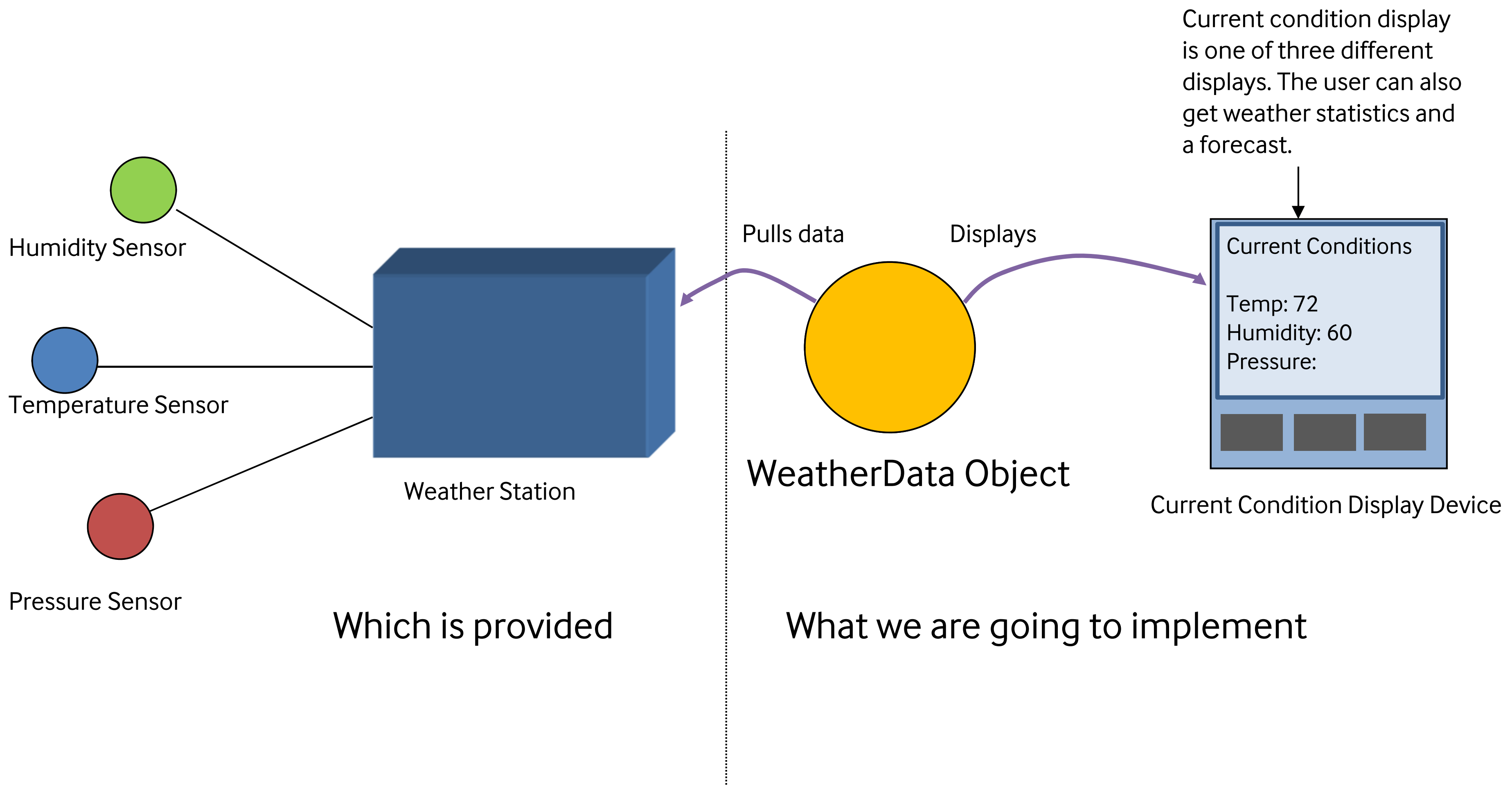
Observer Pattern

Observer Pattern

- Also known as
  - Publish/Subscribe model
- Purpose
  - Lets one or more objects be notified of state changes in other objects within the system.
- Use When
  - Loose coupling is needed for communications
  - State changes in one or more objects should trigger behavior in other objects
  - Broadcasting capabilities are required.
  - An understanding exists that objects will be blind to the expense of notification.

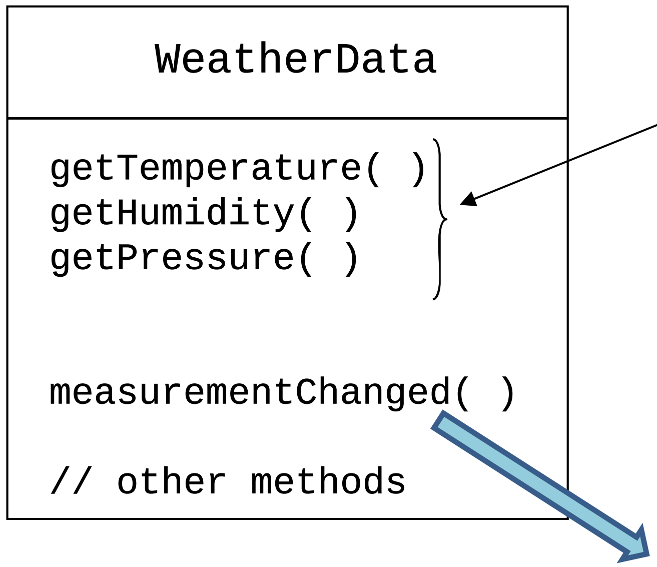
The Application Overview


Current condition display is one of three different displays. The user can also get weather statistics and a forecast.
Humidity Sensor
Temperature Sensor
Pressure Sensor
Weather Station
Pulls data
Displays
WeatherData Object
Current Conditions
Temp: 72
Humidity: 60
Pressure:
Current Condition Display Device
Which is provided
What we are going to implement

Create an application using the WeatherData object to update three displays.

Figure from [HF]

The WeatherData class



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

We don’t care how these variables are set; the WeatherData object knows how to get updated information from the Weather Station

```
/* This method gets called whenever the
 * measurements have been updated. * /

 public void measurementsChanged( ) {
     // Your code goes here
 }
```

What do we know?

- Display elements that use the weather data:
  - Current conditions display
  - Statistics display
  - Forecast display
- They must be updated each time WeatherData has new measurements
- 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.
  - Currently, we know about only the initial three display types: current conditions, statistics, and forecast

The First Attempt

```
public class WeatherData {

   // instance variable declarations

  public void measurementsChanged() {

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

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

        // other WeatherData methods here
}
```

The First Attempt

```
public class WeatherData {

   // instance variable declarations

  public void measurementsChanged() {

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

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

        // other WeatherData methods here
}
```

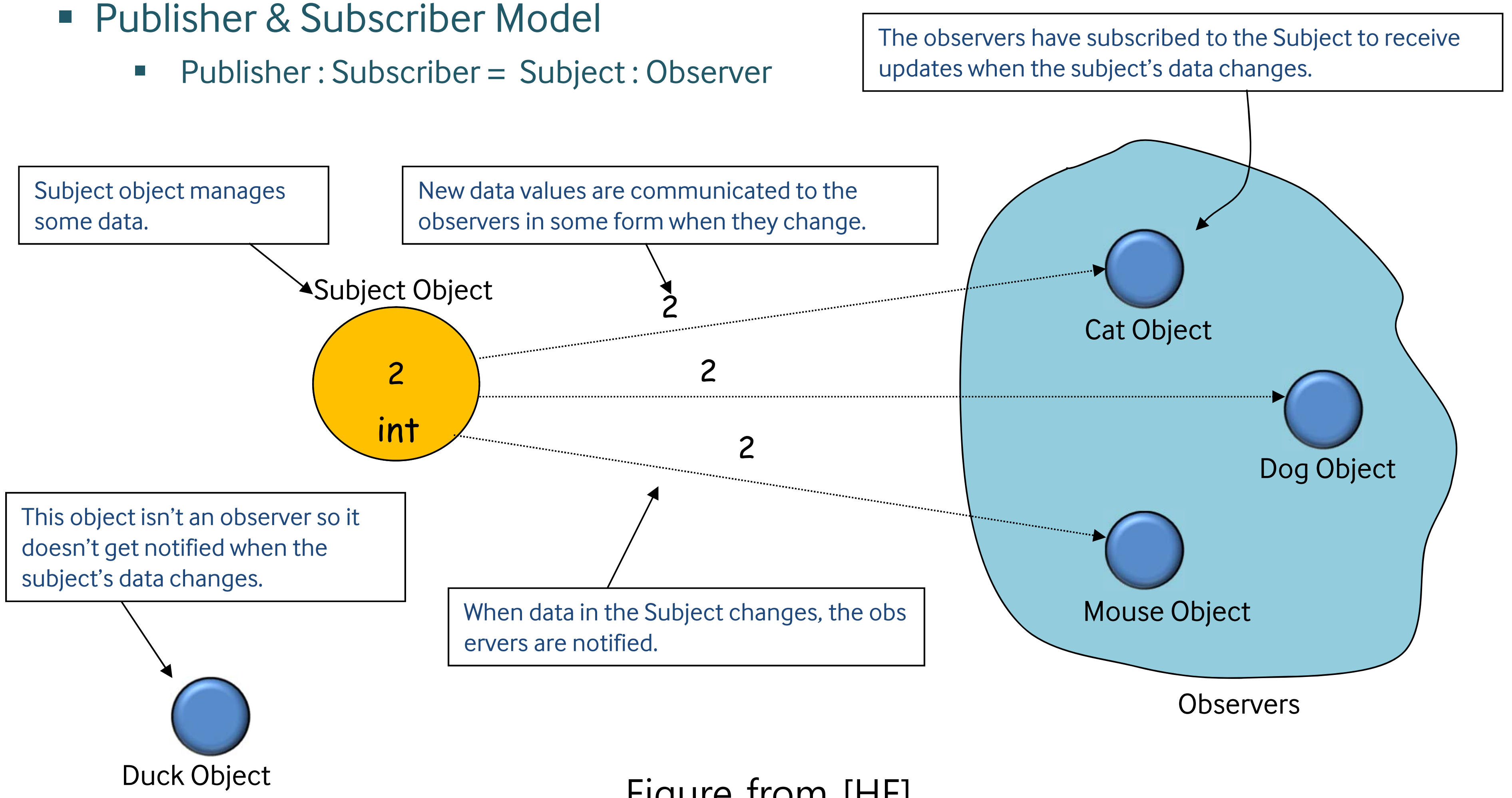
By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

Meet the Observer Pattern

- A news paper **publisher** goes into business and begins publishing newspapers
- You **subscribe** to a particular publisher
  - Every time there's new edition it gets delivered to you. As long as you remain a subscriber; you get new newspapers
- You **unsubscribe** when you don't want papers anymore
  - They stop delivery
- While the publisher remains in business, people, hotels, airlines and other business **constantly subscribe and unsubscribe** to the newspaper.

Observer Pattern

- Publisher & Subscriber Model
  - Publisher : Subscriber = Subject : Observer



The observers have subscribed to the Subject to receive updates when the subject's data changes.
Subject object manages some data.
New data values are communicated to the observers in some form when they change.
Subject Object
2
int
2
2
2
Cat Object
Dog Object
Mouse Object
Observers
This object isn't an observer so it doesn't get notified when the subject's data changes.
When data in the Subject changes, the obs ervers are notified.
Duck Object

Figure from [HF]

Duck wants to subscribe

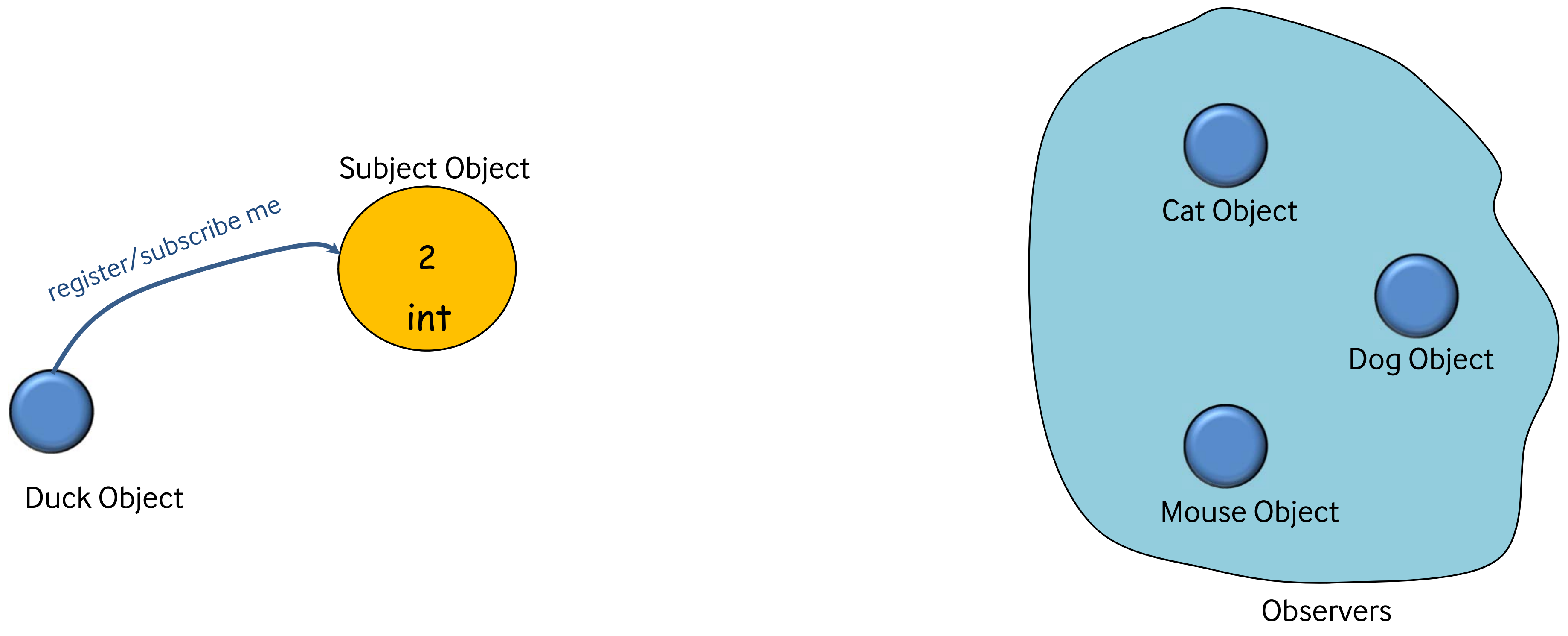


Figure from [HF]

Duck is now an observer, too

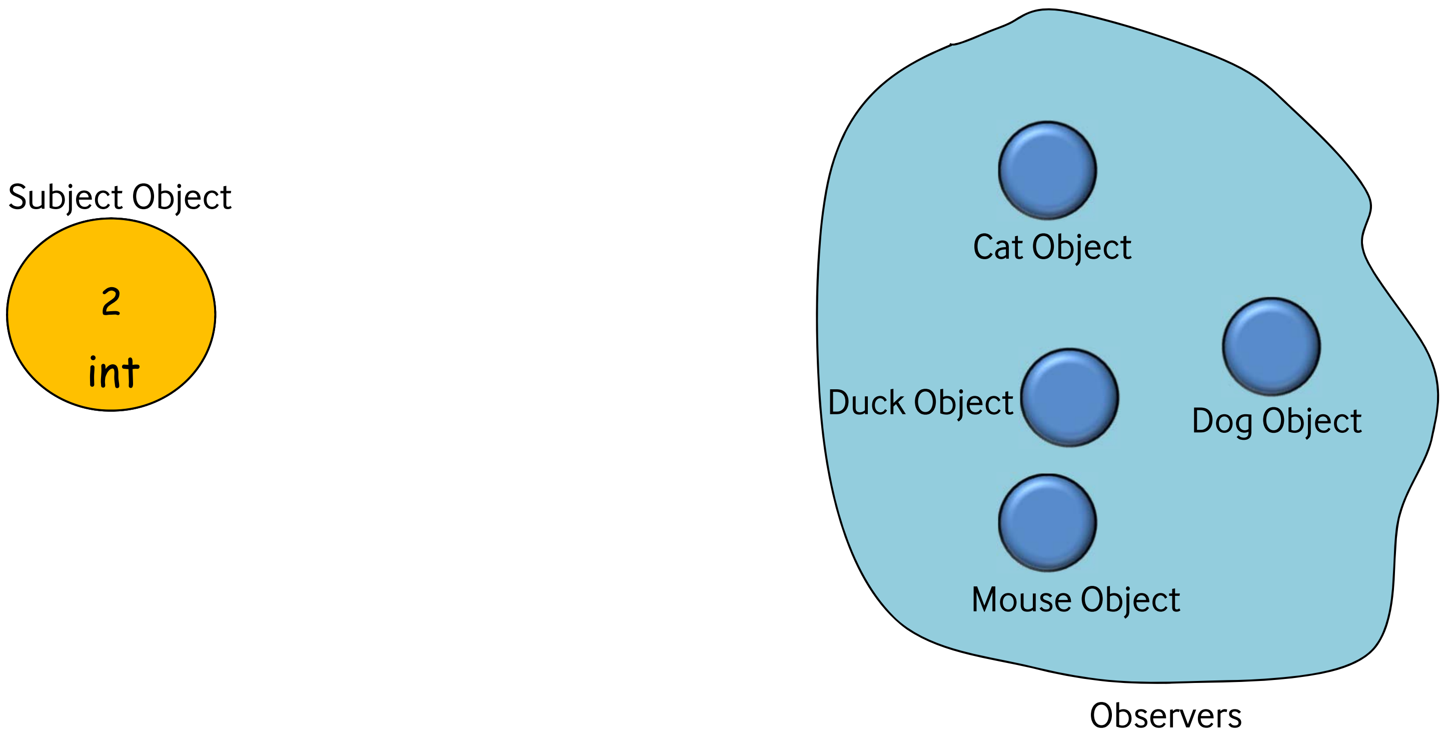


Figure from [HF]

Notifying the observers

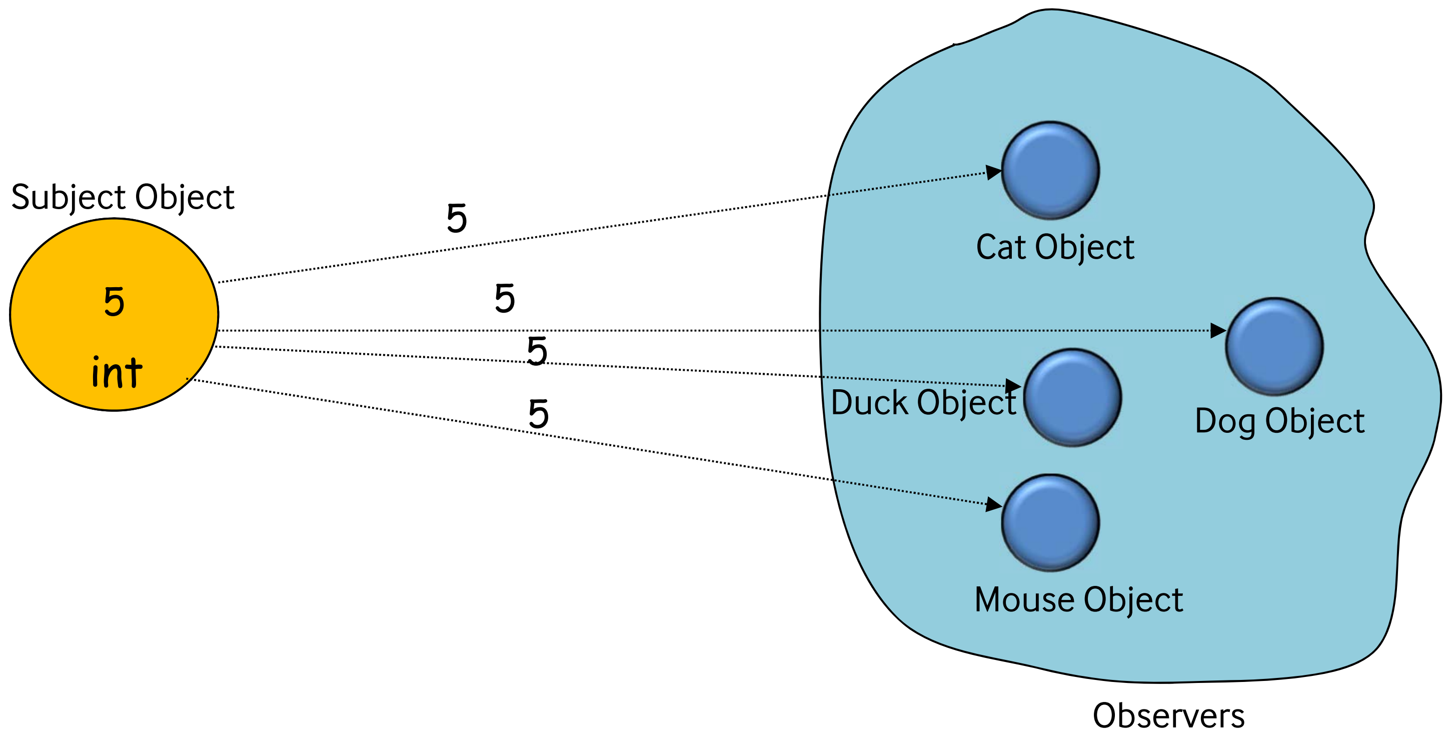


Figure from [HF]

Mouse wants to unsubscribe

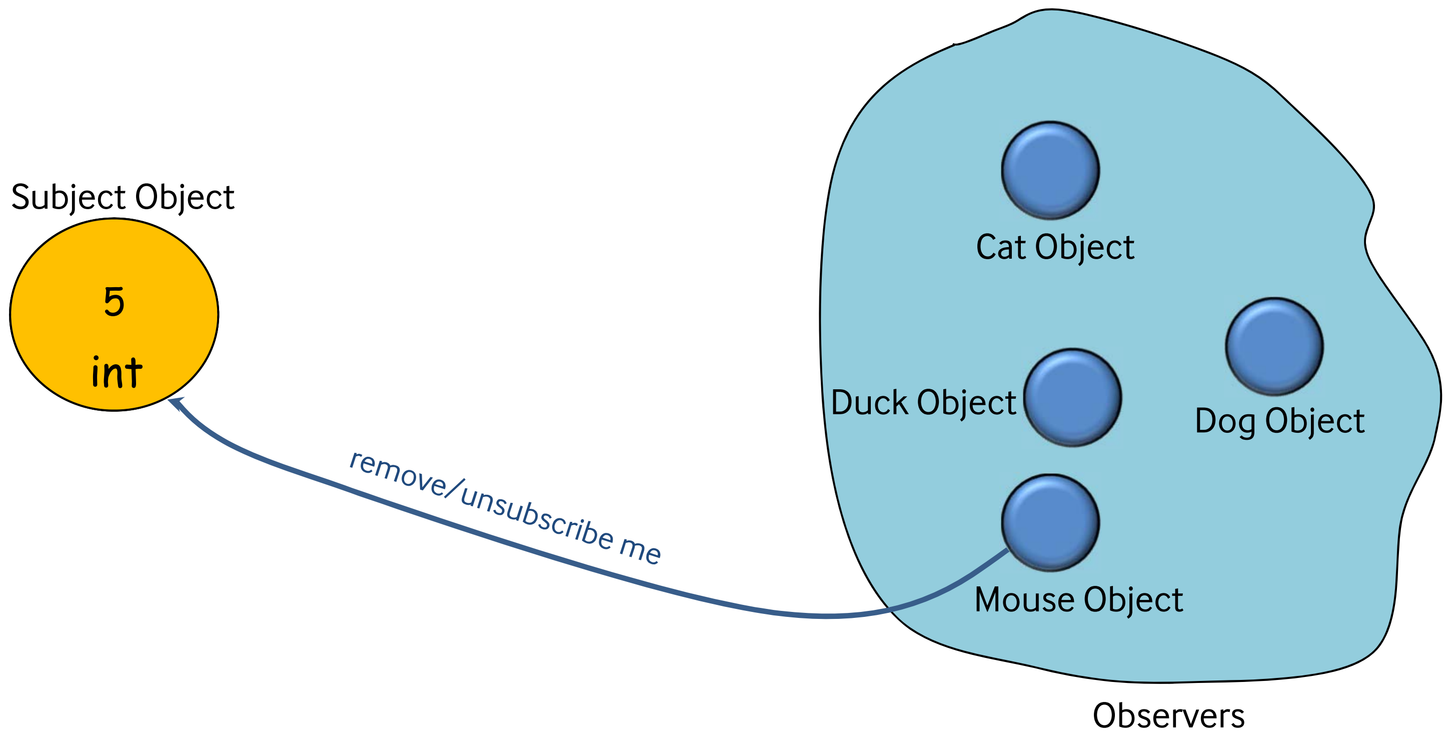


Figure from [HF]

Mouse wants to unsubscribe

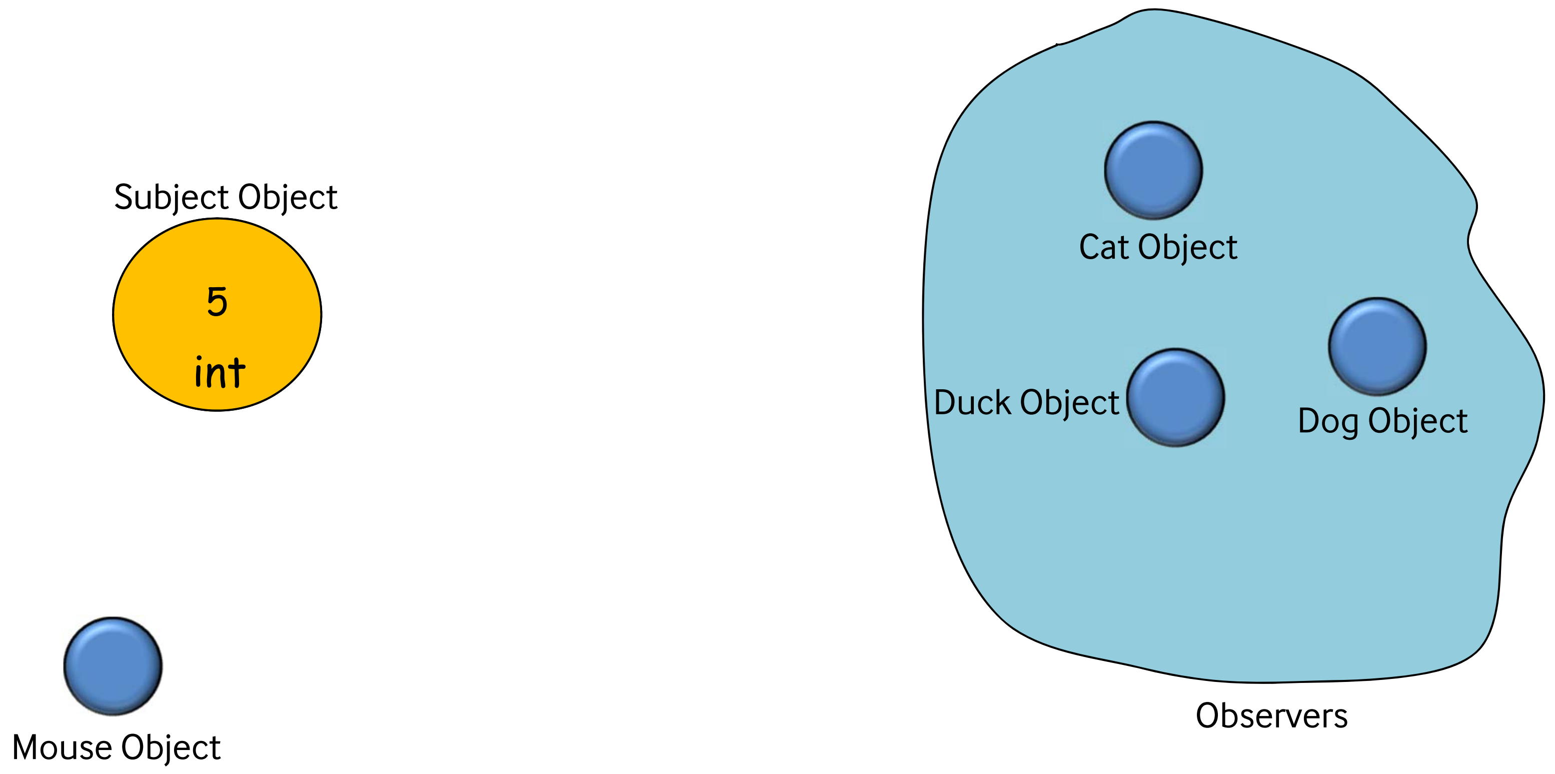


Figure from [HF]

Notifying the observers

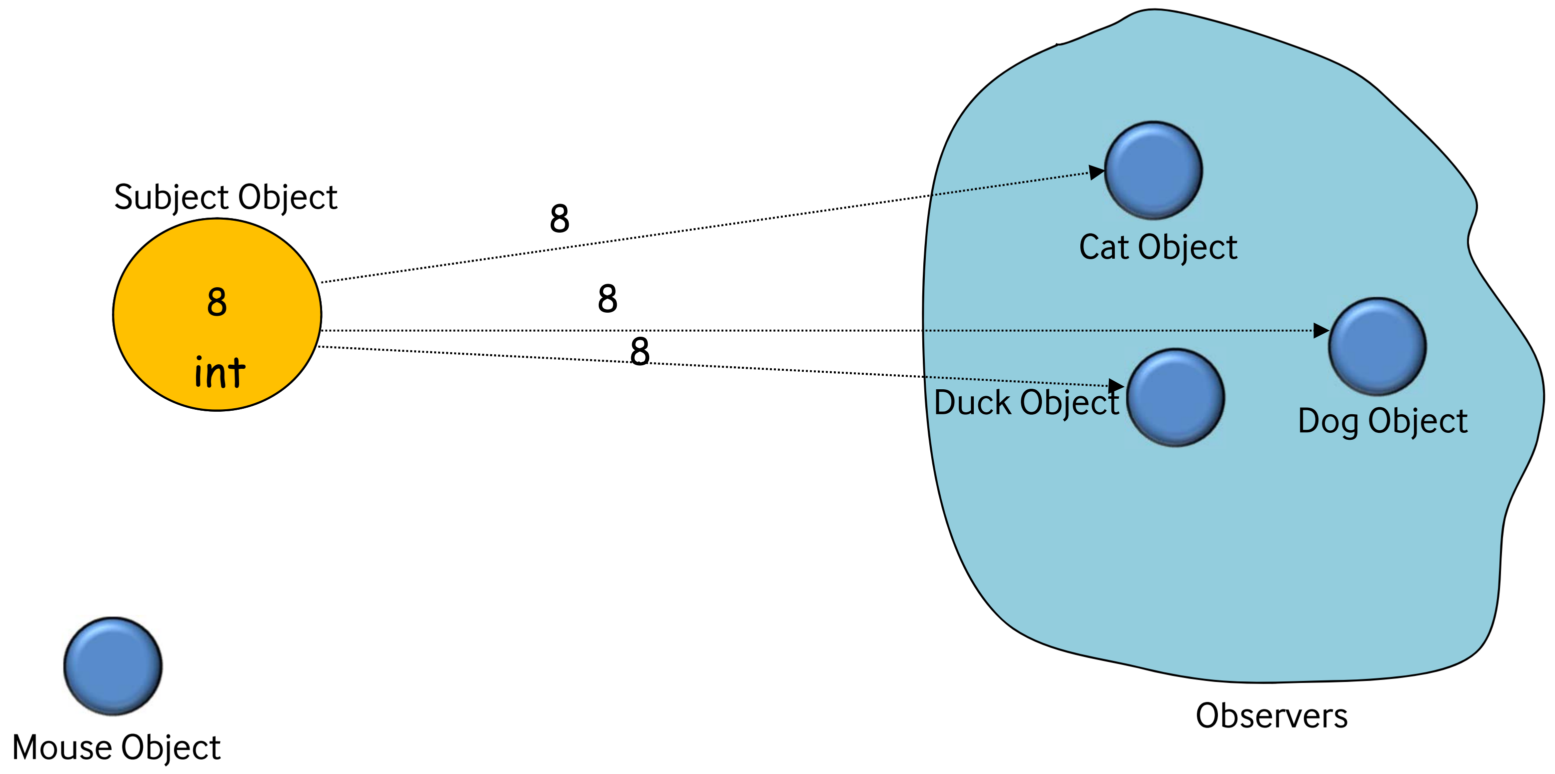


Figure from [HF]

The Observer Pattern Defined

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.

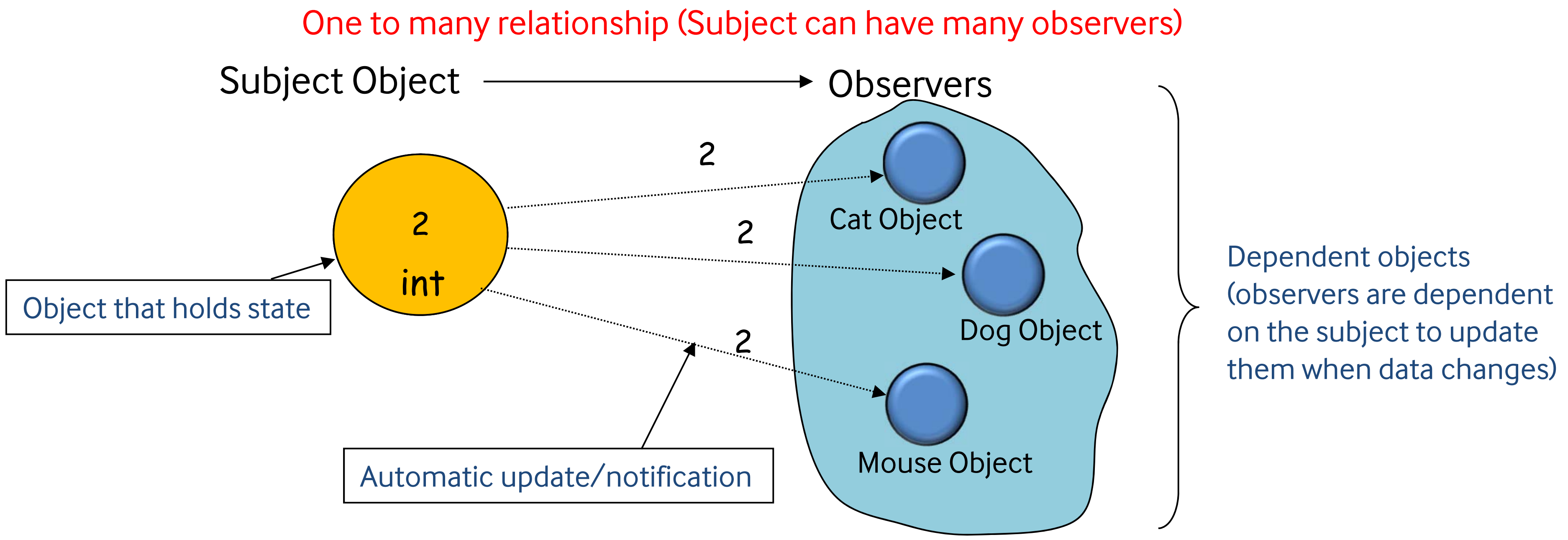

One to many relationship (Subject can have many observers)
Subject Object
Observers
2
2
2
2
int
Cat Object
Dog Object
Mouse Object
Object that holds state
Automatic update/notification
Dependent objects (observers are dependent on the subject to update them when data changes)

Figure from [HF]

Class Diagram for Observer Pattern

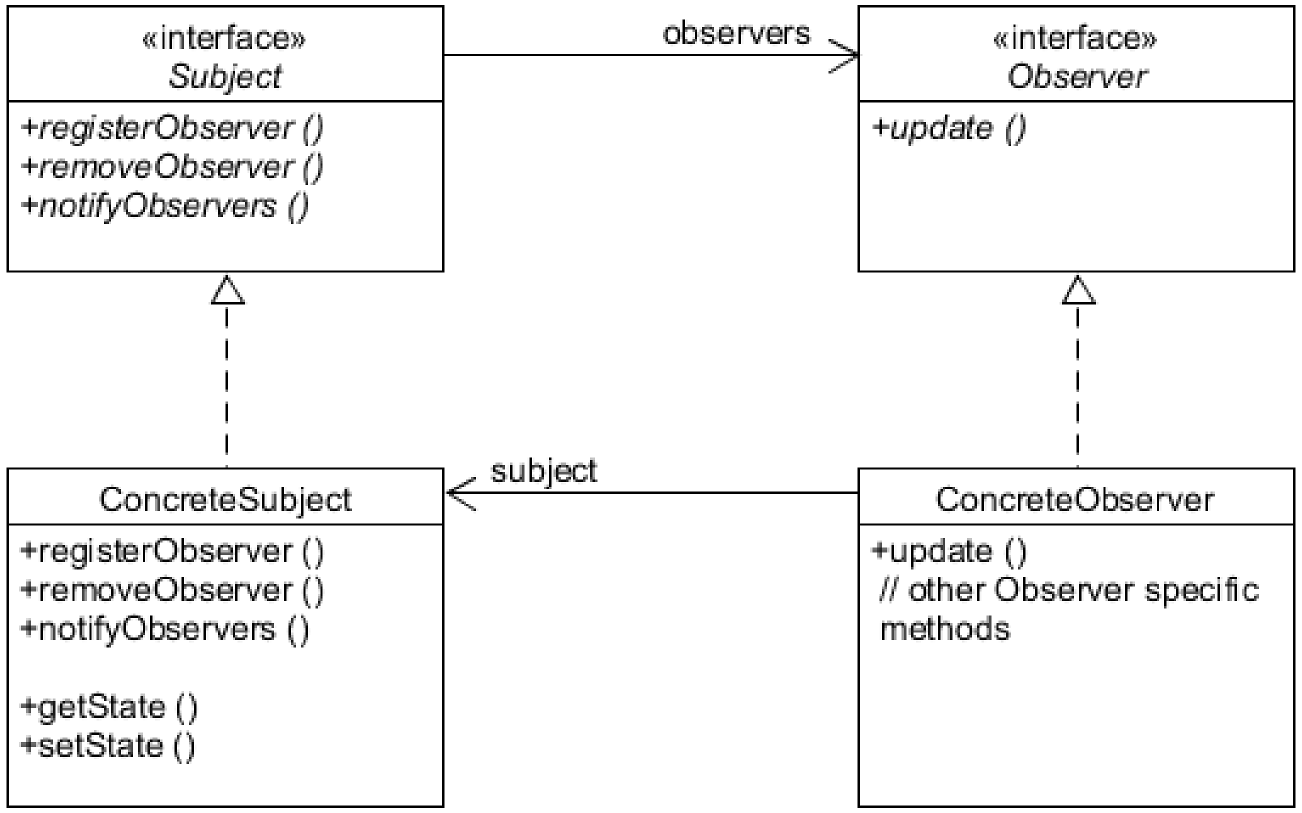


Figure from [HF]

The Power of Loose Coupling

- Principle of loose coupling
  - Strive for loosely coupled designs between object that interact

- When two objects are **loosely coupled**, they can interact, but have very **little knowledge of each other**
- Observer Pattern provides an object design where subjects and observers are loosely coupled

Designing the Weather Station

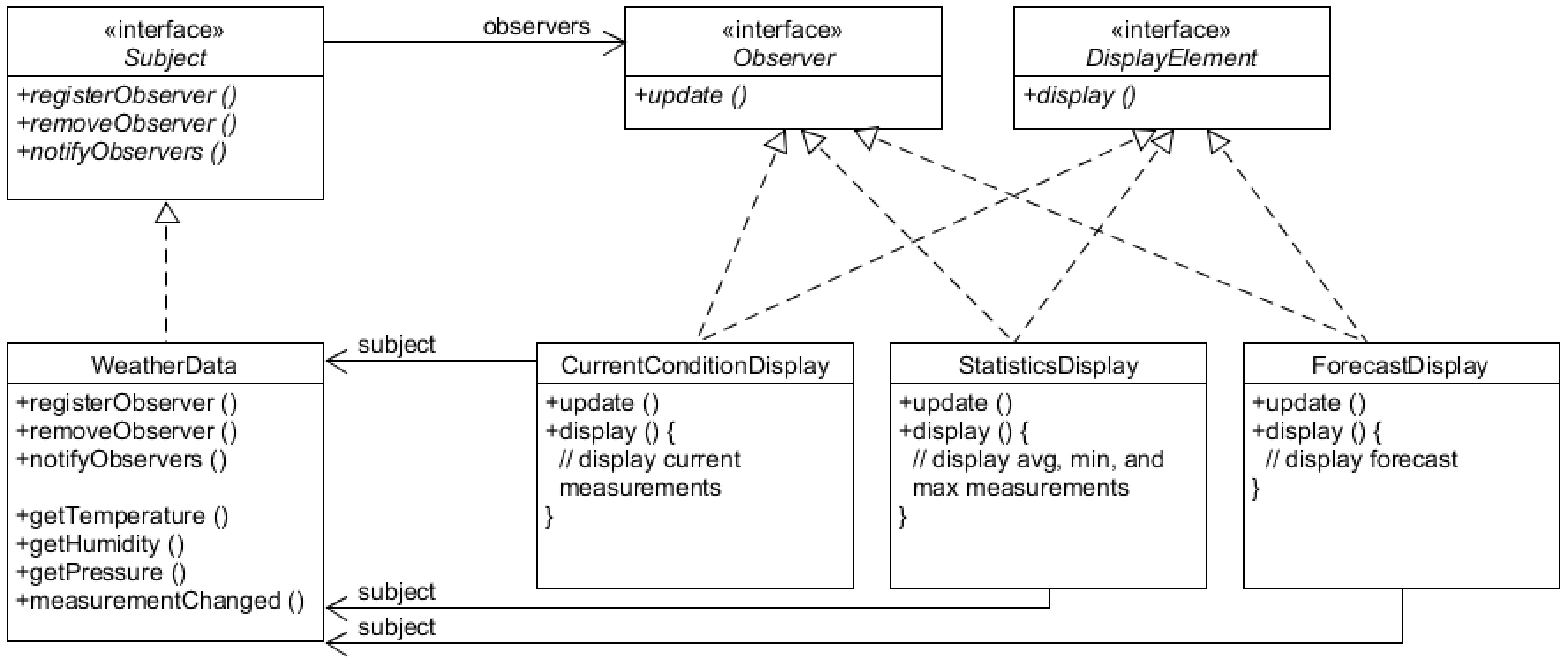


Figure from [HF]

Preparing 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();
}
```

Implementing 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);
                }
        }
```

Implementing WeatherData (cont’)

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

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

        // other WeathreData methods here
}
```

Implementing CurrentConditionDisplay

```
public class CurrentConditionDisplay implements Observer, DisplayElement {
        private float temperature;
        private float humidity;
        private Subject weatherData;

        public CurrentConditionDisplay(Subject weatherData) {
                this.weathreData = 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”);
        }
}
```

Implementing the Weather Station and Output

```
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);
        }
}
```

```
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
```

Using Java's Official Observable Class

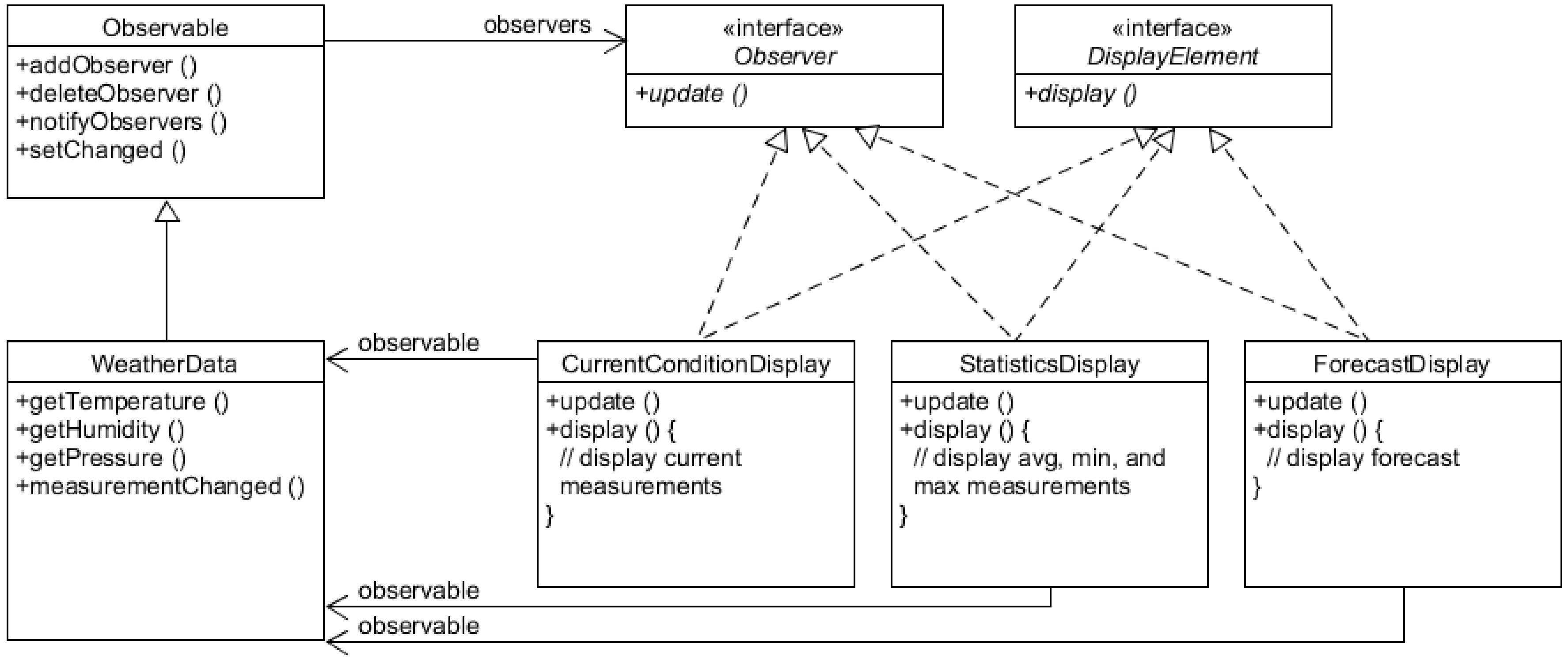


Figure from [HF]

How to Use Observable Class

- Observable (Publisher) side
  - To send notifications
    - Call SetChanged()
    - Call either notifyObservers() or notifyObservers(Object arg)

- Observer (Subscriber) side
  - To become an observer
    - Implement java.util.Observer interface
  - To subscribe
    - Call addObserver() on any Observable object
  - To unsubscribe
    - Call deleteObserver()
  - To get updated
    - Implement update(Observable o, Object arg)

Internals of setChanged()

```
setChanged() {
   changed = true;
}

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

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

Why is the setChanged ( ) necessary?

- setChanged ( ) method is there to give you more flexibility
    - Optimize the notifications
    - Example:
        - If the sensor reading is very sensitive, you may not want to update the observers with every minor fluctuation
        - setChanged ( ) allows you to control the notification points
- Other relevant methods in Observable
    - clearChanged ( )
    - hasChanged ( )

Observable side

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

public class WeatherData extends Observable {
          private float temperature;
          private float humidity;
          private float pressure;

          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;
          }
}
```

Observer side

```
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 = observerble;
                    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”);
          }
}
```

Summary

- Principle of loose coupling
  - Strive for loosely coupled designs between objects that interact
- Observer Pattern
  - Observer defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically