

CS 351
Design of Large Programs
Observer Pattern

Brooke Chenoweth

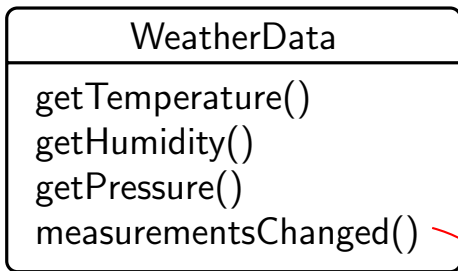
University of New Mexico

Spring 2024

A Weather Monitoring Application

- We have a weather station with humidity, temperature, and pressure sensors
- We are implementing a `WeatherData` object that pulls data from the weather station.
- Create an app that uses the `WeatherData` object to update three different displays:
 - current conditions
 - weather stats
 - forecast

What Needs to be Done?



Update three
different displays

```
/**  
 * Call this method whenever  
 * measurements are updated  
 */  
public void measurementsChanged(){  
 // your code goes here  
}
```

Problem Specification

- The `WeatherData` class has getters and setters for temperature, humidity, and pressure
- The `measurementsChanged()` method is called anytime new weather data is available
 - We don't know or care how!
- We need to implement three different display elements that use the weather data
- The system must be expandable, in case others want to add other display elements later

A First Attempt to Coding

```
public class WeatherData {  
  
    //instance variable declarations  
  
    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);  
    }  
  
    // other methods  
}
```

- What's wrong?

A First Attempt to Coding

```
public class WeatherData {  
  
    //instance variable declarations  
  
    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);  
    }  
  
    // other methods  
}
```

- **What's wrong?**
- Coding to implementations: adding displays requires changing the program

A First Attempt to Coding

```
public class WeatherData {  
  
    //instance variable declarations  
  
    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);  
    }  
  
    // other methods  
}
```

- **What's wrong?**
- Coding to implementations: adding displays requires changing the program
- Encapsulate stuff that changes!

A First Attempt to Coding

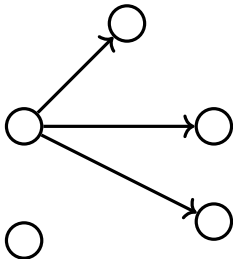
```
public class WeatherData {  
  
    //instance variable declarations  
  
    public void measurementsChanged(){  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
  
        currentConditionsDisplay.update(temp, humidity, pressure);  
        statisticsDisplay.update(temp, humidity, pressure);  
        forecastDisplay.update(temp, humidity, pressure);  
  
    }  
  
    // other methods  
}
```

Not so bad: Here's a common interface!

- **What's wrong?**
- Coding to implementations: adding displays requires changing the program
- Encapsulate stuff that changes!

Publish/Subscribe

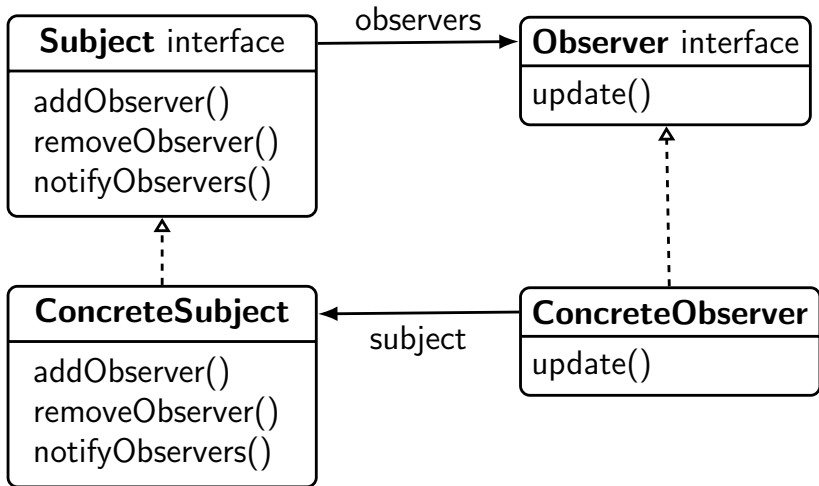
- Just like newspapers and magazines
 - email lists
 - RSS feeds
 - following someone on Twitter
- You subscribe and receive any new additions
- You unsubscribe and stop receiving anything



The Observer Pattern

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

The Observer Pattern



The Observer Pattern

- Objects use the Subject interface to (de)register as observers
- Each subject can have many observers
- All potential observers need to implement the Observer interface and provide the update() method
- A concrete subject always implements the Subject interface and the notifyObservers() method
- Concrete observers can be any class that implements the Observer interface and registers with a concrete subject

The Power of Loose Coupling

- The only thing a subject knows about an observer is that it implements a given interface

The Power of Loose Coupling

- The only thing a subject knows about an observer is that it implements a given interface
- We can add new observers at any time

The Power of Loose Coupling

- The only thing a subject knows about an observer is that it implements a given interface
- We can add new observers at any time
- We never need to modify the subject to add new types of observers

The Power of Loose Coupling

- The only thing a subject knows about an observer is that it implements a given interface
- We can add new observers at any time
- We never need to modify the subject to add new types of observers
- We can reuse subjects or observers independently of each other

The Power of Loose Coupling

- The only thing a subject knows about an observer is that it implements a given interface
- We can add new observers at any time
- We never need to modify the subject to add new types of observers
- We can reuse subjects or observers independently of each other
- Changes to either the subject or an observer will not affect each other

The Power of Loose Coupling

- The only thing a subject knows about an observer is that it implements a given interface
- We can add new observers at any time
- We never need to modify the subject to add new types of observers
- We can reuse subjects or observers independently of each other
- Changes to either the subject or an observer will not affect each other
- **Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependencies between objects.**

Weather Data Interfaces

```
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();  
}
```

Weather Data Interfaces

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

called to notify
all observers
when Subject's
state changes

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

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

Weather Data Interfaces

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

called to notify
all observers
when Subject's
state changes

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

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

Adding an interface for
all display types.

Weather Data Interfaces

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

called to notify
all observers
when Subject's
state changes

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

What's wrong here?

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

Adding an interface for
all display types.

Implementing the Subject Interface

```
public class WeatherData implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }
}
```

Notify Methods

```
public void notifyObservers() {  
    for(Observer observer : observers) {  
        observer.update(temperature, humidity, pressure);  
    }  
}  
  
public void measurementsChanged() {  
    notifyObservers();  
}
```


A Display Element

```
public class CurrentConditionsDisplay
    implements Observer, DisplayElement {

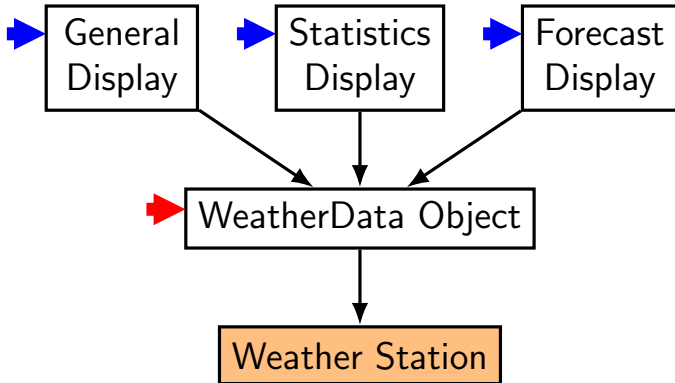
    private float temperature;
    private float humidity;
    private Subject weatherData;

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

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

    public void display() {
        System.out.println("Current conditions: " + temperature
            + " F degrees and " + humidity + "% humidity");
    }
}
```

Object Design Diagram



Triggers

- ▶ notification received from WeatherData object
- ▶ interrupt that triggers measurementsChange