HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

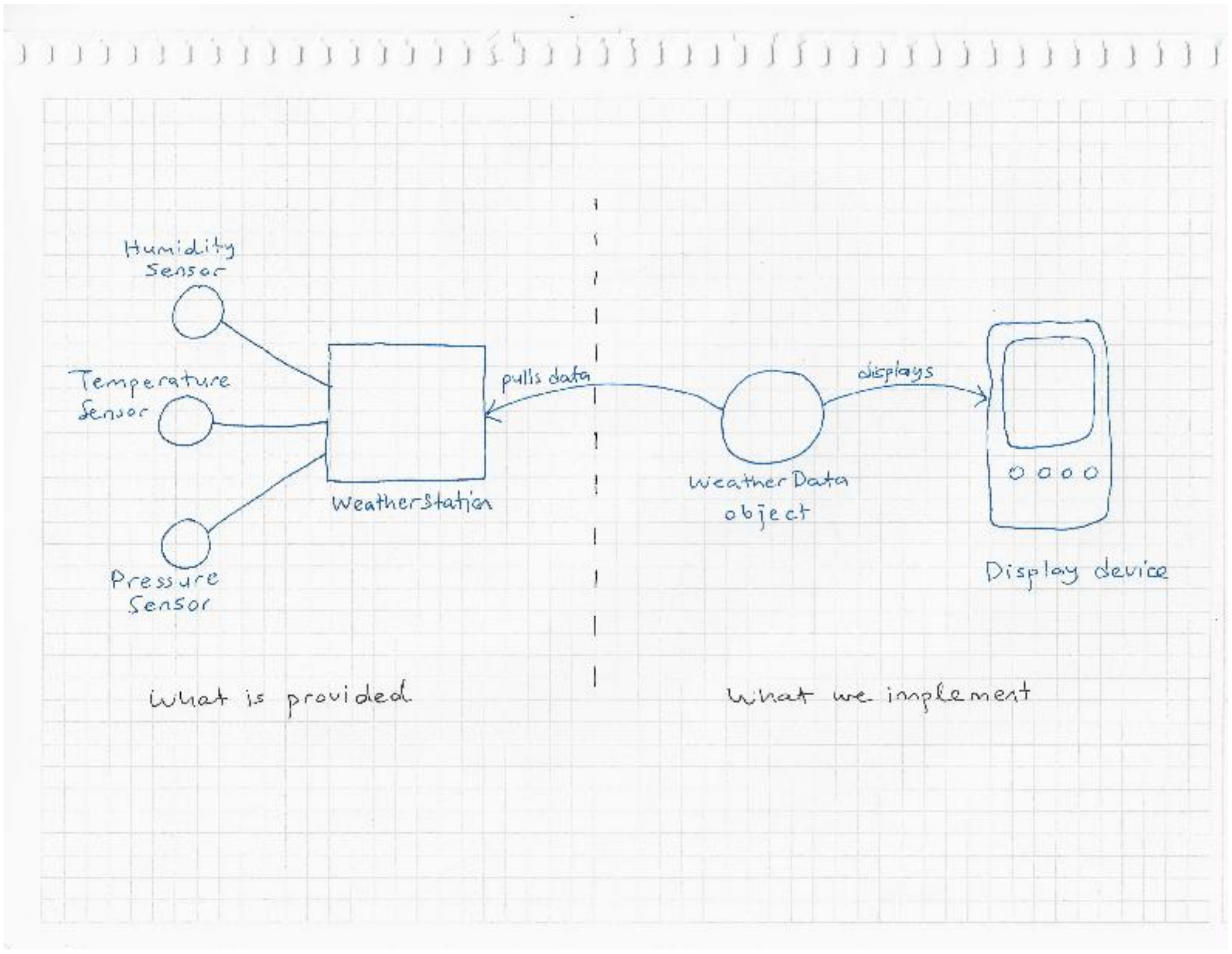
BBM 486 – DESIGN PATTERNS

3. THE OBSERVER PATTERN

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.

The observer pattern is one of the most heavily used patterns in the JDK, and it's incredibly useful. We will illustrate the observer pattern with an example. We are building a next generation, Internet-based Weather Monitoring Station, which tracks weather conditions. We would like the weather station to be expandable through an API so that other developers can build their own weather displays.

1. The initial design of the system has three players: the weather station, the WeatherData object that tracks the data coming from the Weather Station and updates the displays, and the display that shows users the current weather conditions. WeatherData class initially had three methods getTemperature(), getHumidity() and getPressure().



2. We are also given a measurementsChanged() method, which gets called whenever the weather measurements have been updated. Our job is to implement this method so that it updates the three displays for current conditions, weather stats, and forecast. We can add the calls to update the three displays into this method, but by doing so we have no way to add or remove other display elements without making changes to the program.

3. What do we know so far?
    - The WeatherData class has getter methods for three measurement values: temperature, humidity and barometric pressure:
      getTemperature()
      getHumidity
      getPressure()
    - The measurementsChanged() method is called anytime new weather measurement data is available. We do not know or care how this method is called; we just know that it is.
    - We need to implement three display elements that use the weather data: a current conditions display, a statistics display, and a forecast display. These displays must be updated each time WeatherData has new measurements.
    - The system must be expandable – other developers can create new custom display elements and users can add or remove as many display elements as they want to the application.

4. Here is a first implementation possibility:

```
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 WeatherData methods here
}
```

5. Based on our first implementation, we of the following apply?

A. We are coding to concrete implementations, not interfaces.	X
B. For every new display element we need to alter code.	X
C. We have no way to add or remove display elements at run time.	X
D. The display elements don't implement a common interface.	
E. We have not encapsulated the part that changes.	X
F. We are violating encapsulation of the WeatherData class.	

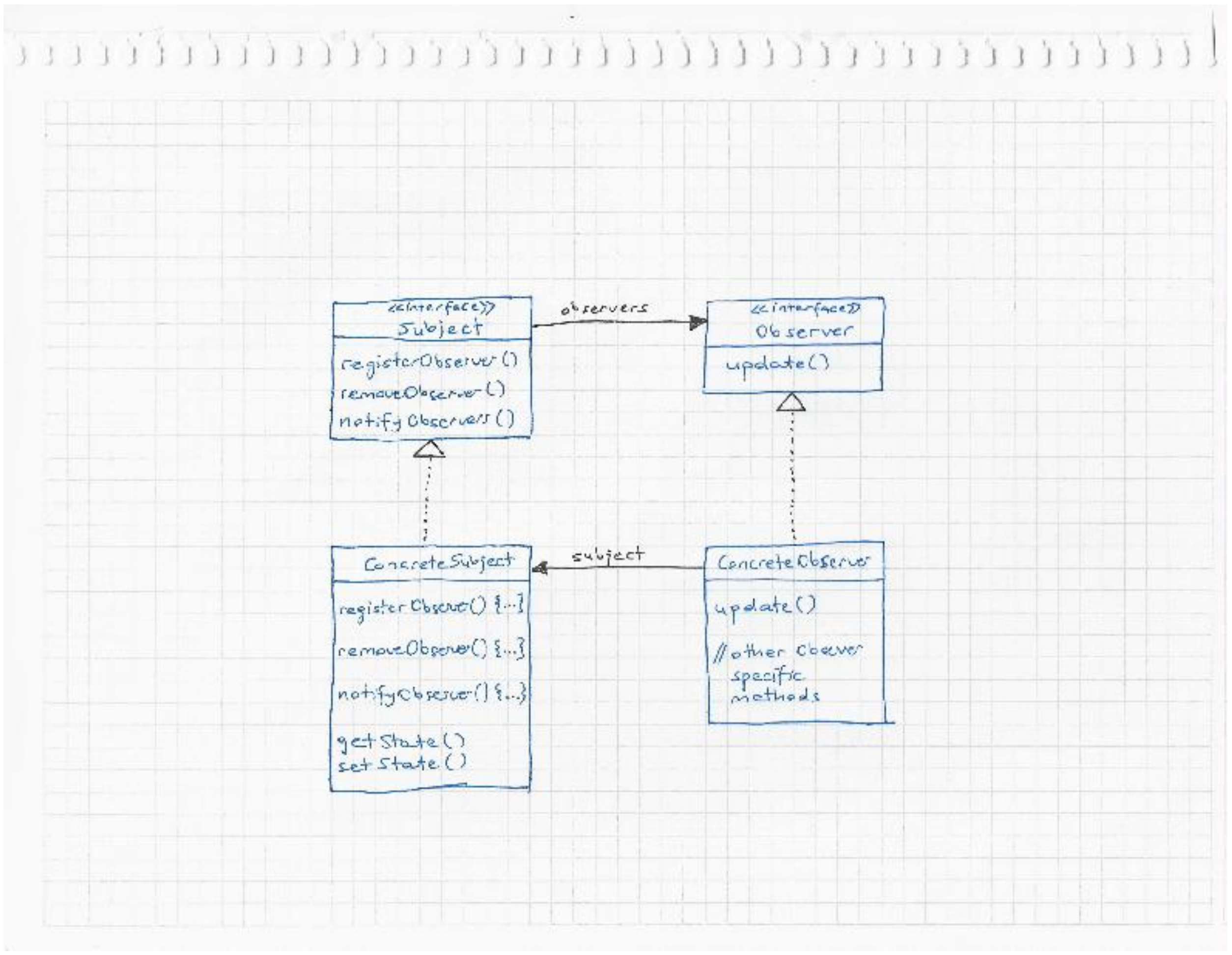
6. What's wrong with our implementation?
   By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.
   At least we seem to be using a common interface to talk to the display elements. They all have an update method that takes the temp, humidity and pressure values.

7. Observer pattern works like newspaper subscriptions. An object subscribes to a subject that it wants to be an observer. When the subject gets a new data value, it sends a notification to all the registered objects. When an object asks to be removed as an observer, it no longer receives updates from the subject.

8. 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.

   Let's relate this definition to what we have been talking about the pattern:
   The subject and observers define the one-to-many relationships. The observers are dependent on the subject such that when the subject's state changes, the observers get notified.

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



10. Subject interface provides registerObserver(), removeObserver() and notifyObservers() methods. Objects use this interface to register as observers and also to remove themselves from being observers. All potential observers need to implement the Observer interface, which has just one method, update(), that gets called when the subject's state changes.

11. A concreteSubject implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method to update all the current observers whenever state changes. Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

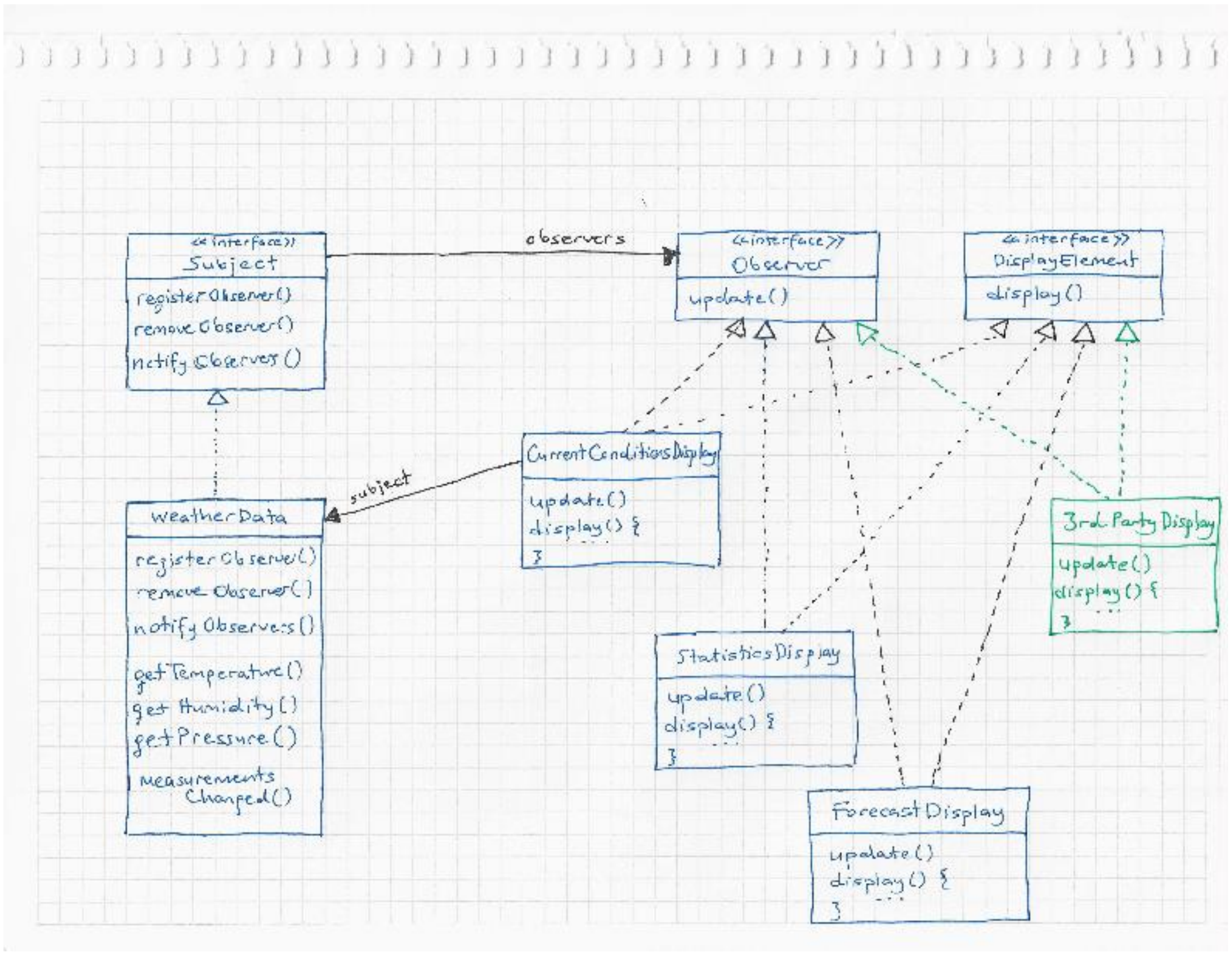
12. The Power of Loose Coupling: When two objects are loosely coupled, they can interact, but have very little knowledge of each other. The Observer Pattern provides an object design where subjects and observers are loosely coupled. Why?
    - The only thing the subject knows about an observer is that it implements a certain interface (Observer interface). It does not need to know the concrete class of the observer, what it does, or anything else about it.

- We can add new observers any time. Because 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. Likewise, we can remove observers whenever we want.
- We never need to modify the subject to add new types of observers. Let's say we have a new concrete class come along that needs to be an observer. We do not need to make any changes to the subject to accommodate the new class type. All we have to do is to implement the Observer interface in the new class and register as an observer.
- We can reuse subjects or observers independently of each other. If we have another use for a subject or an observer, we can easily reuse them because the two are not tightly coupled.
- Changes to either the subject or an observer will not affect the other. Because the two are loosely coupled, we are free to make changes to either, as long as the objects meet their obligations to implement the subject or observer interfaces.

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

14. Loosely coupled designs allow us to build flexible object-oriented systems that can handle change because they minimize the interdependency between objects. Because:
    a. The only thing the subject knows about an observer is that it implements a certain interface.
    b. We can add new observers at any time.
    c. We never need to modify the subject to add new types of observers.
    d. We can reuse subjects or observers independently of each other.
    e. Changes to either the subject or an observer will not affect the other.

15. Designing the Weather Station



16. Implementing the Weather Station

```
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();
}
```

17. Implementing the Subject Interface in WeatherData

```
public class WeatherData implements Subject {
    private ArrayList 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) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
```

18. Building the display elements

```
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 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");
        }
}
```

19. Create a test harness and run the code

```
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, 30.4f);
                weatherData.setMeasurements(82, 70, 29.2f);
                weatherData.setMeasurements(78, 90, 29.2f);
        }
}
```

20. So far, we have rolled our own code for Observer Pattern, but Java has built-in support in several of its APIs. The most general is the Observer interface and the Observable class in the java.util.package. To get a high-level feel for java.util.Observer and java.util.Observable, we can rework the design for the WeatherStation as follows:

21. The built in Observer Pattern works a bit differently than the implementation that we used on the Weather Station. The most obvious difference is that WeatherData (our subject) now extends the

Observable class and inherits the add, delete and notify Observer methods. Here is how we use Java's version:

22. For an object to become an observer: As usual, implement the Observer interface (this time the java.util.Observer interface) and call addObserver() on any Observable object. Likewise, to remove yourself as an observer just call deleteObserver().

    For the Observable to send notifications: First of all you need to be Observable by extending the java.util.Observable superclass. From there it is a two step process: (1) You first must call the setChanged() method to signify that the state has changed in your object. (2) Then, call one of two notifyObservers() methods:

    Either notifyObservers() or notifyObservers(Object arg)

    For an Observer to receive notifications: It implements the update method, as before, but the signature of the method is a bit different:

    ```
    update(Observable o, Object arg);
    ```

    If you want to "push" data to the observers you can pass the data as a data object to the notifyObserver(arg) method. If not, then the Observer has to "pull" the data it wants from the Observable object passed to it.

23. The setChanged() method is used to signify that the state has changed and that notifyObservers() when it is called, should update its observers. If notifyObserver() is called without first calling setChanged(), the observers will not be notified. The setChanged method is used to signify that the state has changed and that notifyObservers(), when it is called, should update its observers.

    ```
    setChanged() {
        changed = true
    }

    notifyObservers (Object arg) {
        if (changed) {
                for every observer on the list {
                        call update(this, arg)
                }
        }
    }

    notifyObservers () {
        notifyObservers(null)
    }
    ```

24. Reworking the Weather Station with the built-in support: First, let's rework WeatherData to use java.util.Observable:

```
import java.util.Observable;
import java.util.Observer;

public class WeatherData extends Observable {
        private float temperature;
        private float humidity;
        private Subject weatherData;

       public WeatherData() { }

        public void measurementsChanged() {
                setChanged();
            notifyObservers();
        }

        public void setMeasurements(float temperature, float humidity, float pressure) {
            this.temperature = temperature;
            this.humidity = humidity;
            this.pressure = pressure;
            measurementsChanged();
        }

       public float getTemperature() {
                return temperature;
       }

       public float getHumidity() {
                return humidity;
       }

       public float getPressure() {
                return pressure;
       }
}
```

25. Now let’s rework the CurrentConditionsDisplay:

```
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
       Observable observable;
       private float temperature;
       private float humidity;
```

```
        public CurrentConditionsDisplay(Observable observable) {
                this.observable = observable;
                observable.addObserver(this);
        }

        public void update(Observable obs, Object arg) {
                if (obs instanceof WeatherData) {
                        WeatherData weatherData = (WeatherData) obs;
                        this.temperature = weatherData.getTemperature();
                        this.humidity = weatherData.getHumidity();
                        display();
                }
        }

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

26. Unfortunately, the java.util.Observable implementation has a number of problems that limit its usefulness and reuse:
    - As you have noticed, Observable is a class, not an interface, and worse, it does not even implement an interface. Since Observable is a class, you have to subclass it. That means you can’t add on the Observable behavior to an existing class that already extends another superclass. This limits its reuse potential.
    - Because, there isn’t an Observable interface, you cannot even create your own implementation that plays well with Java’s built-in Observer API.
    - If you look at the Observable API, the setChanged() method is protected. This means, you cannot call setChanged() unless you have subclassed Observable. You cannot even create an instance of the Observable class and compose it with your own objects, you have to subclass it. The design violates a second design principle here: *favor composition over inheritance.*