CSC40232: SOFTWARE ENGINEERING

Professor: Jane Cleland-Huang

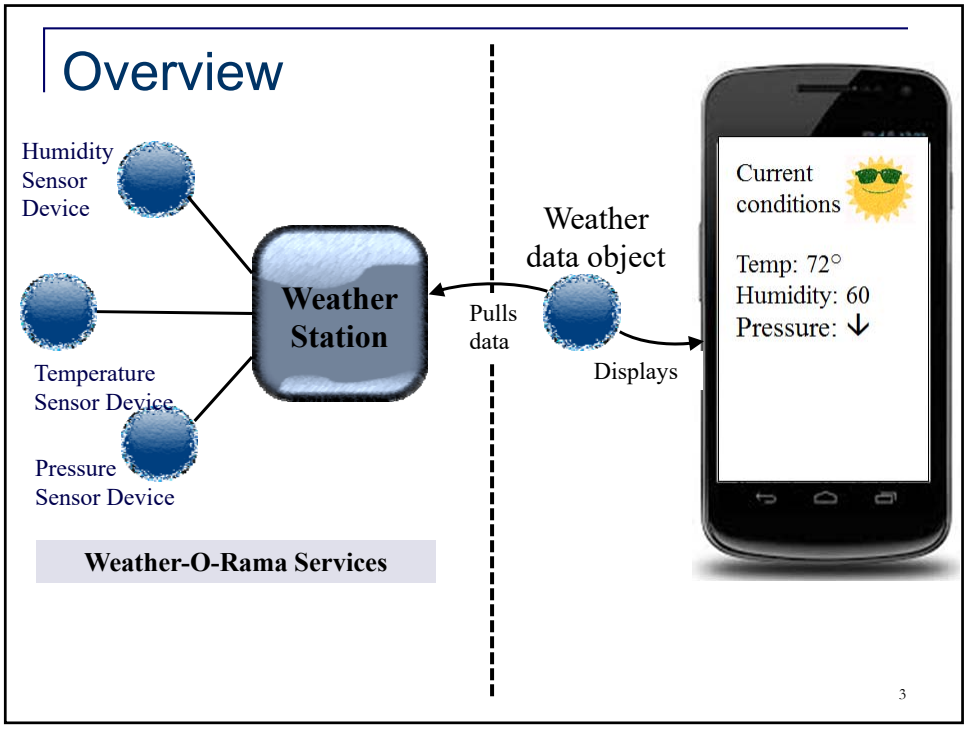
Lecture 3: Observer Pattern
Wednesday, January 18th
sarec.nd.edu/courses/SE2017

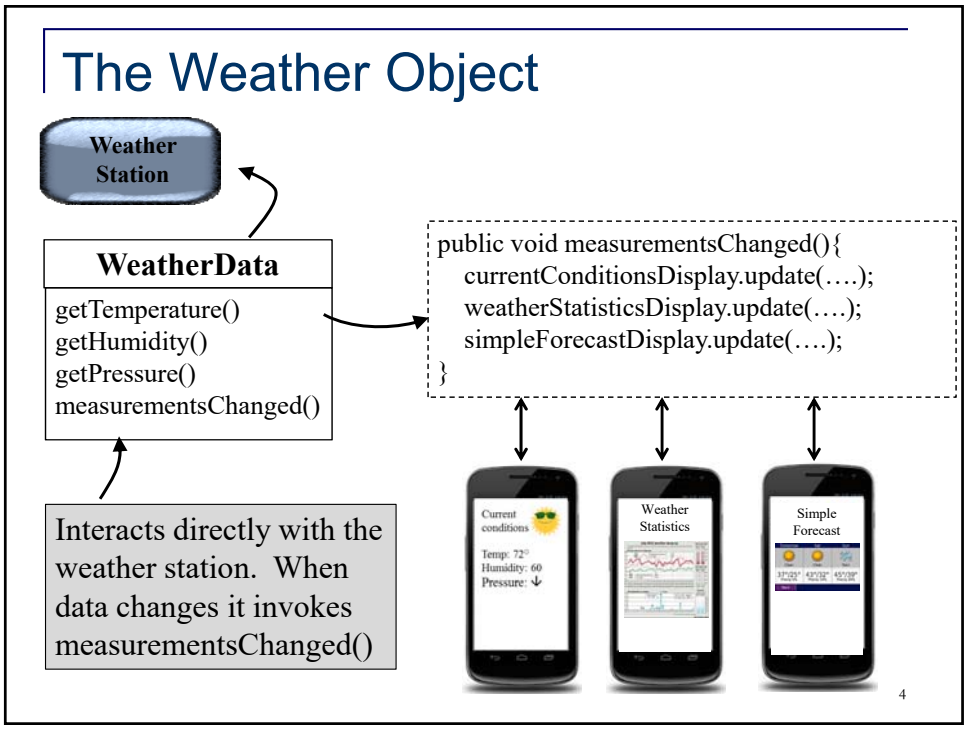
Department of Computer Science and Engineering

A Weather Station

- Create an application that initially provides three display elements:
  - Current conditions
  - Weather statistics
  - Simple forecast
- Also provide an API so that other developers can write their own weather displays.

2


Overview
Humidity Sensor Device
Temperature Sensor Device
Pressure Sensor Device
Weather Station
Weather-O-Rama Services
Pulls data
Weather data object
Displays
Current conditions
Temp: 72°
Humidity: 60
Pressure: ↓
3


The Weather Object
Weather Station
WeatherData
getTemperature()
getHumidity()
getPressure()
measurementsChanged()
Interacts directly with the weather station. When data changes it invokes measurementsChanged()
public void measurementsChanged(){
currentConditionsDisplay.update(….);
weatherStatisticsDisplay.update(….);
simpleForecastDisplay.update(….);
}
Current conditions
Temp: 72°
Humidity: 60
Pressure: ↓
Weather Statistics
Simple Forecast
4

Misguided Solution

```
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
}
```

Grab the most recent measuremets by calling the WeatherData's getter methods (already implemented).

Now update the displays...

Call each display element to update its display, passing it the most recent measurements.

1. We are coding to concrete implementations, not interfaces. True

2. For every new display we need to alter code. True

3. We have no way to add (or remove) display elements at runtime. True

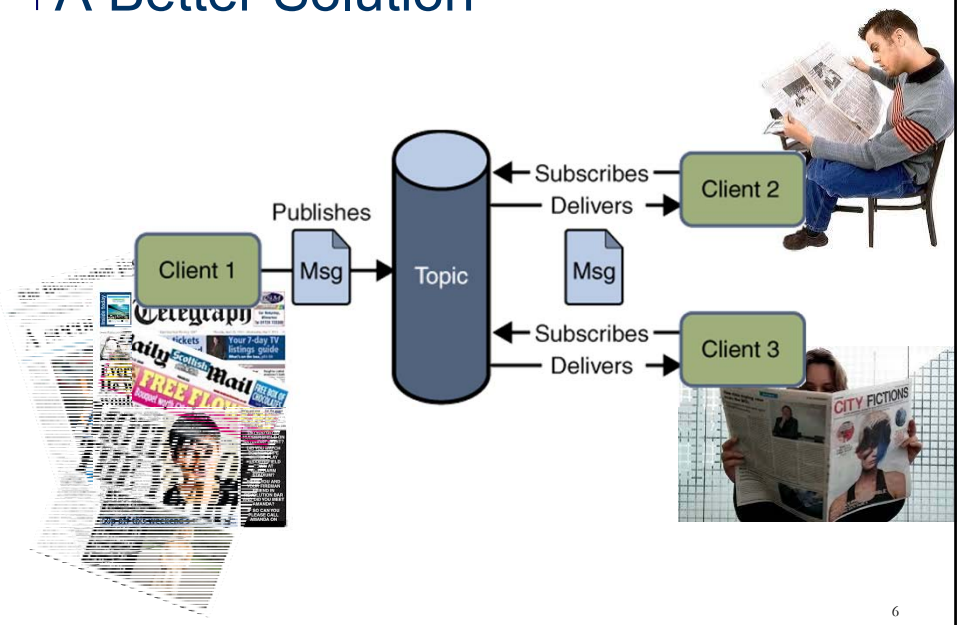
4. The display elements don't implement a common interface. Looks like they might.

5. We haven't encapsulated the part that changes. True

6. We are violating encapsulation of the weather class. Not really

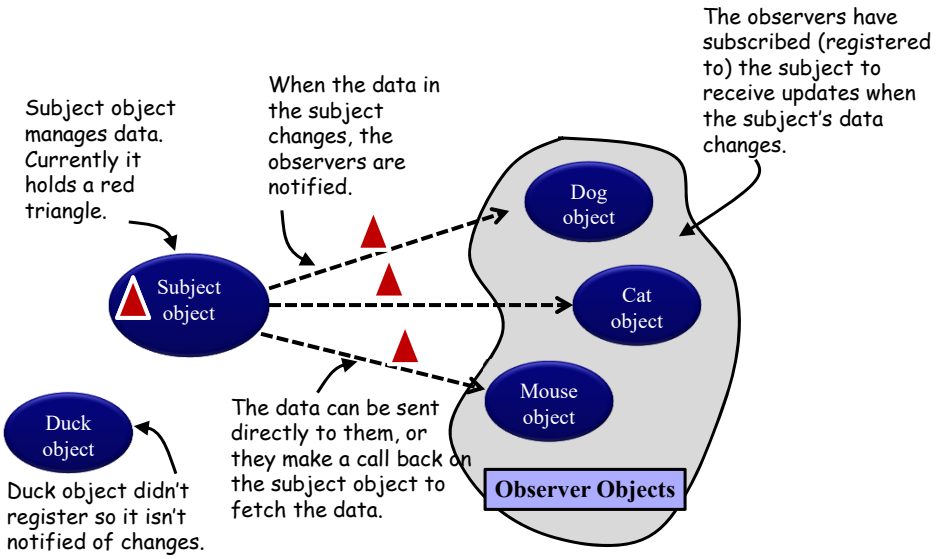
5

A Better Solution



6

The Observer Pattern



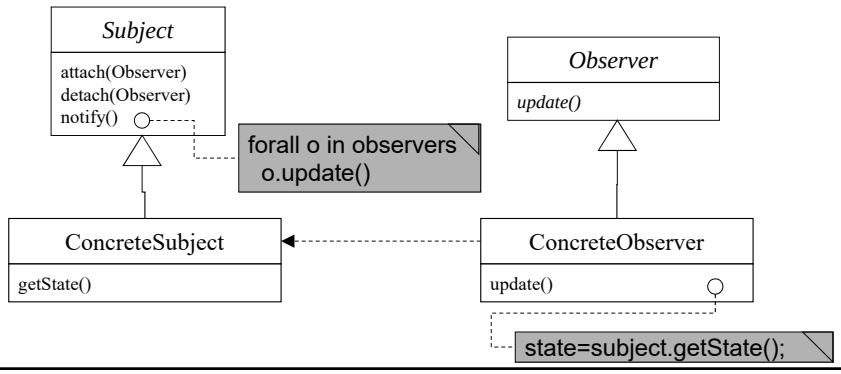
7

Observer

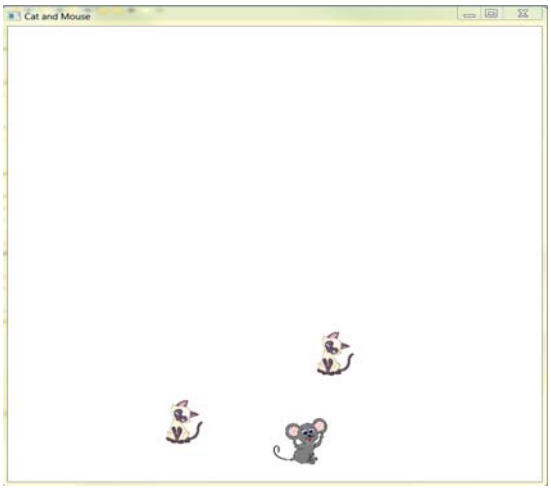
- Many-to-one dependency between objects
- Use when there are two or more views on the same "data"
- aka "Publish and subscribe" mechanism
- Choice of "push" or "pull" notification styles

Defines a one-to-many relationship between a set of objects.

When the state of one object changes, all of its dependents are notified.



Cat and Mouse



In this simple animation the mouse moves around the canvas – sometimes disappearing down an invisible mouse-hole and then reappearing elsewhere.

The cats **observe** the current position of the mouse and try to follow him around the canvas.

❶ Traditional (from scratch) implementation.

❷ Java inbuilt implementation

Disclaimer: The animation is only for display in class. The code I share with you is non-GUI based. Next lecture we will start working with GUIs.

9

```java
public interface Observer {
    public void notify(Subject s);
}
```

Observer interface and **concrete observer class** implemented '**from scratch**' in java.

❶ Observer interface requires all concrete classes to implement a notify(Subject s) method.

❷ The notify(Subject s) operation is called by Subject class when the subject changes.

❸ The concrete observer (cat) makes a call-back onto the subject (mouse) object to get the data it wants. i.e. the mouse position.

```java
public class Cat implements Observer{
    Point mousePosition;   // Current position of the mouse
    Point catPosition;     // Current position of this cat
    Random rand = new Random();

    @Override
    public void notify(Subject s) {
       if(s instanceof Mouse){
           mousePosition = ((Mouse)s).getPosition();
           moveCat();
       }
    }

    public Cat(){
        catPosition = new Point(rand.nextInt(800), rand.nextInt(800));
    }

    public void moveCat(){
        if(rand.nextInt(2)==1){  // Slow down the cat.
            if (catPosition.x - mousePosition.x < 0)
                catPosition.x++;
            else
                catPosition.x--;

            if (catPosition.y - mousePosition.y < 0)
                catPosition.y++;
            else
                catPosition.y--;
        }
        System.out.println("Cat: " + catPosition.x + " " + catPosition.y);
    }
}
```

```java
public class Mouse implements Subject {
    List<Observer> observers = new LinkedList<Observer>();
    Point myPosition = new Point(400,500);
    Point targetPosition;
    Random rand;

    public Mouse(){
        targetPosition = new Point(0,0);
        setTargetPosition();
    }

    private void setTargetPosition(){
        rand = new Random();
        targetPosition.x = rand.nextInt(800);
        targetPosition.y = rand.nextInt(800);
    }

    private void setMousePosition(){
        rand = new Random();
        myPosition.x = rand.nextInt(800);
        myPosition.y = rand.nextInt(800);
    }

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

    @Override
    public void removeObserver(Observer o) {
        if(observers.contains(o))
            observers.remove(o);
    }
```

```java
public interface Subject {

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

Subject interface
and **concrete subject class** implemented '**from scratch**' in java.

Subject interface requires all concrete classes to implement three methods:

- Register observers
- Remove observers
- Notify observers

```java
@Override
public void notifyObservers() {
    for(Observer catObserver: observers){
        catObserver.notify(this);
    }
}

public Point getPosition(){
    return myPosition;
}

public void mouseMove(){

    while (Math.abs(myPosition.x-targetPosition.x) +
            Math.abs(myPosition.y-targetPosition.y) < 20){
        setTargetPosition();
        setMousePosition();
    }

    if (myPosition.x - targetPosition.x < 0)
       myPosition.x++;
    else
       myPosition.x--;

    if (myPosition.y - targetPosition.y < 0)
        myPosition.y++;
    else
        myPosition.y--;

    System.out.println("Mouse: " + myPosition.x + " " + myPosition.y);
    notifyObservers();
}
```

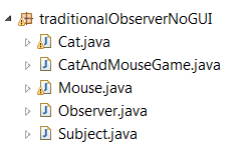
Here we see notification at work:

❶ The mouseMove method is executed, the mouse moves, its state changes and it invokes **notifyObservers()**

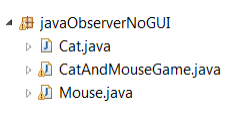
❷ All currently registered observers are notified of the change. "this" subject object is passed to them and used for call-backs. (See the cat example a few slides back!)

Sketch it!!

Java Inbuilt Observer



Classes in my Eclipse package explorer when I implement the solution from scratch.



Classes in my Eclipse package explorer when I implement the solution using Java Inbuilt observer.

We can use the Java inbuilt Observer Pattern

<< *Observer*>>	https://docs.oracle.com/javase/8/docs/api/java/util/Observer.html
Observable	https://docs.oracle.com/javase/8/docs/api/java/util/Observable.html

13

Java: Observer Pattern

Java provides an inbuilt Observer Pattern in the form of:

- **An Observable superclass**
  Extend this class if you want something to be observed. i.e. this plays the role of the subject.

- **An Observer interface**
  Implement this interface if you want to register this object as an observer.

14

Java Inbuilt Observer

```
public class Cat implements Observer{
    Point mousePosition;    // Current position of the mouse
    Point catPosition;      // Current position of this cat
    Random rand = new Random();

    @Override
    public void notify(Subject s) {
       if(s instanceof Mouse){
           mousePosition = ((Mouse)s).getPosition();
           moveCat();
       }
    }
```

"Before" (using the custom-built from-scratch Observer interface)

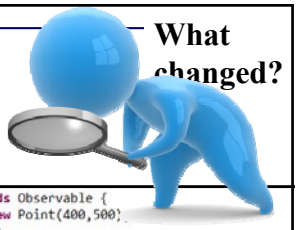
…and "after" (using Java's inbuilt Observer interface).

```
public class Cat implements Observer{
    Point mousePosition;    // Current position of the mouse
    Point catPosition;      // Current position of this cat
    Random rand = new Random();

    @Override
    public void notify(Subject s) {
       if(s instanceof Mouse){
           mousePosition = ((Mouse)s).getPosition();
           moveCat();
       }
    }
```

What changed?

Java Inbuilt Observer



What changed?

From Scratch

```
public class Mouse implements Subject {
    List<Observer> observers = new LinkedList<Observer>();
    Point myPosition = new Point(400,500);
    Point targetPosition;
    Random rand;

    public Mouse(){
    private void setTargetPosition(){
    private void setMousePosition(){
    public void registerObserver(Observer o) {
    public void removeObserver(Observer o) {
    public void notifyObservers() {
    public Point getPosition(){
    public void mouseMove(){

        while (Math.abs(myPosition.x-targetPosition.x) +
               Math.abs(myPosition.y-targetPosition.y) < 20){
            setTargetPosition();
            setMousePosition();
        }

        if (myPosition.x - targetPosition.x < 0)
           myPosition.x++;
        else
           myPosition.x--;

        if (myPosition.y - targetPosition.y < 0)
            myPosition.y++;
        else
            myPosition.y--;

        System.out.println("Mouse: " + myPosition.x + " " + myPosit
        notifyObservers();
    }
}
```

```
public CatAndMouseGame(){
    cat = new Cat();
    cat2 = new Cat();
    mouse = new Mouse();
    mouse.registerObserver(cat);
    mouse.registerObserver(cat2);
}
```

Java inbuilt

```
public class Mouse extends Observable {
    Point myPosition = new Point(400,500);
    Point targetPosition;
    Random rand;

    public Mouse(){
    private void setTargetPosition(){
    private void setMousePosition(){
    public Point getPosition(){

    public void mouseMove(){
        while (Math.abs(myPosition.x-targetPosition.x)
               + Math.abs(myPosition.y-targetPosition.y) < 20){
            setTargetPosition();
            setMousePosition();
        }

        if (myPosition.x - targetPosition.x < 0)
           myPosition.x++;
        else
           myPosition.x--;

        if (myPosition.y - targetPosition.y < 0)
            myPosition.y++;
        else
            myPosition.y--;

        setChanged();
        notifyObservers();
    }
}
```

```
public CatAndMouseGame(){
    cat = new Cat();
    cat2 = new Cat();
    mouse = new Mouse();
    mouse.addObserver(cat);
    mouse.addObserver(cat2);
}
```

Casting

Upcasting:

- Java permits an object of a subclass type to be treated as an object of any superclass type.
- Upcasting is done automatically.
- Supports polymorphism
- Example:
  List<Observer> observers = **new LinkedList<Observer>();**

Downcasting:

- Downcasting must be manually performed by the programmer
- Only sensible if the object being cast is an instance of that type
- We need to test the type before downcasting using:
  if ([object] instanceof [class name]) do something;
- Example:

```
@Override
public void notify(Subject s) {
   if(s instanceof Mouse){
       mousePosition = ((Mouse)s).getPosition();
       moveCat();
   }
}
```

Worksheet

Most likely take-home as we have a busy week planned!