Introduction to Design Patterns

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 19 — 10/28/2008

Lecture Goals

- Cover Material from Chapter 1 and 2 of the Design Patterns Textbook
    - Introduction to Design Patterns
    - Strategy Pattern
    - Observer Pattern

Why Patterns? (I)

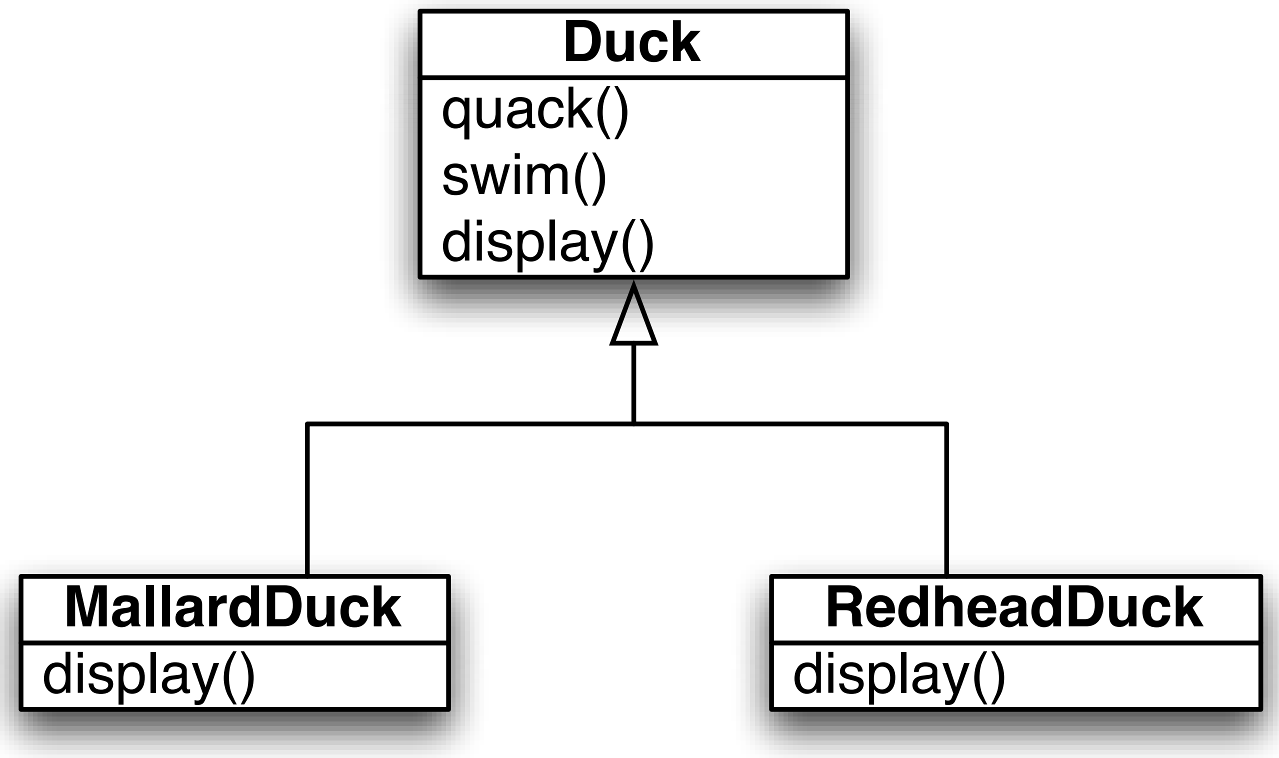
- As the Design Guru says
    - “Remember, knowing concepts like
        - **abstraction**,
        - **inheritance**, and
        - **polymorphism**
    - do **not** make you a good OO designer.
    - A design guru thinks about how to create **flexible designs** that are **maintainable** and that can **cope with change**.”

Why Patterns? (II)

- Someone has already solved your problems (!)
    - Design patterns allow you to exploit the wisdom and lessons learned by other developers who’ve encountered design problems similar to the ones you are encountering
- The best way to use design patterns is to **load your brain with them** and then **recognize places** in your designs and existing applications where you can **apply them**
- Instead of **code reuse**, you get **experience reuse**

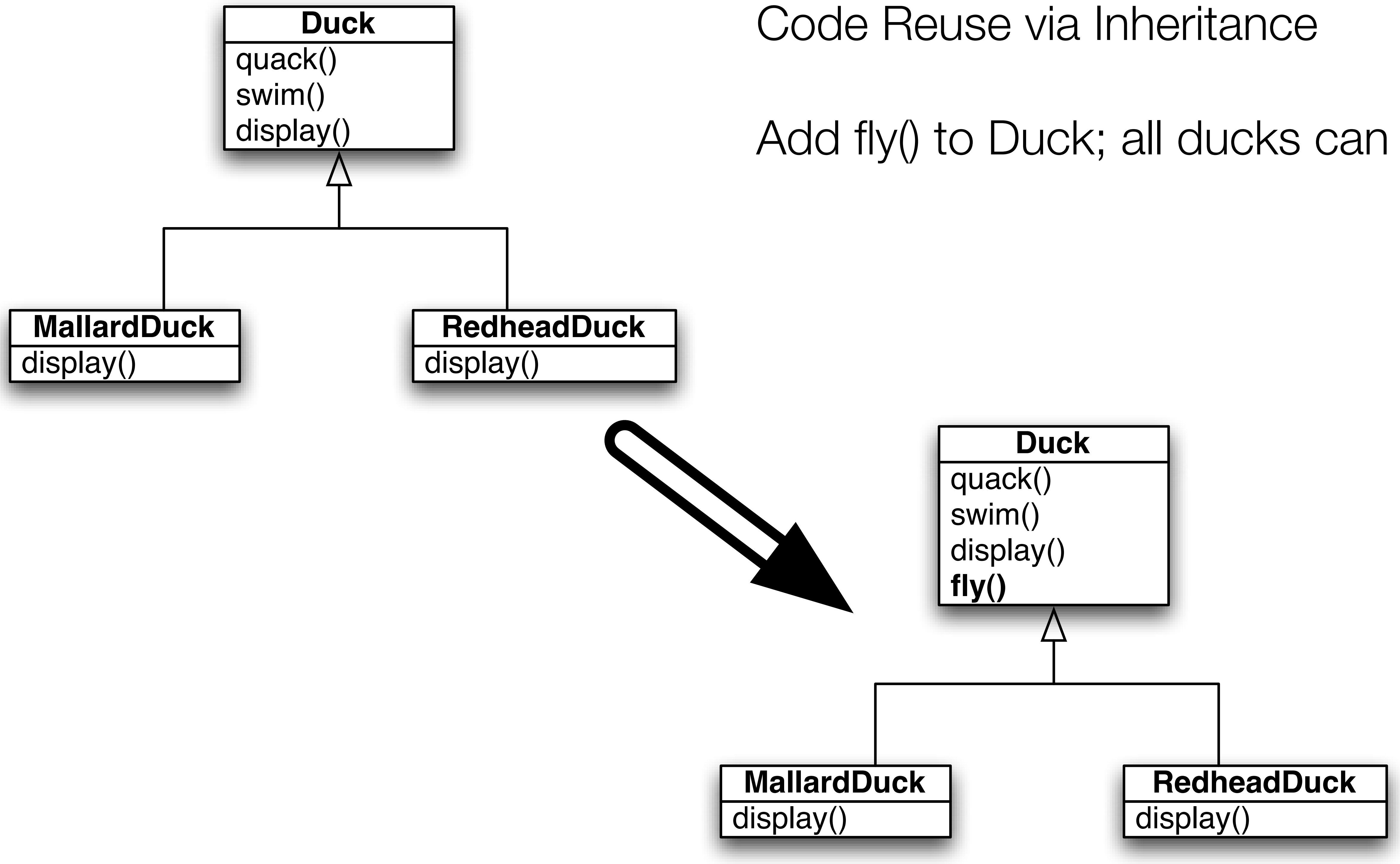
Design Pattern by Example

- SimUDuck: a “duck pond simulator” that can show a wide variety of duck species swimming and quacking
  - Initial State



  - But a request has arrived to allow ducks to also fly. (We need to stay ahead of the competition!)

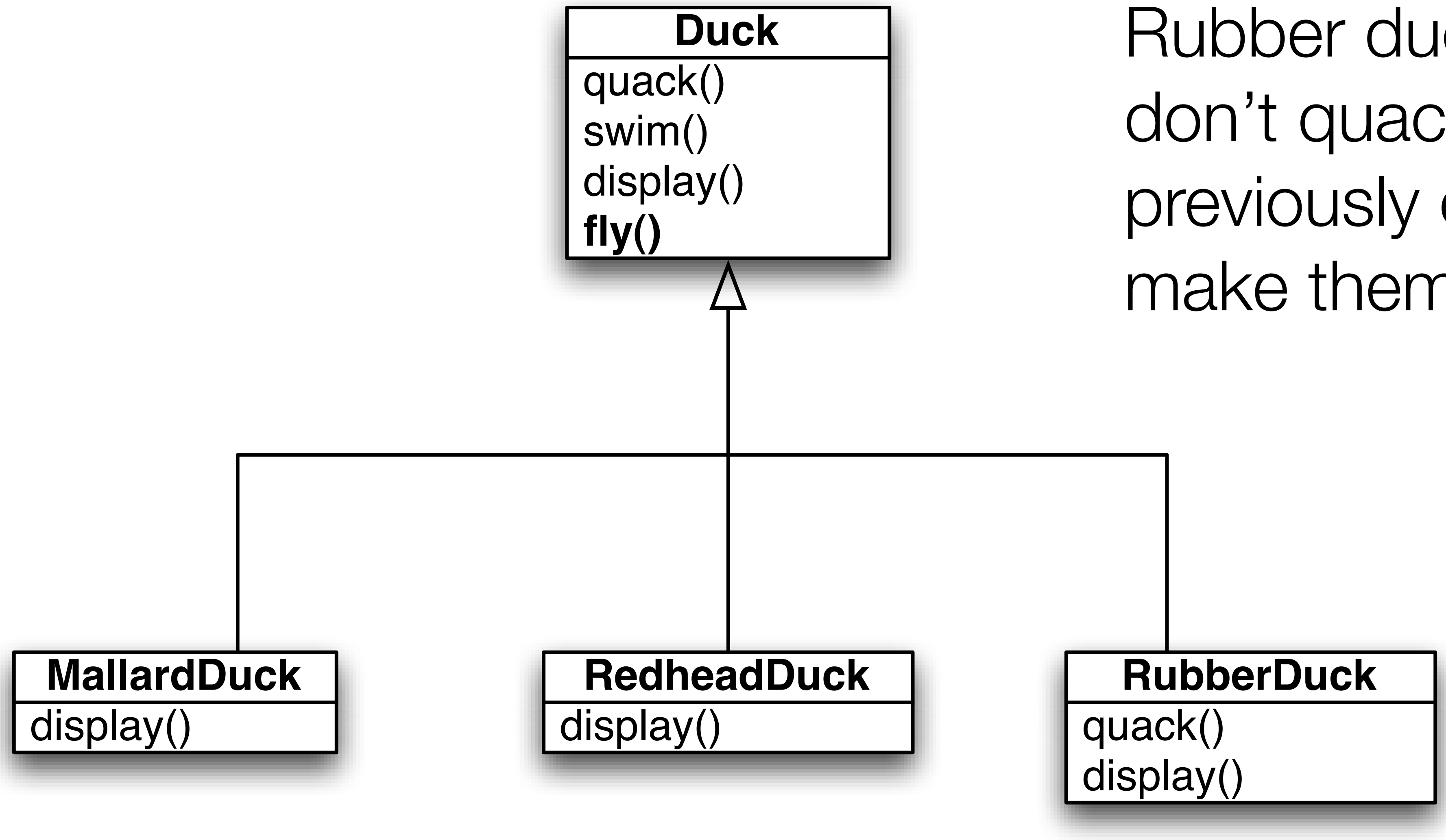
Easy



Code Reuse via Inheritance

Add fly() to Duck; all ducks can now fly

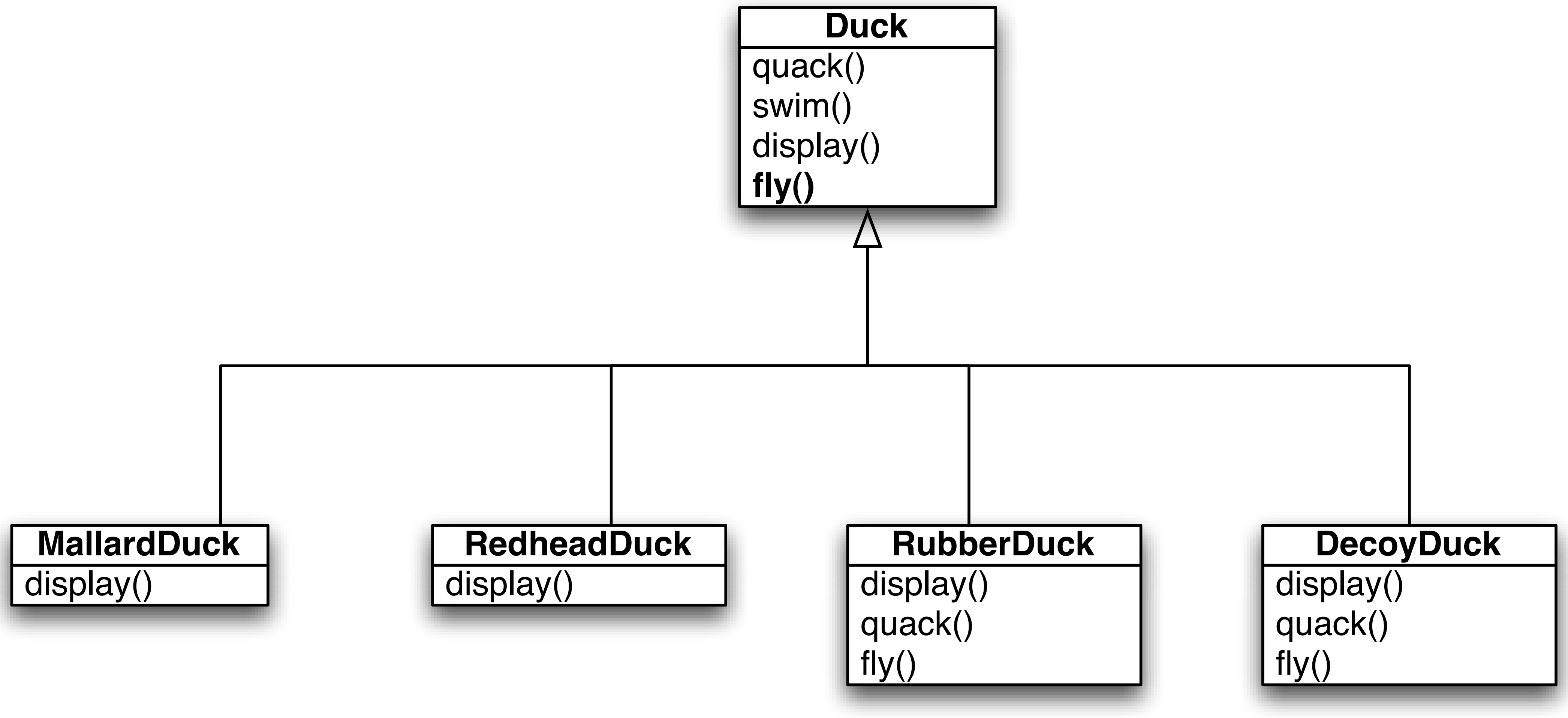
Whoops!



Rubber ducks do not fly! They don't quack either, so we had previously overridden quack() to make them squeak.

We could override fly() in RubberDuck to make it do nothing, but that's less than ideal, especially...

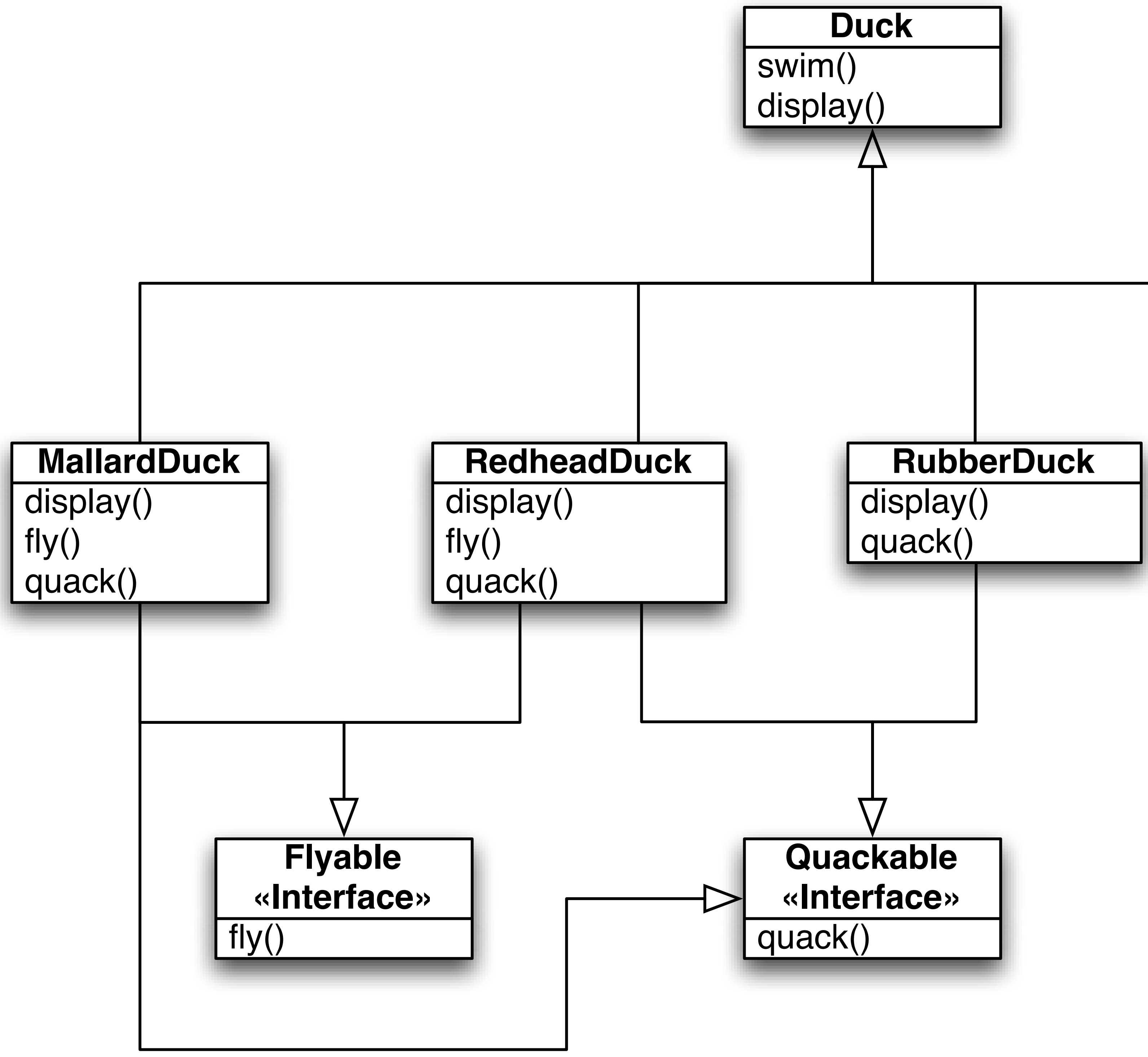
Double Whoops!



…when we might always find other Duck subclasses that would have to do the same thing.

What was supposed to be a good instance of **reuse via inheritance** has turned into a **maintenance headache**!

What about an Interface?



Here we define two interfaces and allow subclasses to implement the interfaces they need.

What are the trade-offs?

Design Trade-Offs

- With inheritance, we get
  - code reuse, only one fly() and quack() method vs. multiple (pro)
  - common behavior in root class, not so common after all (con)
- With interfaces, we get
  - specificity: only those subclasses that need a fly() method get it (pro)
  - no code re-use: since interfaces only define signatures (con)
- Use of abstract base class over an interface? Could do it, but only in languages that support multiple inheritance
  - In this approach, you implement Flyable and Quackable as abstract base classes and then have Duck subclasses use multiple inheritance
    - Trade-Offs?

OO Principles to the Rescue!

- Encapsulate What Varies
  - Recall the InstrumentSpec example from the OO A&D textbook
    - The “what varies” part was the properties between InstrumentSpec subclasses
    - What we needed was “dynamic properties” and the solution entailed getting rid of all the subclasses and storing the properties in a hash table
- For this particular problem, the “what varies” is the behaviors between Duck subclasses
  - We need to pull out behaviors that vary across subclasses and put them in their own classes (i.e. encapsulate them)
- The result: fewer unintended consequences from code changes (such as when we added fly() to Duck) and more flexible code

Basic Idea

- Take any behavior that varies across Duck subclasses and pull them out of Duck
  - Duck’s will no longer have fly() and quack() methods directly
  - Create two sets of classes, one that implements fly behaviors and one that implements quack behaviors
- Code to an Interface
  - We’ll make use of the “code to an interface” principle and make sure that each member of the two sets implements a particular interface
    - For QuackBehavior, we’ll have Quack, Squeak, Silence
    - For FlyBehavior, we’ll have FlyWithWings, CantFly, FlyWhenThrown, …
- Additional benefits
  - Other classes can gain access to these behaviors (if that makes sense) and we can add additional behaviors without impacting other classes

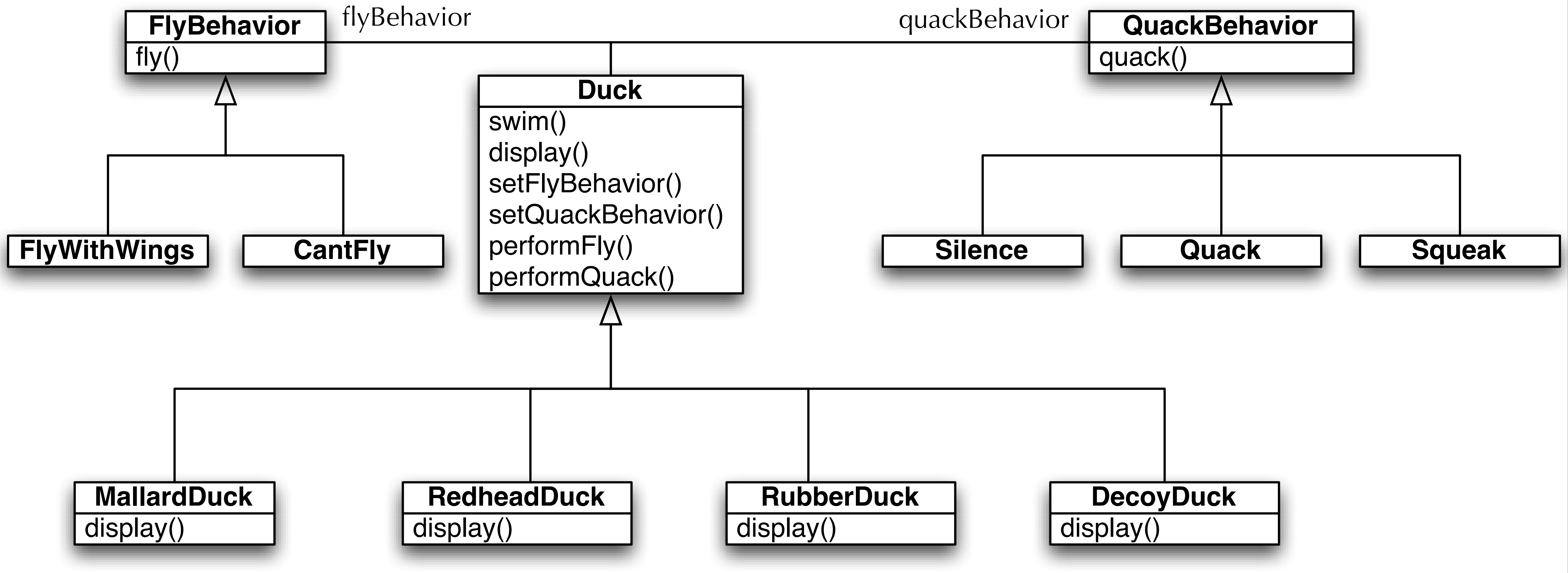
“Code to Interface” Does NOT Imply Java Interface

- We are overloading the word “interface” when we say “code to an interface”
    - We can implement “code to an interface” by defining a Java interface and then have various classes implement that interface
    - Or, we can “code to a supertype” and instead define an abstract base class which classes can access via inheritance.
- When we say “code to an interface” it implies that the object that is using the interface will have a variable whose type is the supertype (whether its an interface or abstract base class) and thus
    - can point at any implementation of that supertype
    - and is shielded from their specific class names
        - A Duck will point to a fly behavior with a variable of type FlyBehavior NOT FlyWithWings; the code will be more loosely coupled as a result

Bringing It All Together: Delegation

- To take advantage of these new behaviors, we must modify Duck to delegate its flying and quacking behaviors to these other classes
  - rather than implementing this behavior internally
- We'll add two attributes that store the desired behavior and we'll rename fly() and quack() to performFly() and performQuack()
  - this last step is meant to address the issue of it not making sense for a DecoyDuck to have methods like fly() and quack() directly as part of its interface
    - Instead, it inherits these methods and plugs-in CantFly and Silence behaviors to make sure that it does the right thing if those methods are invoked
- This is an instance of the principle "Favor composition over inheritance"

New Class Diagram



FlyBehavior and QuackBehavior define a set of behaviors that provide behavior to Duck. Duck is composing each set of behaviors and can switch among them dynamically, if needed. While now each subclass has a performFly() and performQuack() method, at least the user interface is uniform and those methods can point to null behaviors when required.

Duck.java

```java
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {
    }

    public void setFlyBehavior (FlyBehavior fb) {
        flyBehavior = fb;
    }

    public void setQuackBehavior(QuackBehavior qb) {
        quackBehavior = qb;
    }

    abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}

```

Note: “code to interface”, delegation, encapsulation, and ability to change behaviors dynamically

DuckSimulator.java

```java
public class MiniDuckSimulator {

    public static void main(String[] args) {

        Duck    mallard = new MallardDuck();
        Duck    rubberDuckie = new RubberDuck();
        Duck    decoy = new DecoyDuck();

        Duck    model = new ModelDuck();

        mallard.performQuack();
        rubberDuckie.performQuack();
        decoy.performQuack();

        model.performFly();
        model.setFlyBehavior(new FlyRocketPowered());
        model.performFly();
    }
}

```

Note: all variables are of type Duck, not the specific subtypes; “code to interface” in action

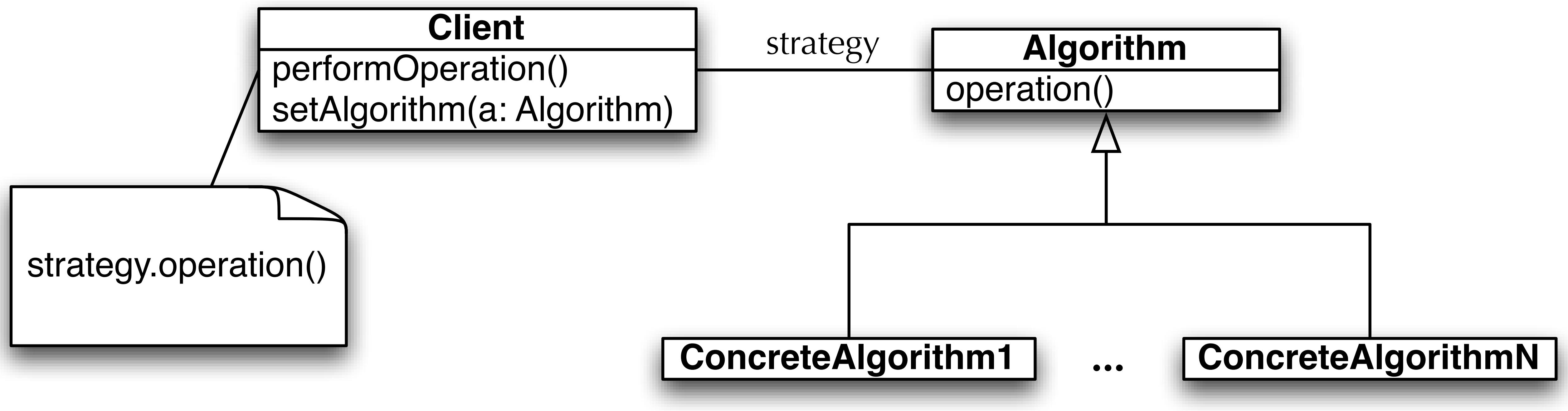
Not Completely Decoupled

- Is DuckSimulator completely decoupled from the Duck subclasses?
    - All of its variables are of type Duck
- No!
    - The subclasses are still coded into DuckSimulator
        - Duck mallard = **new MallardDuck()**;
- This is a type of coupling… fortunately, we can eliminate this type of coupling if needed, using a pattern called **Factory**.
    - We'll see Factory in action later this semester

Meet the Strategy Design Pattern

- The solution that we applied to this design problem is known as the Strategy Design Pattern
  - It features the following OO design concepts/principles:
    - Encapsulate What Varies
    - Code to an Interface
    - Delegation
    - Favor Composition over Inheritance
- Definition: The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it

Structure of Strategy



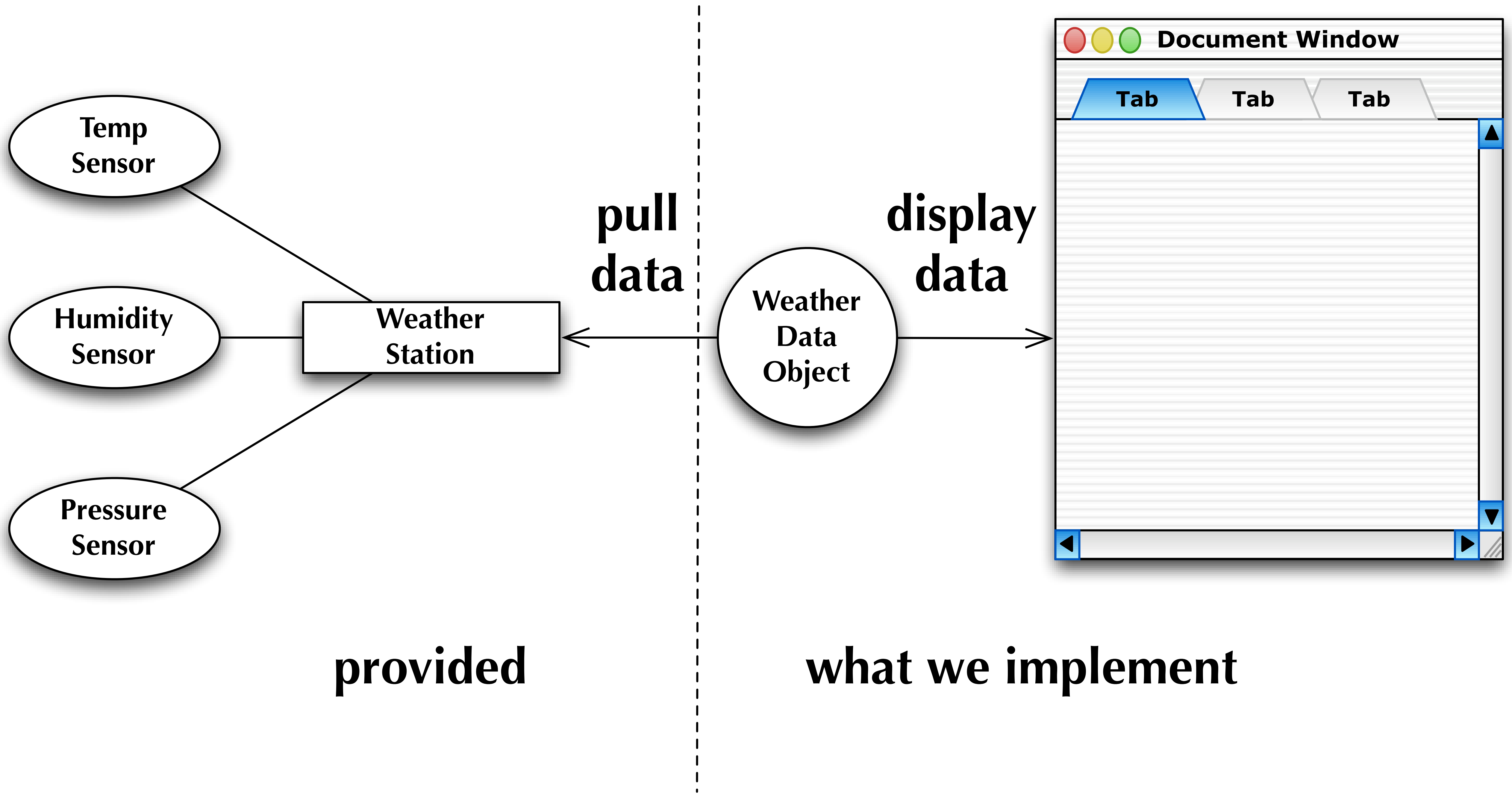
Algorithm is pulled out of Client. Client only makes use of public interface of Algorithm and is not tied to concrete subclasses.

Client can change its behavior by switching among the various concrete algorithms

Observer Pattern

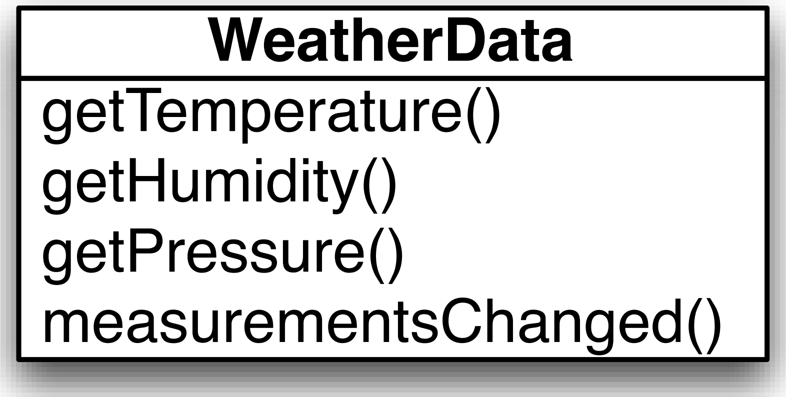
- Don’t miss out when something interesting (in your system) happens!
    - The observer pattern allows objects to keep other objects informed about events occurring within a software system (or across multiple systems)
    - Its dynamic in that an object can choose to receive notifications or not at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

Chapter Example: Weather Monitoring



We need to pull information from the 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… so that's simple!

First pass at measurementsChanged

```java
...

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);

}

...

```

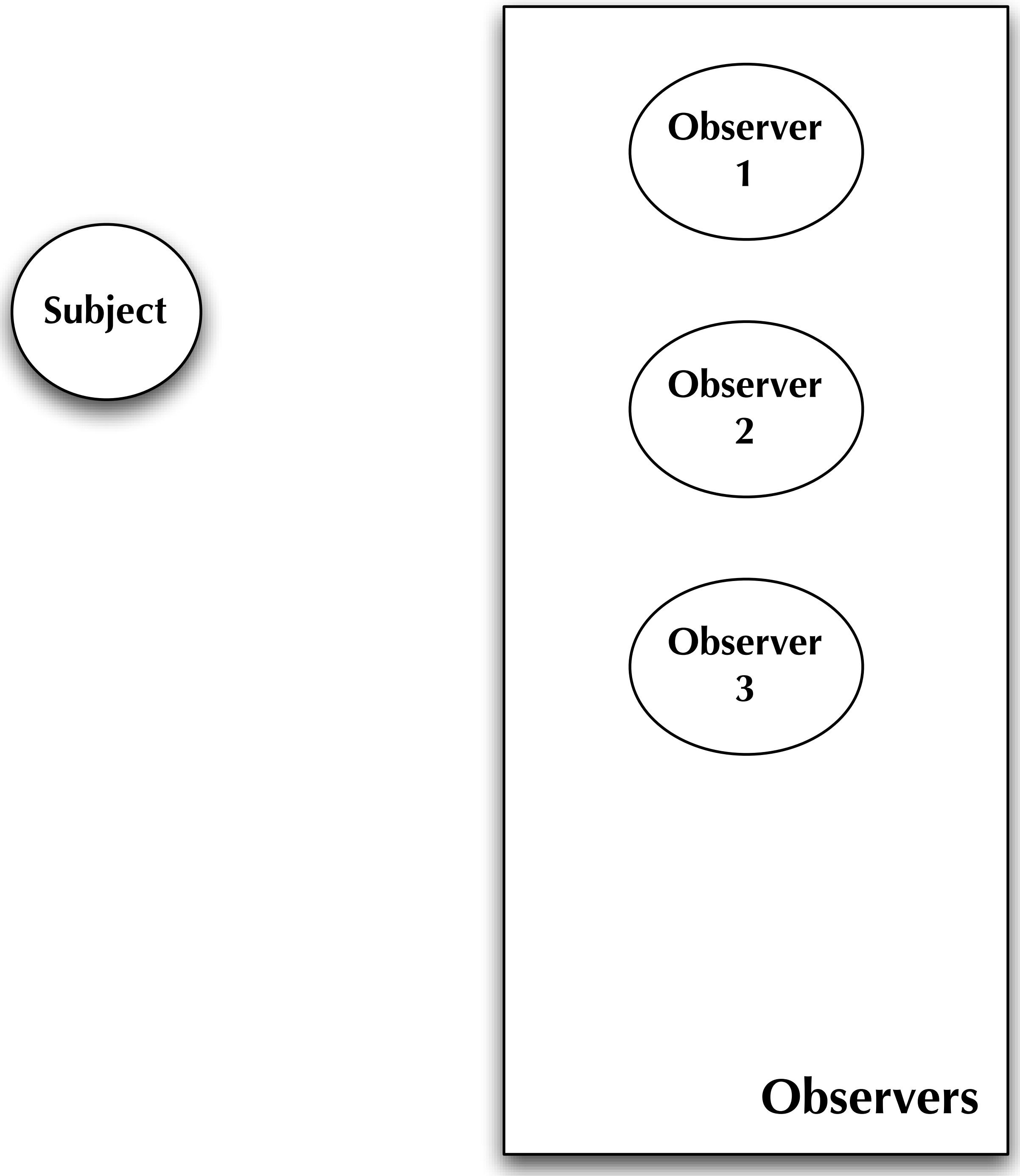
Problems?

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!** Although each implementation has adopted the same interface, so this will make translation easy!

Observer Pattern

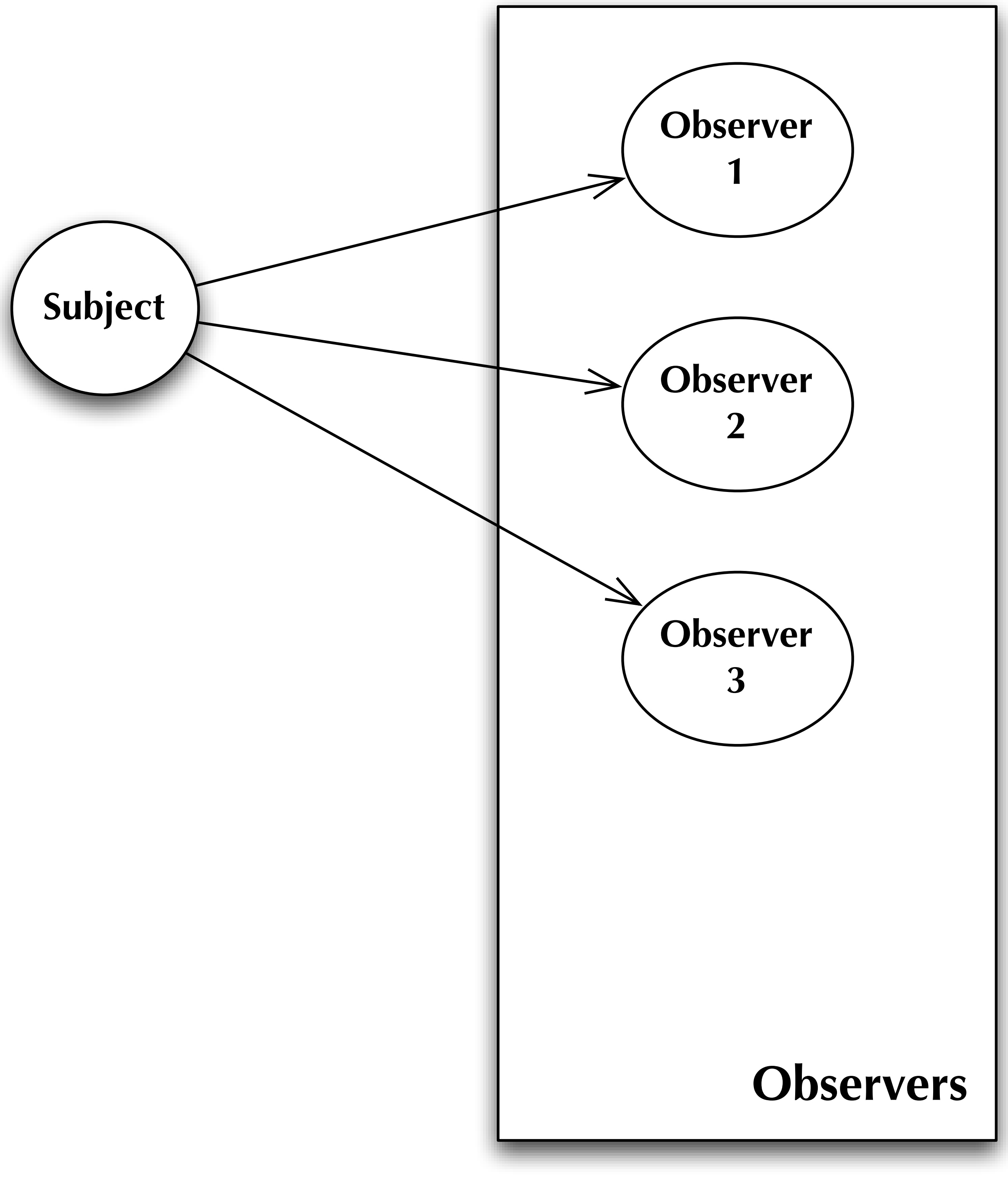
- This situation can benefit from use of the observer pattern
    - This pattern is similar to subscribing to a hard copy newspaper
        - A newspaper comes into existence and starts publishing editions
        - You become interested in the newspaper and subscribe to it
        - Any time an edition 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)



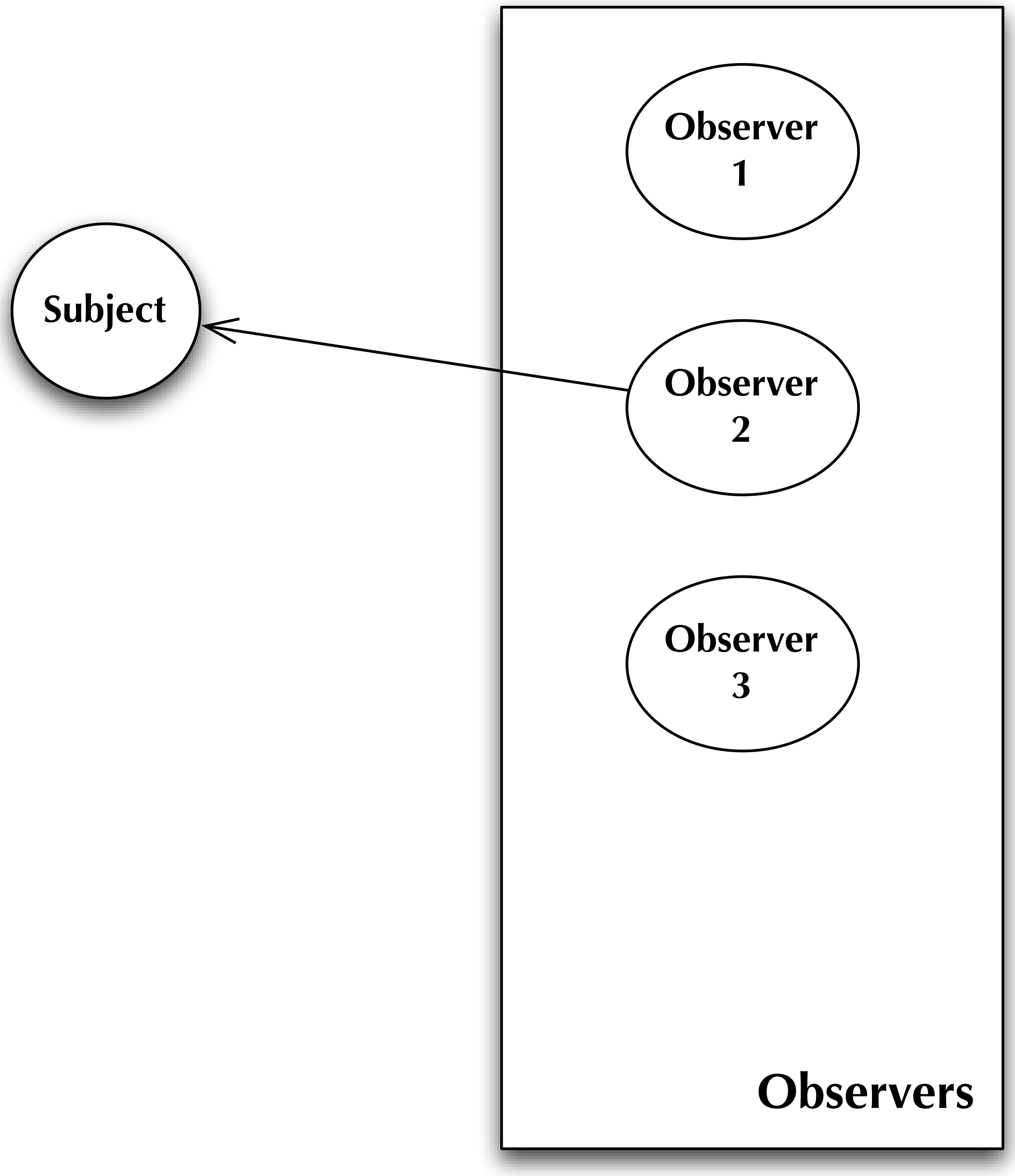
Subject maintains a list of observers

Observer in Action (II)



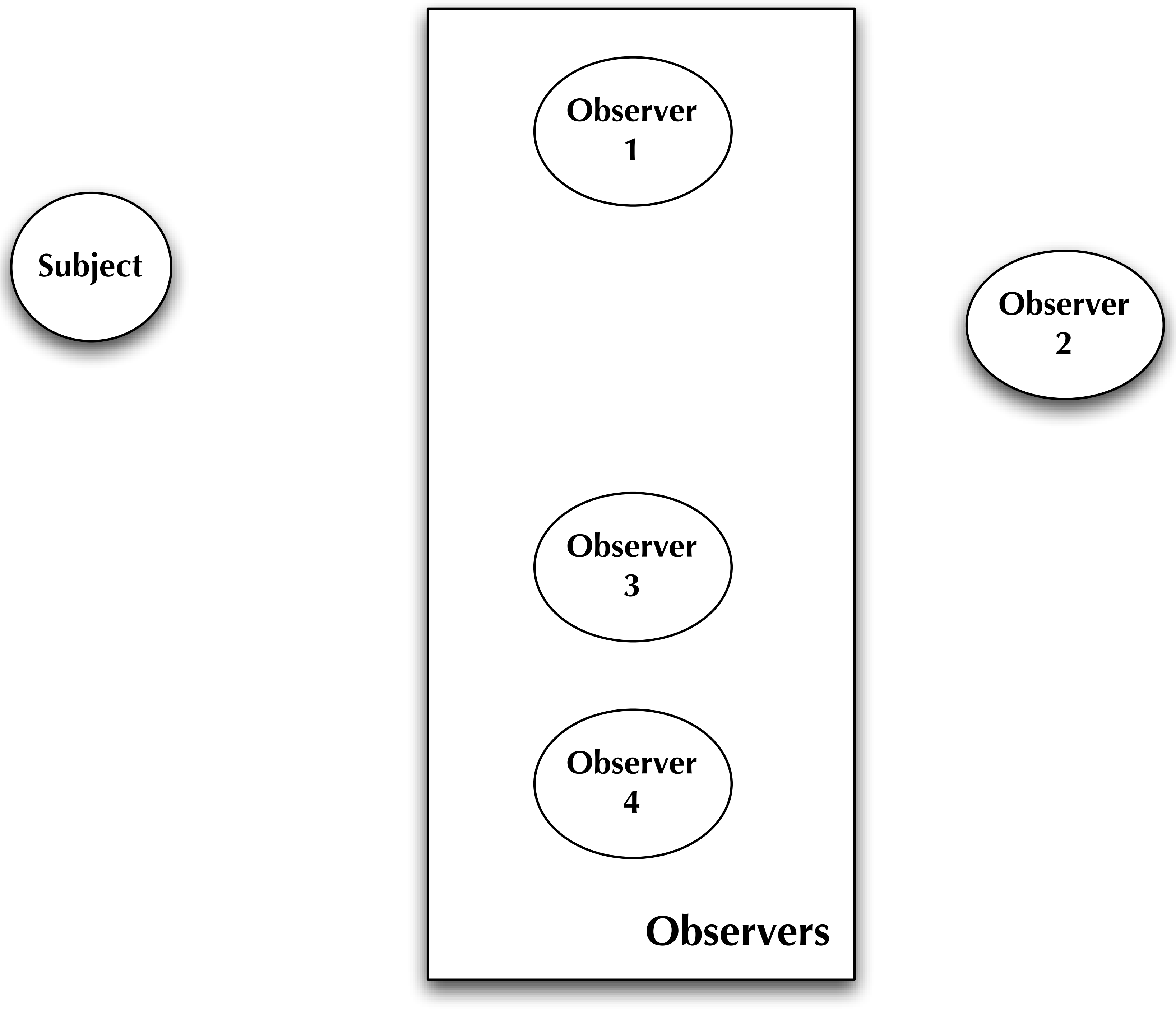
If the Subject changes, it notifies its observers

Observer in Action (III)



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

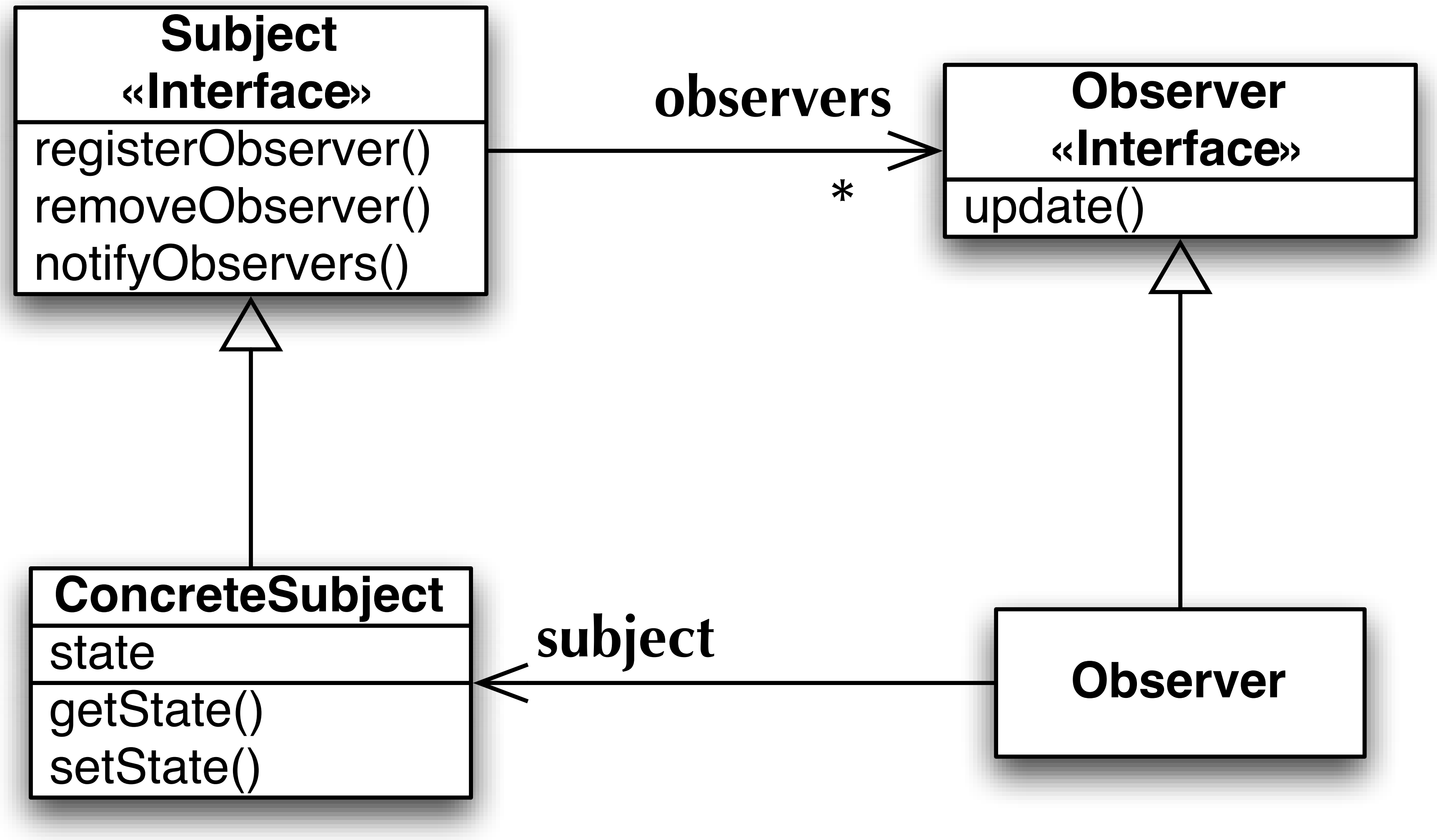
Observer In Action (IV)



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) changes all of its dependents (observers) 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 where ever 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

Demonstration

- Roll Your Own Observer
- 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)
- Observer in Swing
  - Listener framework is just another name for the Observer pattern

The Importance of Shared Vocabulary (I)

- Design Patterns are important because they provide a shared vocabulary to software design
  - (In addition, to being really useful solutions to tricky design problems!)
- Compare:
  - So I created this broadcast class. It tracks a set of listeners and anytime its data changes, it sends a message to the listeners. Listeners can join and leave at any time. It’s really dynamic and loosely-coupled.
- With:
  - I used the Observer Design Pattern

The Importance of Shared Vocabulary (II)

- Shared pattern vocabularies are powerful
    - You communicate not just a name, but a whole set of qualities, services, and constraints associated with the pattern
- Patterns allow you to say more with less
    - Other developers quickly pick up on the design you are proposing
- Talking about patterns, lets you “stay in the design” longer
    - You don’t have to get into nitty gritty details, just how your classes map into the roles provided by the pattern
- Shared vocabularies can empower your development team
    - Experienced team members can talk about design more quickly; junior programmers are motivated to get up to speed, so they can influence the design of the target system

Wrapping Up

- We've seen how patterns embody good OO principles
- We've discussed how they can help keep a team focused on design
- In the weeks ahead, we'll be learning about
    - Decorator
    - Factory
    - Singleton
    - Command
    - Adapter
    - Template Method
    - and more!

Coming Up Next

- Lecture 20: Decorator and Factory Patterns
    - Chapters 3 and 4 of the Design Patterns book