Design Patterns

What Is a Design Pattern?

- A design pattern
  - **Is a common solution to a recurring problem in design**
  - Abstracts a recurring design structure
  - Names & specifies the design structure explicitly
  - Distils design experience
- A design pattern has 4 basic parts:
  - 1. Name
  - 2. Problem
  - 3. Solution
  - 4. Consequences and trade-offs of application
- Language- and implementation-independent
- A "micro-architecture"
- GoF ("the gang of four") catalogue: "Design Patterns: Elements of Reusable Object-Oriented Software," Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995

List of Patterns (not the end..)

	Creational patterns
1	Abstract factory
2	Builder
3	Dependency Injection
4	Factory method
5	Lazy initialization
6	Multiton
7	Object pool
8	Prototype
9	Resource acquisition is initialization (RAII)
10	**Singleton**
	Concurrency patterns
11	Active Object
12	Balking
13	Binding properties
14	Blockchain
15	Compute kernel
16	Double-checked locking
17	Event-based asynchronous
18	Guarded suspension
19	Join
20	Lock
21	Messaging design pattern (MDP)
22	Monitor object
23	Reactor
24	Read-write lock
25	Scheduler
26	Thread pool
27	Thread-specific storage

	Structural patterns
28	**Adapter, Wrapper, or Translator**
29	Bridge
30	Composite
31	Decorator
32	Extension object
33	**Facade**
34	Flyweight
35	Front controller
36	Marker
37	Module
38	Proxy
39	Twin
	Behavioral patterns
40	Blackboard
41	Chain of responsibility
42	Command
43	Interpreter
44	Iterator
45	Mediator
46	Memento
47	Null object
48	**Observer or Publish/subscribe**
49	Servant
50	Specification
51	State
52	Strategy
53	Template method
54	Visitor

The Façade Pattern

Simplify, simplify, simplify!

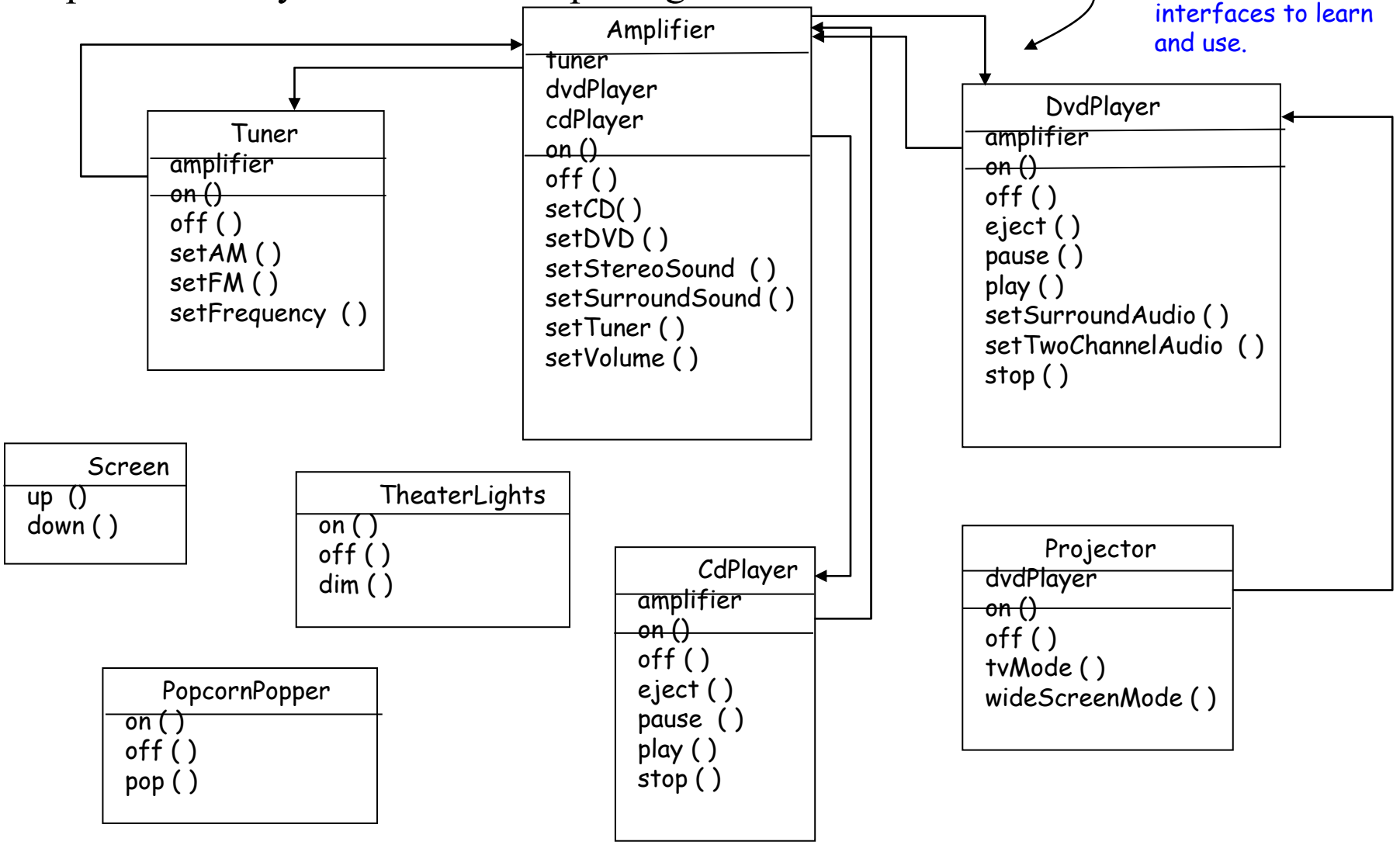
Façade

- Pattern that wraps objects, to simplify the interface
- Aptly named as this pattern hides all the complexity of one or more classes behind a clean, well-lit façade!

Home Sweet Home Theater

Building your own home theater - check out the components that you have/need to put together.

That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use.



Watching a Movie the Hard Way!

1. Turn on the popcorn popper
2. Start the popper popping
3. Dim the lights
4. Put the screen down
5. Turn the projector on
6. Set the projector input to DVD
7. Put the projector on wide-screen mode
8. Turn the sound amplifier on
9. Set the amplifier to DVD input
10. Set the amplifier to surround sound
11. Set the amplifier volume to medium (5)
12. Turn the DVD player on
13. Start the DVD player playing.
14. Whew!

But there's more!
When the movie is done, how do you turn everything off? Do you reverse all the steps?
Wouldn't it be just as complex to listen to a CD or radio?
If you decide to upgrade your system, you're probably going to have to learn a slightly different procedure!

Façade to the Rescue!!

Lights, Camera, Façade!

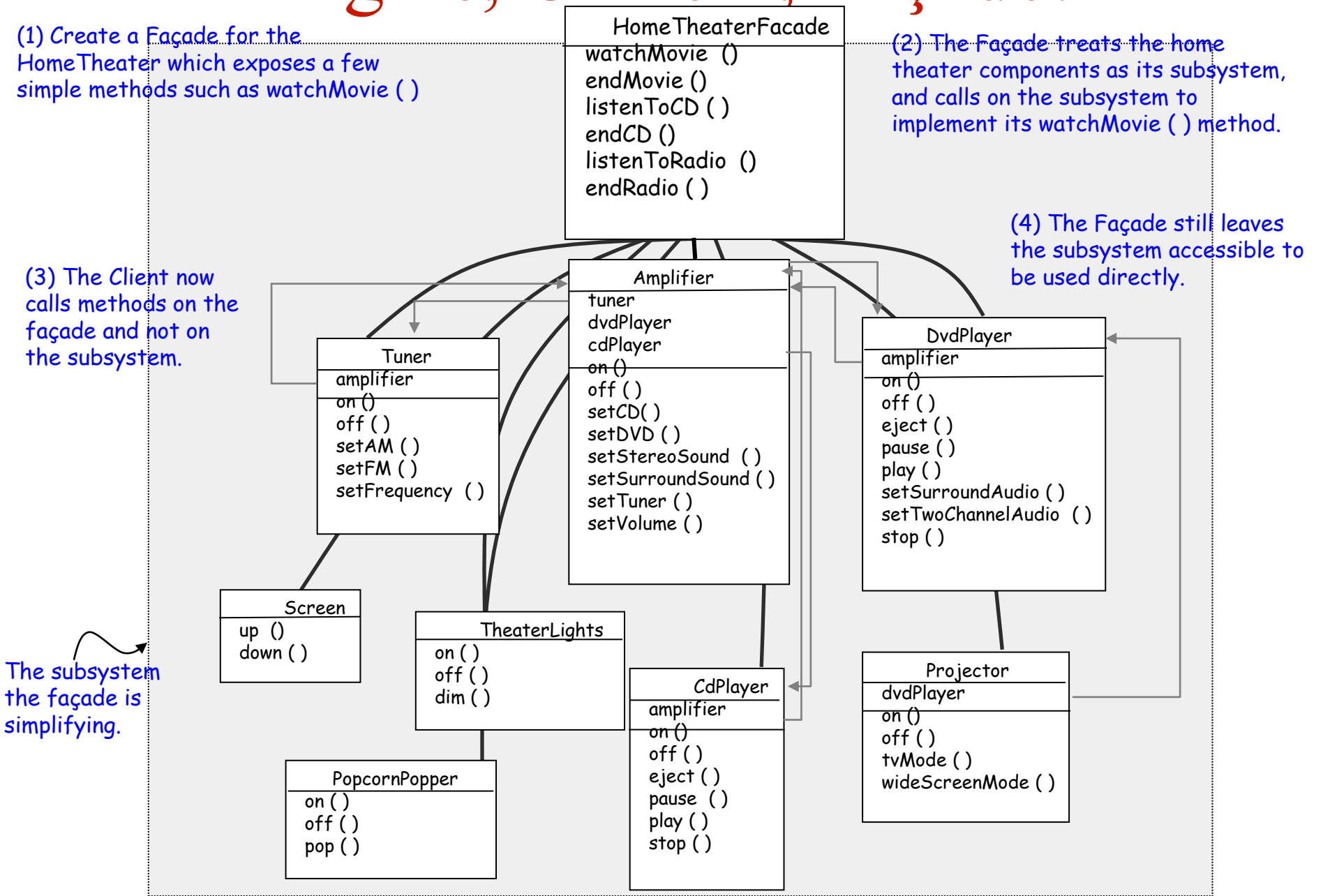
(1) Create a Façade for the HomeTheater which exposes a few simple methods such as watchMovie ( )

(2) The Façade treats the home theater components as its subsystem, and calls on the subsystem to implement its watchMovie ( ) method.

(3) The Client now calls methods on the façade and not on the subsystem.

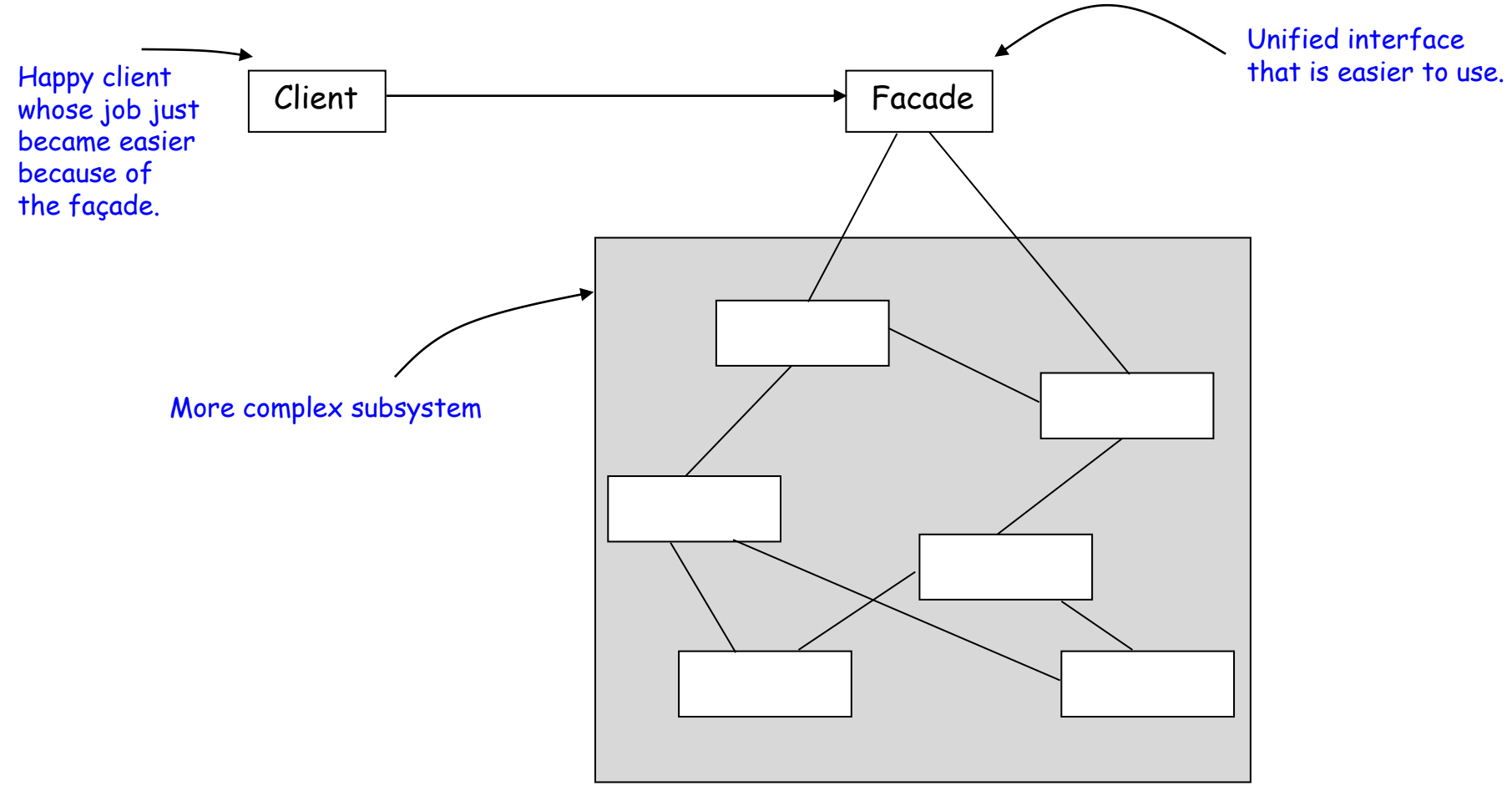
(4) The Façade still leaves the subsystem accessible to be used directly.

The subsystem the façade is simplifying.



The Façade Defined

The **Façade Pattern** provides a unified interface to a set of interfaces in a subsytem. Façade defines a higher-level interface that makes the subsystem easier to use.



Design Principle

- Principle of Least Knowledge - talk only to your immediate friends!

- What does it mean?
  - When designing a system, for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes.
- This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to the other parts.
  - When you build a lot of dependencies between many classes, you are building a fragile system that will be costly to maintain and complex for others to understand!

The Singleton Pattern

"One of a Kind Objects"

What is this?

- Singleton: How to instantiate just one object - one and only one!
- Why?
  - Many objects we need only one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings etc.
  - If more than one instantiated:
    - Incorrect program behavior, overuse of resources, inconsistent results
- Alternatives:
  - Use a *global variable*
    - Downside: assign an object to a global variable then that object might be created when application begins. If application never ends up using it and object is resource intensive --> waste!
  - Use a *static variable*
    - Downside: how do you prevent creation of more than one class object?

The Little Singleton (contd.)

- Is there any class that could use a private constructor?
- What's the meaning of the following?

```
public MyClass {
    public static MyClass getInstance ( ) { }
}
```

- Instantiating a class with a private constructor:

```
public MyClass {
    private MyClass ( ) { }
    public static MyClass getInstance ( ) { }
}
```

The Classic Singleton Pattern

```
public class Singleton {
    private static Singleton uniqueInstance;
    // other useful instance variables

    private Singleton ( ) { }
    public static Singleton getInstance ( ) {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton ( );
        }
        return uniqueInstance;
    }

    // other useful methods
}
```

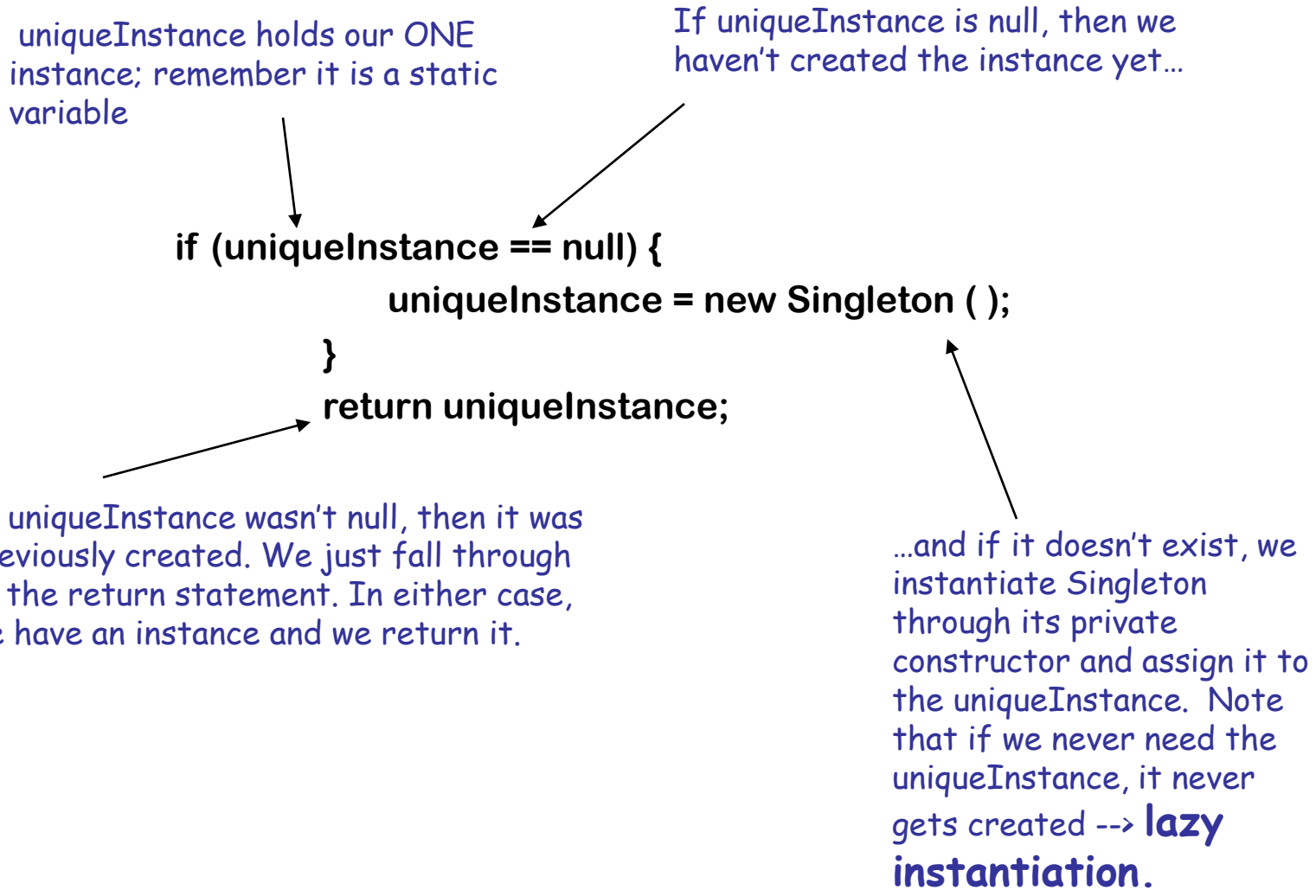
Constructor is declared private; only singleton can instantiate this class!

We have a static variable to hold our one instance of the class Singleton.

The getInstance ( ) method gives us a way to instantiate the class and also return an instance of it.

Of course, Singleton is a regular class so it has other useful instances and methods.

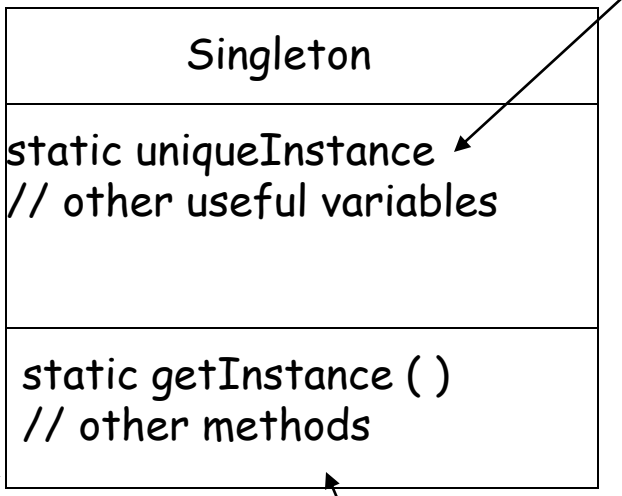
Code Up Close



Singleton Pattern Defined

The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.

The getInstance ( ) method is static, which means it is a class method, so you can conveniently access this method anywhere in your code using Singleton.getInstance ( ). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.



The uniqueInstance class variable holds our one and only one instance of Singleton.

A class implementing a Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

Singleton Summary

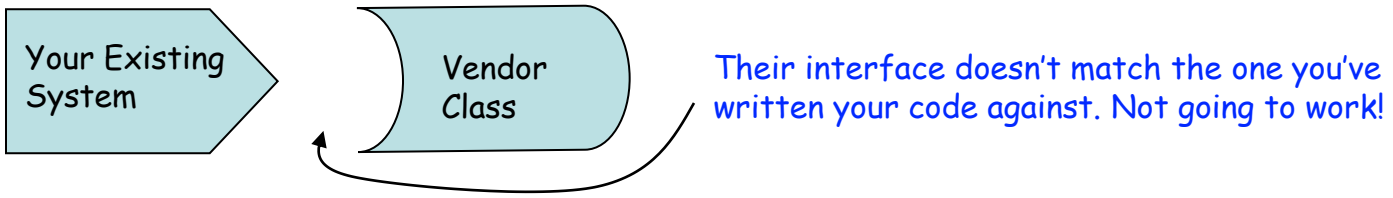
- The Singleton Pattern ensures you have at most one instance of a class in your application
- The Singleton Pattern also provides a global access point to that instance.
- Java's implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a static variable

The Adapter Pattern

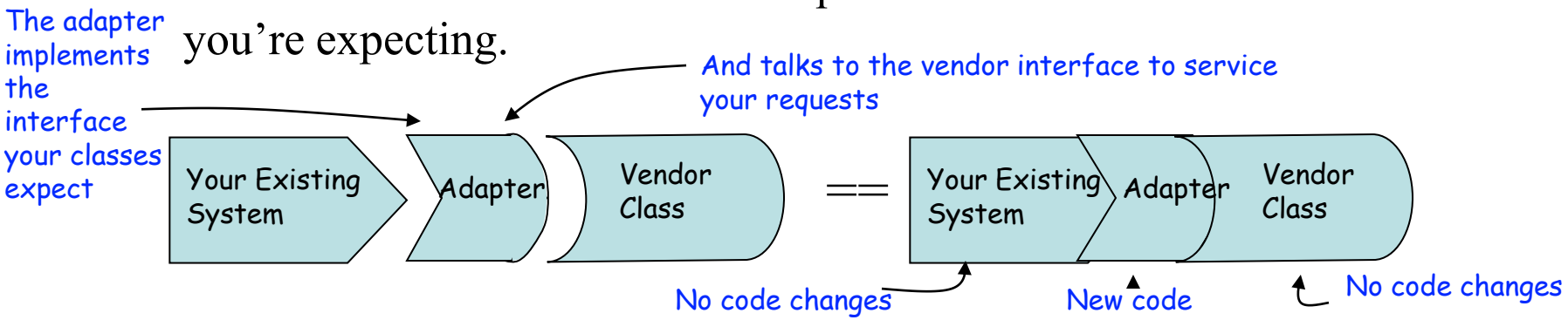
Putting a Square Peg in a Round Hole!

Adapters

- Real world is full of them!
  - Some examples?
- Object oriented adapters
  - Scenario: you have an existing software system that you need to work a new vendor library into, but the new vendor designed their interfaces differently than the last vendor.



  - What to do? Write a class that adapts the new vendor interface into the one you're expecting.



If it walks like a duck…..

- If it walks like a duck and quacks like a duck, then it might be a ~~duck~~ turkey wrapped with a duck adapter….

```
public interface Duck {
    public void quack ();
    public void fly ();
}
public class MallardDuck implements Duck {
    public void quack () {
        System.out.println("Quack");
    }
    public void fly ( ) {
        System.out.println ("I am flying");
    }
}
```

Meet the fowl!

```
public interface Turkey {
    public void gobble ();
    public void fly ( );
}

public class WildTurkey implements Turkey {
    public void gobble ( ) {
        System.out.println("Gobble Gobble");
    }
    public void fly ( ){
        System.out.println("I'm flying a short distance")
    }
}
```

Concrete implementations are similar -- just print out the actions.

Now….

- Lets say you are short on Duck objects and would like to use some Turkey objects in their place.
  - Can't use them outright because they have a different interface.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;
     public TurkeyAdapter (Turkey turkey) {
        this.turkey = turkey;
     }
     public void quack ( ) {
        turkey.gobble ( );
     }
     public void fly ( ){
        for (int j = 0; j<5; j++)
            turkey.fly ( );
     }
  }
}
```

First, you need to implement the interface of the type you are adapting to. This is the interface your client expects.

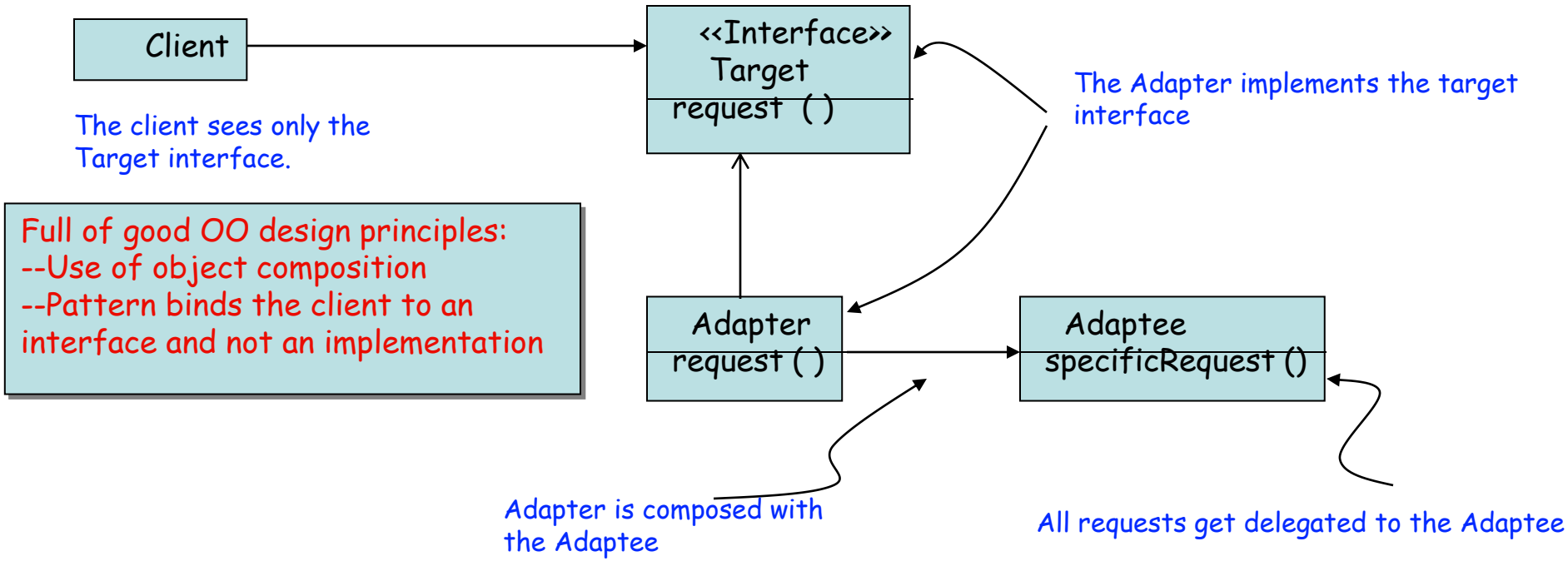
Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy; just call the gobble method.

Even though both interfaces have a fly ( ) method, Turkeys fly in short spurts -- they can't do long distance flying like ducks. To map between a Duck's fly ( ) method and a Turkey's we need to call the Turkey's fly ( ) method five times to make up for it.

The Adapter Pattern Defined

The **Adapter Pattern** converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



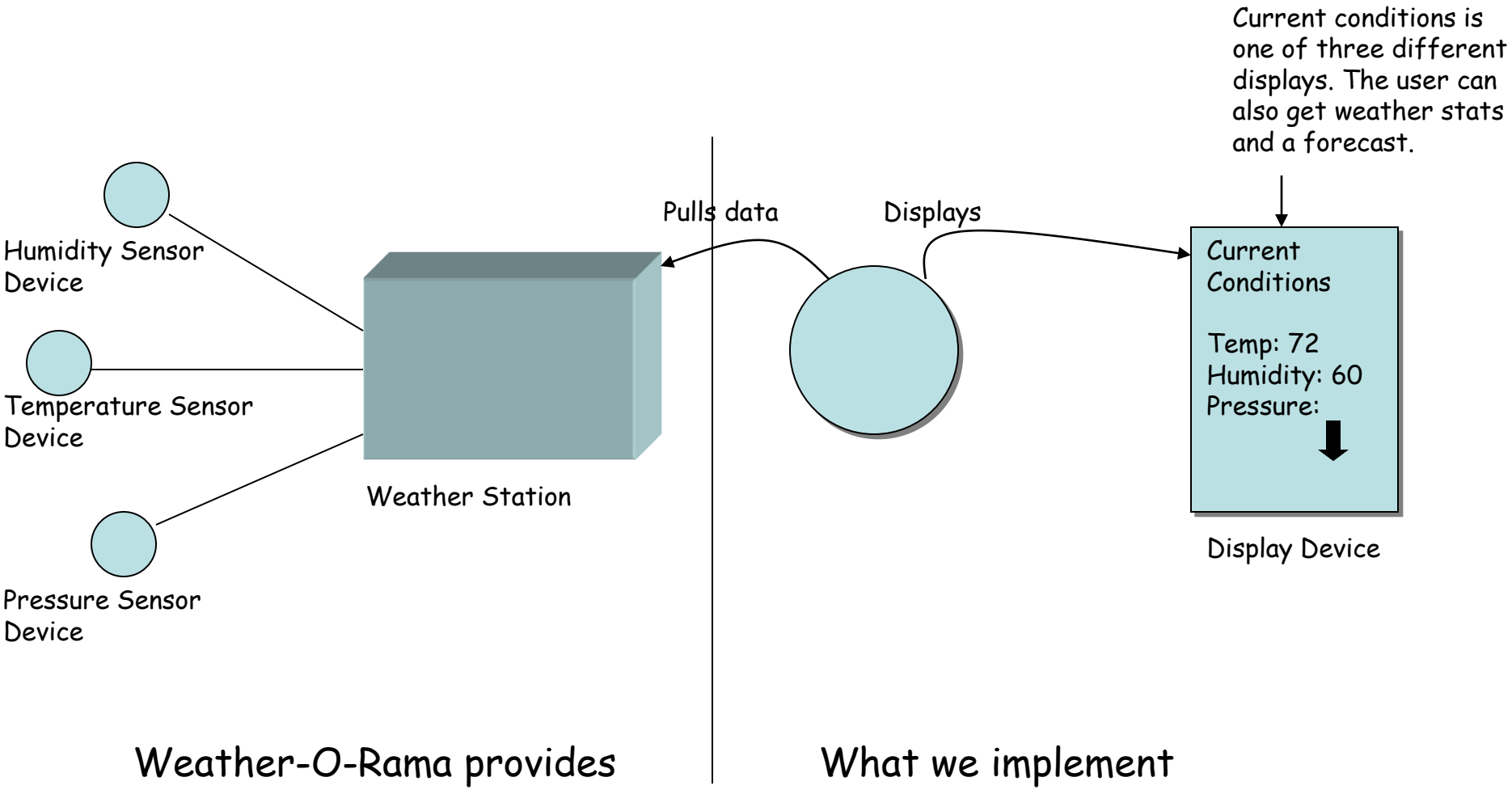
Adapter Summary

- Pattern that wraps objects to change its interface
- When you need to use an existing class and its interface is not the one you need, use an adapter.
- An adapter changes an interface into one a client expects.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- An adapter wraps an object to change its interface, a decorator (another pattern) wraps an object to add new behaviors and responsibilities.

Observer Pattern

Keeping your Objects in the Know!

The Weather-O-Rama!



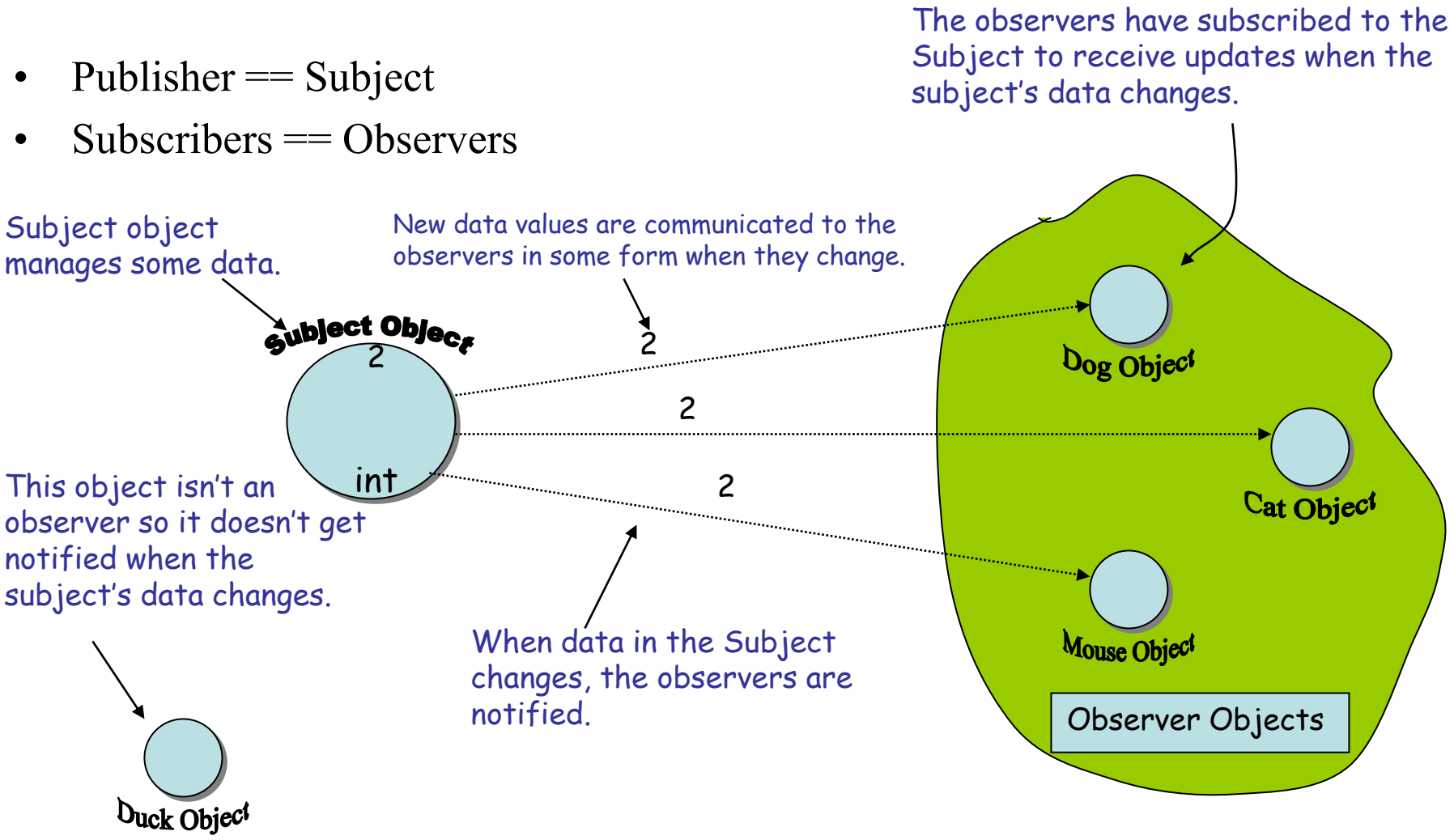
The Job: Create an app that uses the WeatherData object to update three displays for current conditions, weather stats, and a forecast.

Time for the Observer!

- The Newspaper or Magazine subscription model:
  - A newspaper publisher goes into business and begins publishing newspapers
  - You subscribe to a particular newspaper, and every time there is a new edition, its gets delivered to you. As long as you remain a subscriber, you get new newspapers.
  - You unsubscribe when you don't want the newspapers anymore -- and they stop being delivered
  - While the publisher remains in business people, hotels, airlines etc constantly subscribe and unsubscribe to the newspaper.

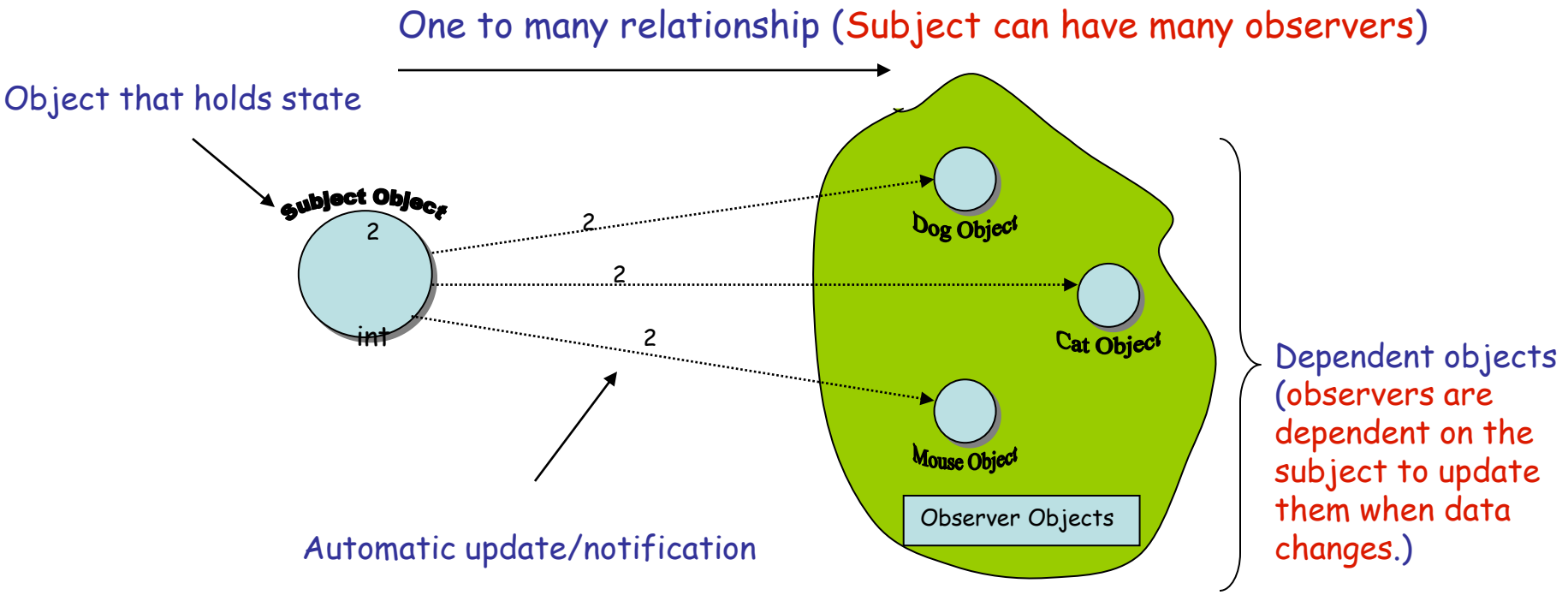
Publishers + Subscribers = Observer Pattern

- Publisher == Subject
- Subscribers == Observers



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.

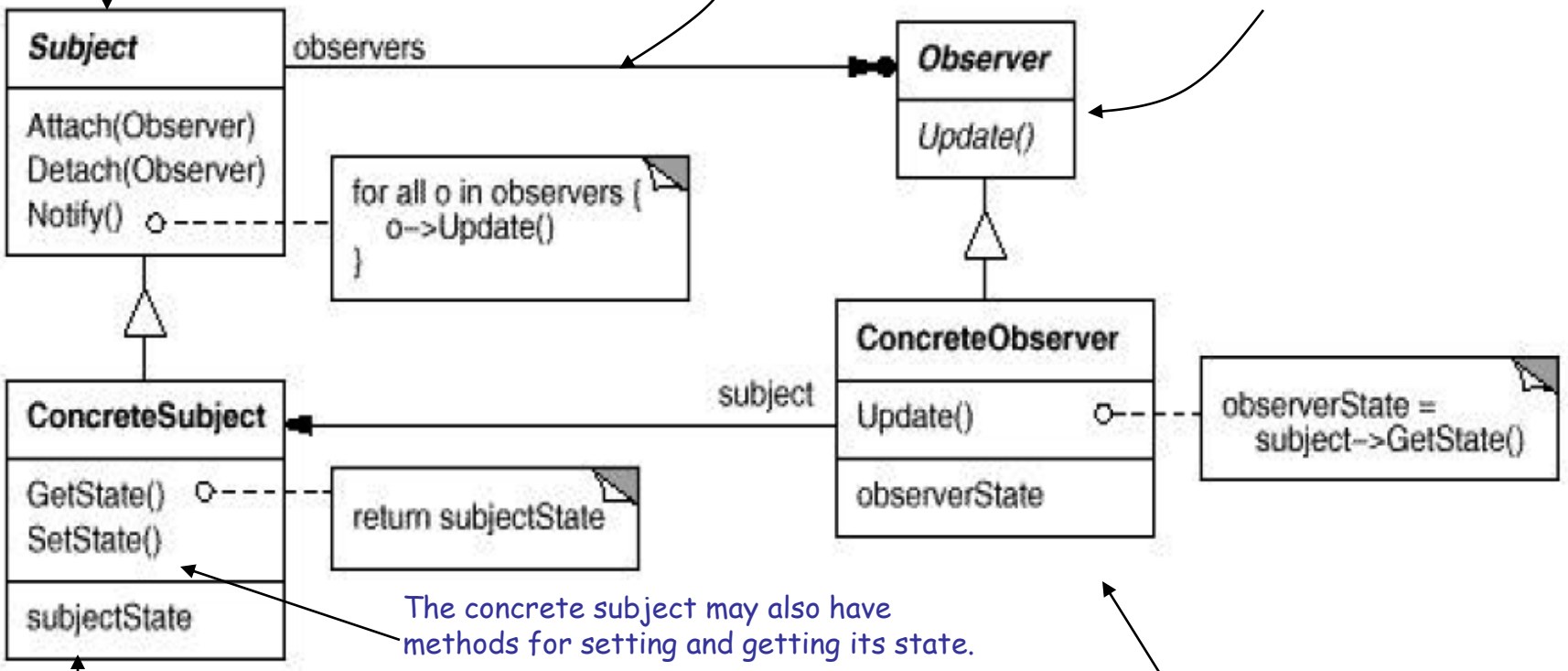


Observer Class Diagram

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers

All potential observers need to implement the Observer interface. This interface has just one method, update ( ), that gets called when the Subject's state changes.

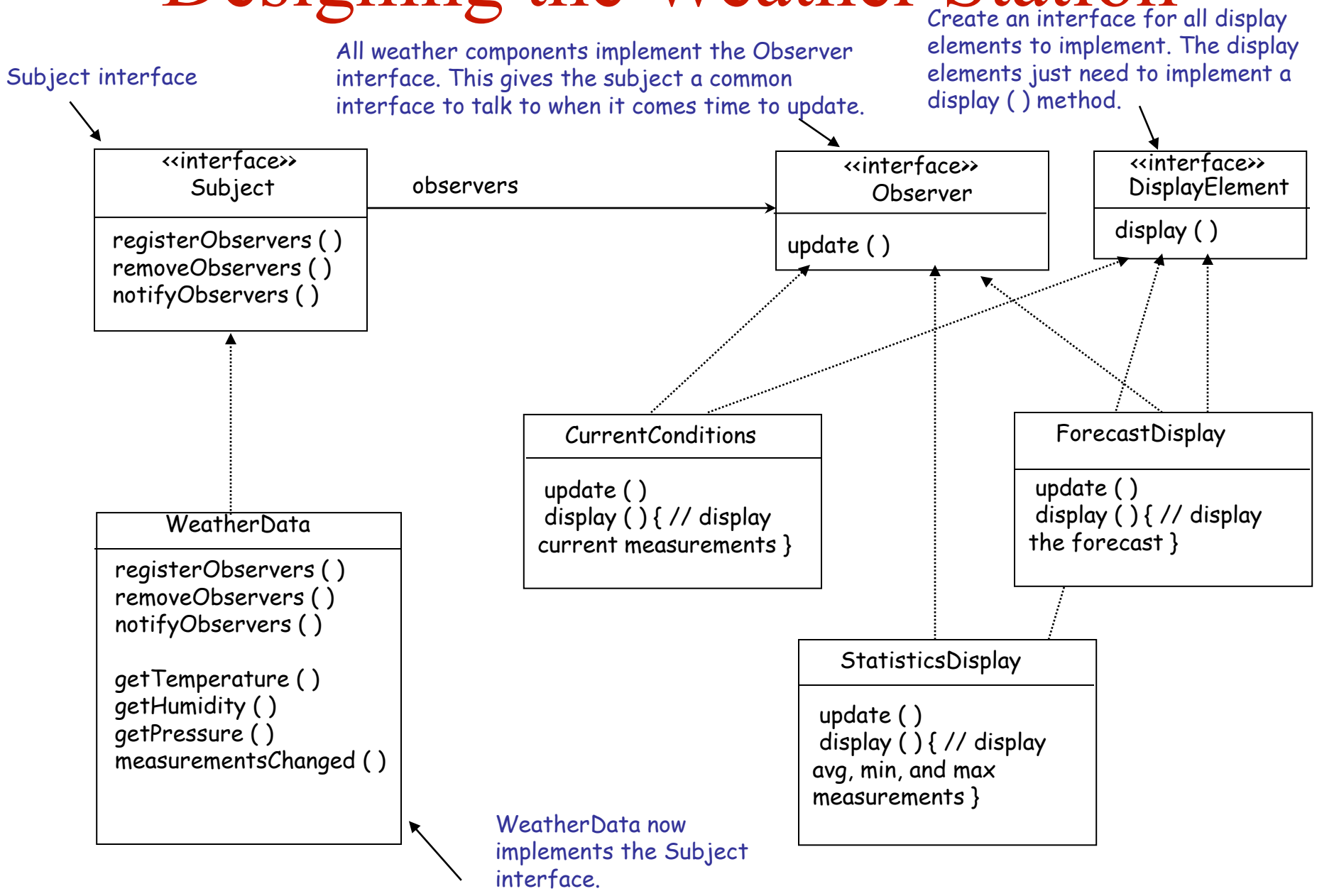


The concrete subject may also have methods for setting and getting its state.

A concrete subject always implements the Subject interface. In addition to the register (attach) and remove (detach) methods, the concrete subject implements a notify() method to notify 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.

Designing the Weather Station



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 ( );
}
```

Both of these methods take an Observer as an argument, that is the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

The Observer interface is implemented by all observers, so they all have to implement the update ( ) method.

These are the state values the Observers get from the Subject when a weather measurement changes.

The DisplayElement interface just includes one method, display ( ), that we will call when the display element needs to be displayed.

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 j = observer.indexOf(o);
            if (j >= 0) {
                  observers.remove(j);
            } }
      public void notifyObservers ( ) {
            for (int j = 0; j < observers.size(); j++) {
                  Observer observer = (Observer)observers.get(j);
                  observer.update(temperature, humidity, pressure);
            }}
      public void measurementsChanged ( ) {
            notifyObservers ( ); }
      // add a set method for testing + other methods.
}
```

Added an ArrayList to hold the Observers, and we create it in the constructor

Here we implement the Subject Interface

Notify the observers when measurements change.

Observer Summary

- **OO Principle** in play: *Strive for loosely coupled designs between objects that interact.*
- Main points:
  - The Observer pattern defines a *one to many relationship* between objects
  - Subjects (observables), update Observers using a common interface
  - Observers are loosely coupled in that the Observable knows nothing about them, other than they implement the Observer interface.
  - You can push or pull data from the Observable when using the pattern *("pull" is considered more correct*)
  - Don't depend on a specific order of notification for your Observers
  - Java has several implementations of the Observer Pattern including the general purpose java.util.Observable
  - Watch out for issues with java.util.Observable
  - Don't be afraid to create our own version of the Observable if needed
  - Swing makes heavy use of the Observer pattern, as do many GUI frameworks
  - You find this pattern in other places as well including JavaBeans and RMI.