Observer

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 12

Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

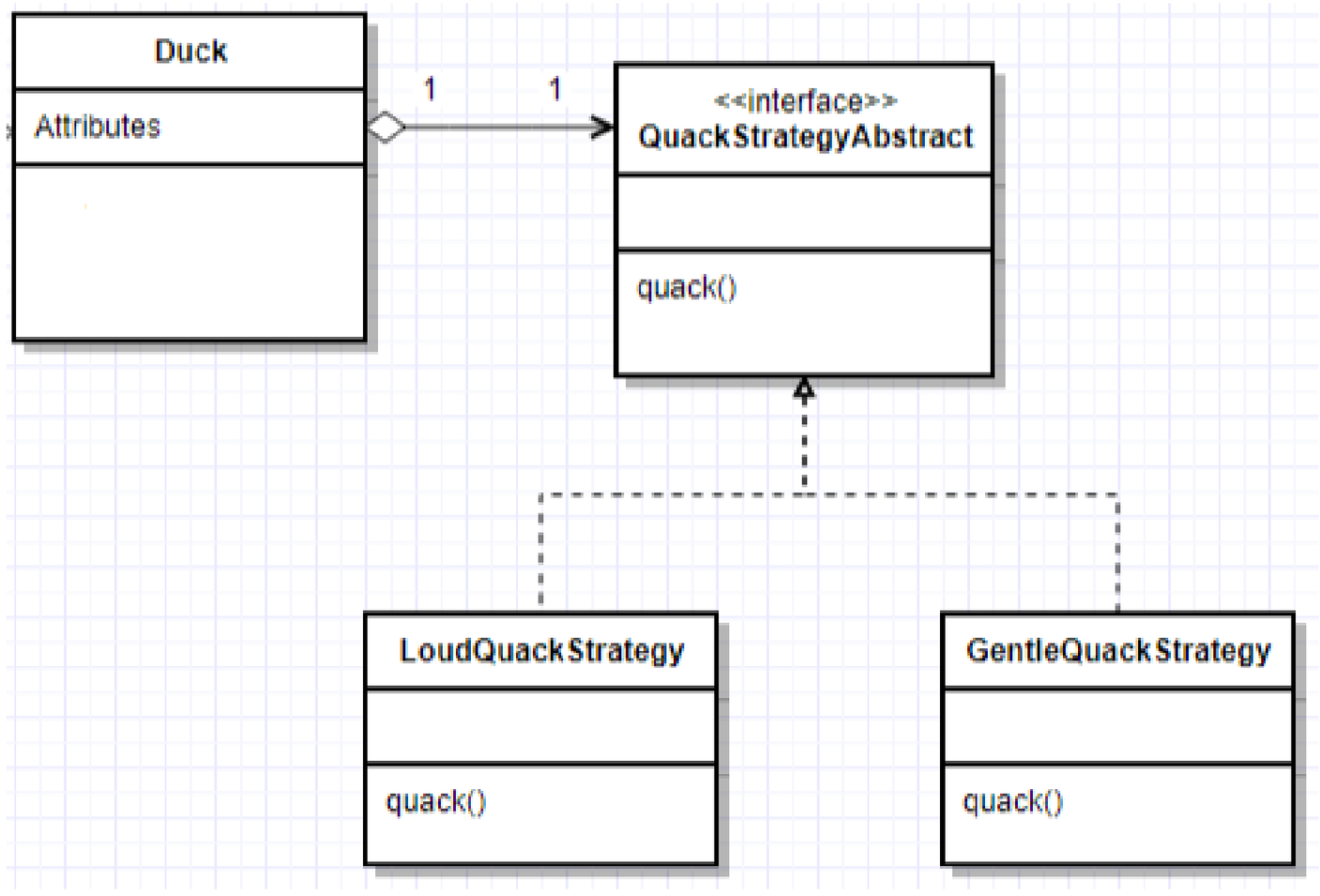
Before we start, Interface vs. Abstract Class

- Similar, but not the same…
- Which should you use, abstract classes or interfaces?
- Using abstract classes…
  - You're probably providing implementations for abstract methods
  - You want to share code among several closely related classes
  - You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private)
  - You want to declare non-static or non-final fields; enables you to define methods that can access and modify the state of the object to which they belong
  - **An abstract class is really an IS-A superclass (that you won't instantiate)**
- Using interfaces if any of these statements apply to your situation:
  - You expect that unrelated classes would implement your interface
    - Examples: the interfaces Comparable and Cloneable are implemented by many unrelated classes
  - You want to specify the behavior of a particular data type, but are not concerned about who implements its behavior
    - Or you use a default method to provide an alternative to the class implementing the method
    - Or you use a static method to allow a class to call the method directly from the interface
  - You want to take advantage of multiple inheritance of type (via implements)
  - All fields are public, static, and final; all methods are public
  - **An interface is more like an API, defining a reusable way to access methods**
- https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

What Strategy looks like in Python

In this example of using Strategy, we're

- moving the quack behavior out of the Duck class
- Making an interface for the abstract strategy for the behavior (actually an abstract class)
- Making concrete strategies to reference in Duck instances for the quack method implementation



Example from: https://medium.com/@sheikhsajid/design-patterns-in-python-part-1-the-strategy-pattern-54b24897233e

Python - Strategy

This module, strategy.py, will hold our strategy implementation.

We have two concrete implementations of the quack strategy, loud and gentle.

Python doesn’t directly have interfaces, but it does have an abstract class library (abc).

```
import abc  # Python's built-in abstract class library

class QuackStrategyAbstract(object):
    """You do not need to know about metaclasses.
    Just know that this is how you define abstract
    classes in Python."""
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def quack(self):
        """Required Method"""

class LoudQuackStrategy(QuackStrategyAbstract):
    def quack(self):
        print "QUACK! QUACK!!"

class GentleQuackStrategy(QuackStrategyAbstract):
    def quack(self):
        print "quack!"
```

Python - Strategy

In our main duck.py code, we'll import the strategy definitions, assign the strategies to new classes of ducks, and then instantiated ducks can use the assigned strategies. We could assign the strategy at runtime as well.

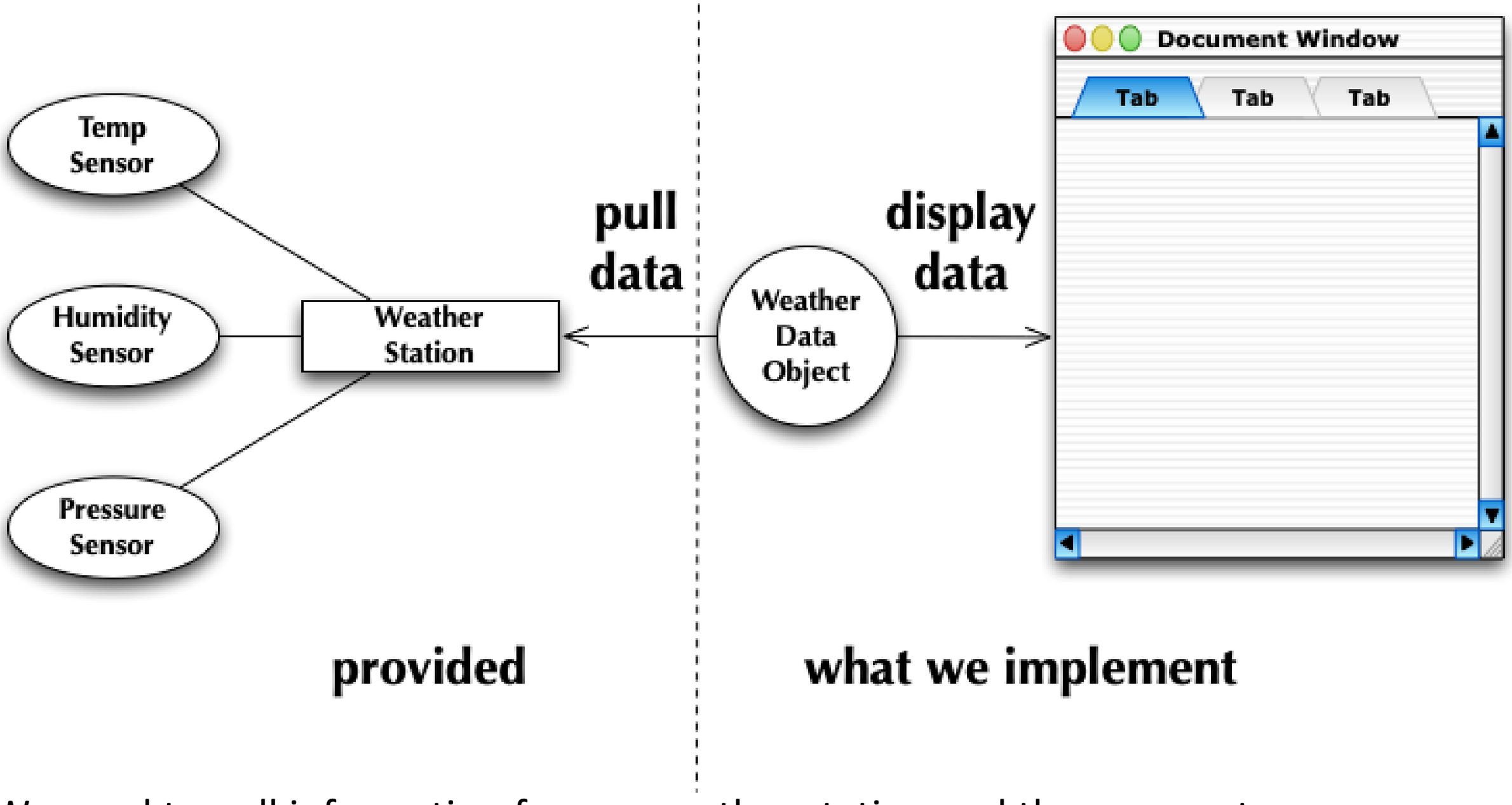
```
from strategy import LoudQuackStrategy
from strategy import GentleQuackStrategy

loud_quack = LoudQuackStrategy()
gentle_quack = GentleQuackStrategy()

class Duck(object):
    def __init__(self, quack_strategy):
        self._quack_strategy = quack_strategy

    def quack(self):
        self._quack_strategy.quack()
```

```
# Types of Ducks
class ToyDuck(Duck):
    def __init__(self):
        super(ToyDuck, self).__init__(gentle_quack)

class RobotDuck(Duck):
    def __init__(self):
        super(RobotDuck, self).__init__(loud_quack)

robo = RobotDuck()
robo.quack()  # QUACK! QUACK!!
```

Observer Pattern

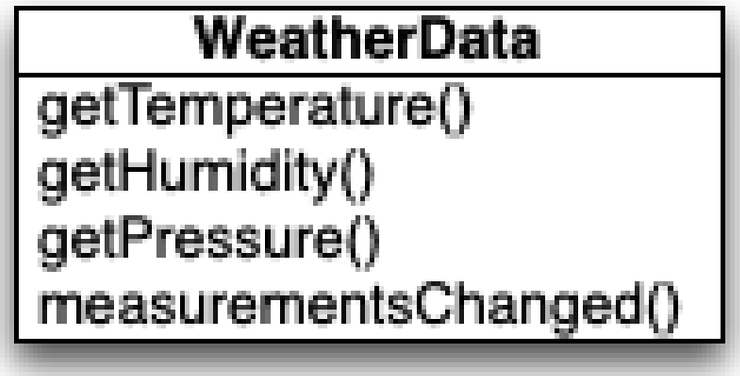
- Don’t miss out when something interesting (in your system) happens!
  - The observer pattern allows objects to keep other objects informed about events occurring within a software system (or across multiple systems)
  - It’s dynamic in that an object can choose to receive or not receive notifications at run-time
  - Observer happens to be one of the most heavily used patterns in the Java Development Kit
  - and indeed is present in many other frameworks

Weather Monitoring



We need to pull information from a weather station and then generate "current conditions, weather stats, and a weather forecast".

WeatherData Skeleton



We receive a partial implementation of the WeatherData class from our client.

They provide three getter methods for the sensor values and an empty measurementsChanged() method that is guaranteed to be called whenever a sensor provides a new value

We need to pass these values to our three displays… simple!

First pass at measurementsChanged

```
1  ...
2
3  public void measurementsChanged() {
4
5      float temp     = getTemperature();
6      float humidity = getHumidity();
7      float pressure = getPressure();
8
9      currentConditionsDisplay.update(temp, humidity, pressure);
10     statisticsDisplay.update(temp, humidity, pressure);
11     forecastDisplay.update(temp, humidity, pressure);
12
13 }
14
15 ...
16
```

Problems?

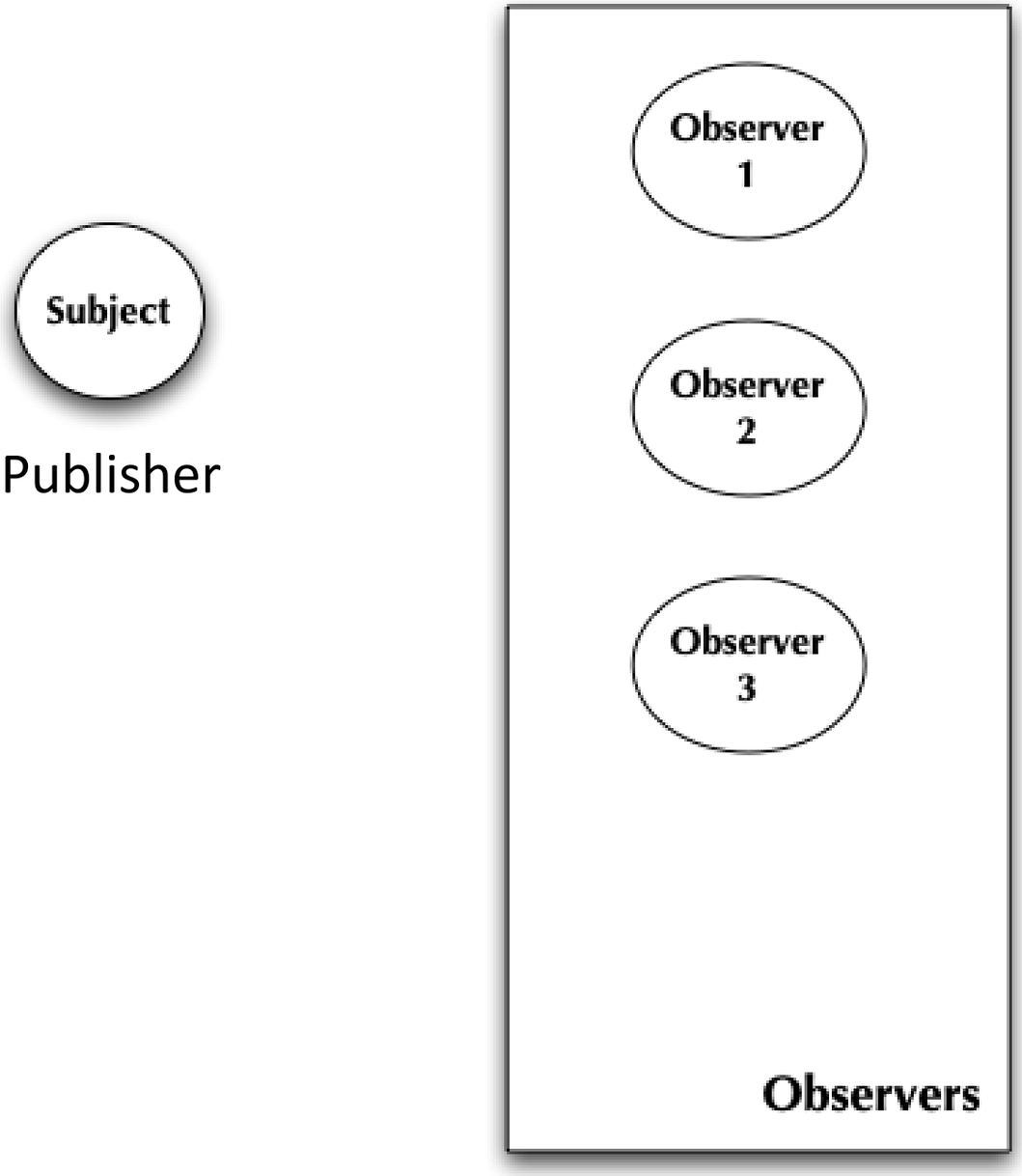
1. The number and type of displays may vary. These three displays are hard coded with no easy way to update them.

2. Coding to **implementations**, not an **interface**! Each implementation has adopted the same interface, so this will make translation easy!

Observer Pattern

- This situation can benefit from use of the observer pattern
- This pattern is similar to subscribing to a newspaper
  - A newspaper comes into existence and starts publishing editions
  - You become interested in the newspaper and subscribe to it
  - Any time an edition becomes available, you are notified (by the fact that it is delivered to you)
  - When you don't want the paper anymore, you unsubscribe
  - The newspaper's current set of subscribers can change at any time
- Observer is just like this but we call the publisher the "subject" and we refer to subscribers as "observers"
- Observer can also be considered a broadcaster, since the publisher is going to send messages out regardless of who subscribes

Observer in Action (I)

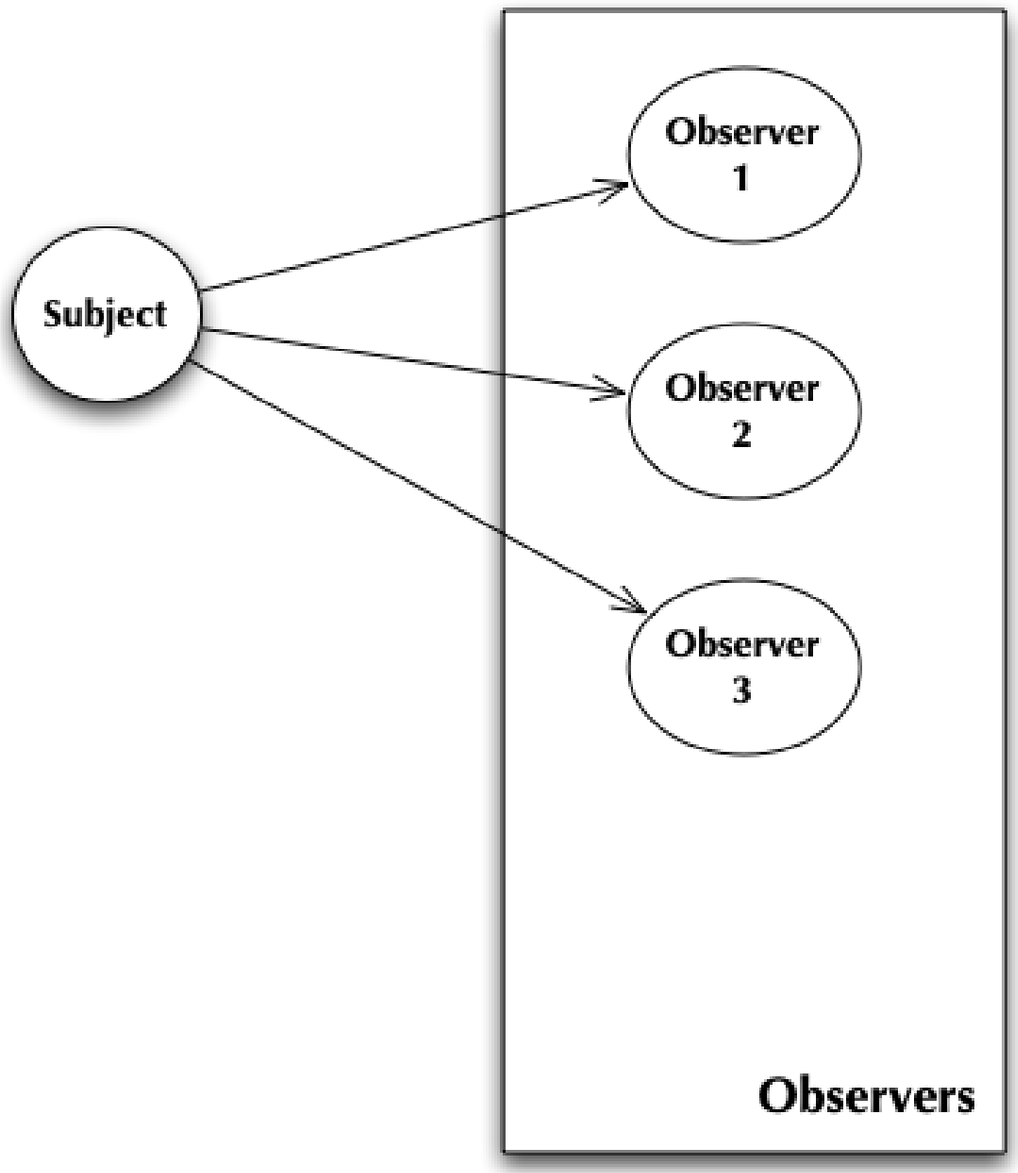


Publisher

Subscribers

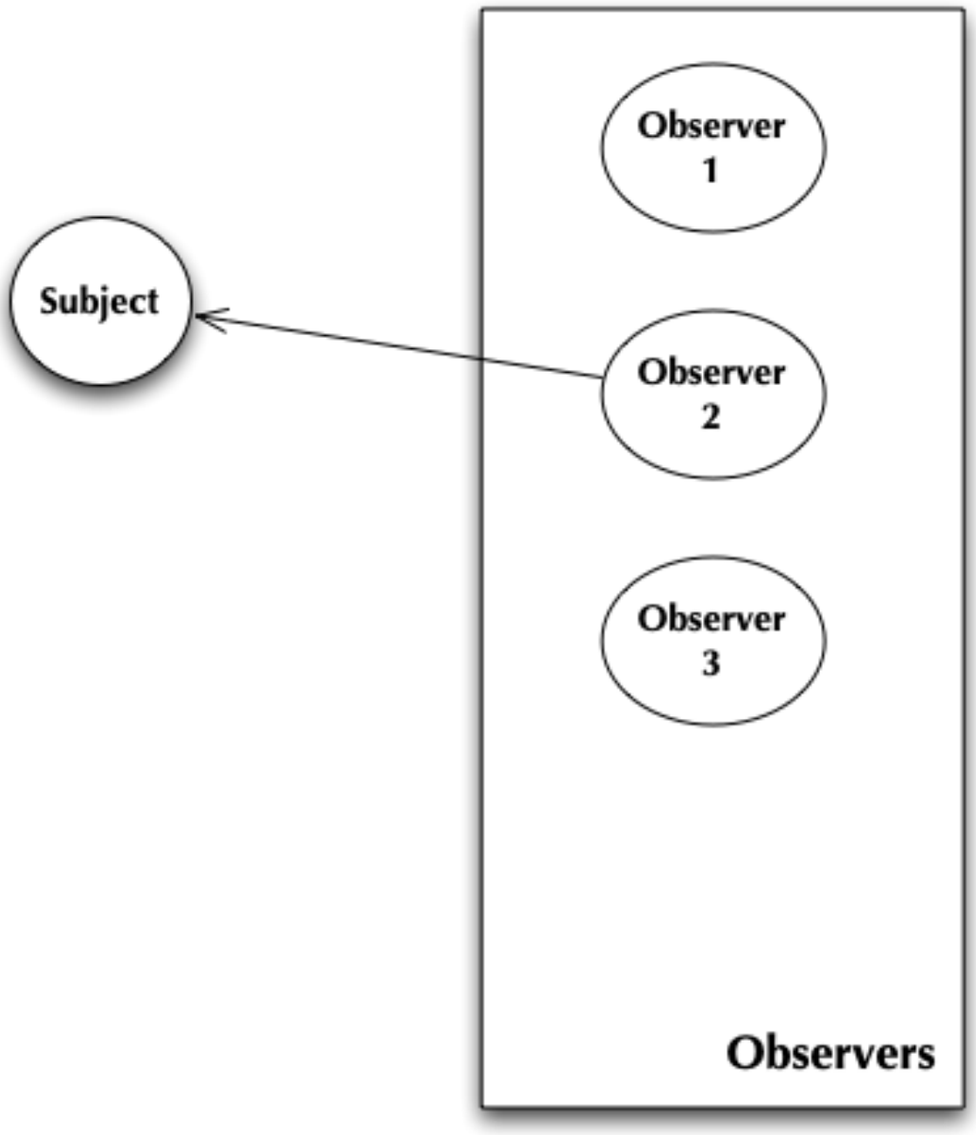
Subject maintains a list of observers

Observer in Action (II)



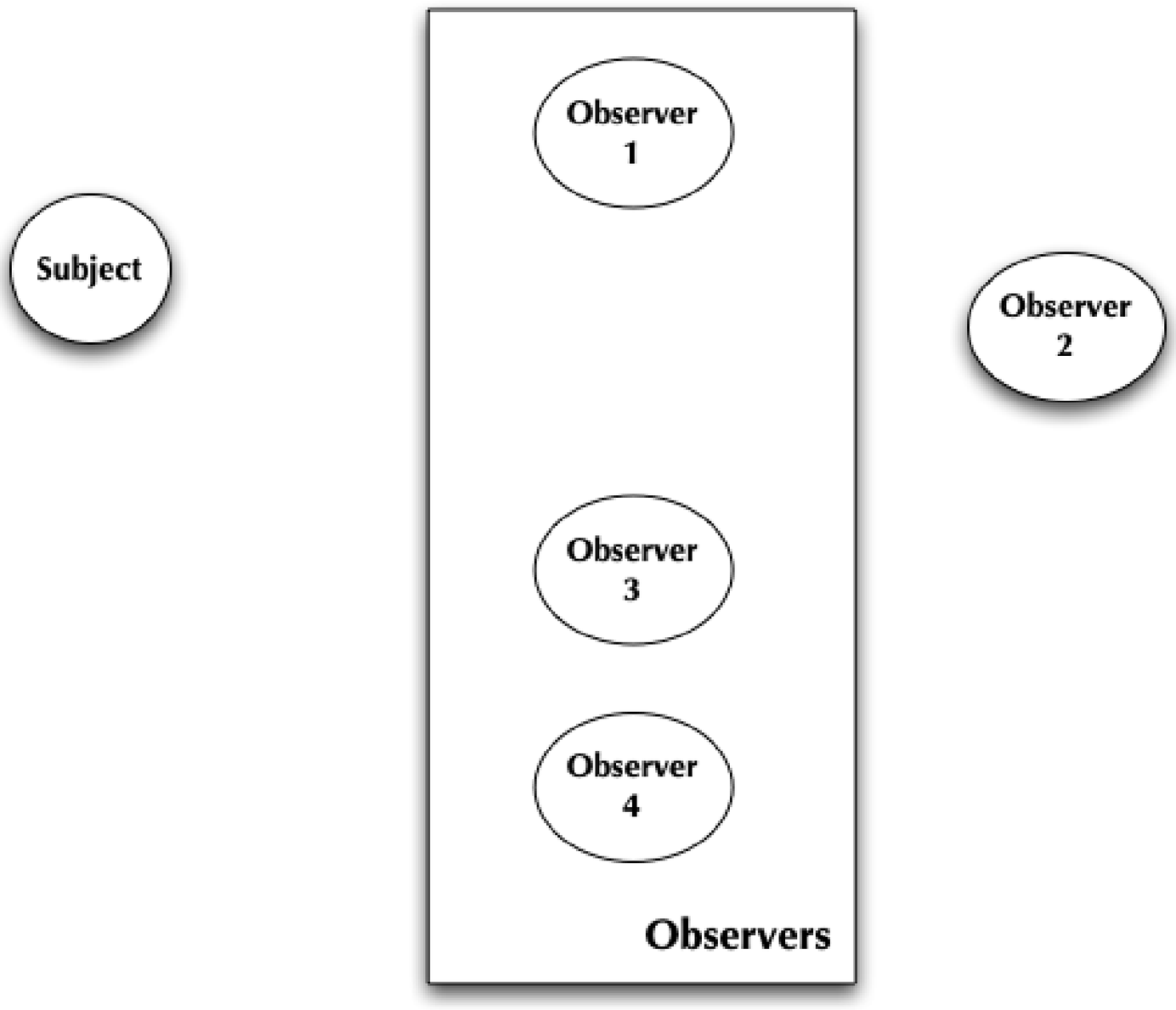
If the Subject changes, it notifies its observers

Observer in Action (III)



If needed, an observer may query its subject for more information

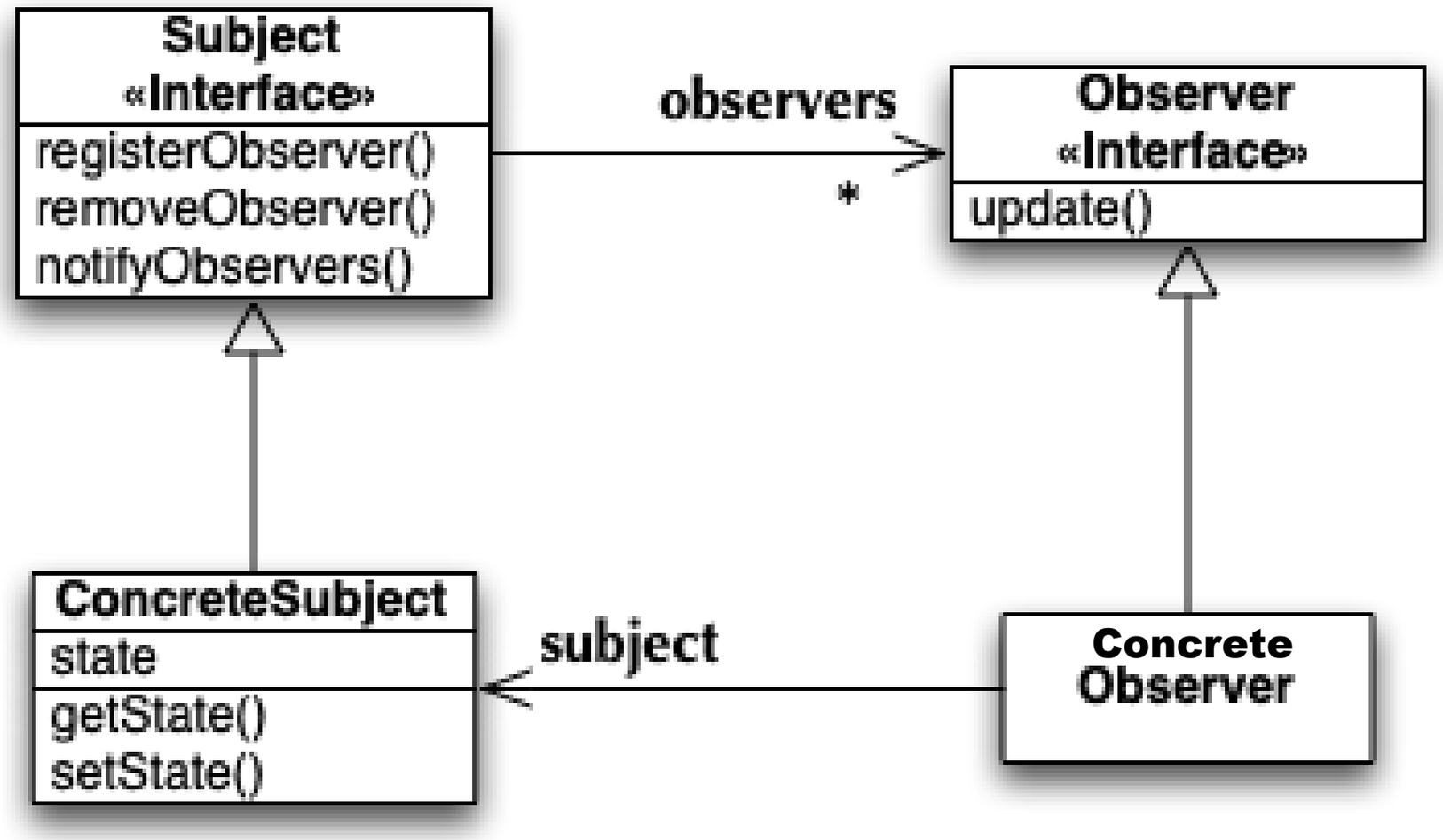
Observer In Action (IV)



At any point, an observer may join or leave the set of observers

Observer Definition and Structure

- The Observer Pattern defines a one-to-many dependency between a set of objects, such that when one object (the subject/publisher) changes, all of its dependents (observers/subscribers) are notified and updated automatically



Observer Benefits

- Observer affords a loosely coupled interaction between subject and observer
  - This means they can interact with very little knowledge about each other
- Consider
  - The subject only knows that observers implement the Observer interface
  - We can add/remove observers of any type at any time
  - We never have to modify subject to add a new type of observer
- We can reuse subjects and observers in other contexts
  - The interfaces plug-and-play anywhere observer is used
- Observers may have to know about the ConcreteSubject class if it provides many different state-related methods
  - Otherwise, data can be passed to observers via the update() method
- Note: The textbook includes a full Java build out of the weather station…

Concerns for Observers

- Keep clarity in the design about why notifications will happen
  - Knowing why notices occur will help in design of observer logic
- Dealing with unexpected updates
  - Observers and observed objects have no knowledge of each other's state or existence
  - A seemingly simple subject change may cause a cascade of updates to observers and all their dependent objects
  - When dependency criteria is not well-defined or maintained, spurious or unexpected updates can cause issues, which may be hard to track down
- Overly simple connectivity?
  - In the base form of the simple update protocol, no details are provided on what changed in the subject
  - Without additional protocol details to help observers discover what changed, they may be forced to work to discover what the subject changes were
- Push or Pull?
  - Including detailed information about changes in notifications is a "push" model
  - Having the observers look for or query for what changed is a "pull" model

http://www.cs.unc.edu/~stotts/GOF/hires/pat5gfso.htm

Observers are built in for Java, but…

- Using java.util.Observable and java.util.Observer
  - Observer is an interface with one defined method: update(subject, data)
  - To notify observers: call setChanged(), then notifyObservers(data)
  - Observable is a CLASS, a subject has to subclass it to manage observers
    - This is an issue, because we can only inherit it, nothing else – reduces its usefulness
    - Also, since it's not an interface, there's not much you can do to make a custom version
    - This is actually trouble for two principles
      - Coding to an interface, not an implementation
      - Favor composition over inheritance
      - We'll talk about the resulting issues with this approach…

- https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Observable.html
- Nice Java example at Javaworld: https://www.javaworld.com/article/2077258/observer-and-observable.html

Observers in Java – An Observable

```
import java.util.Observable;
public class ObservableValue extends Observable
{
   private int n = 0;
   public ObservableValue(int n)
   {
      this.n = n;
   }
   public void setValue(int n)
   {
      this.n = n;
      setChanged();
      notifyObservers();
   }
   public int getValue()
   {
      return n;
   }
}
```

Observers in Java – An Observer

```
import java.util.Observer;
import java.util.Observable;
public class TextObserver implements Observer
{
    private ObservableValue ov = null;
    public TextObserver(ObservableValue ov)
    {
        this.ov = ov;
    }
    public void update(Observable obs, Object obj)
    {
        if (obs == ov)
        {
            System.out.println(ov.getValue());
        }
    }
}
```

Observers in Java – Tied Together

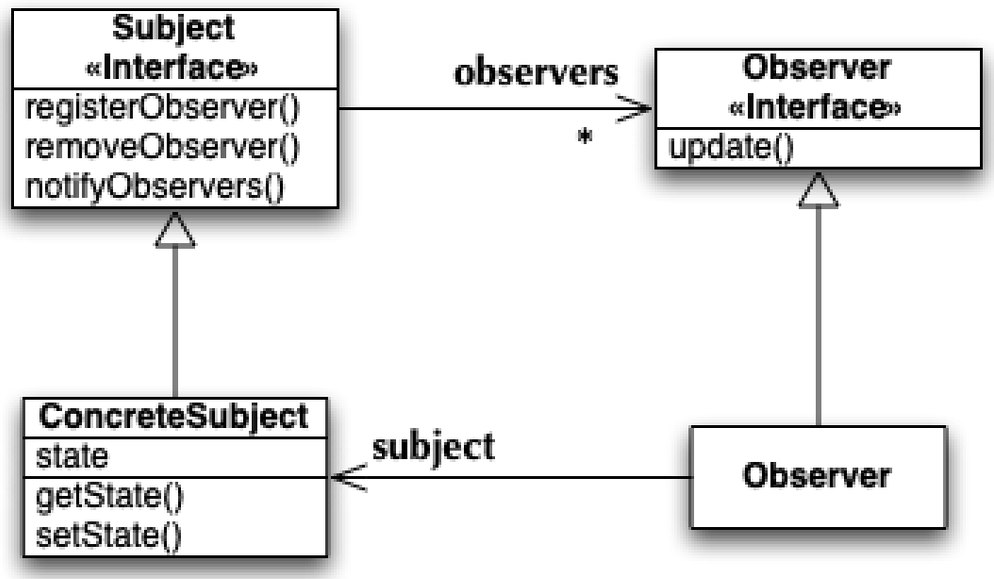
```
public class Main
{
   public Main()
   {
      ObservableValue ov = new ObservableValue(0);
      TextObserver to = new TextObserver(ov);
      ov.addObserver(to);
   }
   public static void main(String [] args)
   {
      Main m = new Main();
   }
}
```

Now, forget that…

- I wanted to review the Java 8 Observer/Observable because it is a clear implementation of the Observer pattern, but…
- Observer/Observable was depreciated in Java 9
  - Order of notifications provided by Observable not specified
  - Observable is not serializable (we'll talk to that later)
  - Not thread safe
  - And overall, a poor model that was never "fixed"
- Suggested Java Alternatives:
  - java.beans PropertyChangeListener interface (replaces Observer) and PropertyChangeSupport (a class for creating Observable objects)
    - https://www.baeldung.com/java-observer-pattern
    - Pretty much a drop in replacement for Observer/Observable)
  - Flow class with 4 interfaces (from java.util.concurrent)
    - Flow.Publisher – for making something observable
    - Flow.Subscriber – for making something observe events
    - Flow.Processor – component that supports both Subscribe and Publish
    - Flow.Subscription – used to link Publishers and Subscribers
    - https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Flow.html
  - Observer/Observable would be usable for classwork, but I'd look at these methods
- Full discussion here: https://stackoverflow.com/questions/46380073/observer-is-deprecated-in-java-9-what-should-we-use-instead-of-it

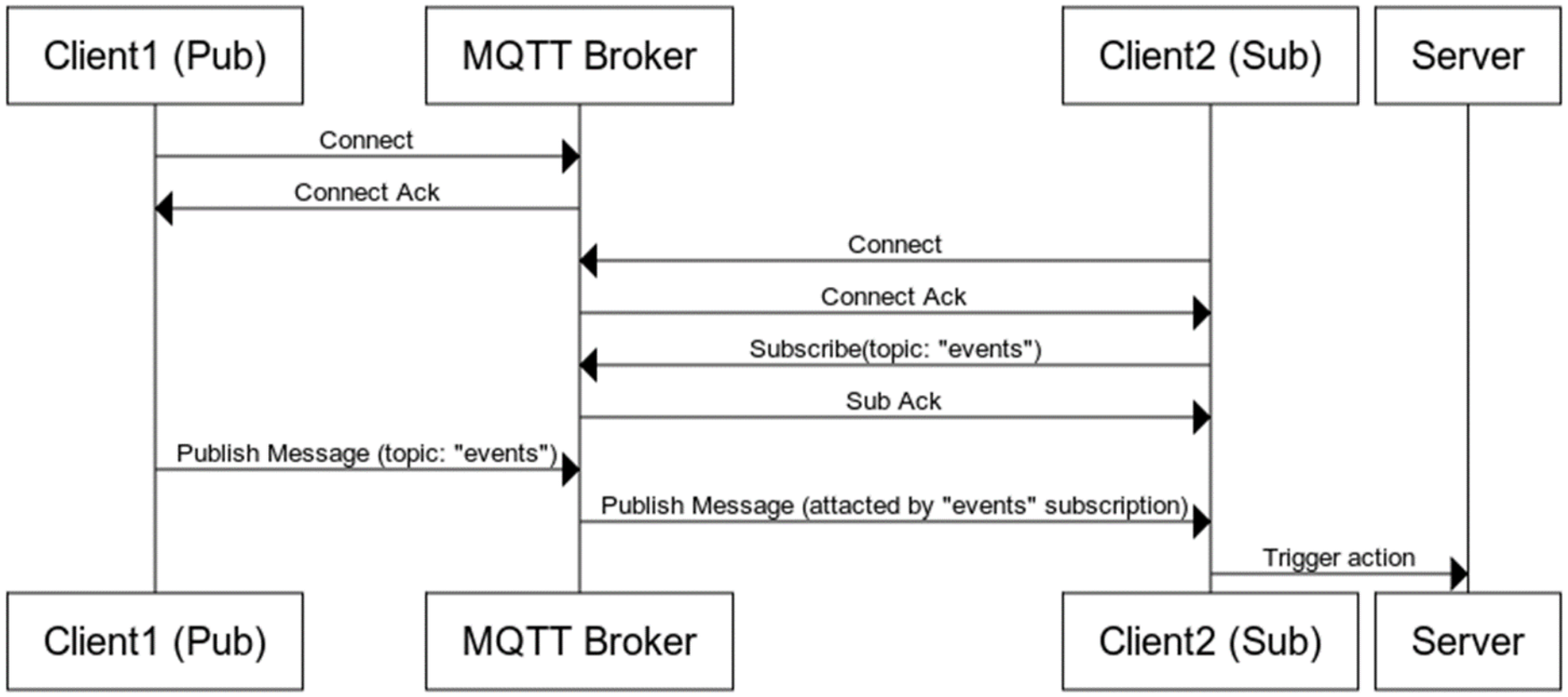
You could… Write your own Observer, Observable interfaces

- Observer (Subscriber)
  - Register with any observable you want information from
  - Provide an update() method to get notified if things change
  - Call the Observable's getState() method to get the changed information



- Observables (Publishers/Subjects)
  - Allow register and remove observer objects
  - Call notifyObservers update() methods when something happens
  - Allow for get and set of state; getState will be called by Observers

You could… Use another Pub/Sub model (analogous to Observed/Observer)



- We saw this model in the UML lecture:
  - https://stackoverflow.com/questions/32538535/node-and-mqtt-do-something-on-message
- You can use an MQTT broker with Java from a standard message queuing app like RabbitMQ; or you can use other pub/sub approaches from a socket library like ZeroMQ (more later on these in the Proxy pattern lecture…)

Python - Observer

In this example of using Observer, we're using Python classes to set up the plumbing for an Observable class and and Observer class.

This could be done with abstract classes as well to prevent any mistaken direct instantiations…

```
# Subject aka Observable or Publisher
class Subject:
    def register_observer(self, observer):
        raise NotImplementedError

    def remove_observer(self, observer):
        raise NotImplementedError

    def notify_observers(self):
        raise NotImplementedError

# Observer aka Subscriber
class Observer:
    def update(self, temp):
        raise NotImplementedError
```

Example from:
https://github.com/jtortorelli/head-first-design-patterns-python/blob/master/src/python/chapter_2/weather_o_rama.py

Python - Observer

We'll make a weather publisher (observable, subject) by inheriting from Subject and defining methods and local attributes we need.

Note that the Subject here is maintaining a list of observers and functions to add, remove, and notify them.

Whenever temperature is set in set_measurements, a changed even causes all registered observers to be notified.

```
# Weather Publisher
class WeatherData(Subject):

    def __init__(self):
        self.temperature = None
        self.observers = []

    def register_observer(self, observer):
        self.observers.append(observer)

    def remove_observer(self, observer):
        self.observers.remove(observer)

    def notify_observers(self):
        for observer in self.observers:
            observer.update(self.temperature)

    def measurements_changed(self):
        self.notify_observers()

    def set_measurements(self, temperature):
        self.temperature = temperature
        self.measurements_changed()
```

Example from:
https://github.com/jtortorelli/head-first-design-patterns-python/blob/master/src/python/chapter_2/weather_o_rama.py

Python - Observer

When we make the Observer or Subscriber, we get a reference to the Publisher – here the WeatherData object – and we register ourselves for updates.

Now, whenever the WeatherData object changes, we'll be notified and we'll do what we want with the new data (in this case, display it).

Here we plumbed this out ourselves, you might grab a library for this instead, like PyPubSub (https://pypi.org/project/PyPubSub/)

```
# Weather Subscriber
class CurrentConditionsDisplay(Observer):
    def __init__(self, weather_data):
        self.weather_data = weather_data
        self.weather_data.register_observer(self)
        self.temperature = None

    def update(self, temp):
        self.temperature = temp
        self.display()

    def display(self):
        print(f"Current conditions: {self.temperature} F")

# Main program
wd = WeatherData()
ccd = CurrentConditionsDisplay(wd)

wd.set_measurements(80)
wd.set_measurements(82)
wd.set_measurements(78)
```

Example from:
https://github.com/jtortorelli/head-first-design-patterns-python/blob/master/src/python/chapter_2/weather_o_rama.py

Summary

- Observer provides a one-to-many dependency; when one object changes state, all the dependents are automatically notified.
- It is one of the most popular patterns from the GoF patterns library
- It helps keep **Loose Coupling** between objects; able to notify other objects about changes without knowing about them
- Design Principle: **Strive for loosely coupled designs between objects that interact**
- The Java 8 Observer/Observable implementation will work, but it has issues and was depreciated in Java 9
- Use either PropertyChangeListener interface (replaces Observer) and PropertyChangeSupport (a class for creating Observable objects) or the interfaces from the Flow class (from java.util.concurrent)
- In Python, you can easily develop an observer/observable or use a library like PyPubSub

Next Steps

- Project 2 code and updated UML is due Wed 9/22 (part 2)
- New quiz due Wed 9/22
- New Piazza discussion topic for participation grade
- Make sure you sign up for Piazza and Canvas notifications
- Get access to the Head First Patterns book, we'll point to readings there as we get into the OO patterns
- I'm reviewing graduate proposals, comments posted soon
- Recitation for programming or class questions today: 2-4 PM (on Piazza and Canvas)
- Coming soon – Decorator (Chapter 3 in book), Factories (Chapter 4 in book), and some Conceptual Modelling approaches…
- Project 3 for you on Wednesday
- Office hours and GitHub user IDs for class staff are on Piazza and Canvas
- If you'd like to see Dwight, Max, or Roshan at times other than their office hours, ping them on Piazza or email
- If you'd like to see Bruce outside of office hours, use my appointment app: https://brucem.appointlet.com