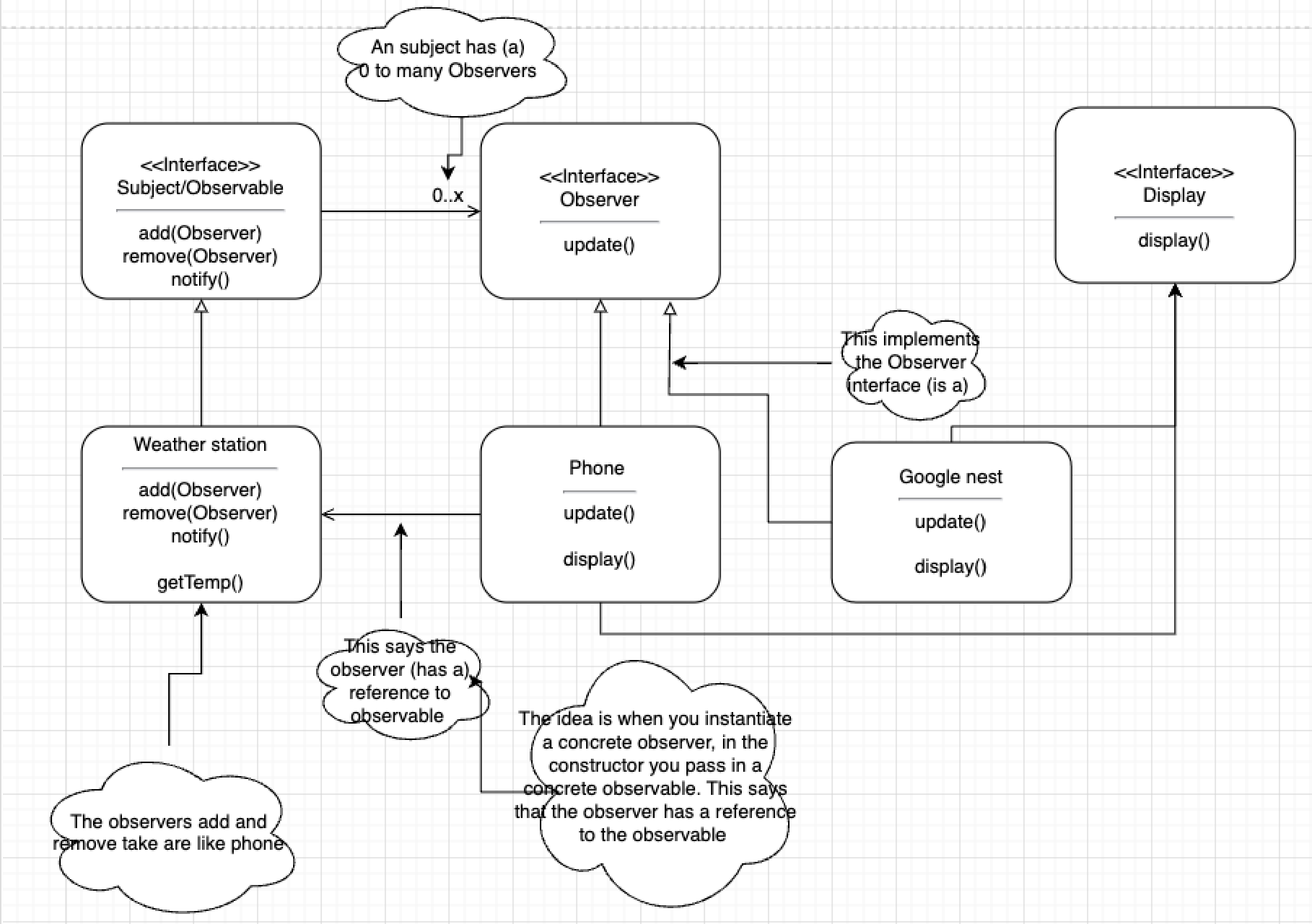
The thing, that has the state that is changing, is called the subject or observable. The thing that is observing the subject is called the observer. The subject can have many observers. The subject is the one that notifies the observers when the state changes. The observers are the ones that are notified and updated.



It is a one-to-many relationship between objects. When the state of one object changes, all the objects that are observing it are notified and updated.

A use case for this is for tracking the changes in a stock price. The stock price is the subject and the observers are the people that are interested in the stock price. When the stock price changes, the observers are notified and updated.

UML Diagram


An subject has (a)
0 to many Observers
<<Interface>>
Subject/Observable
add(Observer)
remove(Observer)
notify()
0..x
<<Interface>>
Observer
update()
<<Interface>>
Display
display()
This implements the Observer interface (is a)
Weather station
add(Observer)
remove(Observer)
notify()
getTemp()
Phone
update()
display()
Google nest
update()
display()
This says the observer (has a) reference to observable
The idea is when you instantiate a concrete observer, in the constructor you pass in a concrete observable. This says that the observer has a reference to the observable
The observers add and remove take are like phone

Example 1

```
public class PropertyInt{
    private List<Consumer<Integer>> listeners = new ArrayList<>();
    private int value;

    public void addListener(Consumer<Integer> listener){
        listeners.add(listener);
    }

    public void setValue(int value){
        this.value = value;
        // for(Consumer<Integer> listener : listeners){
        //     listener.accept(value);
        // }
        listeners.forEach(listener -> listener.accept(value));
    }

    public int getValue(){
        return value;
    }
}

public class PattersMain{
    public static void main(String[] args) {
        PropertyInt propertyInt = new PropertyInt();
        propertyInt.addListener(x -> System.out.println("Value changed to " + x));
        propertyInt.setValue(10);
    }
}
```

Example 2

This is a full example implemented by me

```
package org.example;

public interface Subject {
    void add(Observer observer);
    void remove(Observer observer);

    void notifyObservers();

}
```

```
package org.example;

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

```
package org.example;

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

```
package org.example;

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

public class WeatherStation implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String weather;
```

```java

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

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

    @Override
    public void notifyObservers() {
        for(Observer o: observers){
            o.update();
        }
    }

    public String getWeather(){
        return weather;
    }


    public void setWeather(String weather){
        this.weather = weather;
        notifyObservers();
    }
}
```

```java
package org.example;

public class Phone implements Observer{

    private WeatherStation weather;
    private String weatherString;

    public Phone(WeatherStation weather){
        this.weather = weather;
    }


    @Override
    public void update() {
        this.weatherString = weather.getWeather();
    }


    public void display(){
        System.out.println("Phone: " + weatherString);
    }
}
```

```
package org.example;

public class Main {
    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();
        Phone phoneDisplay = new Phone (weatherStation);

        weatherStation.add(phoneDisplay);
        weatherStation.setWeather("20");
        phoneDisplay.display();


    }
}
```