More Design Patterns

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 24 — 11/10/2011

© Kenneth M. Anderson, 2011

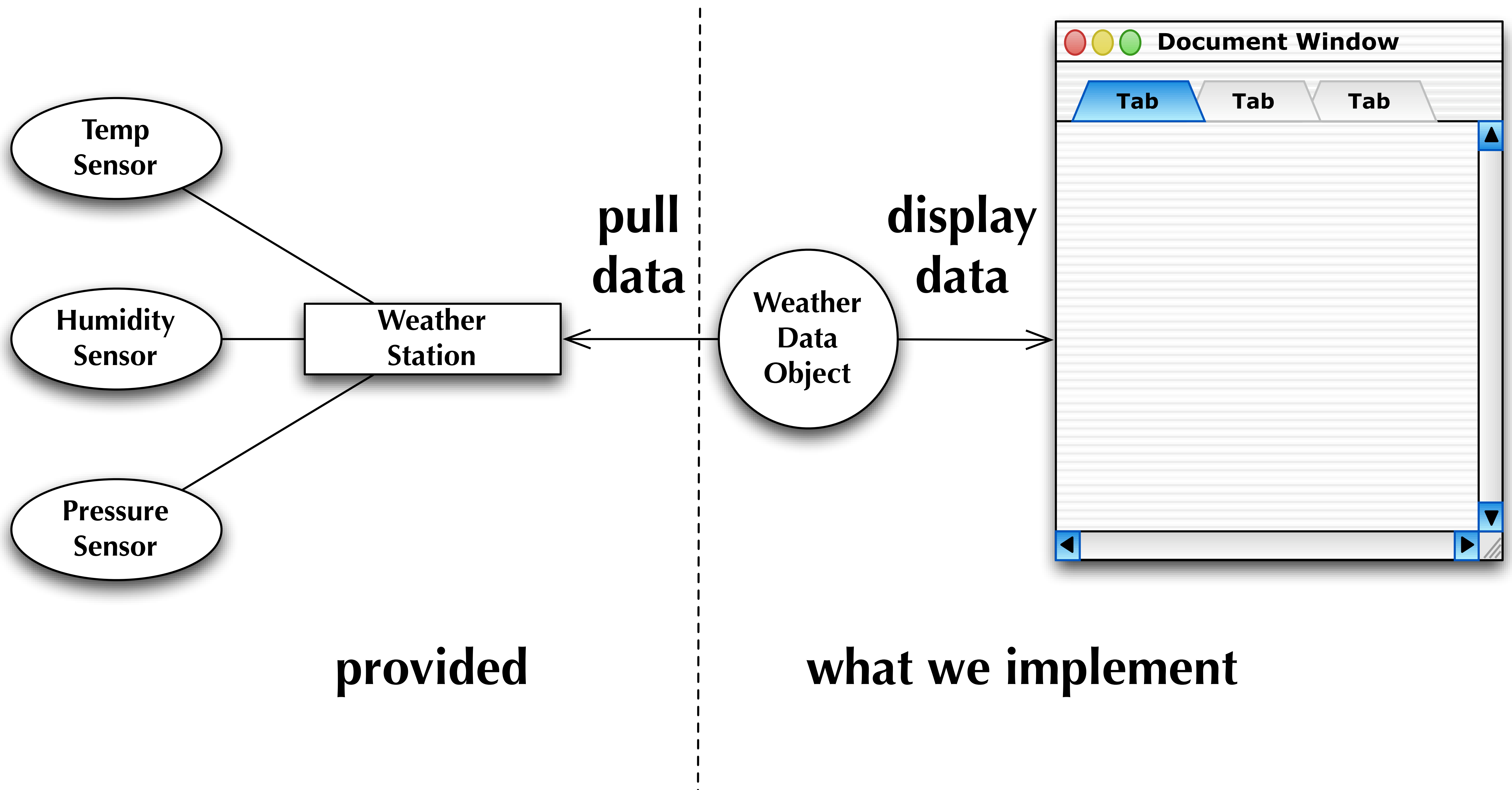
Goals of the Lecture

- Cover the material in Chapters 18 & 19 of our textbook
  - Observer
  - Template Method
- And, include two bonus patterns
  - State
  - Flyweight

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 frameworks

Weather Monitoring



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

WeatherData Skeleton

WeatherData
getTemperature()
getHumidity()
getPressure()
measurementsChanged()

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
```

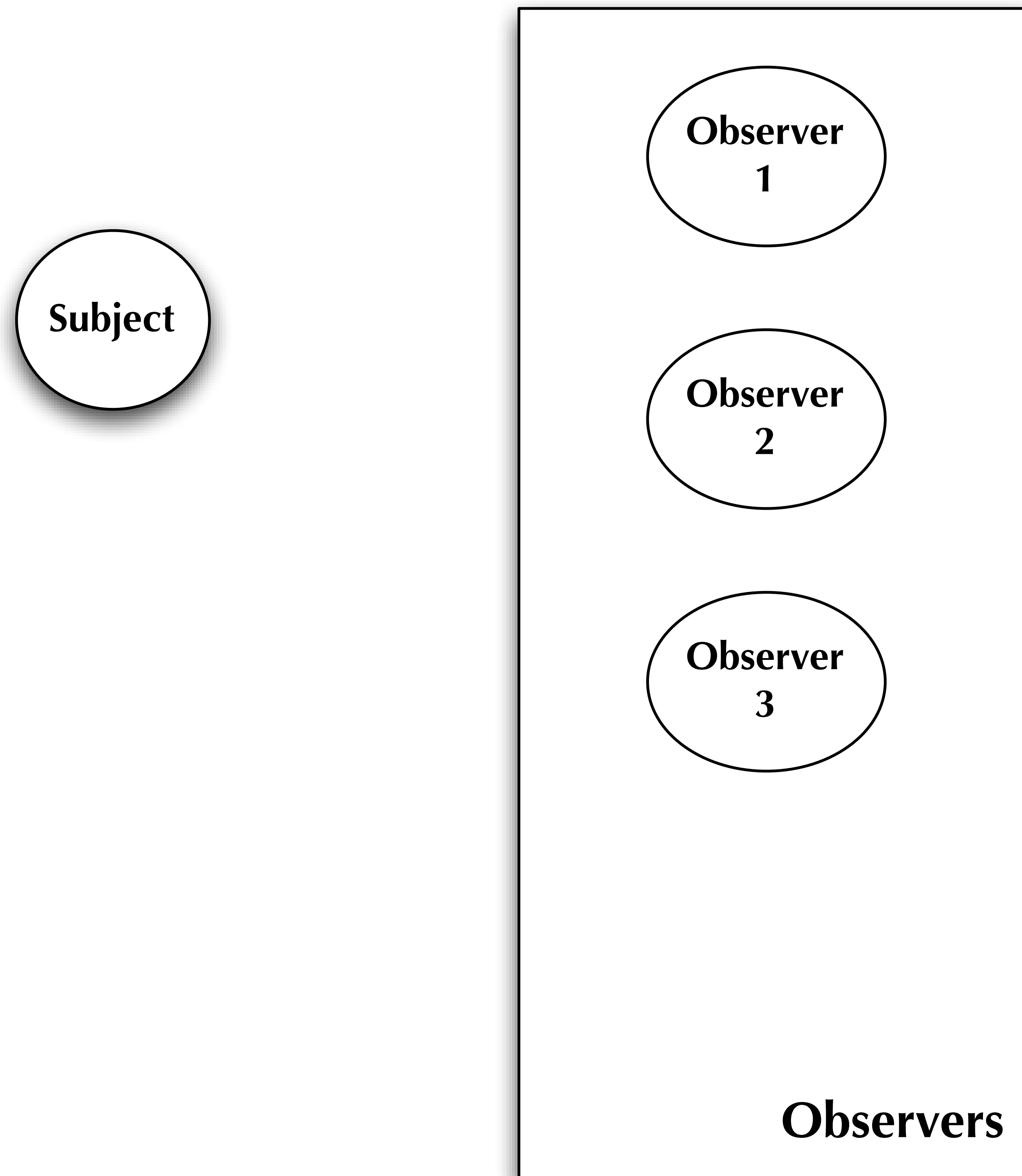
Problems?

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

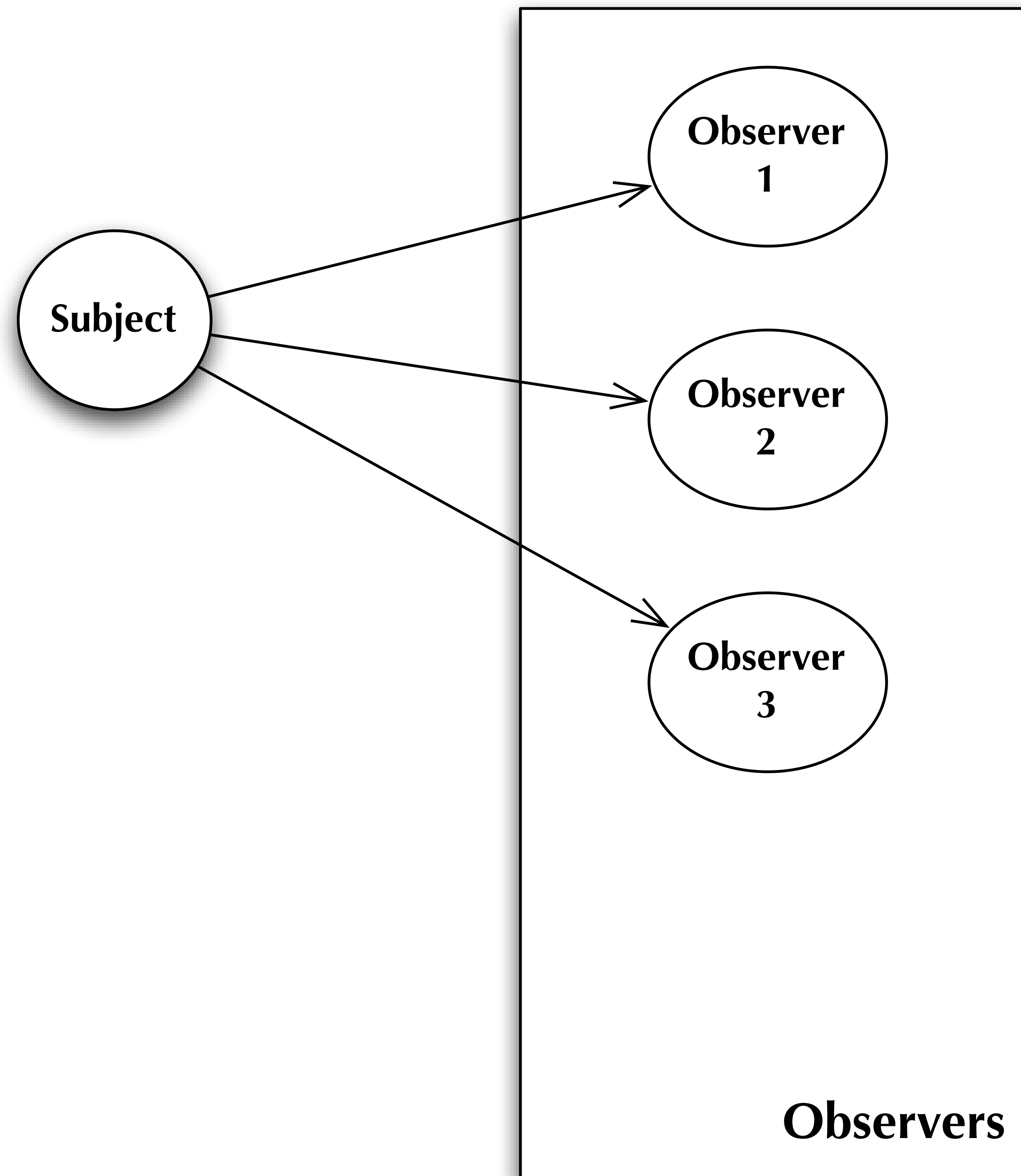
- This situation can benefit from use of the observer pattern
  - This pattern is similar to subscribing to a hard copy newspaper
    - A newspaper comes into existence and starts publishing editions
    - You become interested in the newspaper and subscribe to it
    - Any time an edition becomes available,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



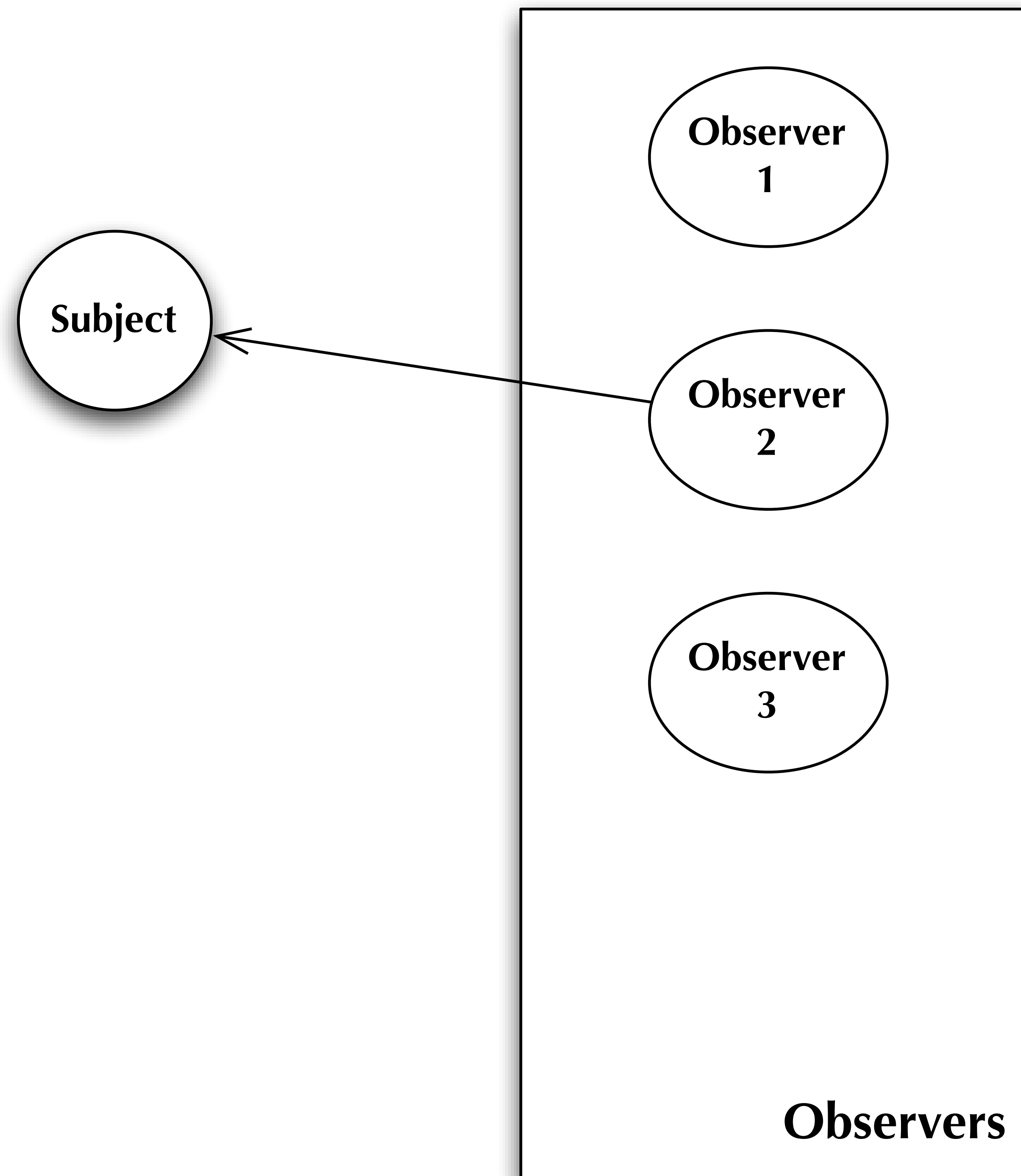
Subject maintains a list of observers

Observer in Action (II)



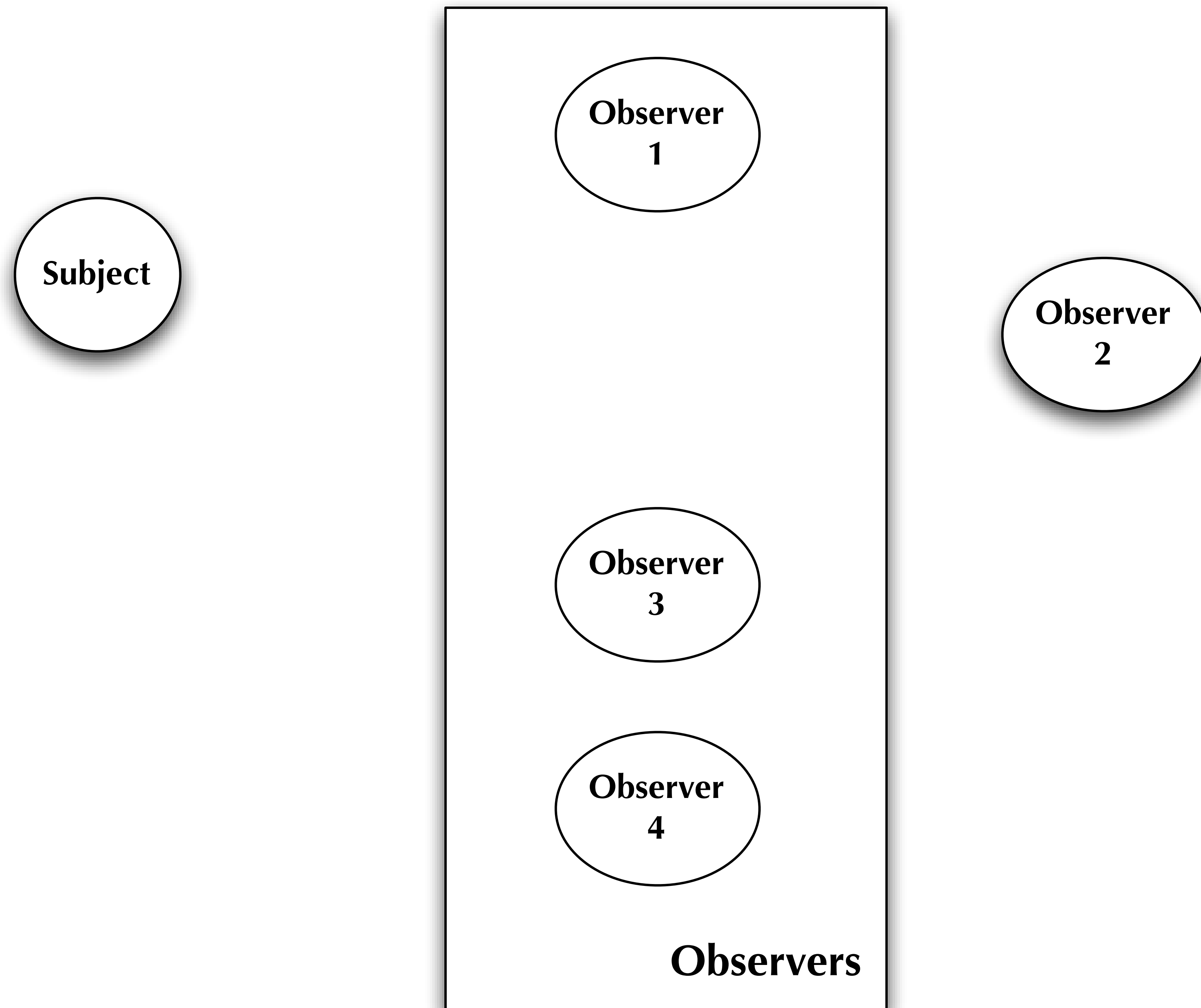
If the Subject changes, it notifies its observers

Observer in Action (III)



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

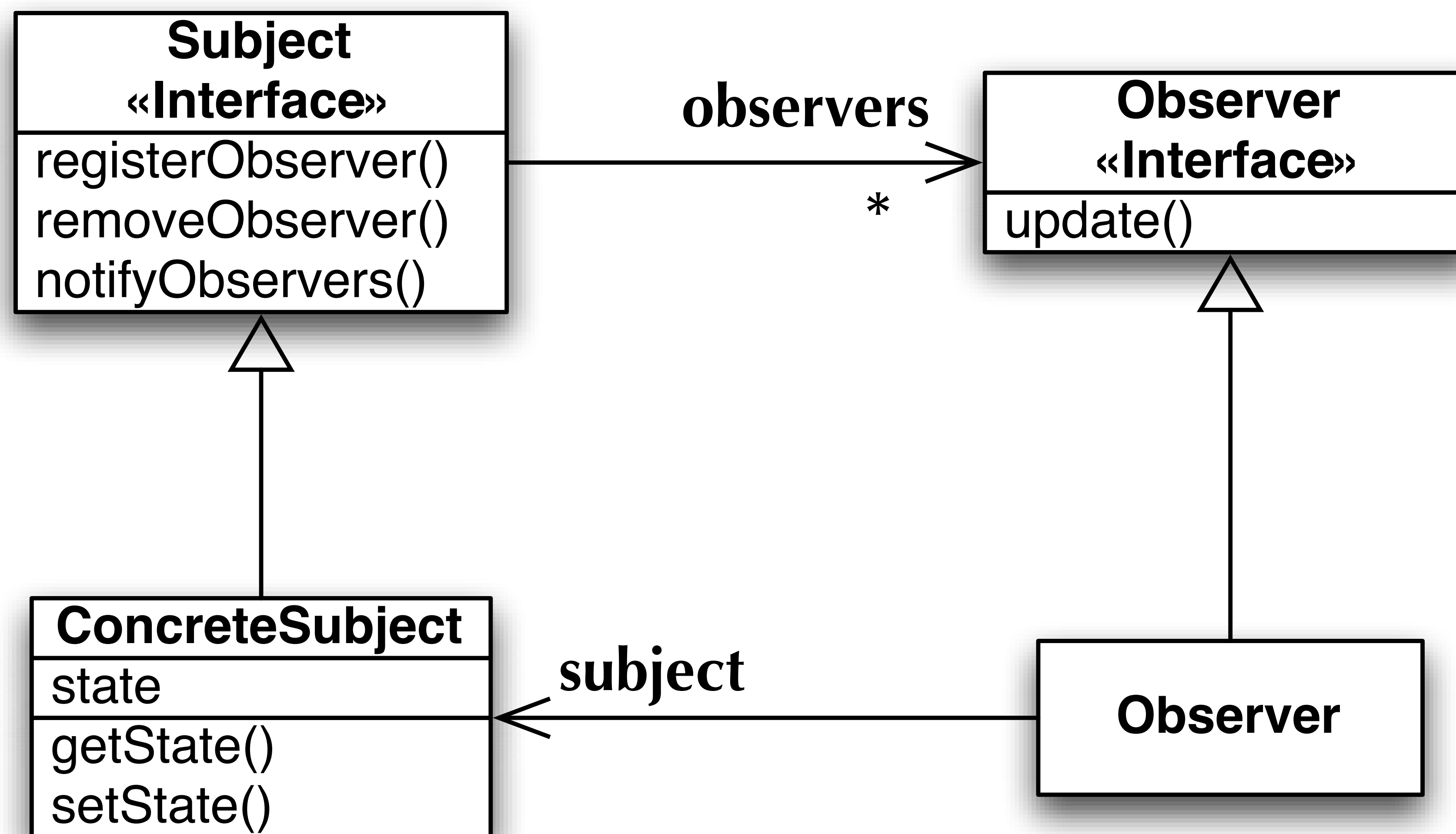
Observer In Action (IV)



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 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

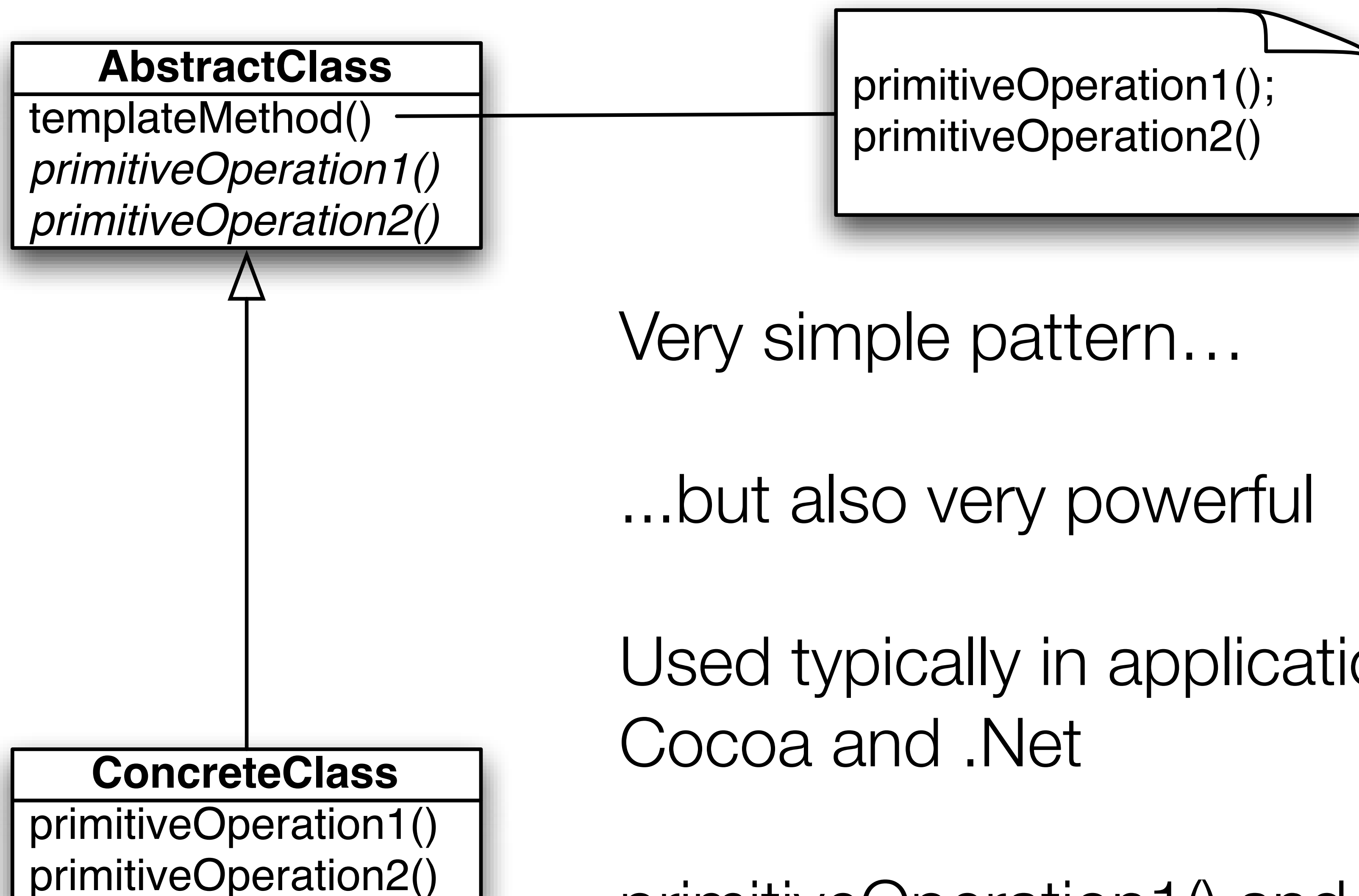
Demonstration

- 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
- Observer in Cocoa
  - Notifications (system defined as well as application defined)

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. Cocoa and .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

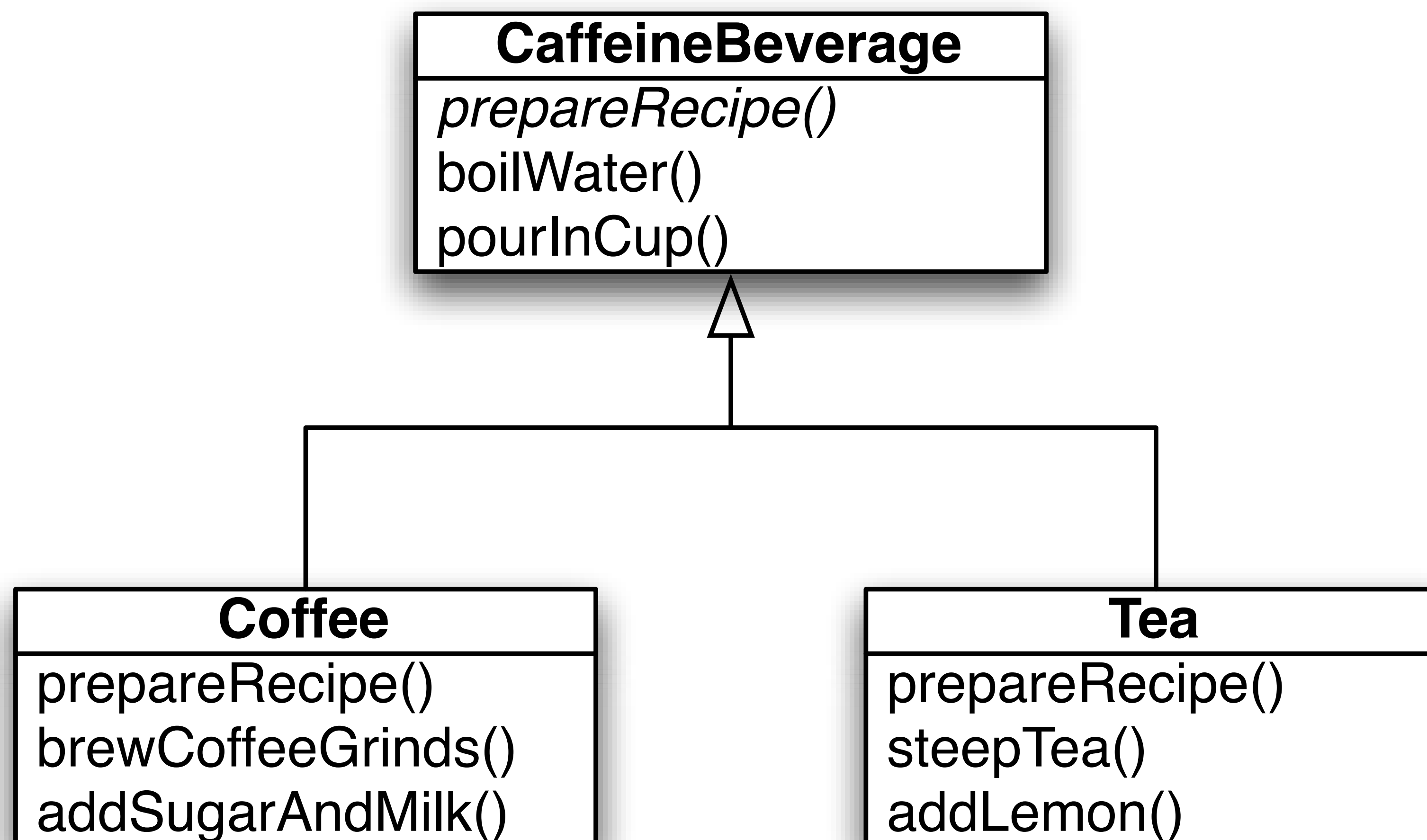
```java
public class Coffee {

    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

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

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

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

```

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 the keyword "final" in languages that support it

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

Adding a Hook to Coffee

Demonstration

```
1  import java.io.*;
2
3  public class CoffeeWithHook extends CaffeineBeverageWithHook {
4
5      public void brew() {
6          System.out.println("Dripping Coffee through filter");
7      }
8
9      public void addCondiments() {
10         System.out.println("Adding Sugar and Milk");
11     }
12
13     public boolean customerWantsCondiments() {
14
15         String answer = getUserInput();
16
17         if (answer.toLowerCase().startsWith("y")) {
18             return true;
19         } else {
20             return false;
21         }
22     }
23
24     private String getUserInput() {
25         String answer = null;
26
27         System.out.print("Would you like milk and sugar with your coffee (y/n)? ");
28
29         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
30         try {
31             answer = in.readLine();
32         } catch (IOException ioe) {
33             System.err.println("IO error trying to read your answer");
34         }
35         if (answer == null) {
36             return "no";
37         }
38         return answer;
39     }
40 }
```

New Design Principle: Hollywood Principle

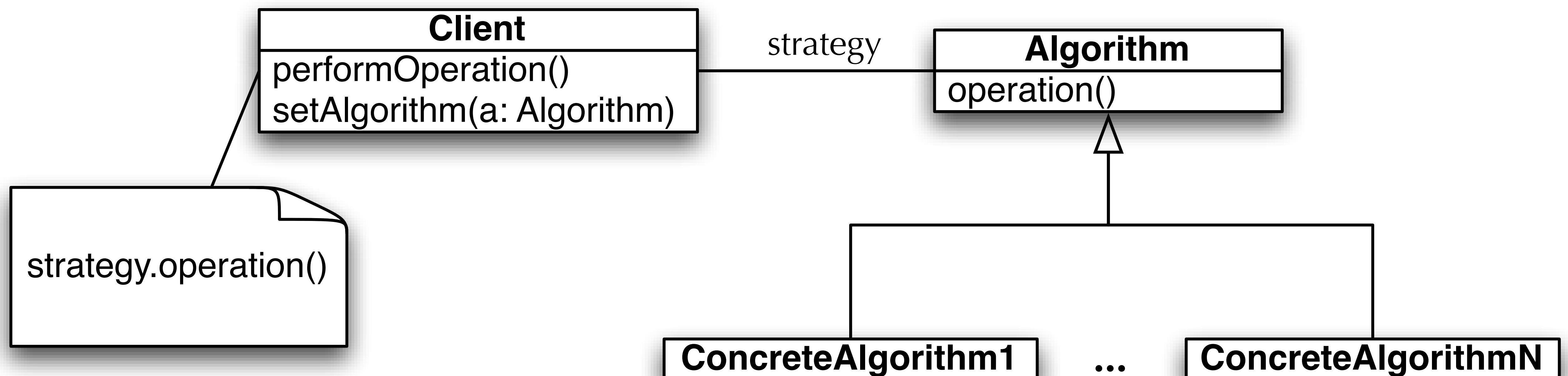
- Don’t call us, we’ll call you
- Or, in OO terms, high-level components call low-level components, not the other way around
  - In the context of the template method pattern, the template method lives in a high-level class and invokes methods that live in its subclasses
- This principle is similar to 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
- Examples in the Java API
  - Sorting using compareTo() method
  - Frames in Swing
  - Applets
- Demonstration

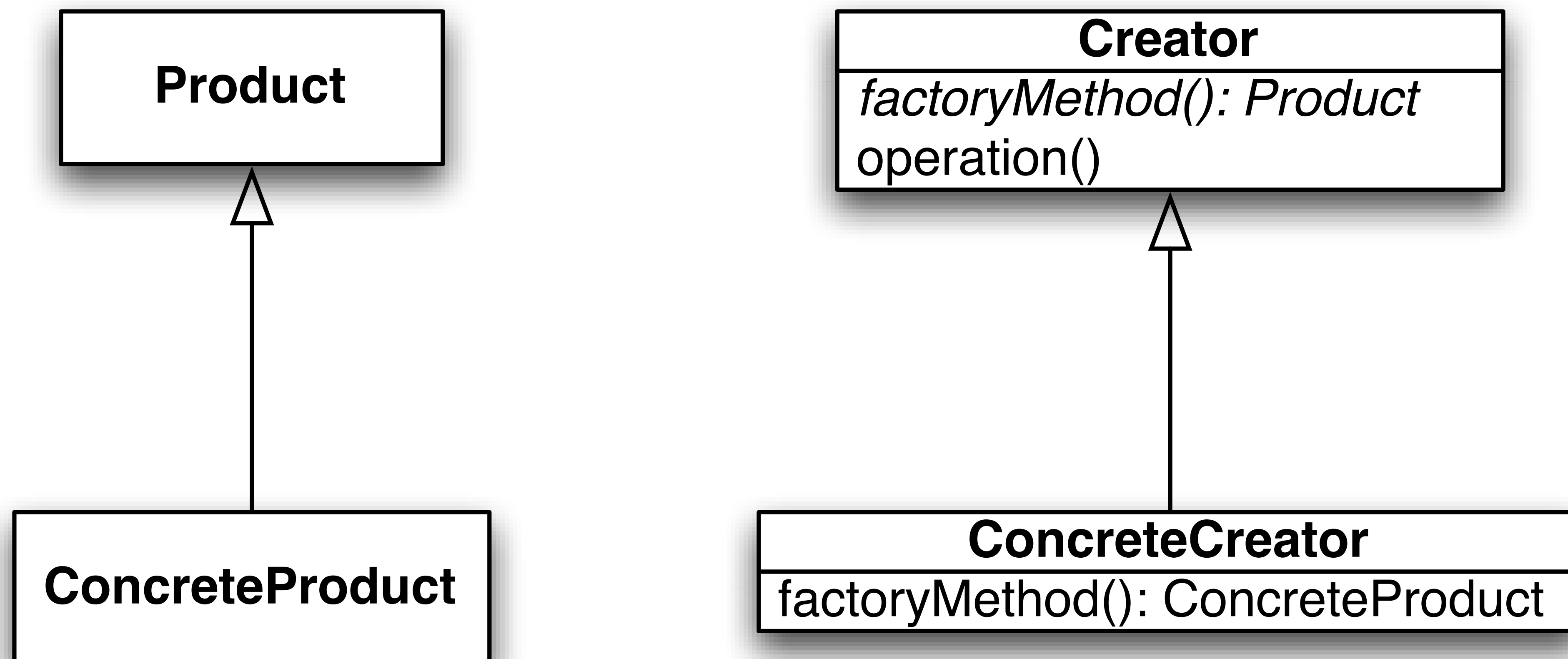
Template Method vs. Strategy (I)

- 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

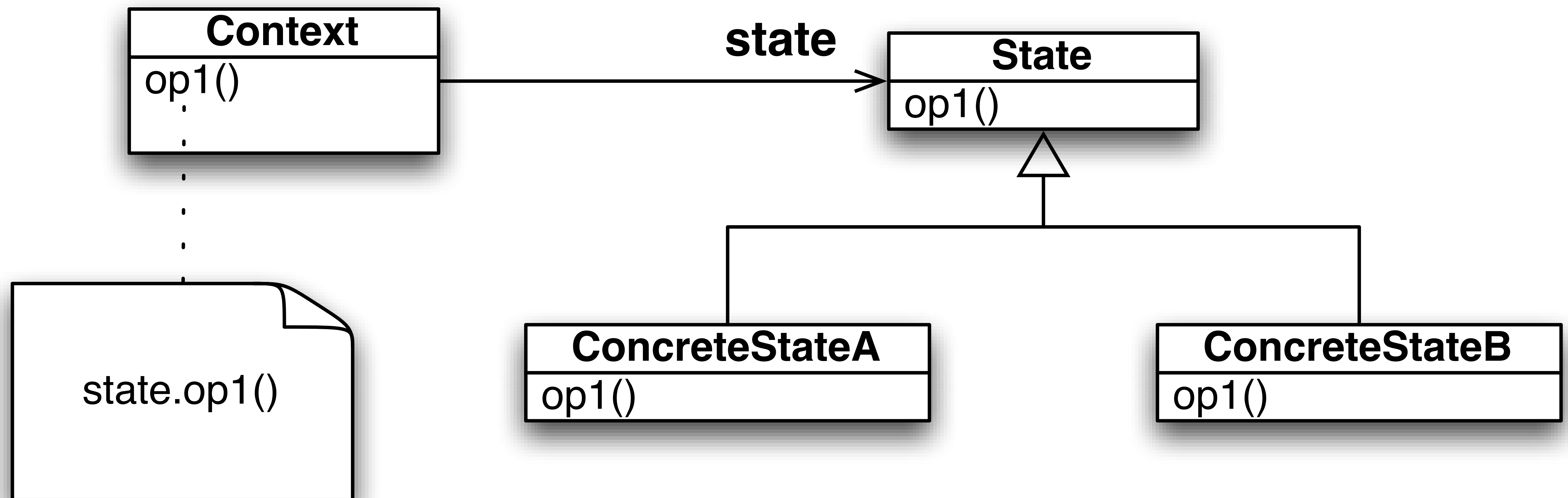
Template Method vs. Strategy (III)

- 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

State Pattern: Definition

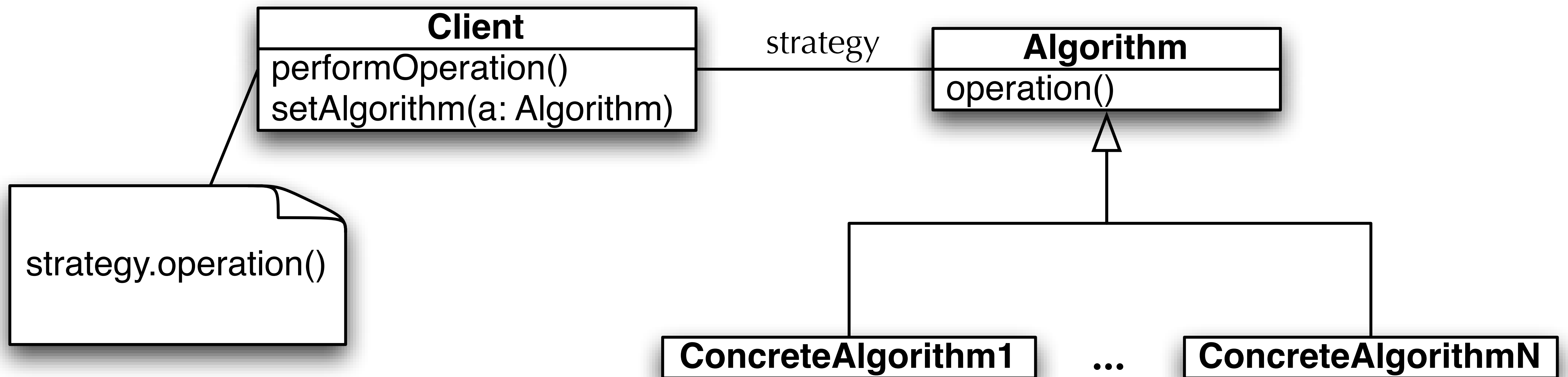
- The state pattern provides a clean way for an object to vary its behavior based on its current "state"
    - That is, the object's public interface doesn't change but each method's behavior may be different as the object's internal state changes
- Definition: The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
    - If we associate a class with behavior, then
        - since the state pattern allows an object to change its behavior
        - it will seem as if the object is an instance of a different class each time it changes state

State Pattern: Structure



Look Familiar?

Strategy Pattern: Structure

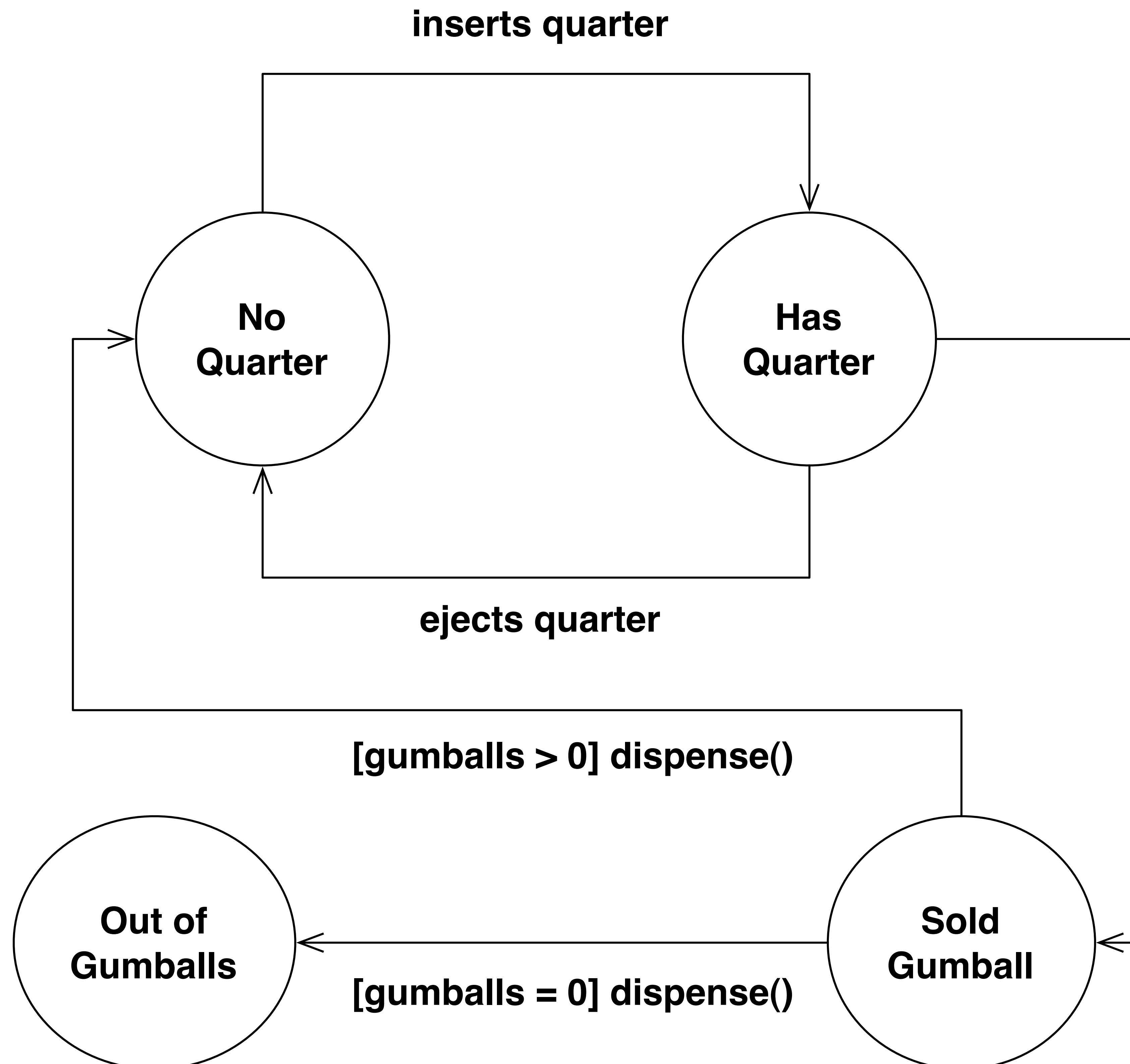


Strategy and State Patterns: Separated at Birth?!

Strategy and State are structurally equivalent; their intent however is different.

Strategy is meant to share behavior with classes without resorting to inheritance; it allows this behavior to be configured at run-time and to change if needed; State has a very different purpose, as we shall see.

Example: State Machines for Gumball Machines



Each circle represents a state that the gumball machine can be in.

Each label corresponds to an event (method call) that can occur on the object

Modeling State without State Pattern

- Create instance variable to track current state
  - Define constants: one for each state
    - For example
      - final static int SOLD_OUT = 0;
      - int state = SOLD_OUT;
- Create class to act as a state machine
  - One method per state transition
    - Inside each method, we code the behavior that transition would have given the current state; we do this using conditional statements
      - Demonstration

Seemed Like a Good Idea At The Time...

- This approach to implementing state machines is intuitive
  - and most people would stumble into it if asked to implement a state machine for the first time
- But the problems with this approach become clear as soon as change requests start rolling in
  - With each change, you discover that a lot of work must occur to update the code that implements the state machine
    - Indeed, in the Gumball example, you get a request that the behavior should change such that roughly 10% of the time, it dispenses two gumballs rather than one
      - Requires a change such that the “turns crank” action from the state “Has Quarter” will take you either to “Gumball Sold” or to “Winner”
      - The problem? You need to add one new state and update the code for each action

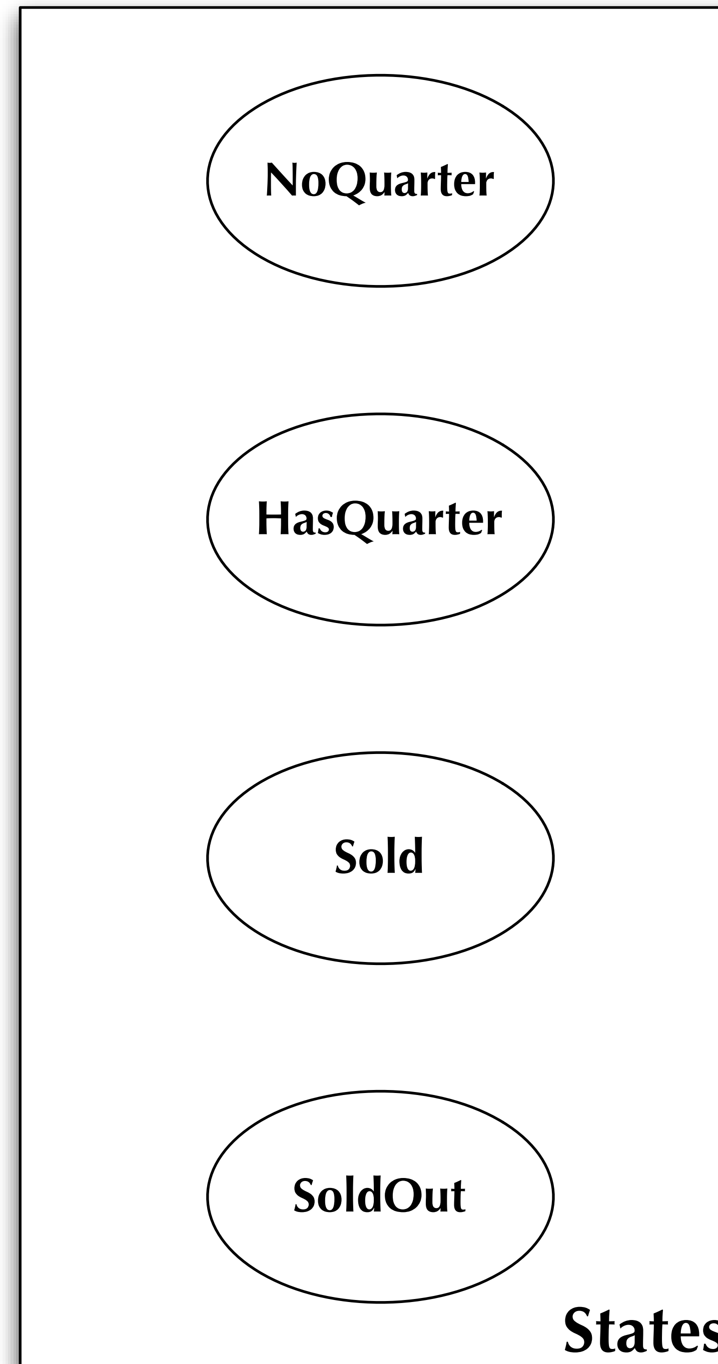
Design Problems with First Attempt

- Does not support Open Closed Principle
  - A change to the state machine requires a change to the original class
    - You can't place new state machine behavior in an extension of the original class
- The design is not very object-oriented: indeed no objects at all except for the one that represents the state machine, in our case GumballMachine.
- State transitions are not explicit; they are hidden amongst a ton of conditional code
- We have not "encapsulated what varies"

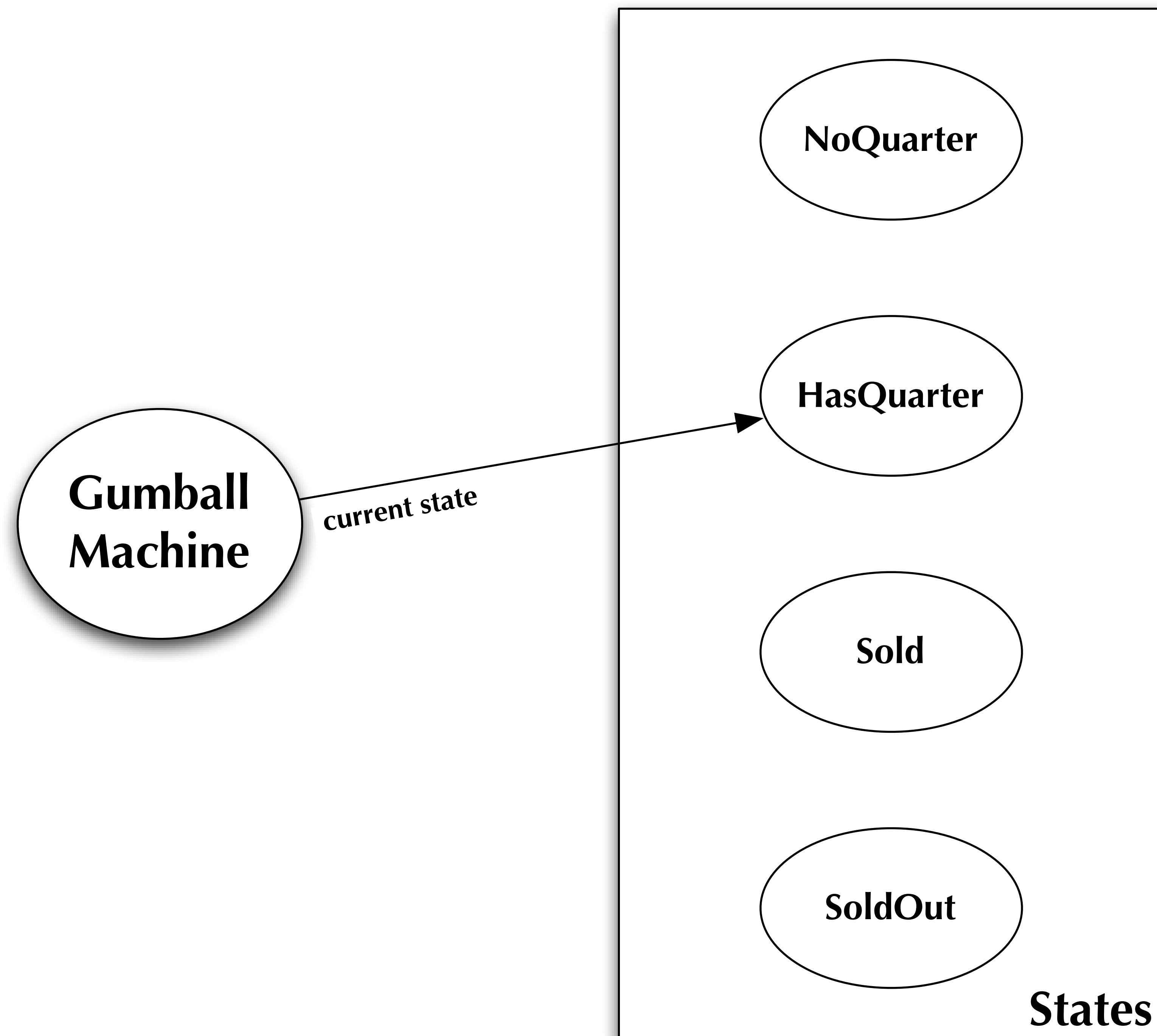
2nd Attempt: Use State Pattern

- Create a State interface that has one method per state transition
- Create one class per state in state machine. Each such class implements the State interface and provides the correct behavior for each action in that state
- Change GumballMachine class to point at an instance of one of the State implementations and delegate all calls to that class. An action may change the current state of the GumballMachine by making it point at a different State implementation
- Demonstration

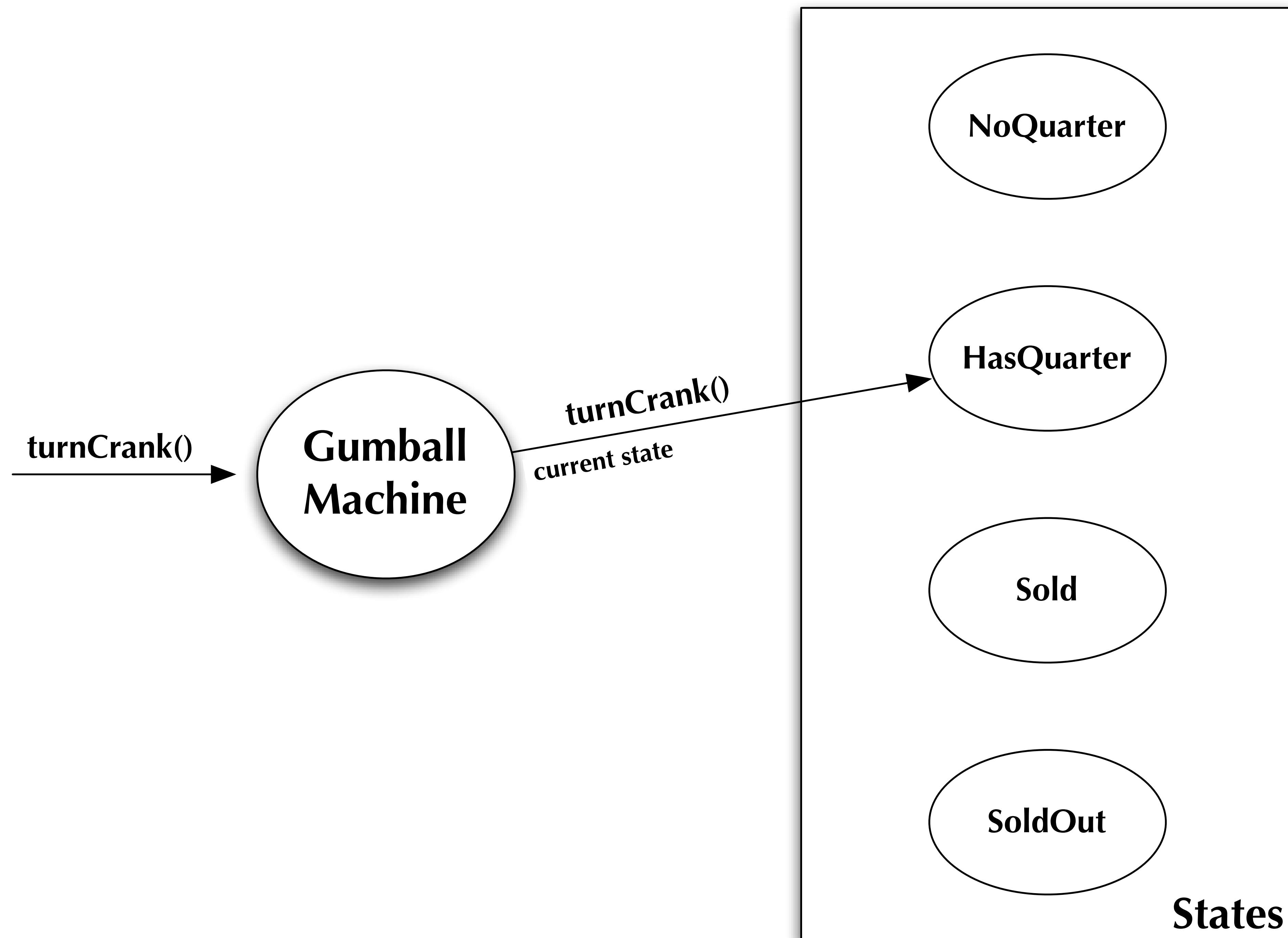
State Pattern in Action (I)



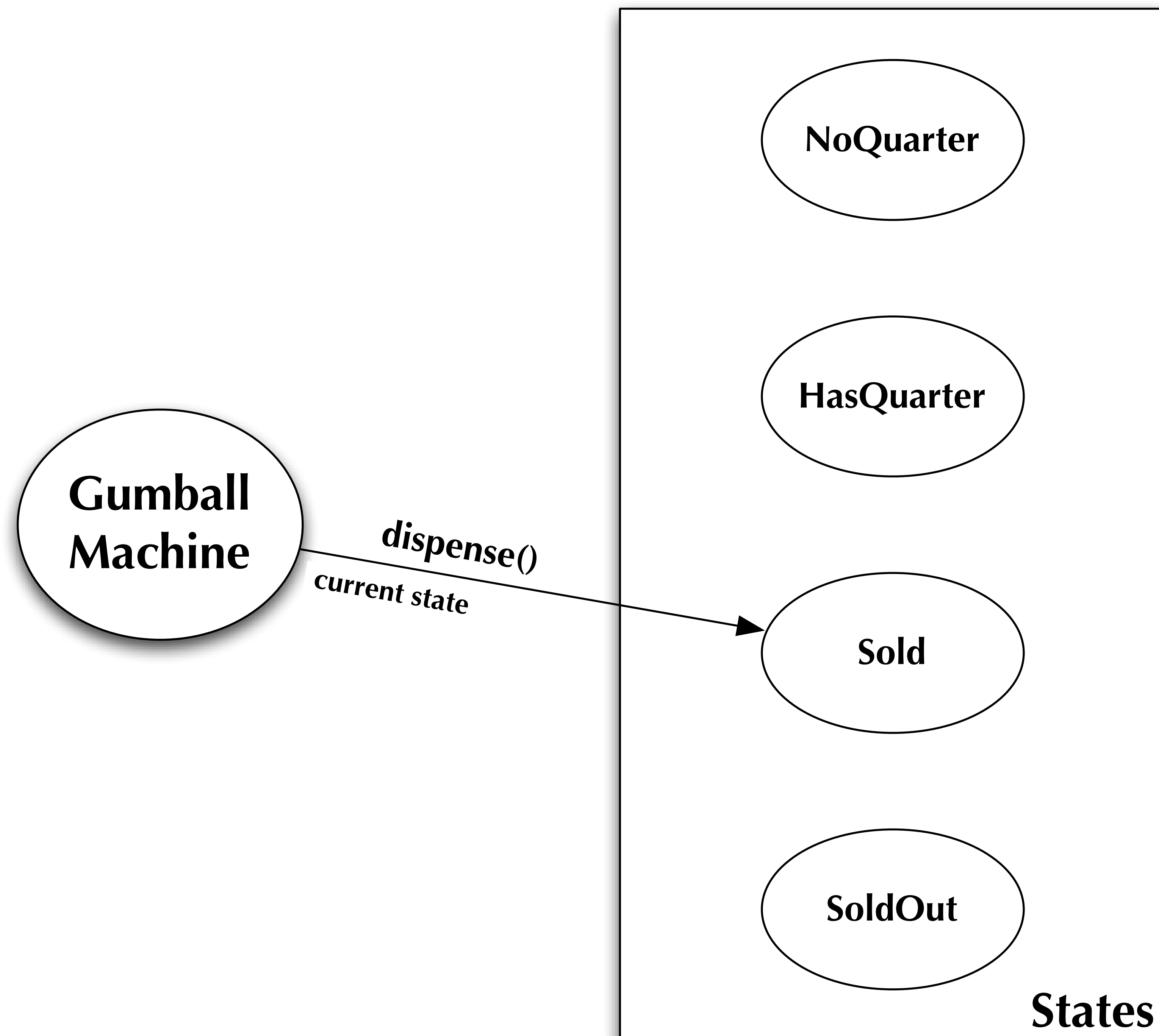
State Pattern in Action (II)



State Pattern in Action (III)



State Pattern in Action (IV)



Third Attempt: Implement 1 in 10 Game

- Demonstrates flexibility of State Pattern
  - Add a new State implementation: WinnerState
    - Exactly like SoldState except that its dispense() method will dispense two gumballs from the machine, checking to make sure that the gumball machine has at least two gumballs
      - You can have WinnerState be a subclass of SoldState and just override the dispense() method
  - Update HasQuarterState to generate random number between 1 and 10
    - if number == 1, then switch to an instance of WinnerState else an instance of SoldState
- Demonstration

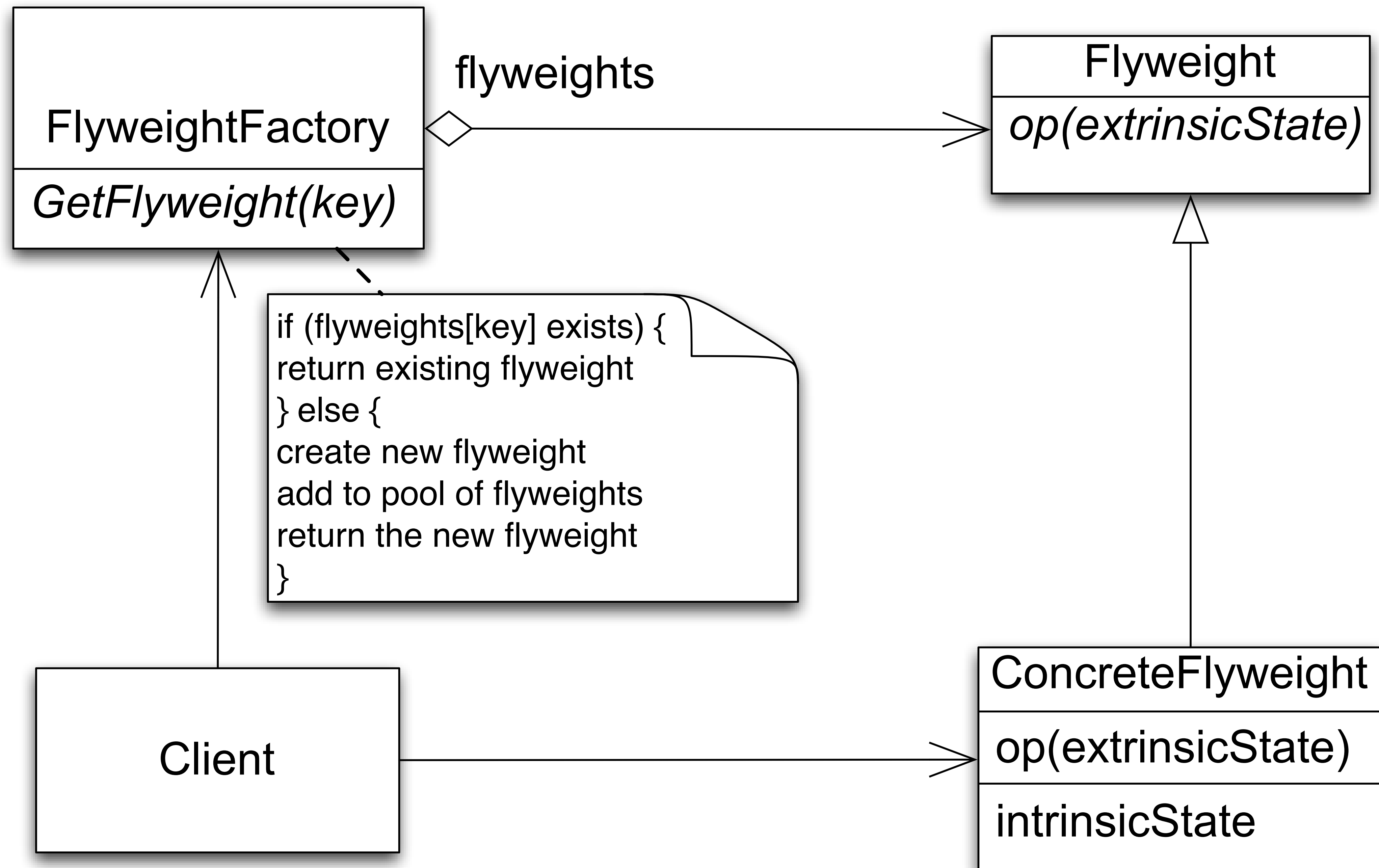
Bonus Pattern: Flyweight

- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently
- Motivation
  - Imagine a text editor that creates one object per character in a document
  - For large documents, that is a lot of objects!
    - but for simple text documents, there are only 26 letters, 10 digits, and a handful of punctuation marks being referenced by all of the individual character objects

Flyweight, continued

- Applicability
  - Use flyweight when all of the following are true
    - An application uses a large number of objects
    - Storage costs are high because of the sheer quantity of objects
    - Most object state can be made extrinsic
    - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed
    - The application does not depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects

Flyweight's Structure and Roles



Flyweight, continued

- Participants
  - Flyweight
    - declares an interface through which flyweights can receive and act on extrinsic state
  - ConcreteFlyweight
    - implements Flyweight interface and adds storage for intrinsic state
  - UnsharedConcreteFlyweight
    - not all flyweights need to be shared; unshared flyweights typically have children which are flyweights
  - FlyweightFactory
    - creates and manages flyweight objects
  - Client
    - maintains extrinsic state and stores references to flyweights

Flyweight, continued

- Collaborations
  - Data that a flyweight needs to process must be classified as intrinsic or extrinsic
    - Intrinsic is stored with flyweight; Extrinsic is stored with client
  - Clients should not instantiate ConcreteFlyweights directly
- Consequences
  - Storage savings is a tradeoff between total reduction in number of objects verses the amount of intrinsic state per flyweight and whether or not extrinsic state is computed or stored
    - greatest savings occur when extrinsic state is computed

Flyweight, continued

- Demonstration
- Simple implementation of flyweight pattern
    - Focus is on factory and flyweight rather than on client
    - Demonstrates how to do simple sharing of characters

Wrapping Up

- Observer
  - Flexibly monitor an object's state changes
- Template Method
  - Specify overall structure of an algorithm; allow some variation via overridden methods
- 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

© Kenneth M. Anderson, 2011

Coming Up Next

- Lecture 25: More Patterns
- Lecture 26: Textbook Wrap Up
- Homework 6 Due Next Friday

© Kenneth M. Anderson, 2011