OBSERVER DESSIGN PATTERN

WELCOME TO OUR GROUP

Hao Trinh The

10421017

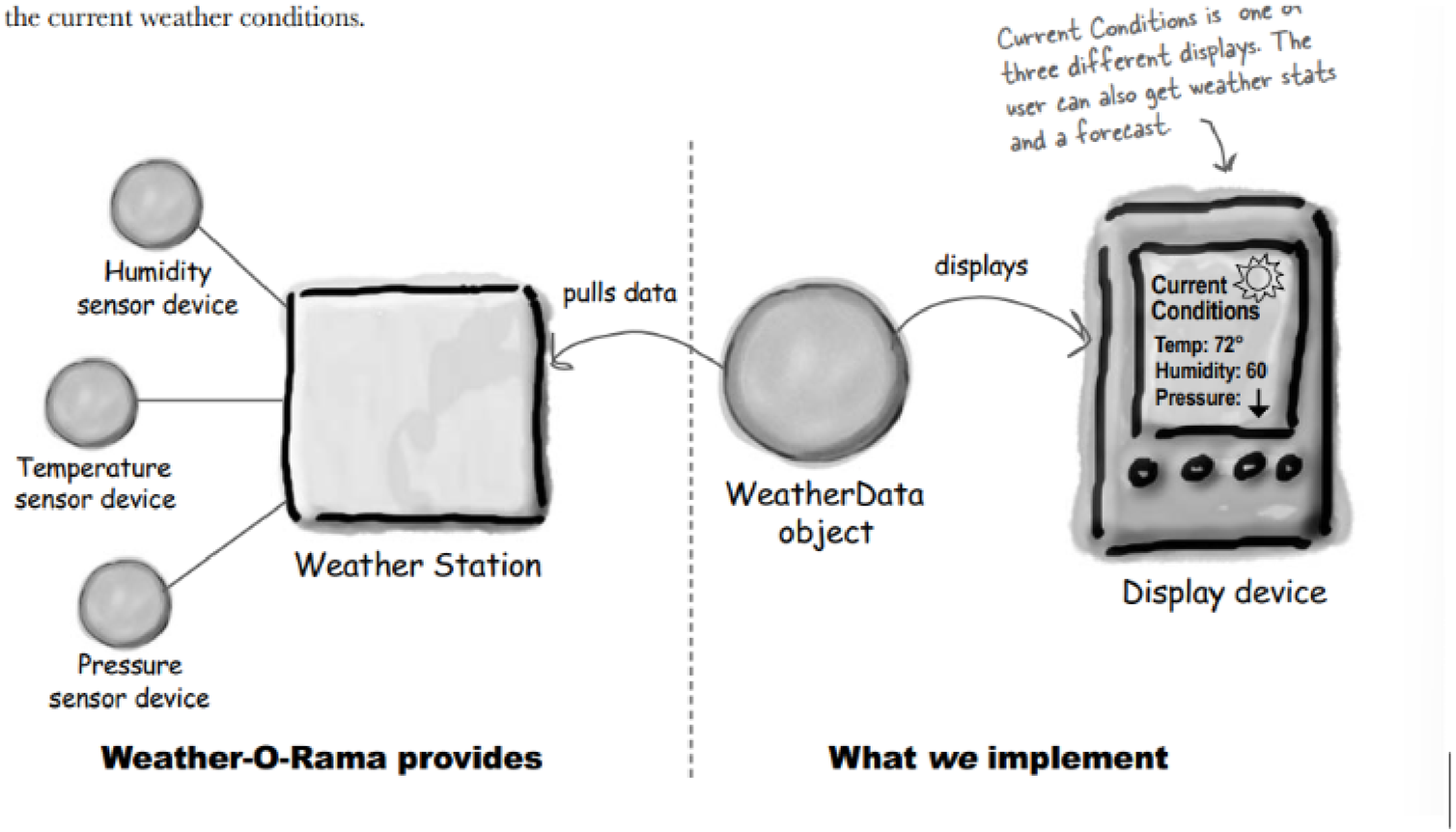
Huy Vo Vuong Bao

10421021

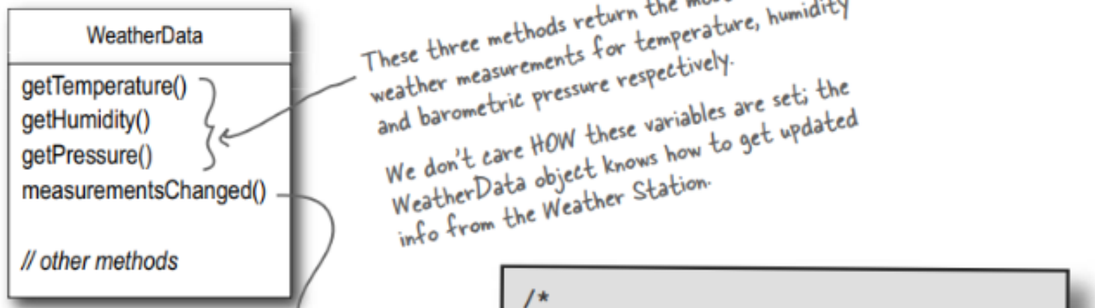

CASE
STUDY

WEATHER MONITORING APPLICATION:

the current weather conditions.



EXTRACTING THE WEATHER DATA CLASS



These three methods return the most recent weather measurements for temperature, humidity and barometric pressure respectively.

We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

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

WeatherData.java

The developers of the WeatherData object left us a clue about what we need to add...

Remember, this Current Conditions is just ONE of three different display screens.

Display device

Our job is to implement measurementsChanged() so that it updates the three displays for current conditions, weather stats, and forecast.

SOMETHING WRONG HAPPEN!!!

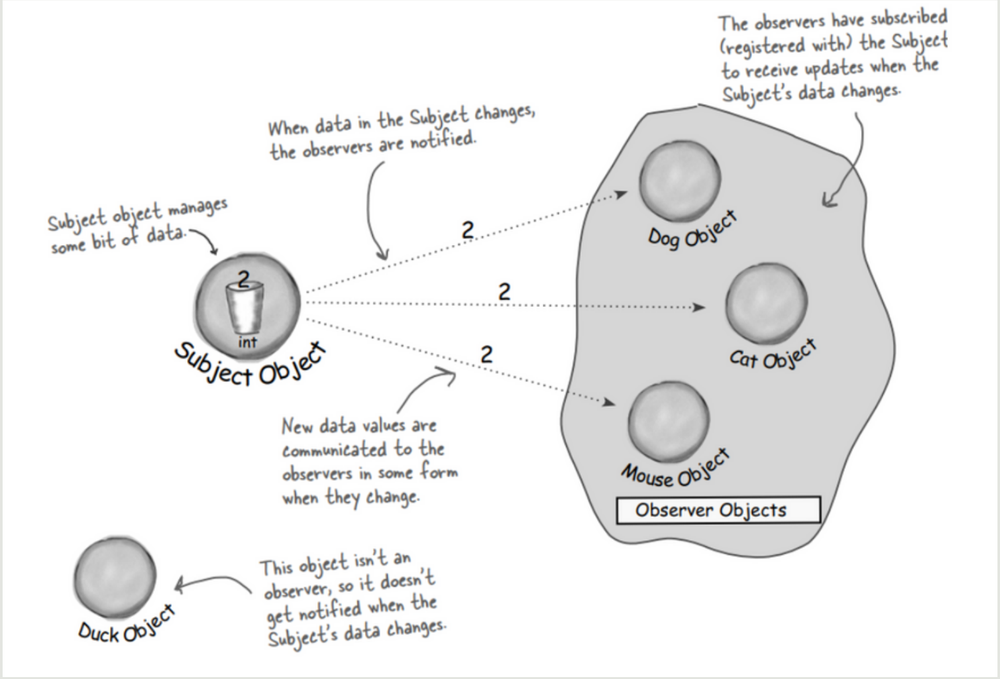
```
public void measurementsChanged() {

    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

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

Area of change, we need to encapsulate this.

By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

At least we seem to be using a common interface to talk to the display elements... they all have an update() method takes the temp, humidity, and pressure values.

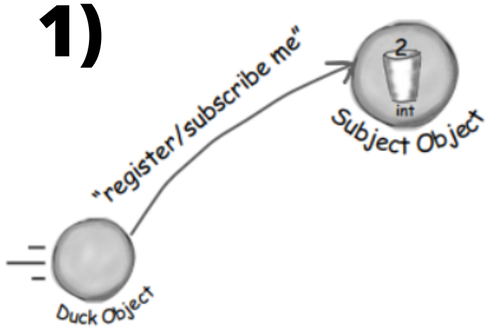
Observer Design Pattern = Publishers + Subscribers

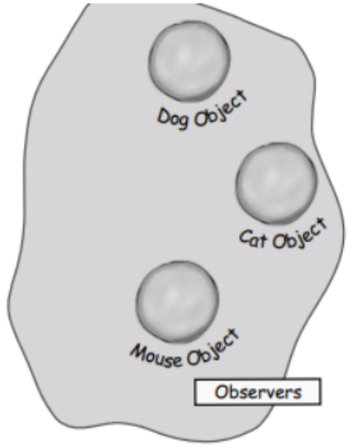


Process of Subscribing un Unsubscribing

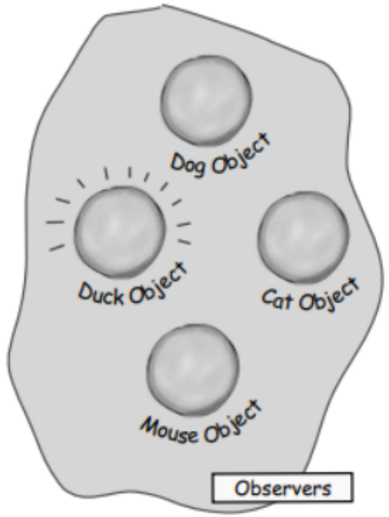
SUBSCRIBING

1)

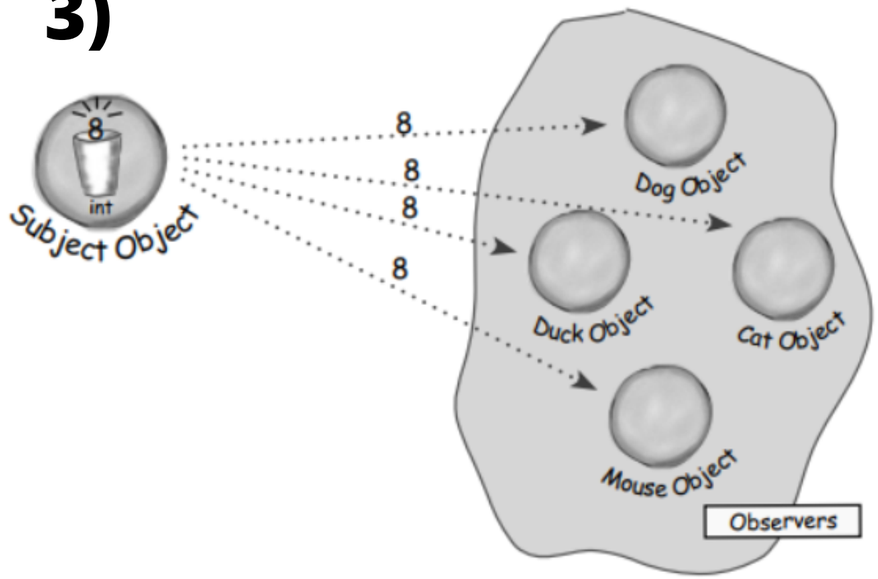




2)

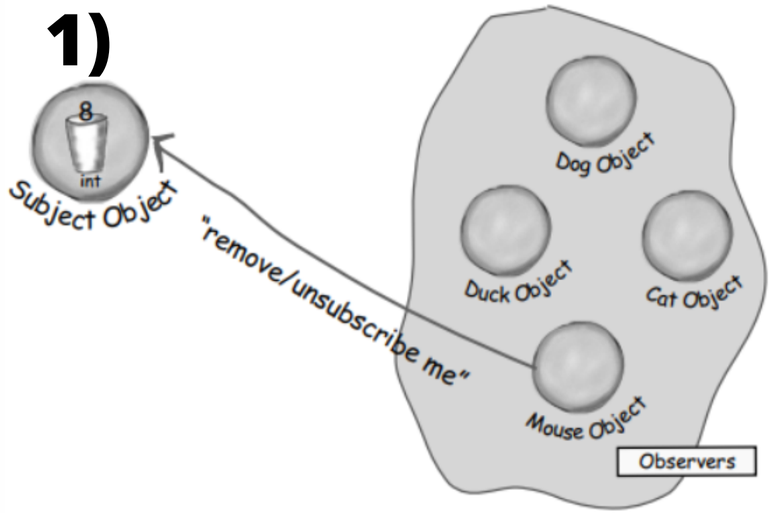


3)

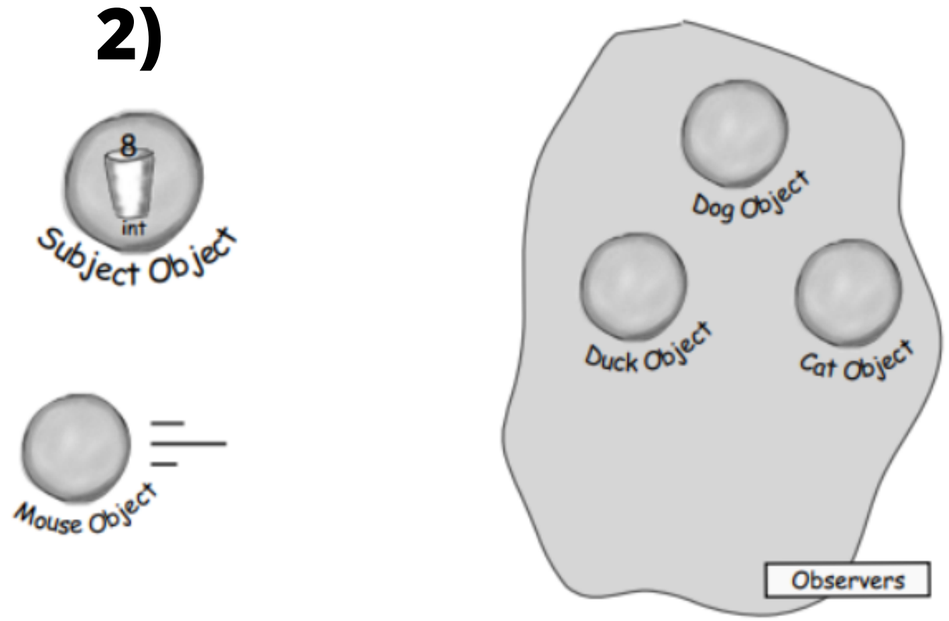


UNSUBSCRIBING

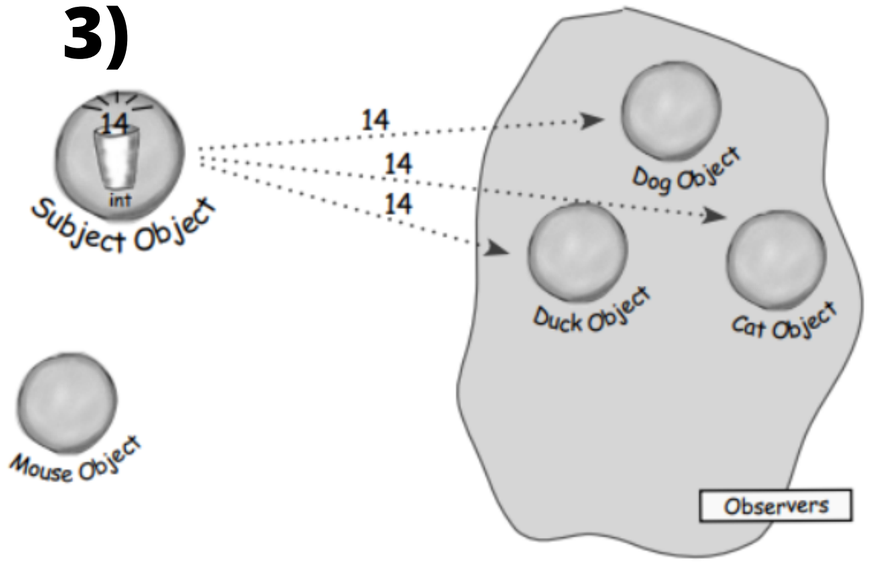
1)



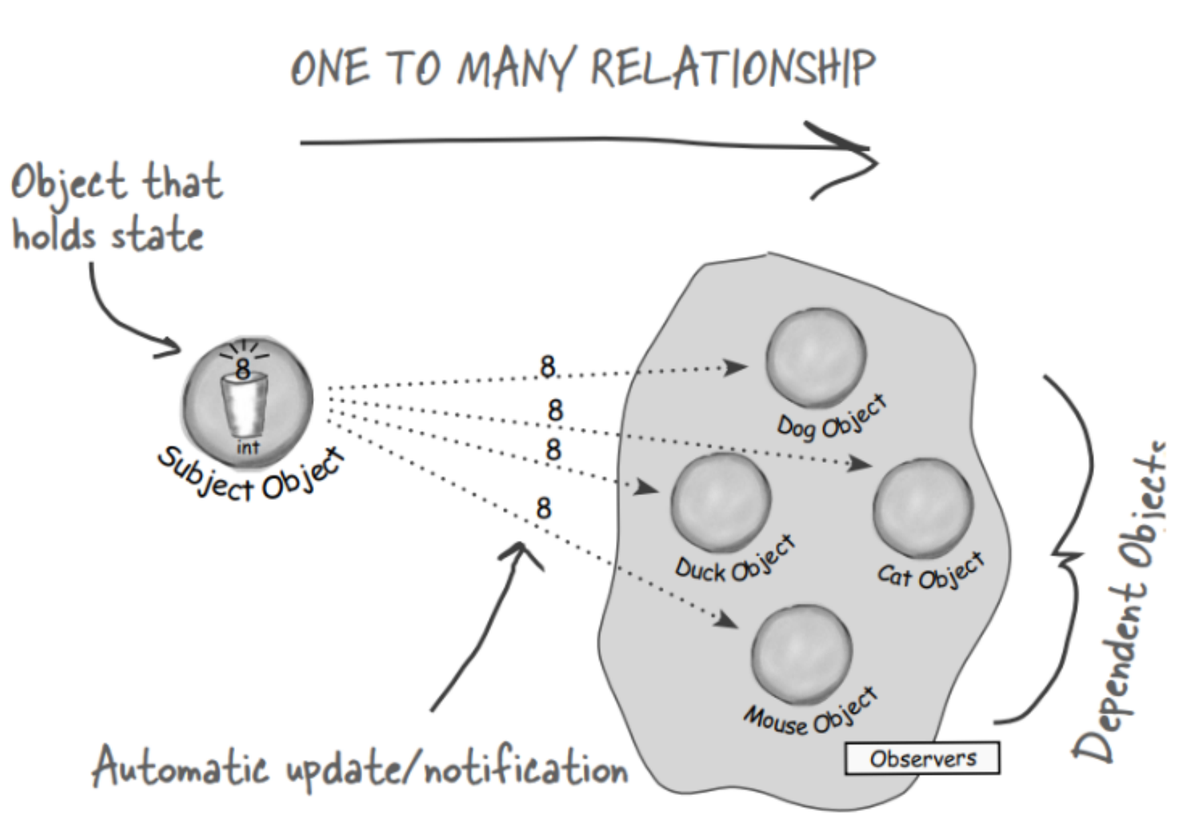
2)

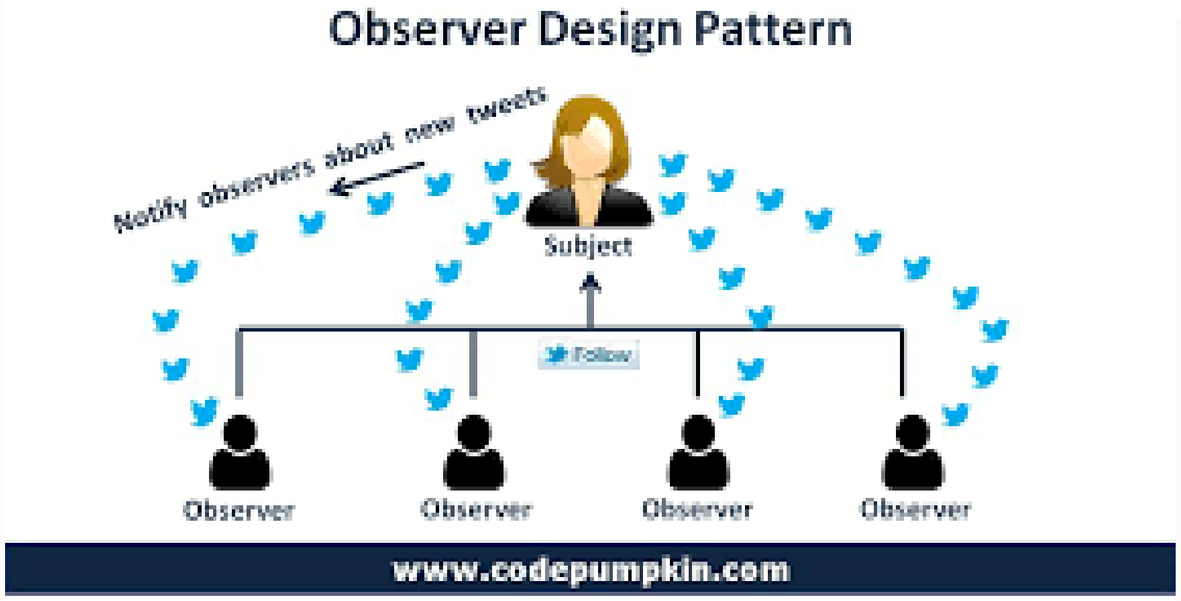


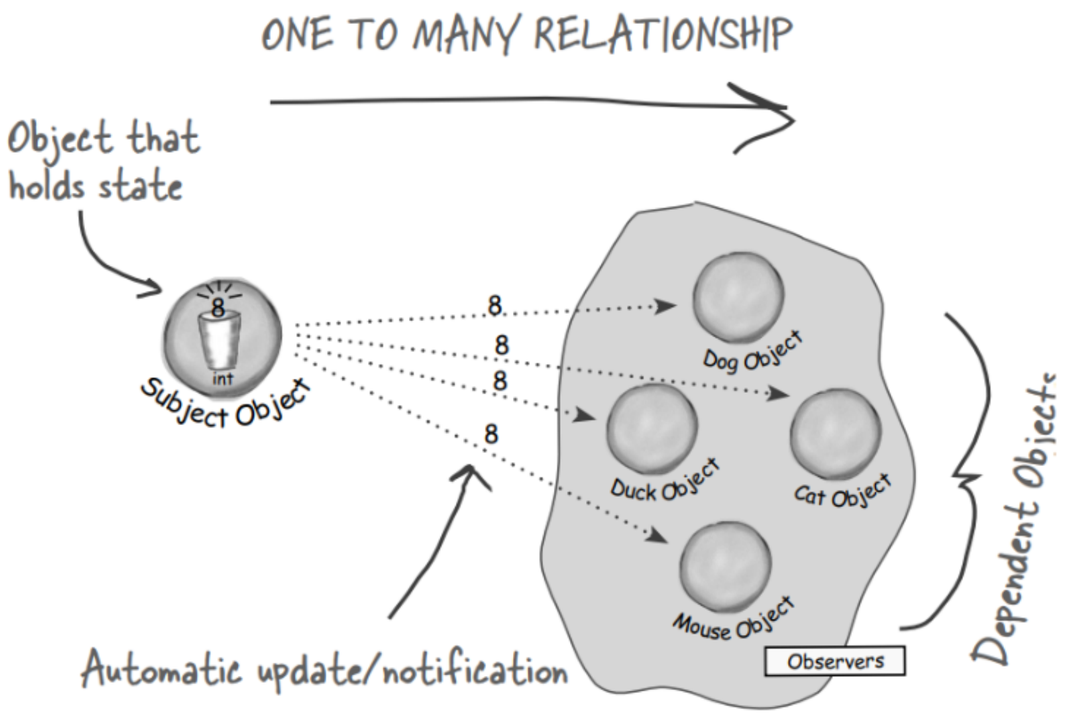
3)



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

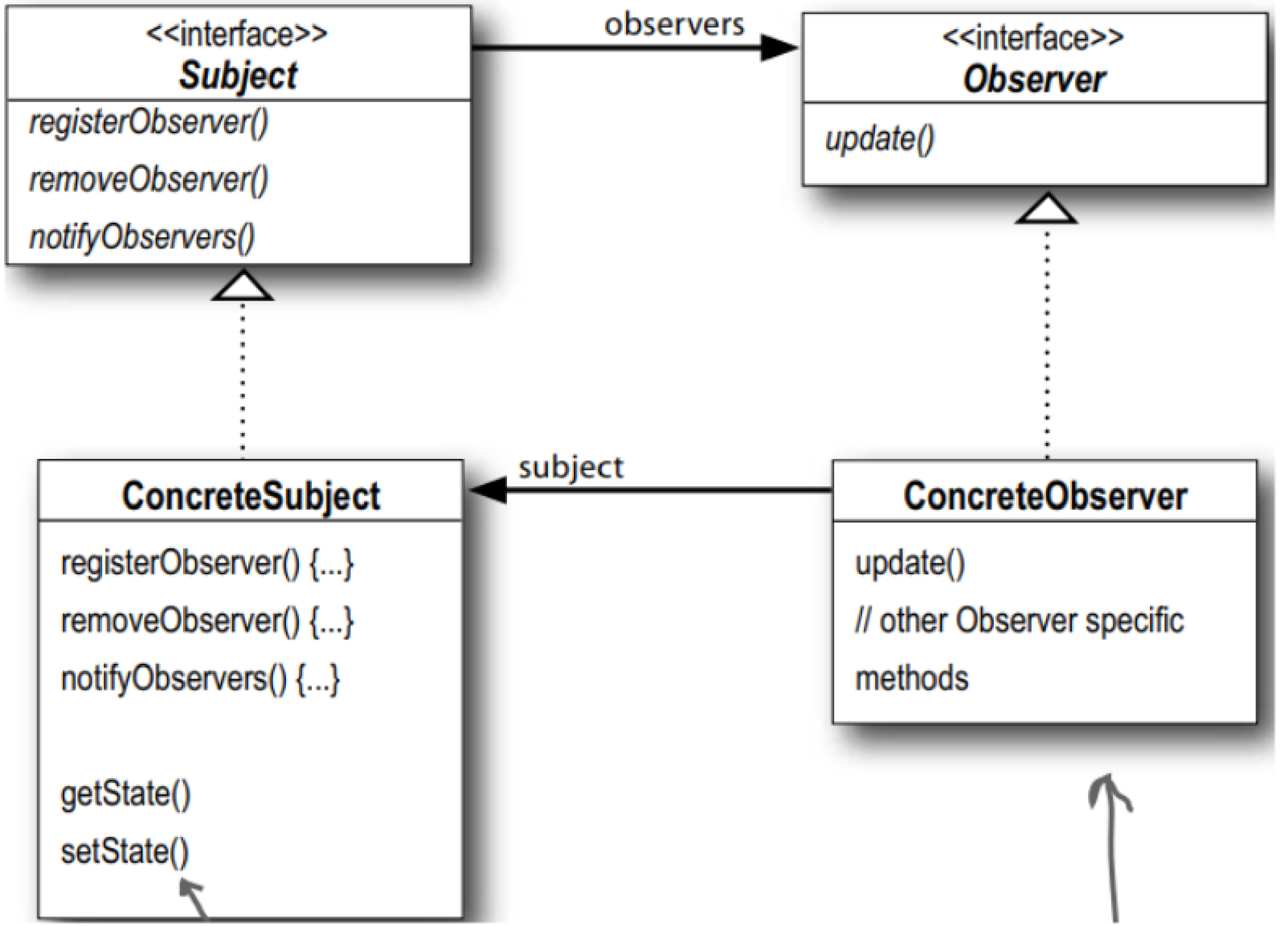




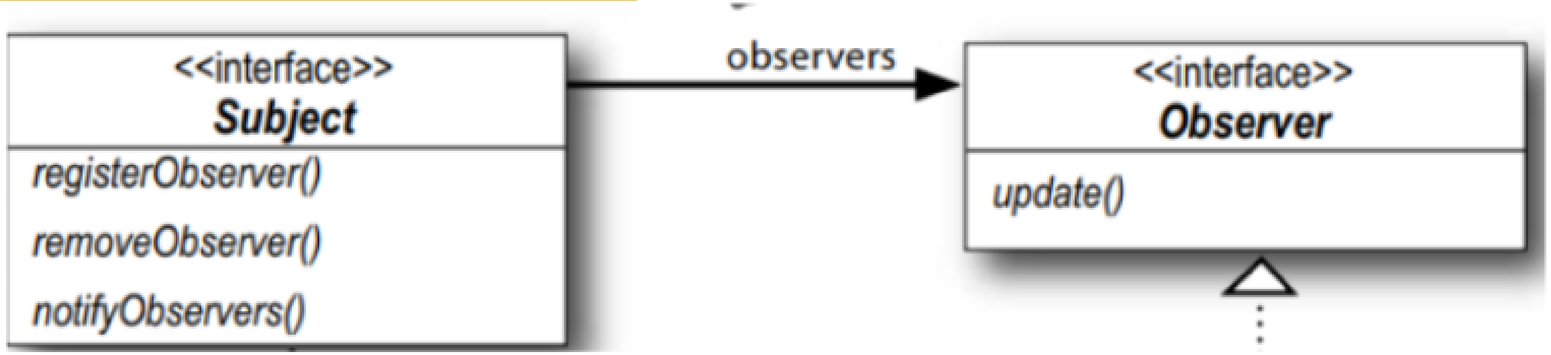


THE OBSERVER PATTERN DEFINES A ONE-TO-MANY DEPENDENCY BETWEEN OBJECTS SO THAT WHEN ONE OBJECT CHANGES STATE, ALL OF ITS DEPENDENT ARE NOTIFIED AND UPDATED AUTOMATICALLY.

The Class Diagram

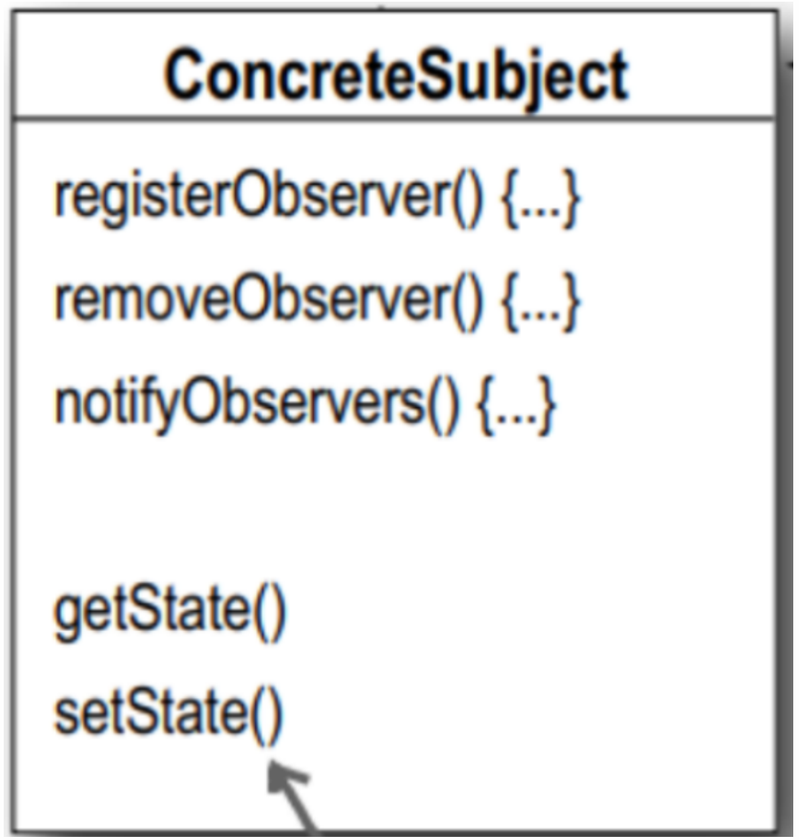


CODE IMPLEMENTATION



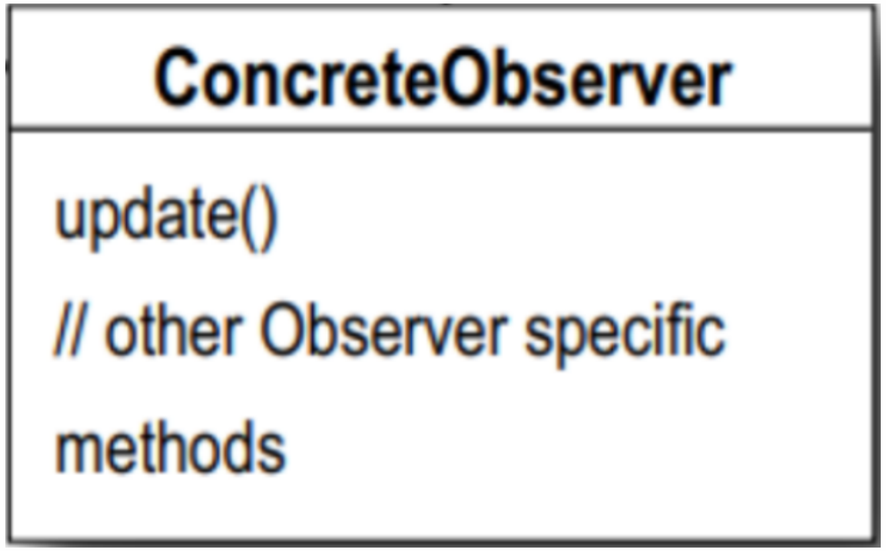

```
class IObserver{
    public:
        virtual ~IObserver(){};
        virtual void Update(string state) =0;
};
class ISubject{
    public:
        virtual ~ISubject(){};
        virtual void registerObserver(IObserver *observer)=0;
        virtual void removeObserver(IObserver *Observer)=0;
        virtual void notifyObserver()=0;
};
```

CODE IMPLEMENTATION




```
class Subject: public ISubject{
    private:
        list<IObserver*>list_observer;
        string state = "Default state";
    public:
        void registerObserver(IObserver *observer) override{
            list_observer.push_back(observer);
        }
        void removeObserver(IObserver *observer) override{
            list_observer.remove(observer);
        }
        void notifyObserver() override{
            list<IObserver*>::iterator iter = list_observer.begin();
            while(iter != list_observer.end()){
                (*iter)->Update(this->state);
                ++iter;
            }
        }
        void setState(string state){
            this->state = state;
            notifyObserver();
        }
        string getState(){
            return this->state;
        }
};
```

CODE IMPLEMENTATION



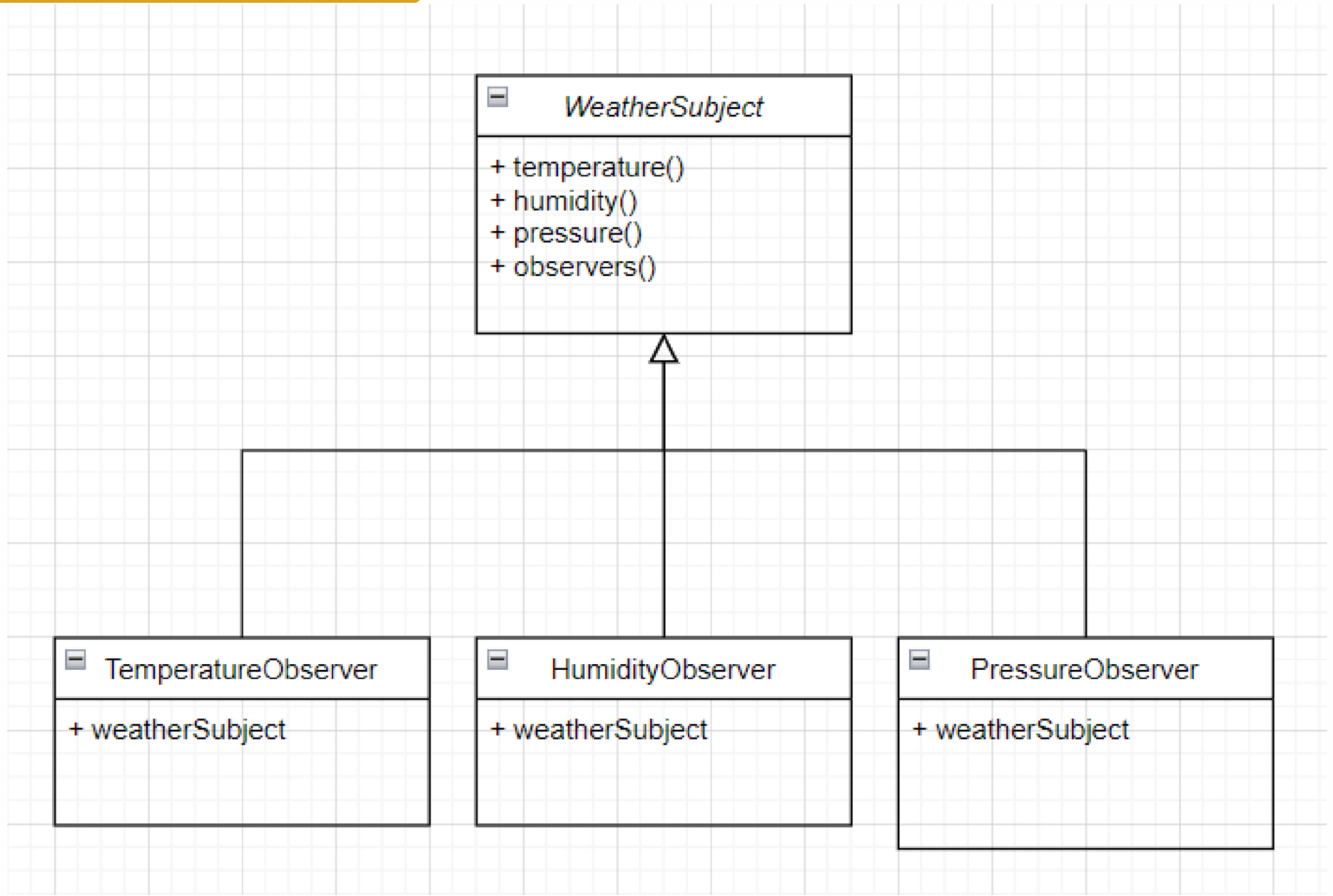

```
class Observer: public IObserver{
    private:
        string state;
        Subject &subject_;
        static int static_number;
        int number;
    public:
        Observer(Subject &subject) : subject_(subject){
            this->subject_.registerObserver(this);
            this->state = this->subject_.getState();
            this->number = Observer::static_number;
            ++Observer::static_number;
        }
        void Update(string state) override{
            this->state = state;
        }
        void Info(){
            cout <<"Observer "<<this->number<<" has state: "<<this->state <<endl
        }
};
```

CODE IMPLEMENTATION

```cpp
int Observer::static_number =1;
void check(vector <Observer*> ob_list, Subject* subject){
    cout <<"Current state of subject is: " <<(*subject).getState()<< endl;
    for(int i=0; i< ob_list.size(); i++){
        ob_list[i]->Info();
    }
    cout<<endl;
}
```

```cpp
int main(){

    Subject *subject = new Subject();

    Observer *observer1 = new Observer(*subject);
    Observer *observer2 = new Observer(*subject);
    Observer *observer3 = new Observer(*subject);

    vector<Observer*> observer_list{observer1,observer2, observer3};
    check(observer_list, subject);

    subject->setState("Sehr Gut");
    check(observer_list, subject);

    Observer *observer4 = new Observer(*subject);
    observer_list.push_back(observer4);
    check(observer_list, subject);

    subject->removeObserver(observer2);
    subject->setState("So So");
    check(observer_list, subject);

    subject->removeObserver(observer3);
    Observer *observer5 = new Observer(*subject);
    observer_list.push_back(observer5);
    subject->setState("Perfect");
    check(observer_list, subject);
}
```

REAL LIFE APPLICATION



CODE FOR REAL LIFE APPLICATION

```cpp
#include <iostream>
#include <vector>

using namespace std;

// The Observer interface
class Observer {
public:
    virtual void update(float temperature, float humidity, float pressure)
= 0;
};

// The Subject interface
class Subject {
public:
    virtual void registerObserver(Observer* observer) = 0;
    virtual void removeObserver(Observer* observer) = 0;
    virtual void notifyObservers() = 0;
};

// The WeatherData class is the concrete Subject
class WeatherData : public Subject {
private:
    vector<Observer*> observers;
    float temperature;
    float humidity;
    float pressure;
public:
        observers.push_back(observer);
    }

    void removeObserver(Observer* observer) override {
        for (auto it = observers.begin(); it != observers.end(); ++it) {
            if (*it == observer) {
                observers.erase(it);
                break;
            }
        }
    }

    void notifyObservers() override {
        for (auto observer : observers) {
            observer->update(temperature, humidity, pressure);
        }
    }

    void setMeasurements(float temperature, float humidity, float
pressure) {
        this->temperature = temperature;
        this->humidity = humidity;
        this->pressure = pressure;
        measurementsChanged();
    }

    void measurementsChanged() {
        notifyObservers();
```

CODE FOR REAL LIFE APPLICATION

```
 The CurrentConditionsDisplay class is a concrete Observer
lass CurrentConditionsDisplay : public Observer {
rivate:
    float temperature;
    float humidity;
    float pressure;
    Subject& weatherData;
ublic:
    CurrentConditionsDisplay(Subject& weatherData) :
```

```
Current conditions: 35C degrees and 65% humidity
Current conditions: 32C degrees and 70% humidity
Current conditions: 22C degrees and 90% humidity
```

```
weatherData(weatherData) {
        weatherData.registerObserver(this);
    }

    void update(float temperature, float humidity, float pressure)
override {
        this->temperature = temperature;
        this->humidity = humidity;
        this->pressure = pressure;
        display();
    }

    void display() {
        cout << "Current conditions: " << temperature << "F degrees and "
<< humidity << "% humidity" << endl;
    }
};

int main() {
    WeatherData weatherData;

    CurrentConditionsDisplay currentConditions(weatherData);

    weatherData.setMeasurements(80, 65, 30.4f);
    weatherData.setMeasurements(82, 70, 29.2f);
    weatherData.setMeasurements(78, 90, 29.2f);

    return 0;
}
```

THANK YOU FOR LISTENING