More Design Patterns, Part 1

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 25 — 04/05/2019

Acknowledgement & Materials Copyright

- Dr. Ken Anderson is a Professor of the Department of Computer Science and the Associate Dean for Education for the College of Engineering & Applied Science
- Ken taught this OOAD class 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

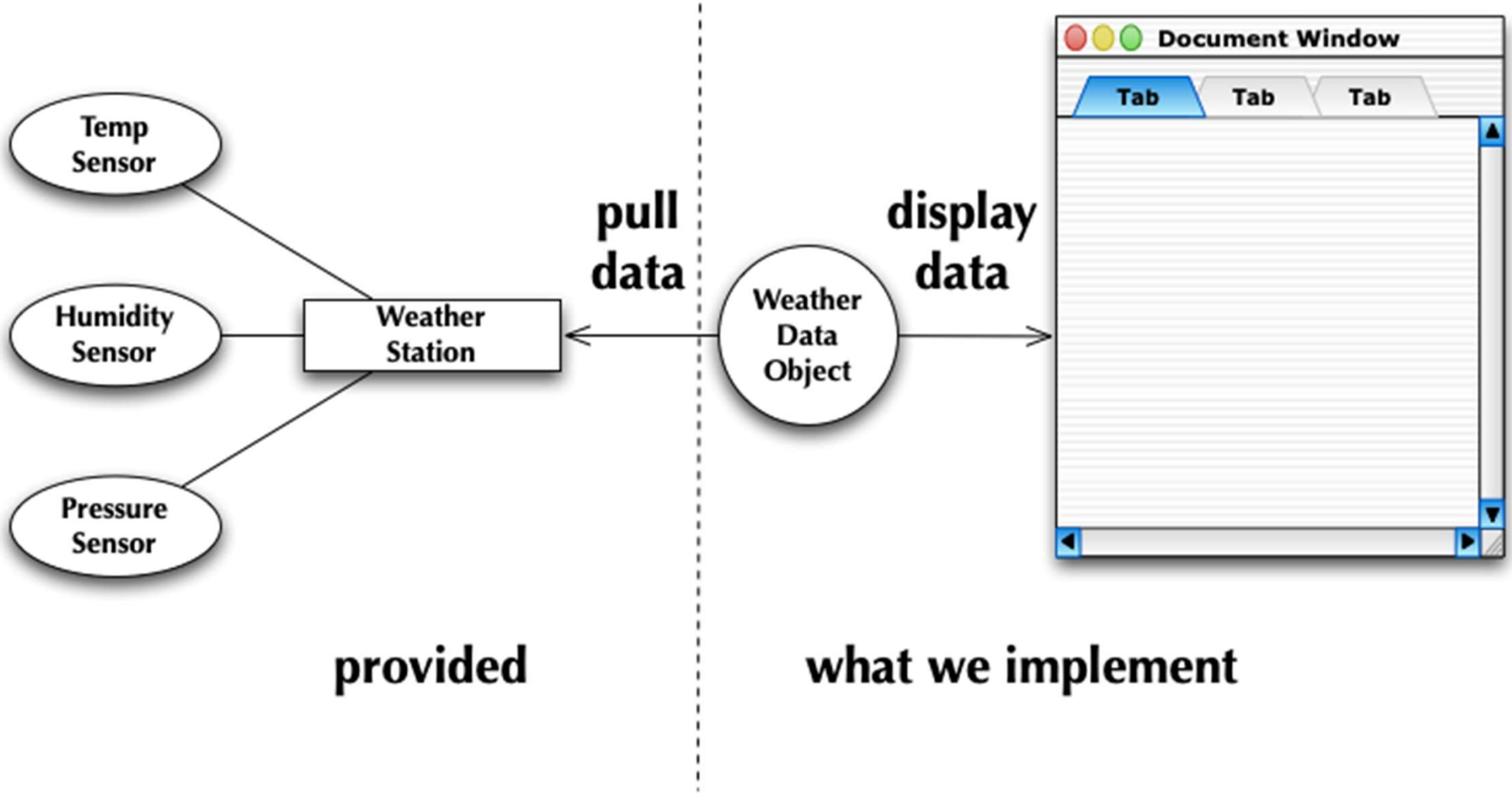
Goals of the Lecture

- Cover the material in Chapters 18 & 19 of our textbook
  - Observer
  - Template Method

Observer Pattern

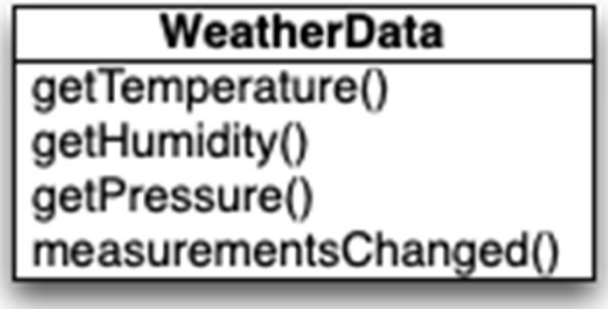
- Don’t miss out when something interesting (in your system) happens!
  - The observer pattern allows objects to keep other objects informed about events occurring within a software system (or across multiple systems)
  - It’s dynamic in that an object can choose to receive or not receive notifications 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?

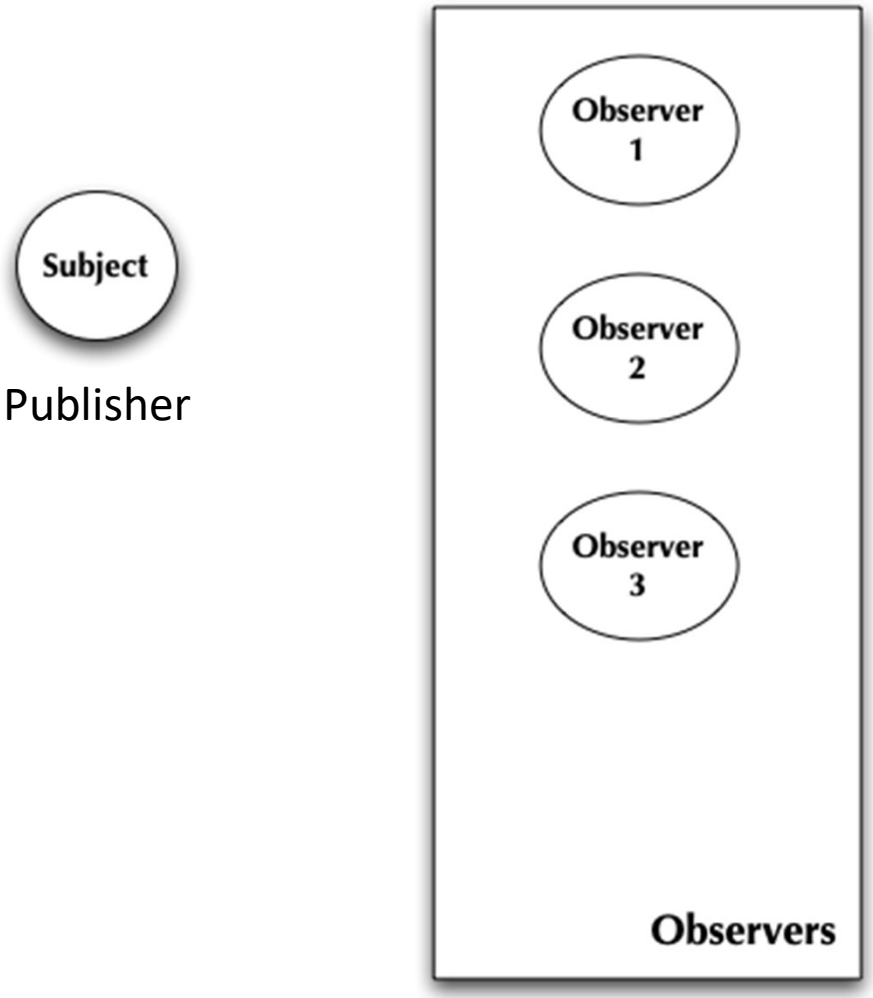
1. The number and type of displays may vary. These three displays are hard coded with no easy way to update them.

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 in Action (I)

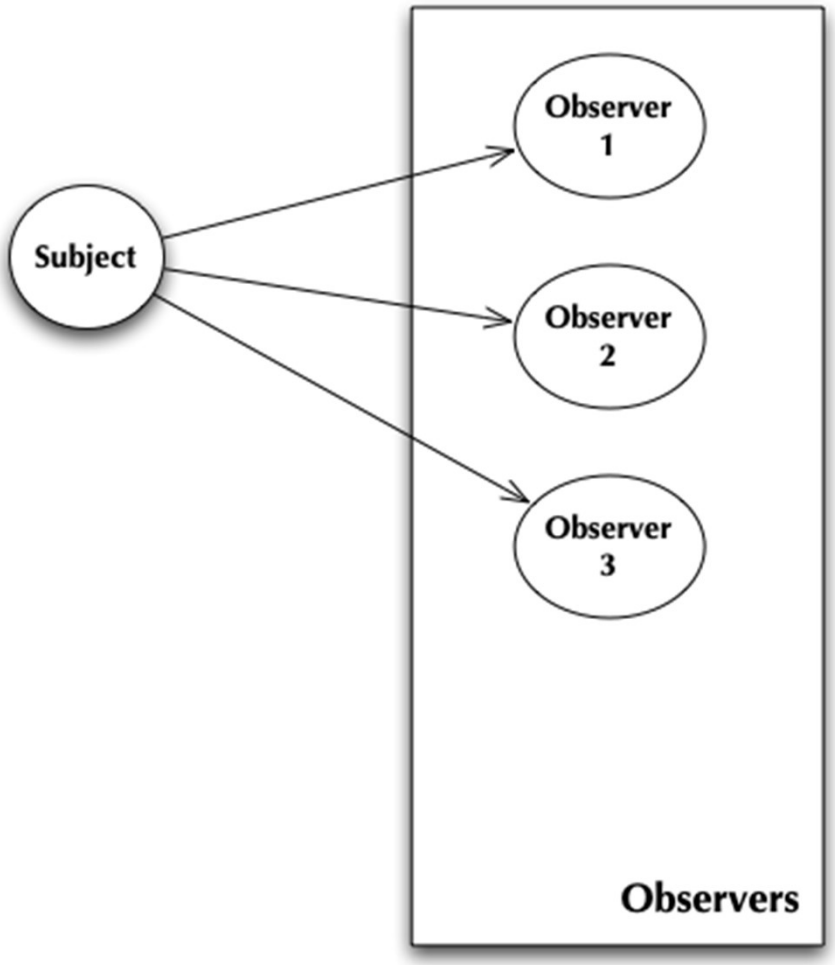


Publisher

Subscribers

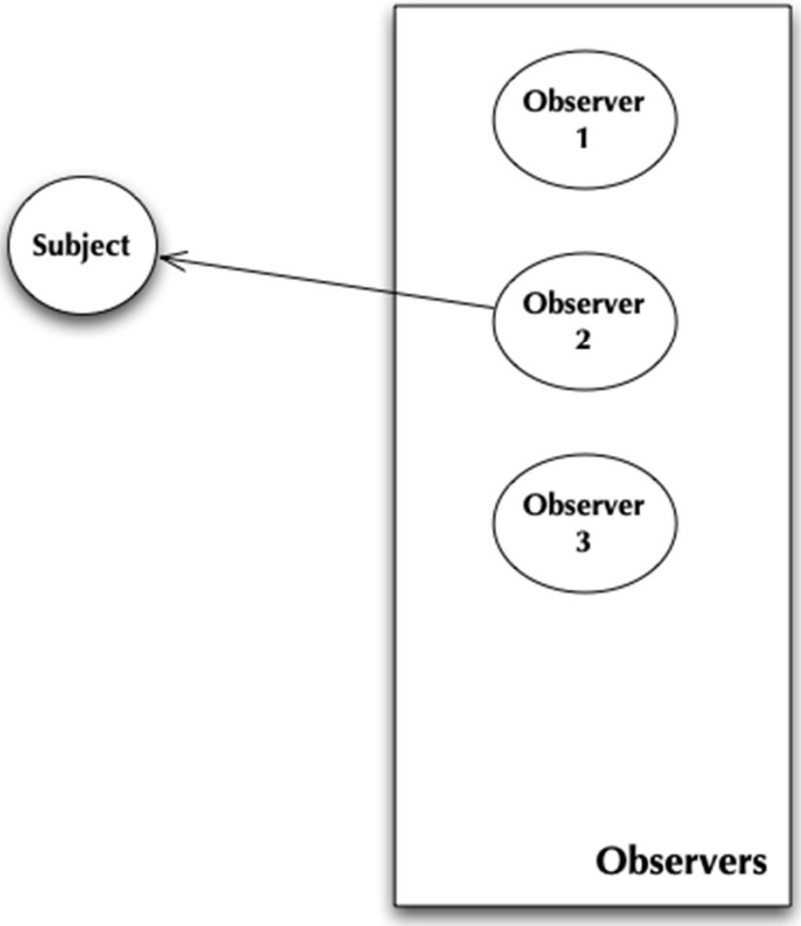
Subject maintains a list of observers

Observer in Action (II)



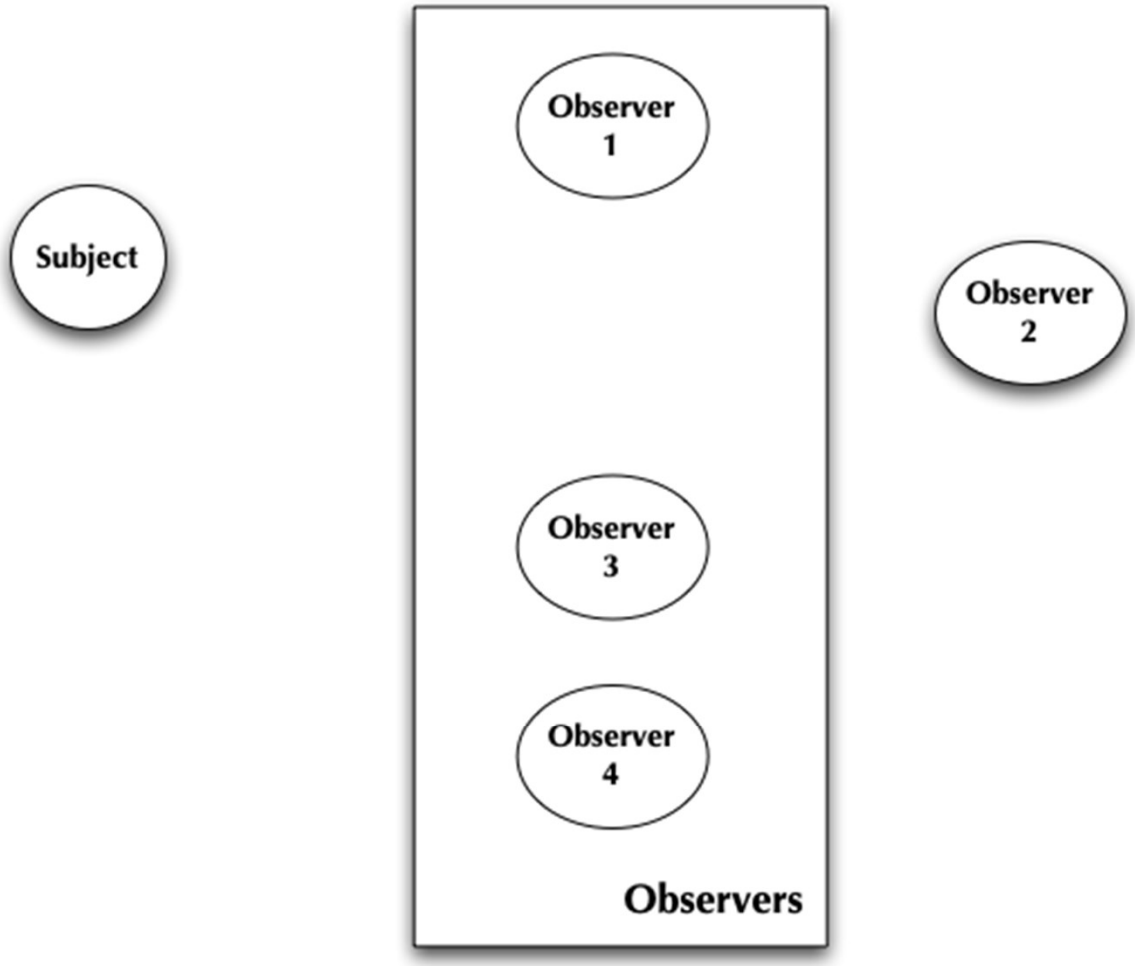
If the Subject changes, it notifies its observers

Observer in Action (III)



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

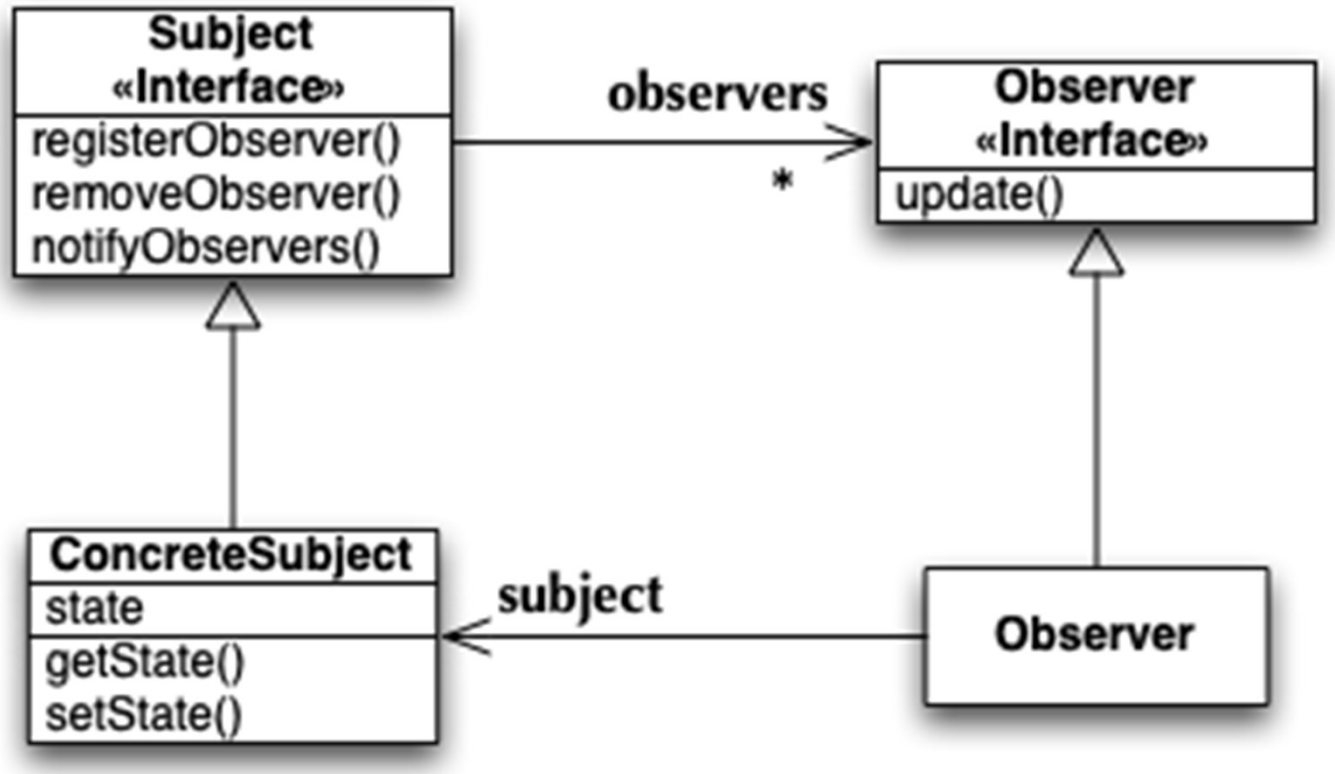
Observer In Action (IV)



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

Observers are built in for Java

- Using java.util.Observable and java.util.Observer
  - Observable is a CLASS, a subject has to subclass it to manage observers
  - Observer is an interface with one defined method: update(subject, data)
  - To notify observers: call setChanged(), then notifyObservers(data)

- 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());
      }
   }
}
```

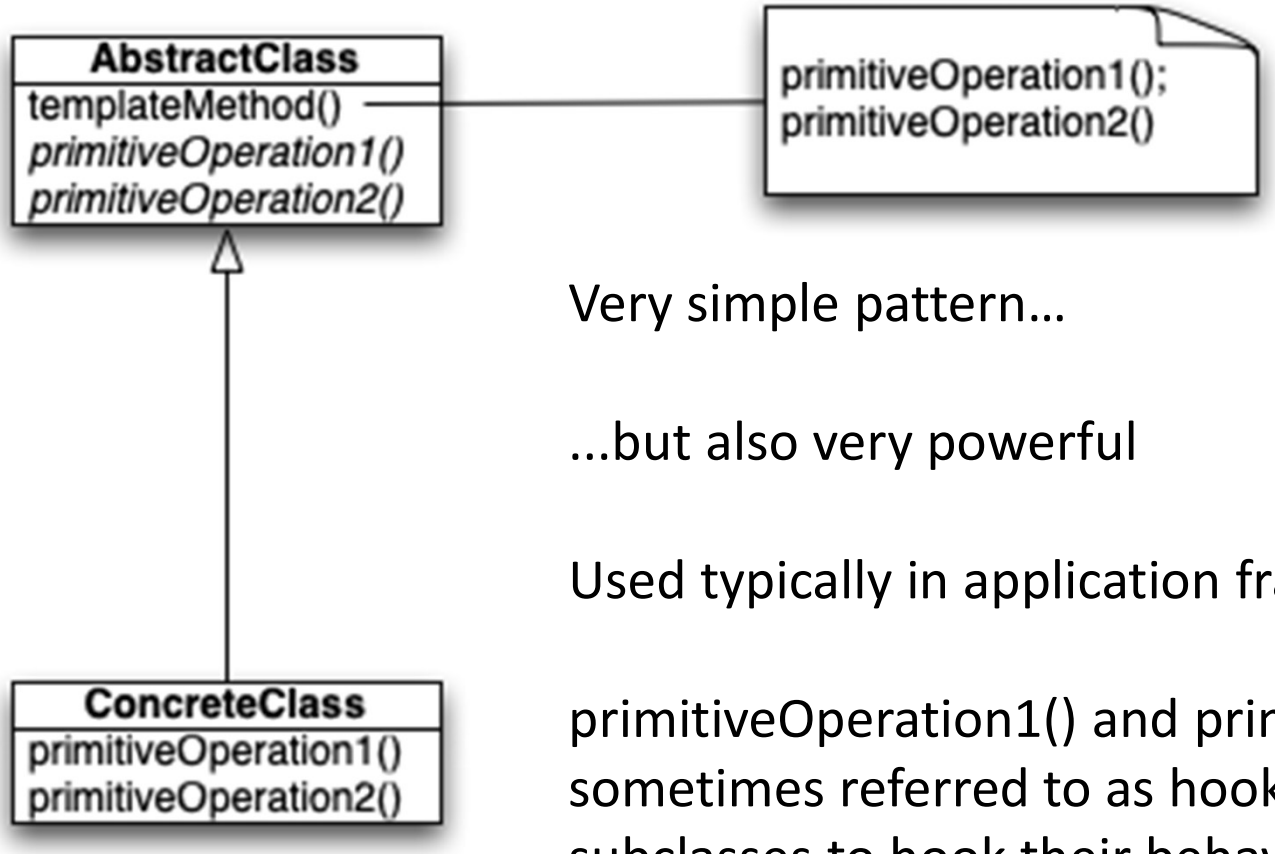
Observers in Java – Tied Together

```
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();
   }
}
```

Template Method: Definition

- The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps
  - Makes the algorithm abstract
  - Each step of the algorithm is represented by a method
  - Encapsulates the details of most steps
    - Steps (methods) handled by subclasses are declared abstract
    - Shared steps (concrete methods) are placed in the same class that has the template method, allowing for code re-use among the various subclasses

Template Method: Structure



Very simple pattern…

...but also very powerful

Used typically in application frameworks, e.g. .Net

primitiveOperation1() and primitiveOperation2() are sometimes referred to as hook methods as they allow subclasses to hook their behavior into the service provided by AbstractClass

Example: Tea and Coffee

- Consider another Starbuzz example in which we consider the recipes for making coffee and tea in a barista's training guide

  - Coffee
    - Boil water
    - Brew coffee in boiling water
    - Pour coffee in cup
    - Add sugar and milk

  - Tea
    - Boil water
    - Steep tea in boiling water
    - Pour tea in cup
    - Add lemon

Coffee Implementation

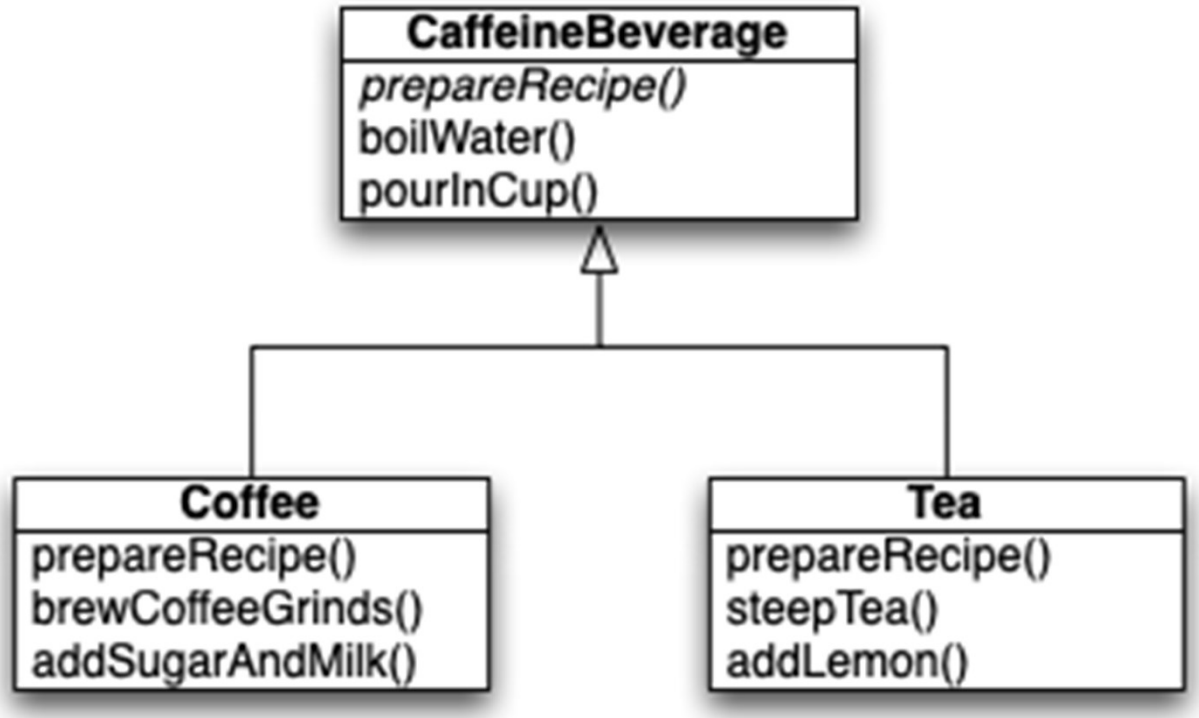
```
1  public class Coffee {
2
3      void prepareRecipe() {
4          boilWater();
5          brewCoffeeGrinds();
6          pourInCup();
7          addSugarAndMilk();
8      }
9
10     public void boilWater() {
11         System.out.println("Boiling water");
12     }
13
14     public void brewCoffeeGrinds() {
15         System.out.println("Dripping Coffee through filter");
16     }
17
18     public void pourInCup() {
19         System.out.println("Pouring into cup");
20     }
21
22     public void addSugarAndMilk() {
23         System.out.println("Adding Sugar and Milk");
24     }
25 }
26
```

Tea Implementation

```
1  public class Tea {
2
3      void prepareRecipe() {
4          boilWater();
5          steepTeaBag();
6          pourInCup();
7          addLemon();
8      }
9
10     public void boilWater() {
11         System.out.println("Boiling water");
12     }
13
14     public void steepTeaBag() {
15         System.out.println("Steeping the tea");
16     }
17
18     public void addLemon() {
19         System.out.println("Adding Lemon");
20     }
21
22     public void pourInCup() {
23         System.out.println("Pouring into cup");
24     }
25 }
26
```

Code Duplication!

- We have code duplication occurring in these two classes
  - boilWater() and pourInCup() are exactly the same
- Lets get rid of the duplication



Similar algorithms

- The structure of the algorithms in prepareRecipe() is similar for Tea and Coffee
- We can improve our code further by making the code in prepareRecipe() more abstract
    - brewCoffeeGrinds() and steepTea() ⇒ brew()
    - addSugarAndMilk() and addLemon() ⇒ addCondiments()
- Excellent, now all we need to do is specify this structure in CaffeineBeverage.prepareRecipe() and make it such that subclasses can't change the structure
    - How do we do that?
    - Answer: By convention OR by using a keyword like "final" in languages that support it
    - In Java – "final" can be used in a variable definition to create a constant, with a method to prevent method overrides, or with a class to prevent inheritance

CaffeineBeverage Implementation

```
1  public abstract class CaffeineBeverage {
2
3      final void prepareRecipe() {
4          boilWater();
5          brew();
6          pourInCup();
7          addCondiments();
8      }
9
10     abstract void brew();
11
12     abstract void addCondiments();
13
14     void boilWater() {
15         System.out.println("Boiling water");
16     }
17
18     void pourInCup() {
19         System.out.println("Pouring into cup");
20     }
21 }
22
```

Note: use of final keyword for prepareReceipe()

brew() and addCondiments() are abstract and must be supplied by subclasses

boilWater() and pourInCup() are specified and shared across all subclasses

Coffee And Tea Implementations

```
1  public class Coffee extends CaffeineBeverage {
2      public void brew() {
3          System.out.println("Dripping Coffee through filter");
4      }
5      public void addCondiments() {
6          System.out.println("Adding Sugar and Milk");
7      }
8  }
9
10 public class Tea extends CaffeineBeverage {
11     public void brew() {
12         System.out.println("Steeping the tea");
13     }
14     public void addCondiments() {
15         System.out.println("Adding Lemon");
16     }
17 }
18
```

Nice and Simple!

What have we done?

- Took two separate classes with separate but similar algorithms
- Noticed duplication and eliminated it by introducing a superclass
- Made steps of algorithm more abstract and specified its structure in the superclass
  - Thereby eliminating another “implicit” duplication between the two classes
- Revised subclasses to implement the abstract (unspecified) portions of the algorithm… in a way that made sense for them

Comparison: Template Method (TM) vs. No TM

- No Template Method
  - Coffee and Tea each have own copy of algorithm
  - Code is duplicated across both classes
  - A change in the algorithm would result in a change in both classes
  - Not easy to add new caffeine beverage
  - Knowledge of algorithm distributed over multiple classes
- Template Method
  - CaffeineBeverage has the algorithm and protects it
  - CaffeineBeverage shares common code with all subclasses
  - A change in the algorithm likely impacts only CaffeineBeverage
  - New caffeine beverages can easily be plugged in
  - CaffeineBeverage centralizes knowledge of the algorithm; subclasses plug in missing pieces

Adding a Hook to CaffeineBeverage

```
1  public abstract class CaffeineBeverageWithHook {
2
3      void prepareRecipe() {
4          boilWater();
5          brew();
6          pourInCup();
7          if (customerWantsCondiments()) {
8              addCondiments();
9          }
10     }
11
12     abstract void brew();
13
14     abstract void addCondiments();
15
16     void boilWater() {
17         System.out.println("Boiling water");
18     }
19
20     void pourInCup() {
21         System.out.println("Pouring into cup");
22     }
23
24     boolean customerWantsCondiments() {
25         return true;
26     }
27 }
28
```

prepareRecipe() altered to have a hook method: customerWantsCondiments()

This method provides a method body that subclasses can override

To make the distinction between hook and non-hook methods more clear, you can add the “final” keyword to all concrete methods that you don’t want subclasses to touch

```java
import java.io.*;

public class CoffeeWithHook extends CaffeineBeverageWithHook {

    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    public boolean customerWantsCondiments() {

        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}

```

Adding a Hook to Coffee

Design Principle: Hollywood Principle

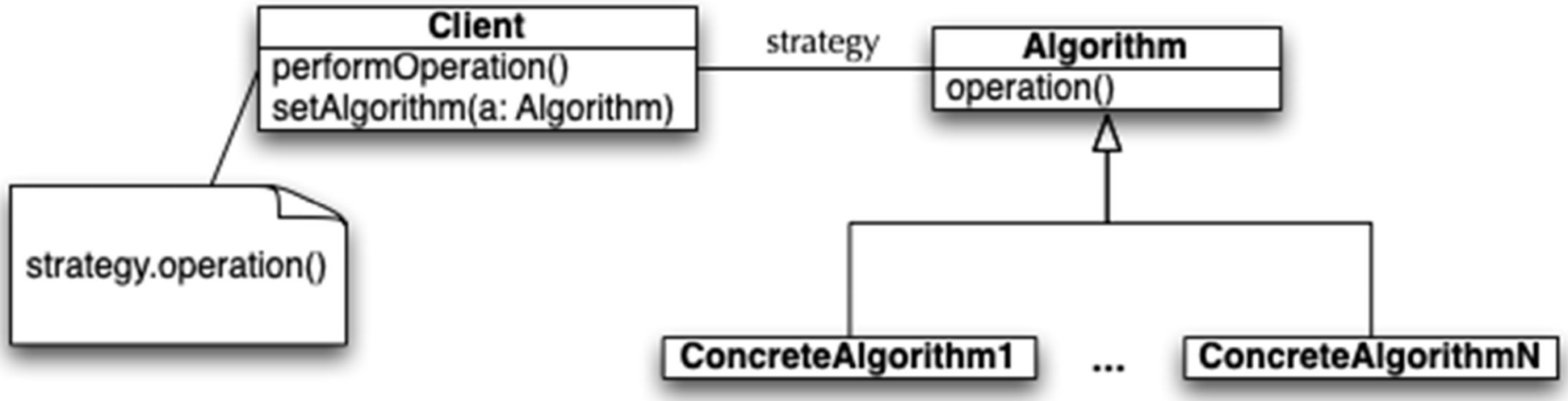
- Don’t call us, we’ll call you
- Or, in OO terms, high-level components call low-level components, not the other way around
  - In the context of the template method pattern, the template method lives in a high-level class and invokes methods that live in its subclasses
- This principle is similar to the dependency inversion principle: “Depend upon abstractions. Do not depend upon concrete classes.”
  - Template method encourages clients to interact with the abstract class that defines template methods as much as possible; this discourages the client from depending on the template method subclasses

Template Methods in the Wild

- Template Method is used a lot since it's a great design tool for creating frameworks
  - the framework specifies how something should be done with a template method
  - that method invokes abstract hook methods that allow client-specific subclasses to "hook into" the framework and take advantage of its services
- Example in the Java API: Sorting using compareTo() method
  - compareTo() is a member of Comparable
  - Implementing the Comparable<T> interface lets you specify ordering methods

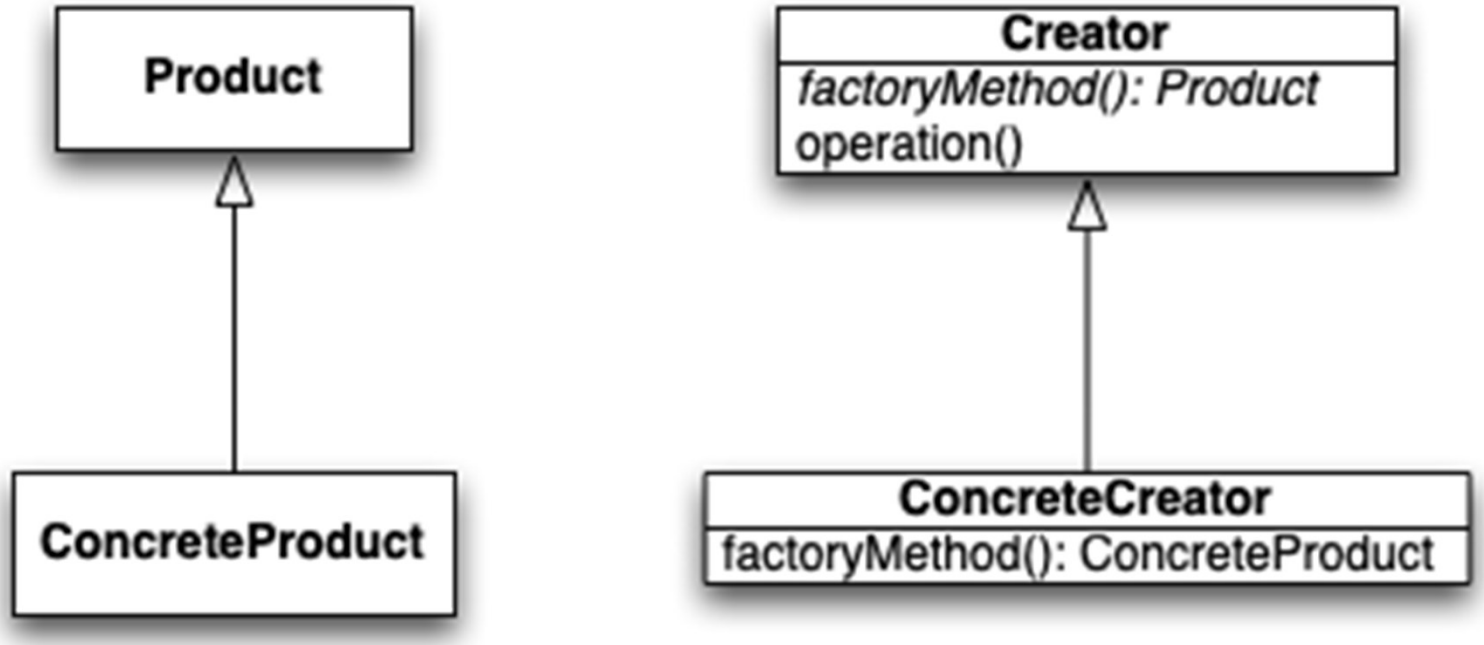
Template Method vs. Strategy (I)

- Both Template Method and Strategy deal with the encapsulation of algorithms
  - Template Method focuses encapsulation **on the steps of the algorithm**
  - Strategy focuses on encapsulating **entire algorithms**
  - You can use both patterns at the same time if you want
- Strategy Structure



Template Method vs. Strategy (II)

- Template Method encapsulate the details of algorithms using inheritance
- Factory Method can now be seen as a specialization of the Template Method pattern



- In contrast, Strategy does a similar thing but uses composition/delegation

Template Method vs. Strategy (III)

- Because it uses inheritance, Template Method offers code reuse benefits not typically seen with the Strategy pattern
- On the other hand, Strategy provides run-time flexibility because of its use of composition/delegation
  - You can switch to an entirely different algorithm when using Strategy, something that you can't do when using Template Method

Wrapping Up

- Observer
  - Flexibly monitor an object's state changes
- Template Method
  - Specify overall structure of an algorithm; allow some variation via overridden methods
- Next time:
  - State
    - Allow an object to completely change its behavior based on its current state
  - Flyweight
    - Make the seeming creation of "lots of little objects" efficient
  - Start on Creational patterns…

Heading towards the finish…

- Lectures 4/8, 4/12, 4/15, 4/19
- Recitations w/Manjunath on 4/10 and 4/17
- Demonstrations of Semester Project
  - I have reserved the classroom through 1:30PM on Fri 4/26, Mon 4/29, and Wed 5/1
  - Next week I will provide signups for 10 minute demonstration slots with me on those days
  - If your team can't make those times, or the slots fill up, you can also arrange demonstrations with Manjunath (more information soon)
  - All projects must be demonstrated – remote students can schedule a Zoom web meeting with me or Manjunath or provide a recorded video
- Graduate Presentations
  - For graduate students on campus (or distance graduate students regularly attending class)
  - We will use class periods on Mon 4/22, Wed 4/24, Fri 4/26, and Mon 4/29 for five 10 minute presentations each period – attendance will be taken for all four days – presentations will be recorded
  - Sign ups for presentation slots will be provided next week
- Final Lecture on 5/1

Next Steps

- Optional additional material
  - Dr. Anderson's lecture on More Design Patterns
    - You can find it on the class Canvas site under Media Gallery, Lecture 25
    - Again, very similar material as I'm using versions of his slides…
- Coming up
  - Next Week, more patterns: State, Flyweight, Singleton, Object Pool, Builder, etc.
  - We'll close out the textbook in the final lectures
- I'll be grading your Quiz 7 PDFs
- Midterm grades are posted, exams will be returned to you on Monday
  - I'll scan mid-terms and e-mail them to remote students
- Things that are due
  - Quiz 8 will be up on Saturday, due Wed 4/10 at 11 AM
  - Homework 5 – Interim Report – Fri 4/12 at 11 AM (50 points)
  - Graduate Presentation – Mon 4/15 11 AM (100 points) **(may NOT be turned in late)**
  - Optional Extra Credit for Quizzes – Wed 4/17 11 AM (up to 10 points total Extra Credit)
  - Homework 6 – Final Project – Fri 4/26 at 11 AM (150 points) **(may NOT be turned in late)**