State, Flyweight & Proxy Patterns

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 28 — 11/11/2008

Lecture Goals

- Cover Material from Chapters 10 and 11 of the Design Patterns Textbook

    - State Pattern

    - Bonus Pattern

        - Flyweight (not from textbook)

    - Proxy Pattern

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
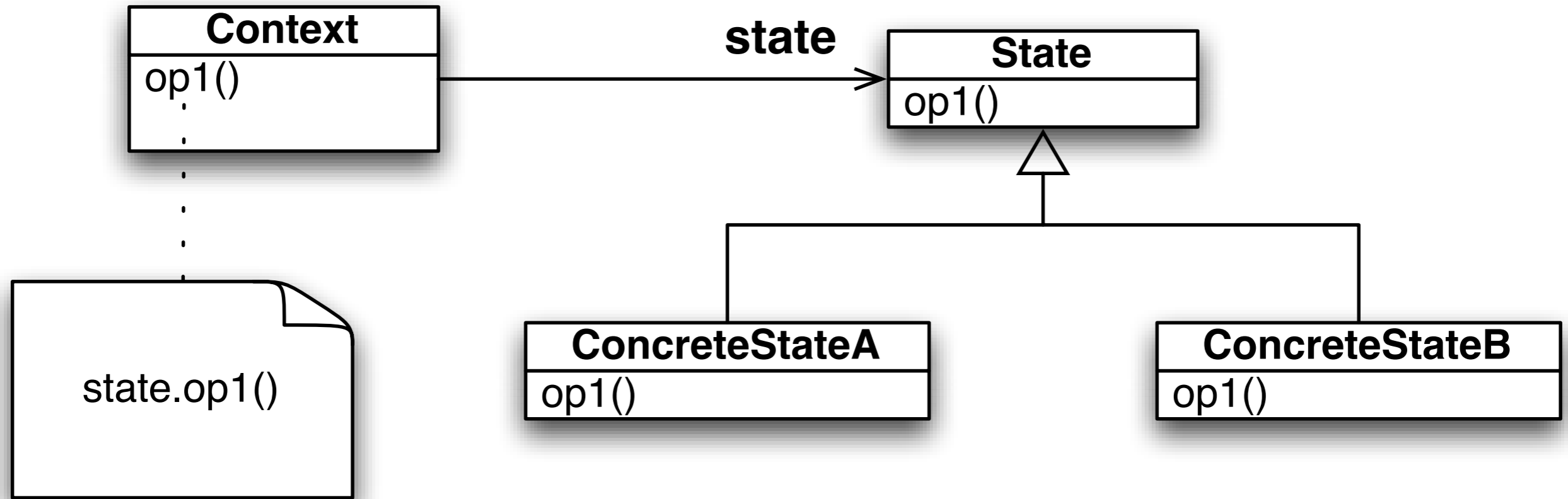
Look Familiar?

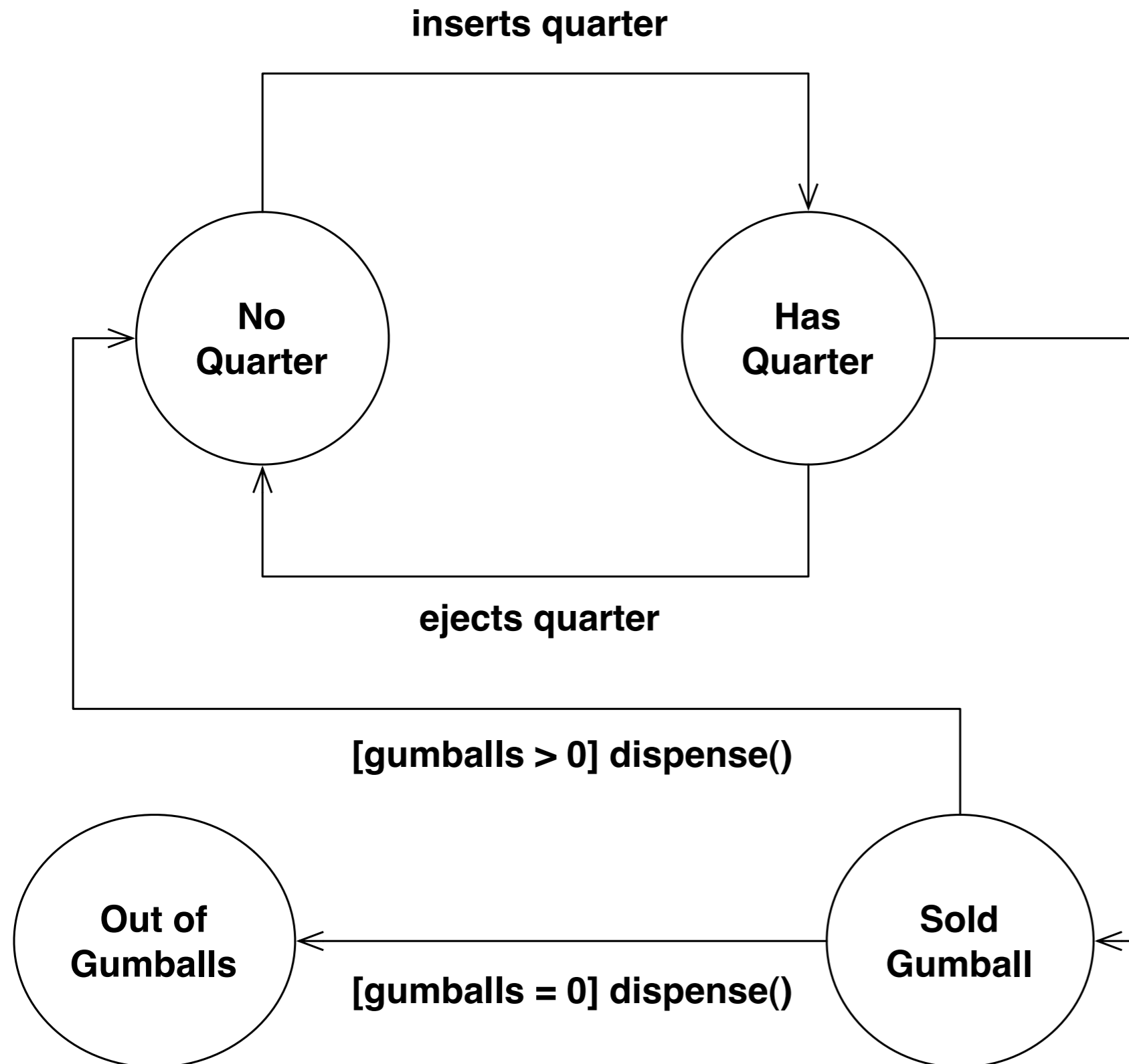# Strategy Pattern: Structure (from Lecture 17)
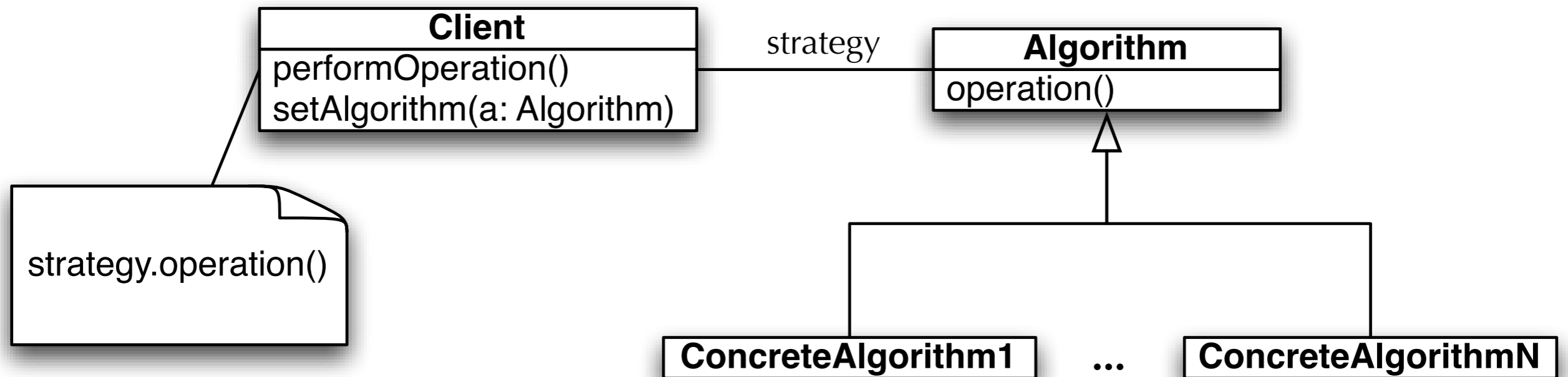
Strategy and State Patterns: Separated at Birth?!

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.

5

Example: State Machines for Gumball Machines

inserts quarter

No Quarter

Has Quarter

ejects quarter

[gumballs > 0] dispense()

turns crank

Out of Gumballs

Sold Gumball

[gumballs = 0] dispense()

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

6

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

7

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
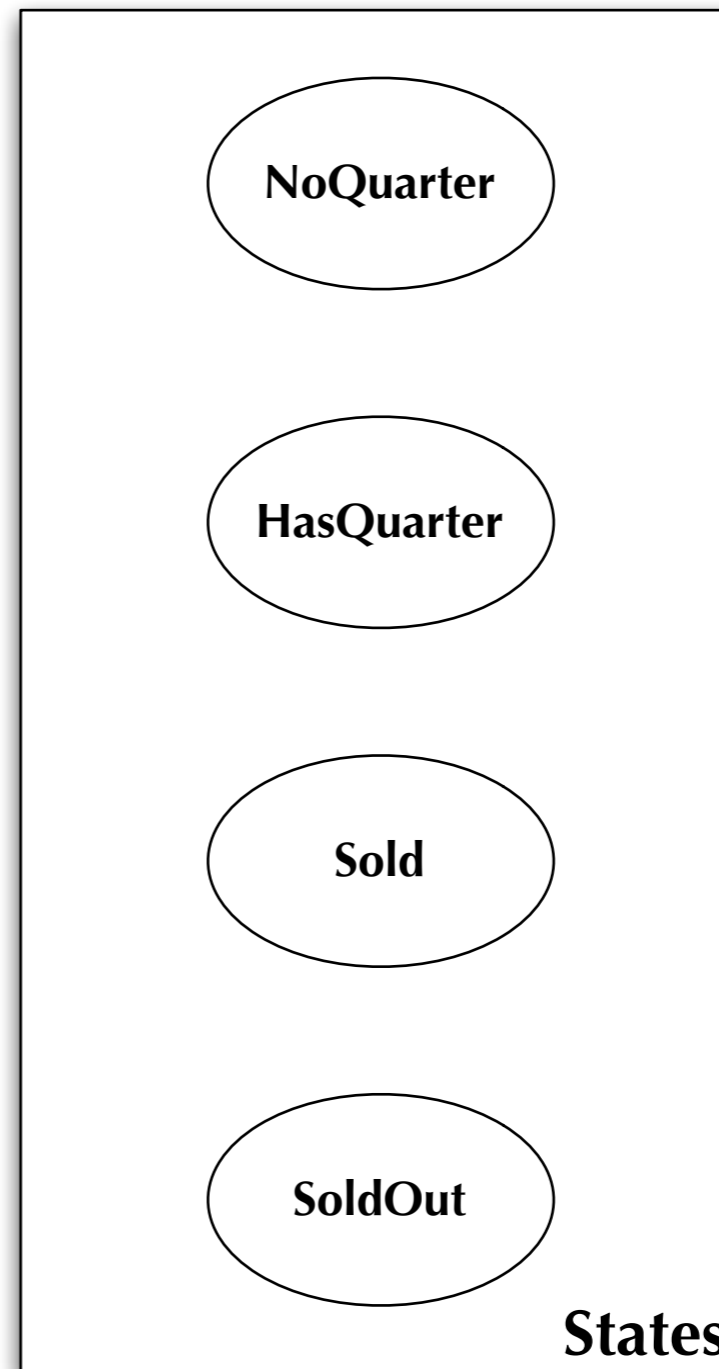
8

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"

- "This code would make a FORTRAN programmer proud" — FORTRAN code can often be very convoluted, aka spaghetti code, no structure, just a mess!
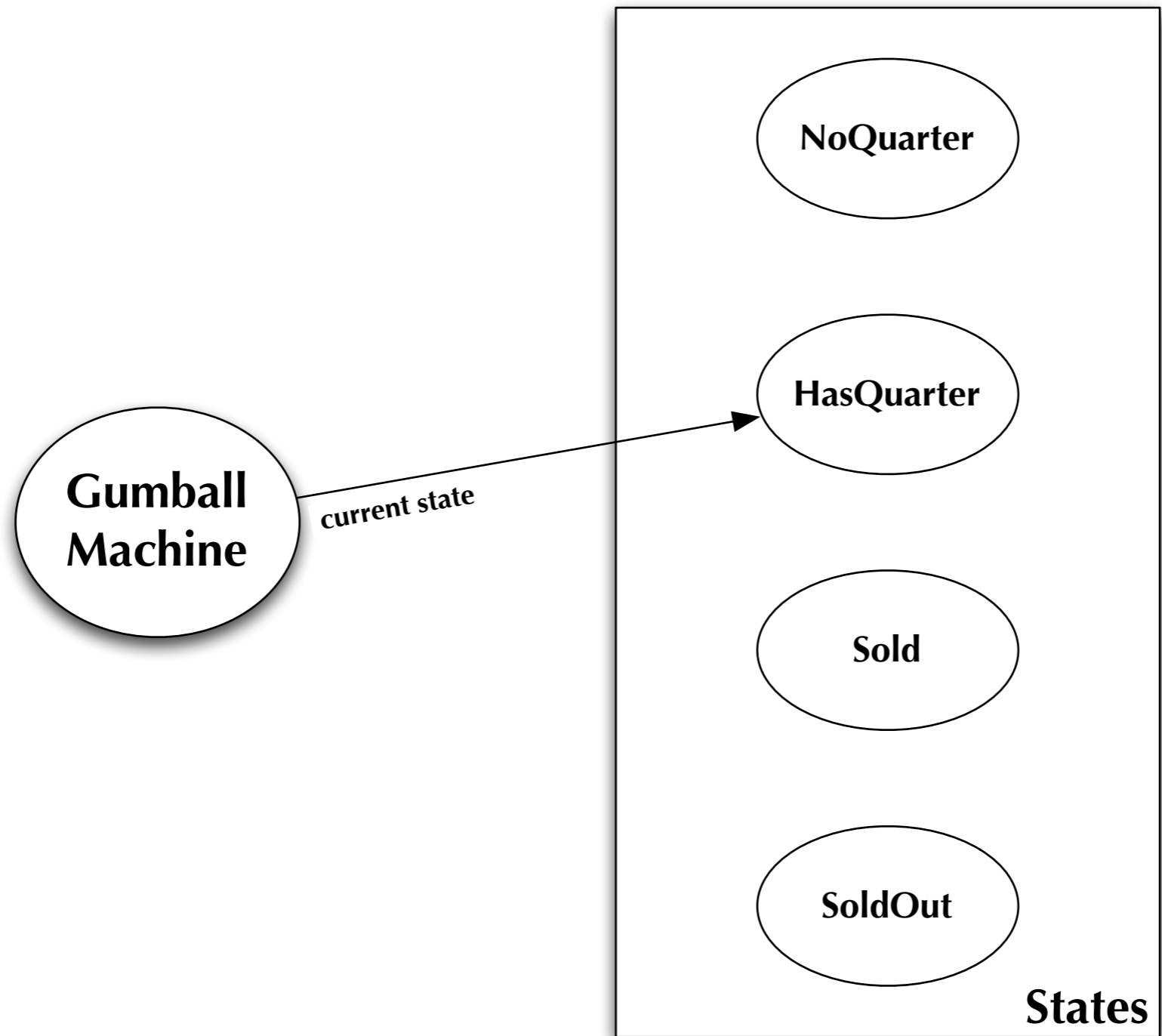
2nd Attempt: Use State Pattern

- Create a State interface that has one method per state transition (called action in the textbook)

- 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)

Gumball Machine

NoQuarter

HasQuarter

Sold

SoldOut

States

State Pattern in Action (II)

State Pattern in Action (III)

turnCrank() → **Gumball Machine**

turnCrank()
current state → **HasQuarter**

States
- NoQuarter
- HasQuarter
- Sold
- SoldOut
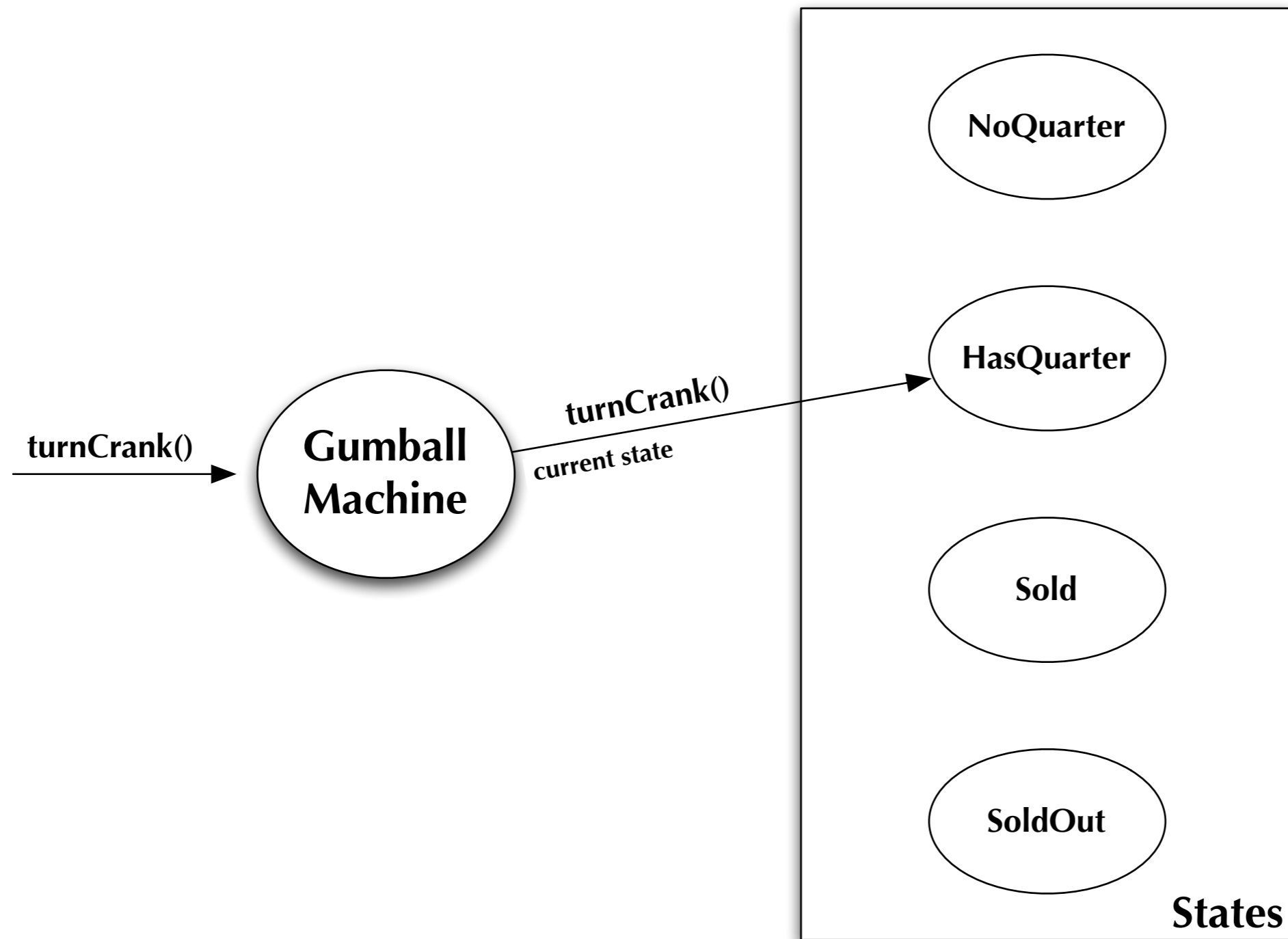
State Pattern in Action (IV)
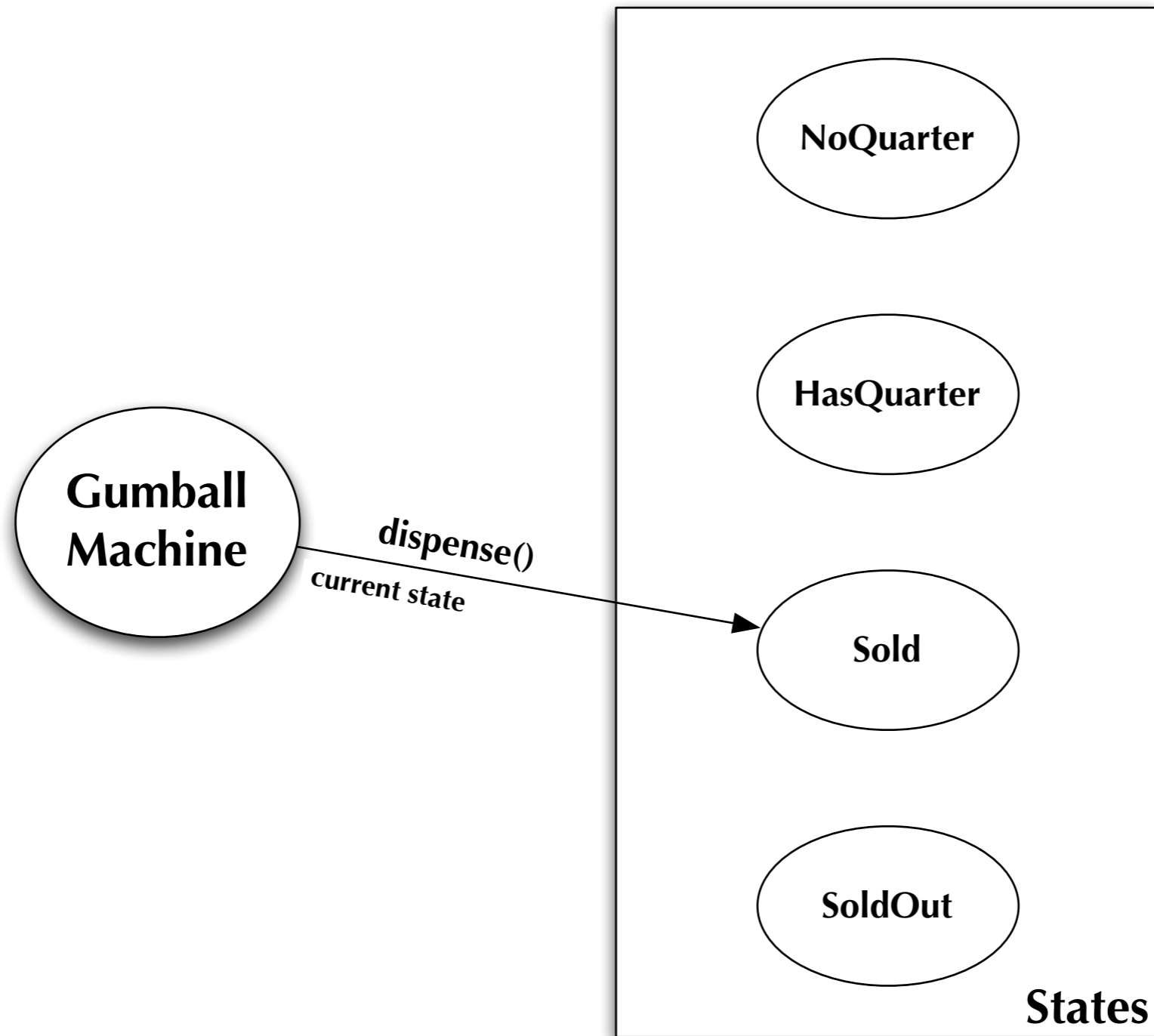
Third Attempt: Implement 1 in 10 Game

- 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, 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's Structure and Roles
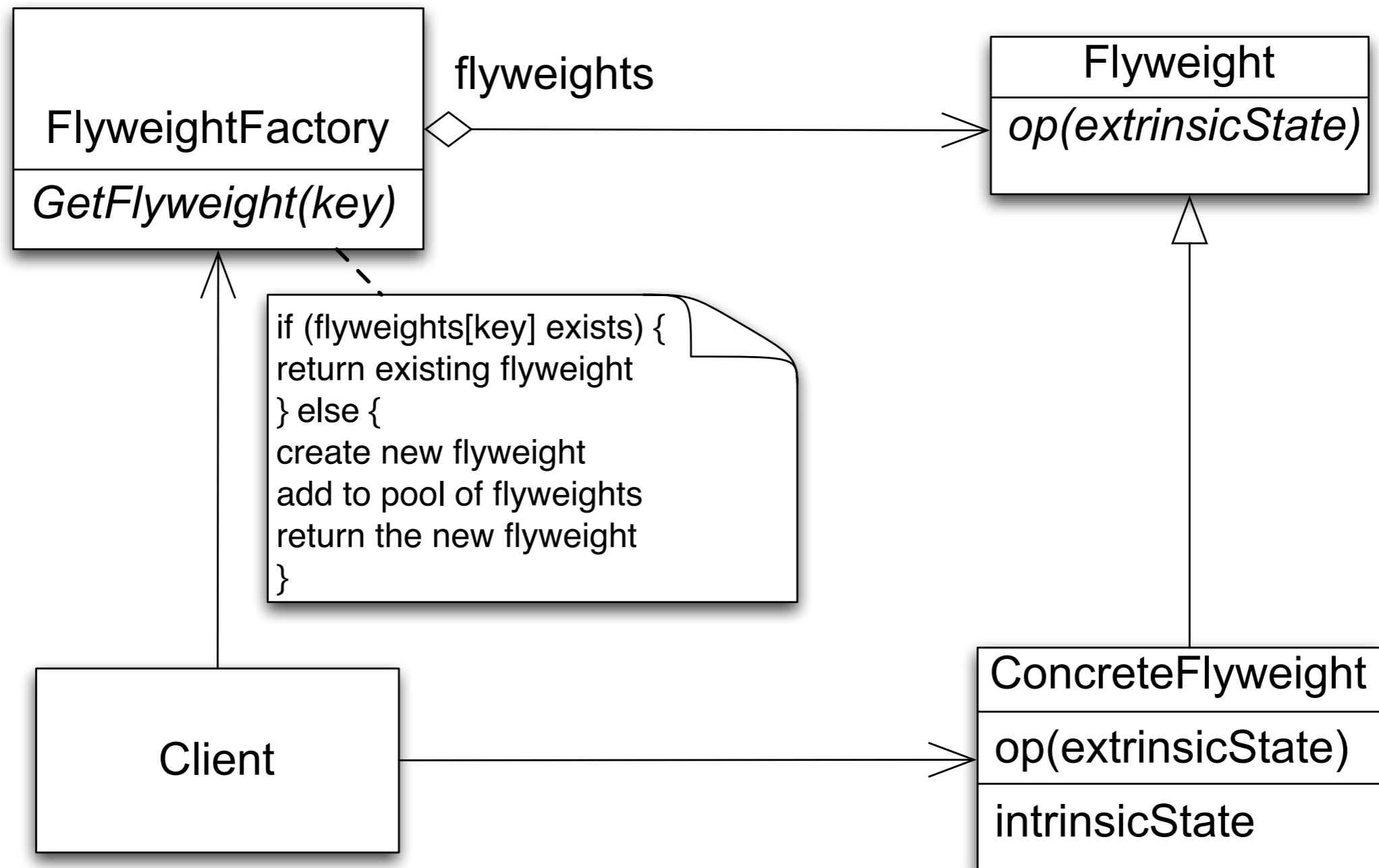
Flyweight, continued

- 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

Proxy Pattern: Definition

- The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.

    - Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create, or in need of securing

- We'll see several examples of the Proxy pattern in use in this lecture, including the Remote Proxy, Virtual Proxy, and Protection Proxy

Proxy Pattern: Structure

```
                    ┌─────────────────────┐
                    │      Subject        │
                    │    «interface»      │
                    ├─────────────────────┤
                    │   request()         │
                    └─────────────────────┘
                              △
                    ┌─────────┴──────────┐
                    │                    │
    ┌────────────────────┐      subject   ┌────────────────────┐
    │   RealSubject      │◁─────────────  │       Proxy        │
    ├────────────────────┤               ├────────────────────┤
    │   request()        │               │   request()        │
    └────────────────────┘               └────────────────────┘
```
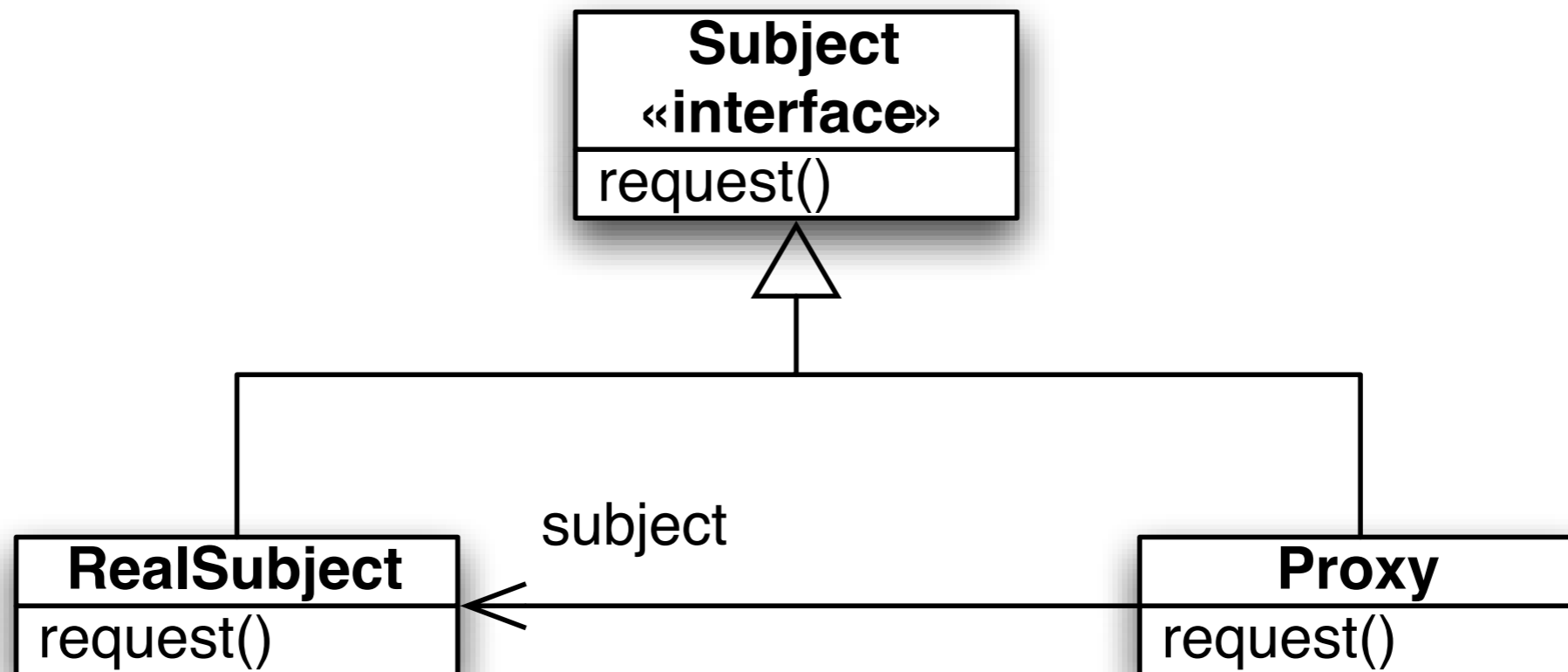
A Proxy and "Real" Subject class both implement a Subject interface.
Clients interact with the Proxy, which controls access to the methods of
the RealSubject class. When appropriate, the Proxy will forward requests
to the RealSubject via delegation. The Proxy may also respond to client
requests itself, or it may reject the request (perhaps by throwing an
exception).

23

Gumball Revisited

- To illustrate the Remote Proxy variation of the Proxy Pattern (in which the Proxy and RealSubject objects are on different machines), we return to the Gumball Machine example of the previous lecture

    - Our client would like a way to monitor gumball machines remotely

        - to enable regular status reports and to allow them to do a better job of keeping the gumball machines full of gumballs

Step 1: Update Gumball Machine

```
 1   public class GumballMachine {
 2
 3       State soldOutState;
 4       State noQuarterState;
 5       State hasQuarterState;
 6       State soldState;
 7       State winnerState;
 8
 9       State state = soldOutState;
10       int count = 0;
11       String location;
12
13       public GumballMachine(String location, int count) {
14           soldOutState = new SoldOutState(this);
15           noQuarterState = new NoQuarterState(this);
16           hasQuarterState = new HasQuarterState(this);
17           soldState = new SoldState(this);
18           winnerState = new WinnerState(this);
19
20           this.count = count;
21           if (count > 0) {
22               state = noQuarterState;
23           }
24           this.location = location;
25       }
```

First, we update the Gumball Machine class to store its location

We also add a getter method for this attribute (not shown)

25

Step 2: Create a Gumball Monitor Class

```java
 1  public class GumballMonitor {
 2      GumballMachine machine;
 3
 4      public GumballMonitor(GumballMachine machine) {
 5          this.machine = machine;
 6      }
 7
 8      public void report() {
 9          System.out.println("Gumball Machine: " + machine.getLocation());
10          System.out.println("Current inventory: " + machine.getCount() + " gumballs");
11          System.out.println("Current state: " + machine.getState());
12      }
13  }
14
```

Simple! The monitor takes an instance of the Gumball machine class and can generate a status report: location, number of gumballs, and the machine's current state.

But something is wrong with this design… what?

26

Going Remote

- The Gumball Monitor is coded to accept a pointer to a local Gumball Machine object

- But we need to monitor Gumball Machines that are not physically present

  - Or in computer speak: We need access to a "remote" Gumball Machine object, one that "lives" in a different JavaVM, or address space.

- To do this, we'll use a technology built into Java, called RMI, short for Remote Method Invocation

Machine Boundary

Gumball Monitor → Proxy → Gumball Machine
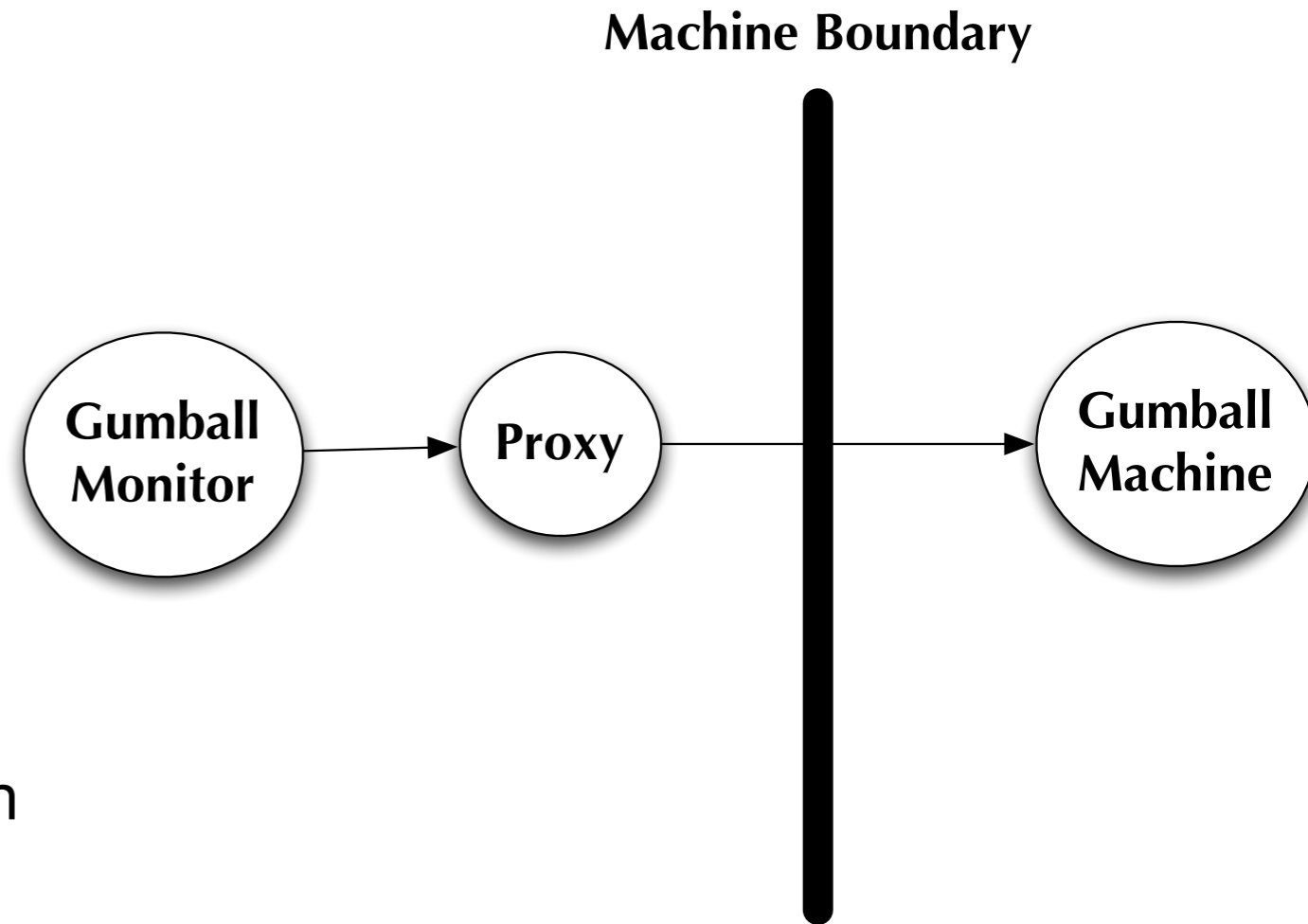
The Gumball Monitor talks to the Proxy object, thinking that its a Gumball Machine object. The Proxy communicates with the "real" Gumball Machine, and returns any results back to the monitor.

27

Approach

- Quick Introduction to RMI

- Change Gumball Machine so that it becomes a "remote service"

- Create a Proxy object that can talk to this "remote service" while looking like a local Gumball Machine to the Gumball Monitor class

Remote Method Invocation (I)

Machine Boundary

Client → **Client Helper** → **Service Helper** → **Service**
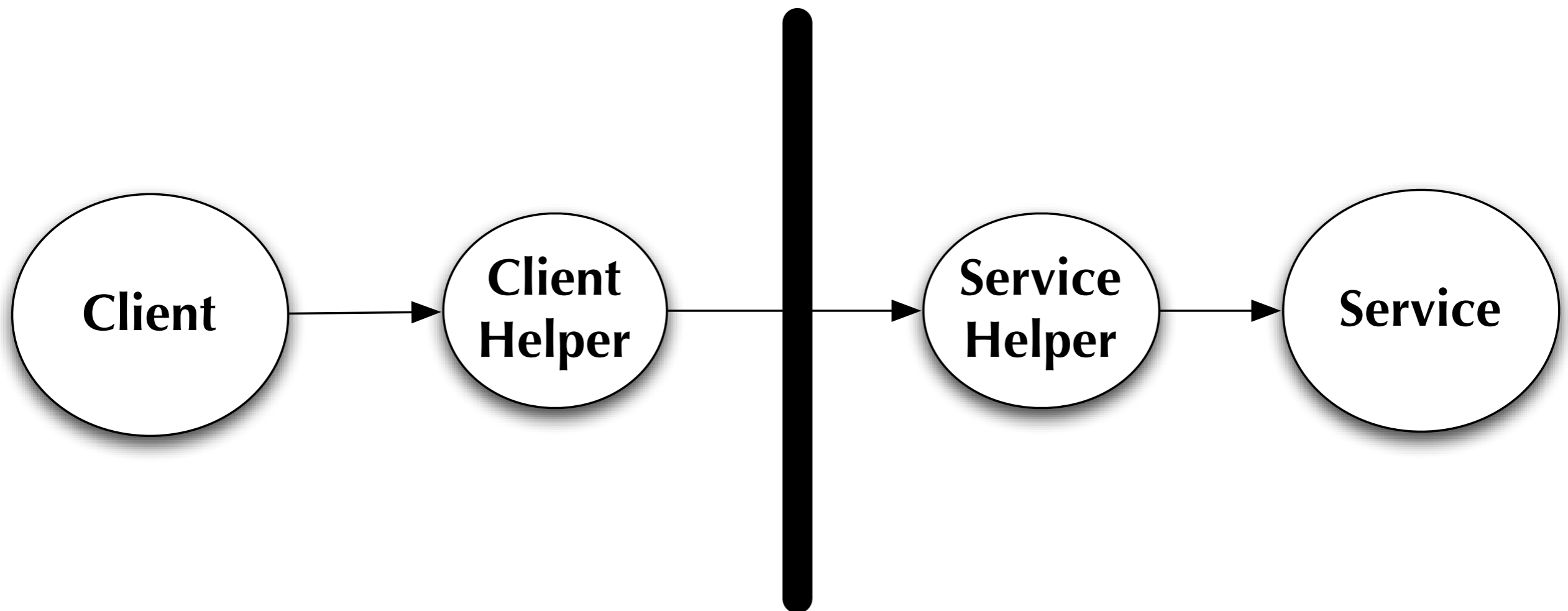
RMI creates "helper" objects that live on the client and service sides of a remote transaction. The client helper acts as a proxy for the remote service and sends method call information to the service helper. The service helper invokes the requested method on the service and returns the results.

29

Remote Method Invocation (II)

- RMI provides tools to automate the creation of the "helper" objects

    - The client helper is often called the "stub"

        - since none of the service methods are actually implemented

    - The service helper is often called the "skeleton"

        - since it often has methods that need to be filled in by the developer to hook it up to the actual service object

    - This architecture is common to many distributed computing frameworks

- These architectures do a lot to make the distributed nature of these method calls hidden from the client object… its not possible to entirely hide this process however, and the client does need to be prepared for exceptions that wouldn't normally occur when invoking methods on a local object

Remote Method Invocation (III)

- Step One: Make a remote interface

  - The interface defines the methods that the remote service provides to the client; both the client helper (stub) and the service implement this interface

- Step Two: Make a remote implementation

  - The actual service object, in this case our Gumball Machine

- Step Three: Generate the stubs and skeletons

  - Using a tool that ships with the Java SDK: rmic

- Step Four: Start the RMI registry

  - So client objects can find service objects at run-time

- Step Five: Start the remote service

  - Before clients can make calls on the remote service, it needs to be running

- Step Six: Run the client, which can now access the remote service

Remote Method Invocation (IV)

- Demonstration on Simple Example

    - Simple server with single method

        - implements MyRemote interface and extends UnicastRemoteObject

    - Server and client run on the same machine

- Warning: I found RMI to be a bit fickle

    - For instance, I got this example to work, but only after I set my classpath to equal "." (i.e. the current directory) and then performed all steps of the example with that particular classpath (sigh)

Gumball RMI Architecture

Machine Boundary
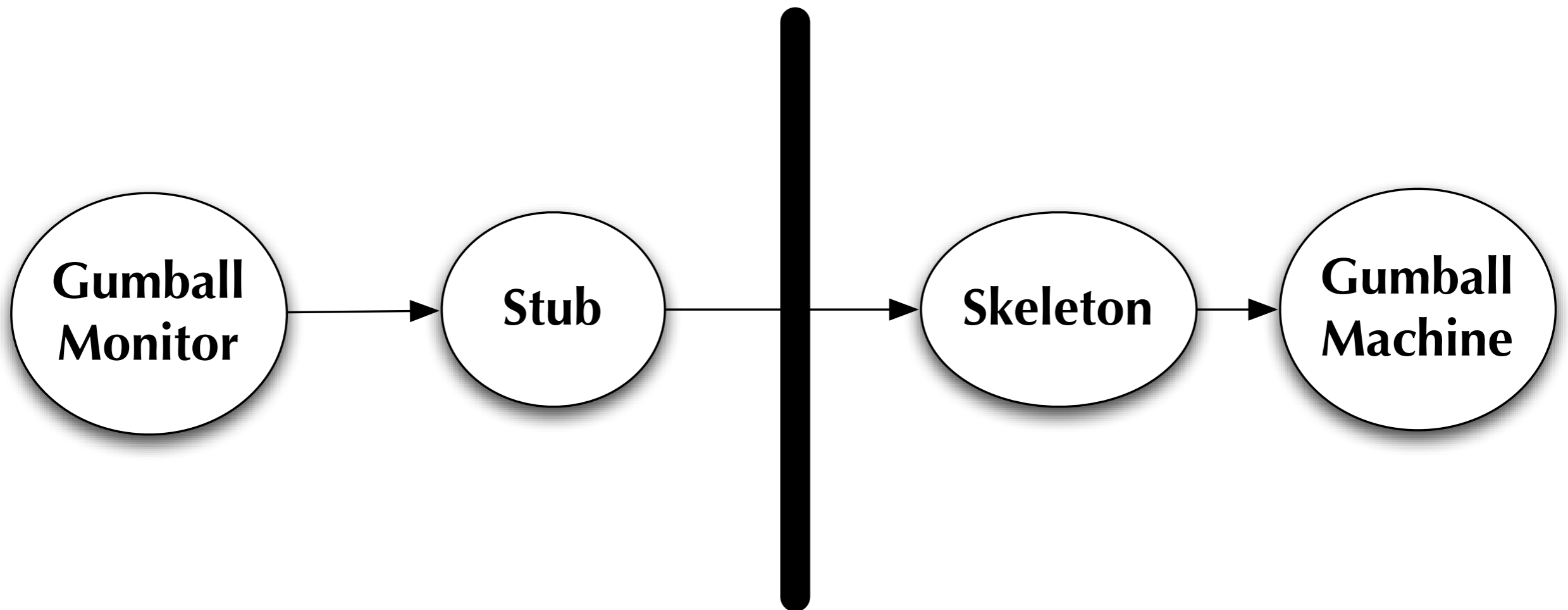
Step 1: Create Remote Interface

```
1  import java.rmi.*;
2
3  public interface GumballMachineRemote extends Remote {
4      public int getCount() throws RemoteException;
5      public String getLocation() throws RemoteException;
6      public State getState() throws RemoteException;
7  }
8
```

Simple translation of GumballMachine API into a Remote interface.

Note: RMI has a restriction that all return types and parameters need to be "serializable" which means that RMI needs to know how to "dismantle" an object of a type, send it across the network, and then assemble the information coming across the wire back into the original object.

See page 451 of the book to see how "State" is made serializable…

34

Step 2: Update Gumball Machine

```java
1  import java.rmi.*;
2  import java.rmi.server.*;
3
4  public class GumballMachine
5          extends UnicastRemoteObject implements GumballMachineRemote
6  {
7      State soldOutState;
8      State noQuarterState;
9      State hasQuarterState;
10     State soldState;
11     State winnerState;
12
13     State state = soldOutState;
14     int count = 0;
15     String location;
16
17     public GumballMachine(String location, int numberGumballs) throws RemoteException {
18         soldOutState = new SoldOutState(this);
19         noQuarterState = new NoQuarterState(this);
20         hasQuarterState = new HasQuarterState(this);
21         soldState = new SoldState(this);
22         winnerState = new WinnerState(this);
23
24         this.count = numberGumballs;
25         if (numberGumballs > 0) {
26             state = noQuarterState;
27         }
28         this.location = location;
```

35

Step 3: Update Gumball Monitor

```
 1  import java.rmi.*;
 2
 3  public class GumballMonitor {
 4
 5      GumballMachineRemote machine;
 6
 7      public GumballMonitor(GumballMachineRemote machine) {
 8          this.machine = machine;
 9      }
10
11      public void report() {
12          try {
13              System.out.println("Gumball Machine: " + machine.getLocation());
14              System.out.println("Current inventory: " + machine.getCount() + " gumballs");
15              System.out.println("Current state: " + machine.getState());
16          } catch (RemoteException e) {
17              e.printStackTrace();
18          }
19      }
20  }
21
```

Step 4: Create "main" program for service

```java
import java.rmi.*;

public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachineRemote gumballMachine = null;
        int count;

        if (args.length < 2) {
            System.out.println("GumballMachineTestDrive <name> <inventory>");
            System.exit(1);
        }

        try {
            count = Integer.parseInt(args[1]);

            gumballMachine =
                new GumballMachine(args[0], count);
            Naming.rebind(args[0], gumballMachine);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Note: NOT the same as the code in the book (which I couldn't get to work).

37

Step 5: Create "main" program for client

```java
1   import java.rmi.*;
2
3   public class GumballMonitorTestDrive {
4
5       public static void main(String[] args) {
6           String[] location = {"rmi://127.0.0.1/santafe",
7                                "rmi://127.0.0.1/boulder",
8                                "rmi://127.0.0.1/seattle"};
9
10          GumballMonitor[] monitor = new GumballMonitor[location.length];
11
12          for (int i=0;i < location.length; i++) {
13              try {
14                  GumballMachineRemote machine =
15                          (GumballMachineRemote) Naming.lookup(location[i]);
16                  monitor[i] = new GumballMonitor(machine);
17              } catch (Exception e) {
18                  e.printStackTrace();
19              }
20          }
21
22          for(int i=0; i < monitor.length; i++) {
23              monitor[i].report();
24          }
25      }
26  }
27
```

Note: NOT the same as the code in the textbook (which I couldn't get to work).

38

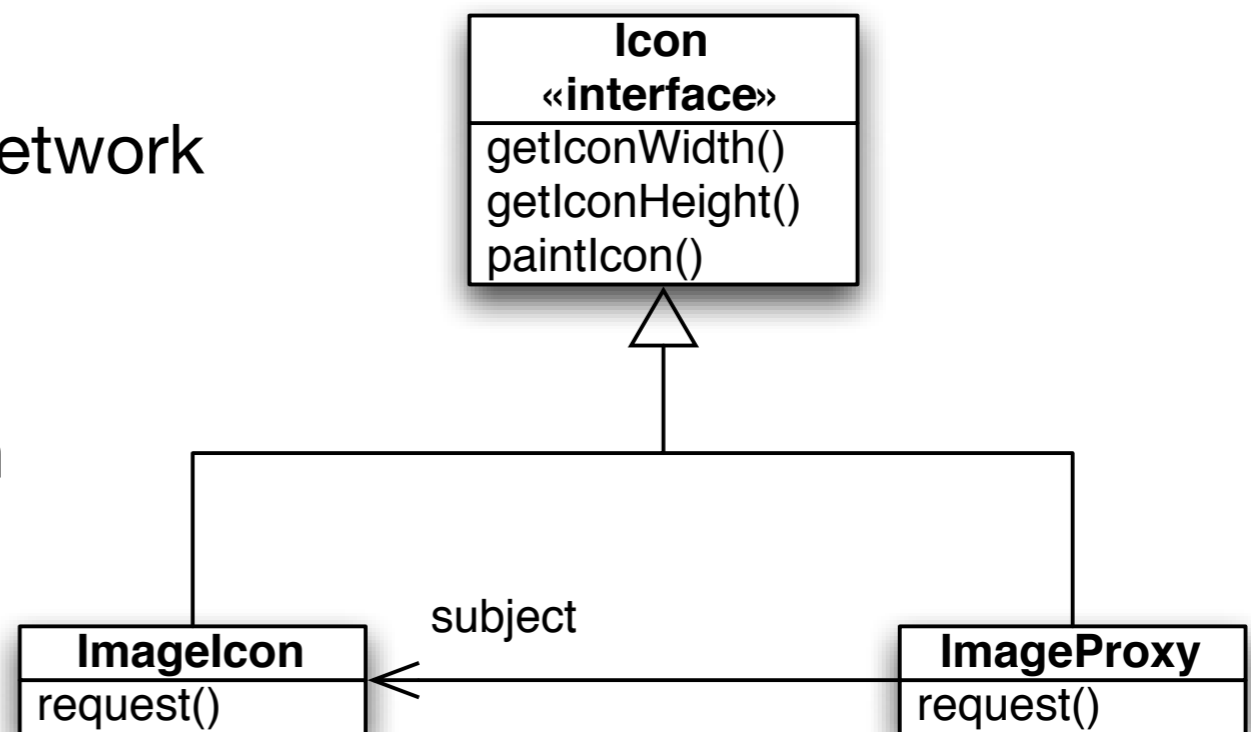# Step 6: Compile, Generate, Run

1. set CLASSPATH equal to "." (i.e. the current directory)

2. javac *.java

3. rmic GumballMachine

4. rmiregistry &

5. java GumballMachineTestDrive boulder 50 &

6. java GumballMachineTestDrive seattle 250 &

7. java GumballMachineTestDrive santafe 150 &

8. java GumballMonitorTestDrive

- **Demonstration**

Virtual Proxy

- Virtual Proxy is a variation of Proxy that provides control over when "expensive" objects are created
    - whereby expensive typically means
        - "takes a long time to create" or
        - "object takes up a lot of memory"
- The virtual proxy ensures that the object is only created when it is absolutely needed and "stands in" for the real object while the "expensive" creation process takes place
- Example: Loading Images Over a Network
    - Icon: Swing Interface
    - ImageIcon: Display Image
    - ImageProxy: Acts like ImageIcon while the image is loading...

Demonstration

Icon
«interface»
getIconWidth()
getIconHeight()
paintIcon()

subject

ImageIcon
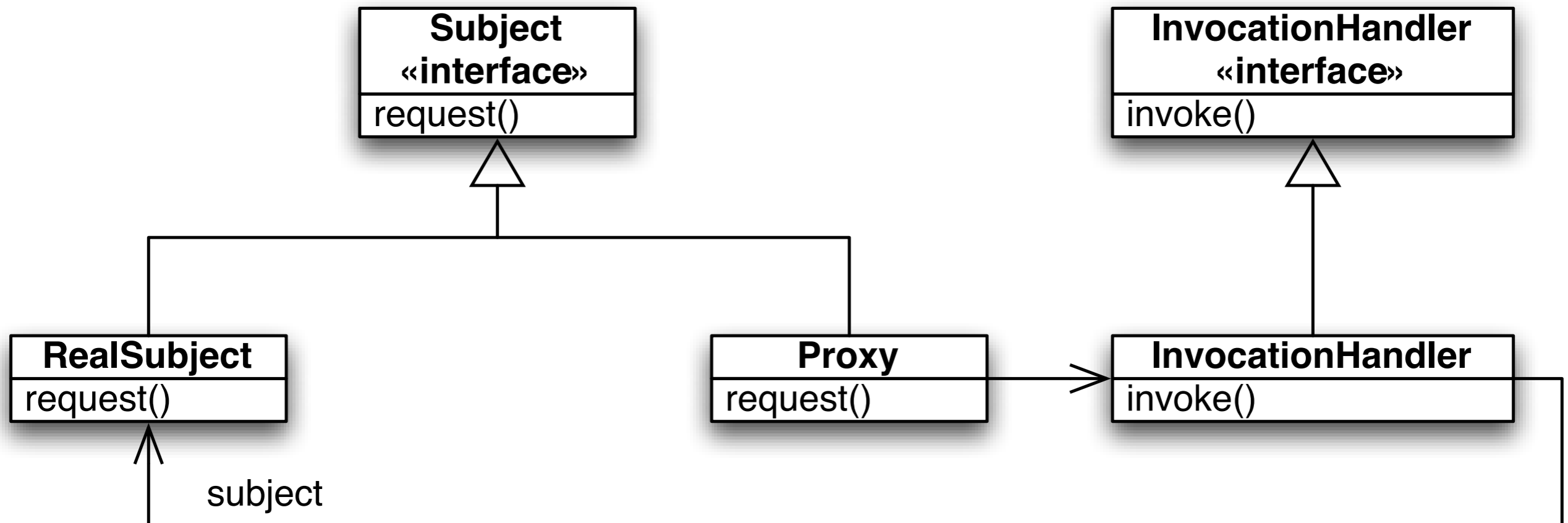request()

ImageProxy
request()

40

Protection Proxy

- A protection proxy is a variation on the Proxy pattern in which the proxy looks at the caller and the method being called and decides if it wants to forward the method call to the real subject

    - In this variation, the proxy is implementing a form of access control on top of the real subject; without this access control, any object that got a reference to the real subject could call any of its methods

- Example:

    - A "Hot or Not" website

        - Model Class: Person

        - Problems

            - Owners calling "setRating(10)" over and over to inflate their rating

            - Non-Owners calling various setter methods to capriciously (or maliciously) change the details of another person

Java's Built-In Proxy Services

- In this example, the book decides to look at Java's built in support of the Proxy pattern: aka Dynamic Proxies

  - Structure of Java's Built-In Proxy support

Subject **«interface»**
request()

InvocationHandler **«interface»**
invoke()

RealSubject
request()

Proxy
request()

InvocationHandler
invoke()

subject

Proxy is now an auto-generated class that is configured at creation time with an invocation handler. This handler determines what requests get forwarded to the RealSubject.

42

Hot Or Not (I): Specify Interface for Subject

```java
 1  public interface PersonBean {
 2
 3      String getName();
 4      String getGender();
 5      String getInterests();
 6      String getStringRep();
 7      double getHotOrNotRating();
 8
 9      void setName(String name);
10      void setGender(String gender);
11      void setInterests(String interests);
12      void setHotOrNotRating(double rating);
13
14  }
```

Note: interface is slightly different from that in book. I changed it to make the output of the test program more understandable!

Differences:
    Added "String getStringRep();"
    Changed: "HotOrNot" methods to work with "double" not "int"

43

Hot Or Not (II): Implement RealSubject

```
 1  public class PersonBeanImpl implements PersonBean {
 2
 3      String name;
 4      String gender;
 5      String interests;
 6      double rating;
 7      int ratingCount = 0;
 8
 9      public String getName() {
10          return name;
11      }
12
13      public String getGender() {
14          return gender;
15      }
16
17      public String getInterests() {
18          return interests;
19      }
20
21      public double getHotOrNotRating() {
22          if (ratingCount == 0) return 0;
23          return (rating/ratingCount);
24      }
```

<rest of class not shown>

Standard "information holder" object with getter/setter routines.

HotOrNot methods keep track of number of ratings submitted and return an average value.

getStringRep() produces a report of all current values of the object.

44

Hot Or Not (III): Implement InvocationHandlers

```java
1  import java.lang.reflect.*;
2
3  public class OwnerInvocationHandler implements InvocationHandler {
4      PersonBean person;
5
6      public OwnerInvocationHandler(PersonBean person) {
7          this.person = person;
8      }
9
10     public Object invoke(Object proxy, Method method, Object[] args)
11             throws IllegalAccessException {
12
13         try {
14             if (method.getName().startsWith("get")) {
15                 return method.invoke(person, args);
16             } else if (method.getName().equals("setHotOrNotRating")) {
17                 throw new IllegalAccessException();
18             } else if (method.getName().startsWith("set")) {
19                 return method.invoke(person, args);
20             }
21         } catch (InvocationTargetException e) {
22             e.printStackTrace();
23         }
24         return null;
25     }
26  }
```

Note: Makes use of Java's Reflection API; any attempt to set your own rating is denied.

45

Hot Or Not (IV): Implement InvocationHandlers

```java
1  import java.lang.reflect.*;
2
3  public class NonOwnerInvocationHandler implements InvocationHandler {
4      PersonBean person;
5
6      public NonOwnerInvocationHandler(PersonBean person) {
7          this.person = person;
8      }
9
10     public Object invoke(Object proxy, Method method, Object[] args)
11             throws IllegalAccessException {
12
13         try {
14             if (method.getName().startsWith("get")) {
15                 return method.invoke(person, args);
16             } else if (method.getName().equals("setHotOrNotRating")) {
17                 return method.invoke(person, args);
18             } else if (method.getName().startsWith("set")) {
19                 throw new IllegalAccessException();
20             }
21         } catch (InvocationTargetException e) {
22             e.printStackTrace();
23         }
24         return null;
25     }
26 }
```

Note: Makes use of Java's Reflection API; any attempt to set a person's attributes (other than rating) is denied.

46

Hot Or Not (V): Create Dynamic Proxies

```
63    PersonBean getOwnerProxy(PersonBean person) {
64
65        return (PersonBean) Proxy.newProxyInstance(
66            person.getClass().getClassLoader(),
67            person.getClass().getInterfaces(),
68            new OwnerInvocationHandler(person));
69    }
70
71    PersonBean getNonOwnerProxy(PersonBean person) {
72
73        return (PersonBean) Proxy.newProxyInstance(
74            person.getClass().getClassLoader(),
75            person.getClass().getInterfaces(),
76            new NonOwnerInvocationHandler(person));
77    }
```

Call Proxy's newProxyInstance() method; provide access to RealSubject's class loader, its interfaces, and the desired invocation handler.

Demon-stration

Client code then gets an instance of RealSubject and wraps it.

Wrapping Up (I)

- The State Pattern allows an object to have many different behaviors that are based on its internal state

    - Unlike a procedural state machine, the State Pattern represents state as a full-blown class

    - The state machine object gets its behavior by delegating to its current state object

    - Each state object has the power to change the state of the state machine object, aka context object

- The Flyweight Pattern is useful for managing situations where you need lots of "small" objects but you don't want them taking up a lot of memory

    - It is an example of a "pattern of patterns" as it requires use of the Factory pattern to control the creation of the "small" objects

Wrapping Up (II)

- Proxy is an extremely flexible pattern that allows you to control access to a particular object

  - We've seen examples of proxies that enable distributed access, control of "expensive" objects, and protection of an object's methods

- The book also mentions the use of proxies to

  - mimic a firewall in controlling access to network resources

  - keep track of the number of objects pointing at a subject

  - cache the results of expensive operations

  - protect an object from being accessed by multiple threads

  - and more

Coming Up Next

- Lecture 24: Patterns of Patterns

    - Read Chapter 12 of the Design Patterns Textbook