State and Flyweight Patterns

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 28 — 11/29/2007

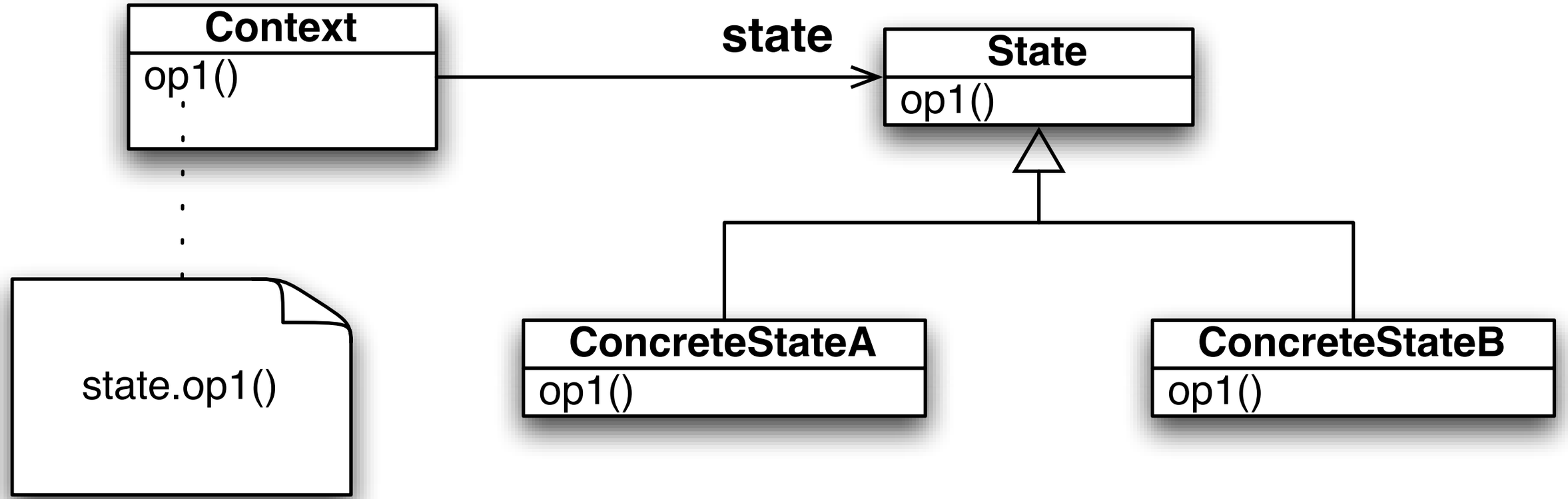
Lecture Goals

- Cover Material from Chapter 10 of the Design Patterns Textbook
    - State Pattern
- Bonus Pattern
    - Flyweight (not from textbook)

State Pattern: Definition

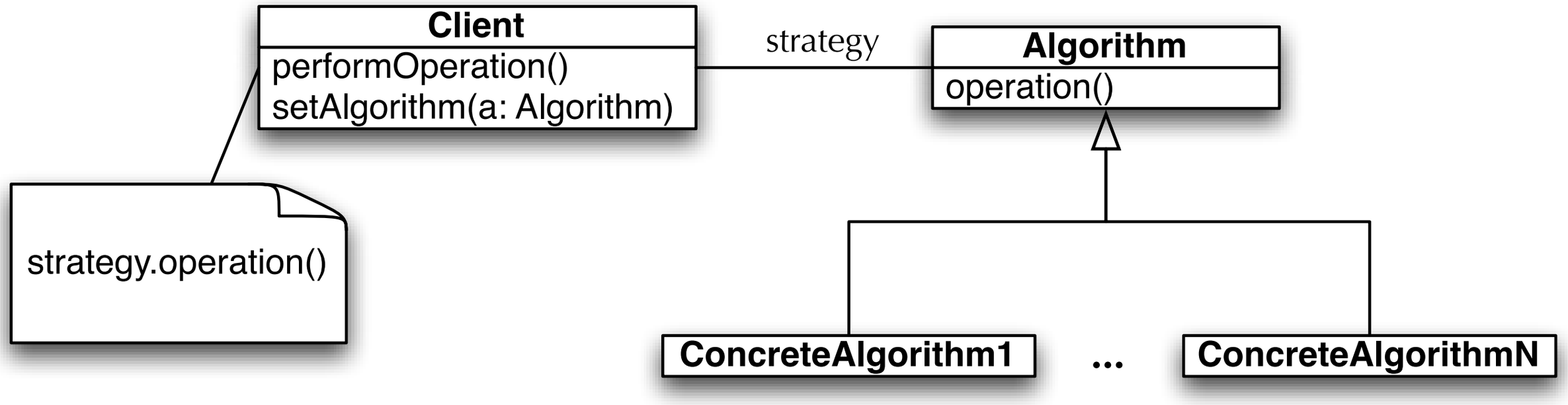
- The state pattern provides a clean way for an object to vary its behavior based on its current “state”
    - That is, the object’s public interface doesn’t change but each method’s behavior may be different as the object’s internal state changes
- Definition: The State Pattern allows an object to alter 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)

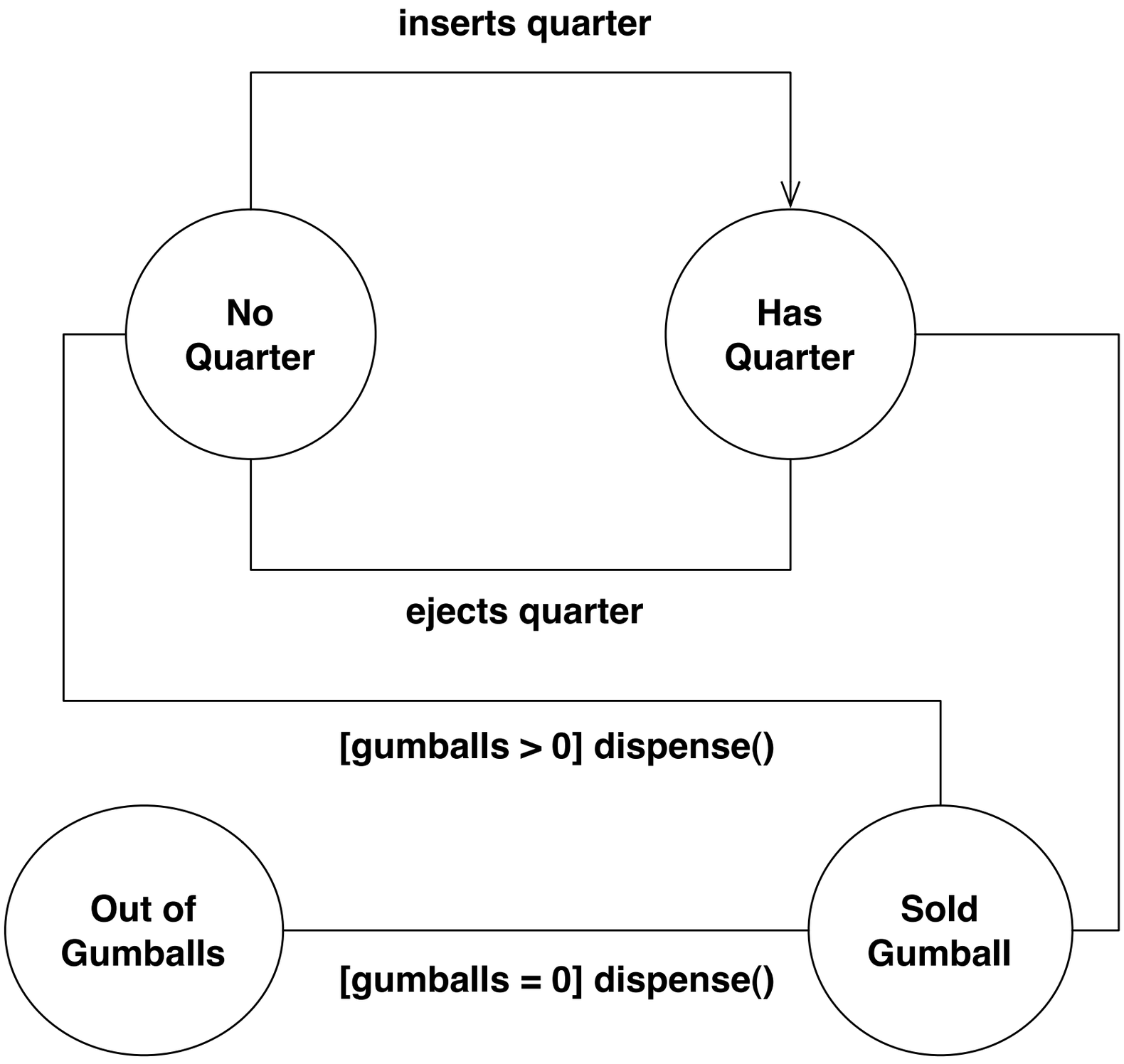


Strategy and State Patterns: Separated at Birth?!

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.

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

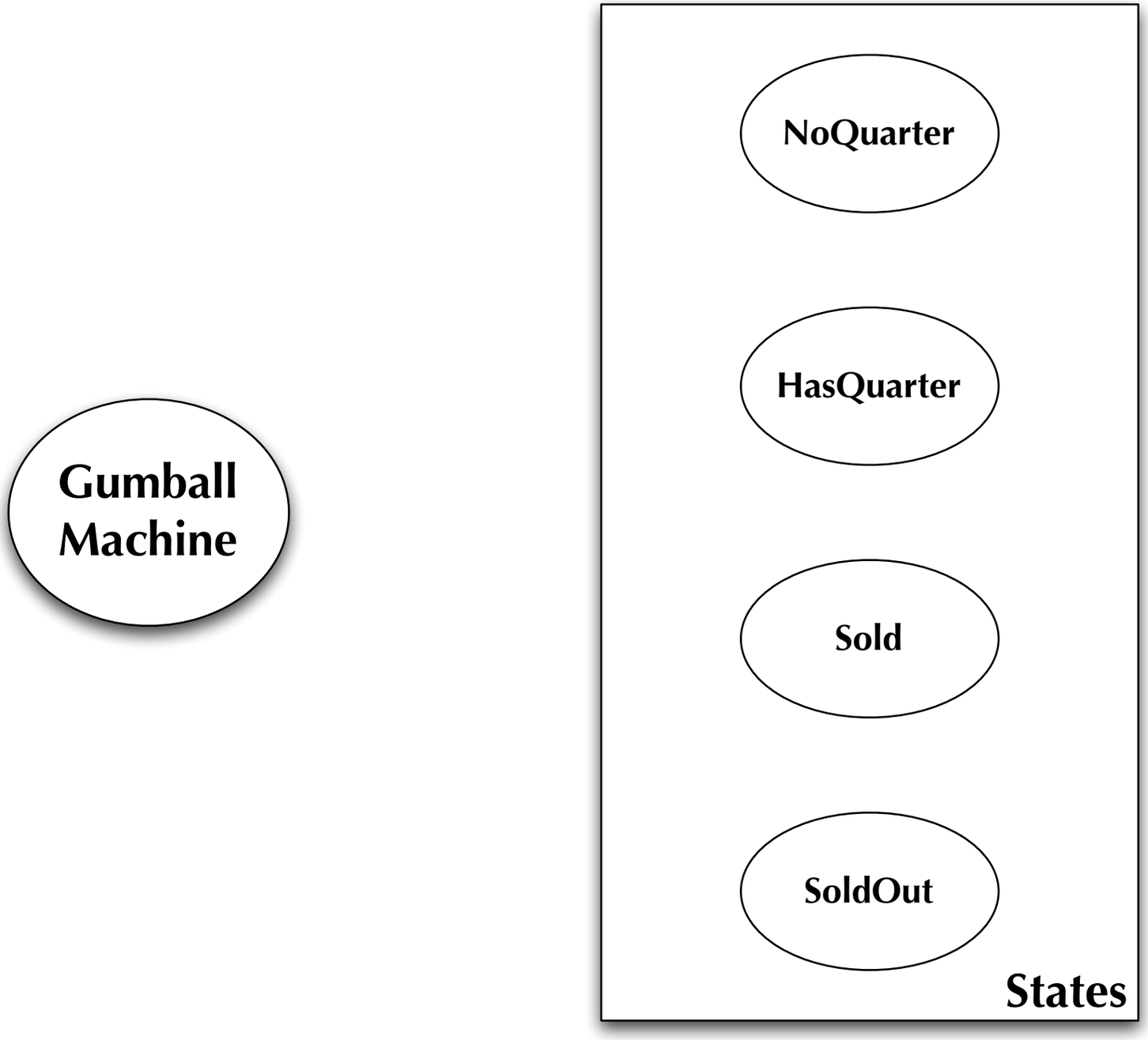
Design Problems with First Attempt

- 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”
- “This code would make a FORTRAN programmer proud” — FORTRAN code can often be very convoluted, aka spaghetti code, no structure, just a mess!

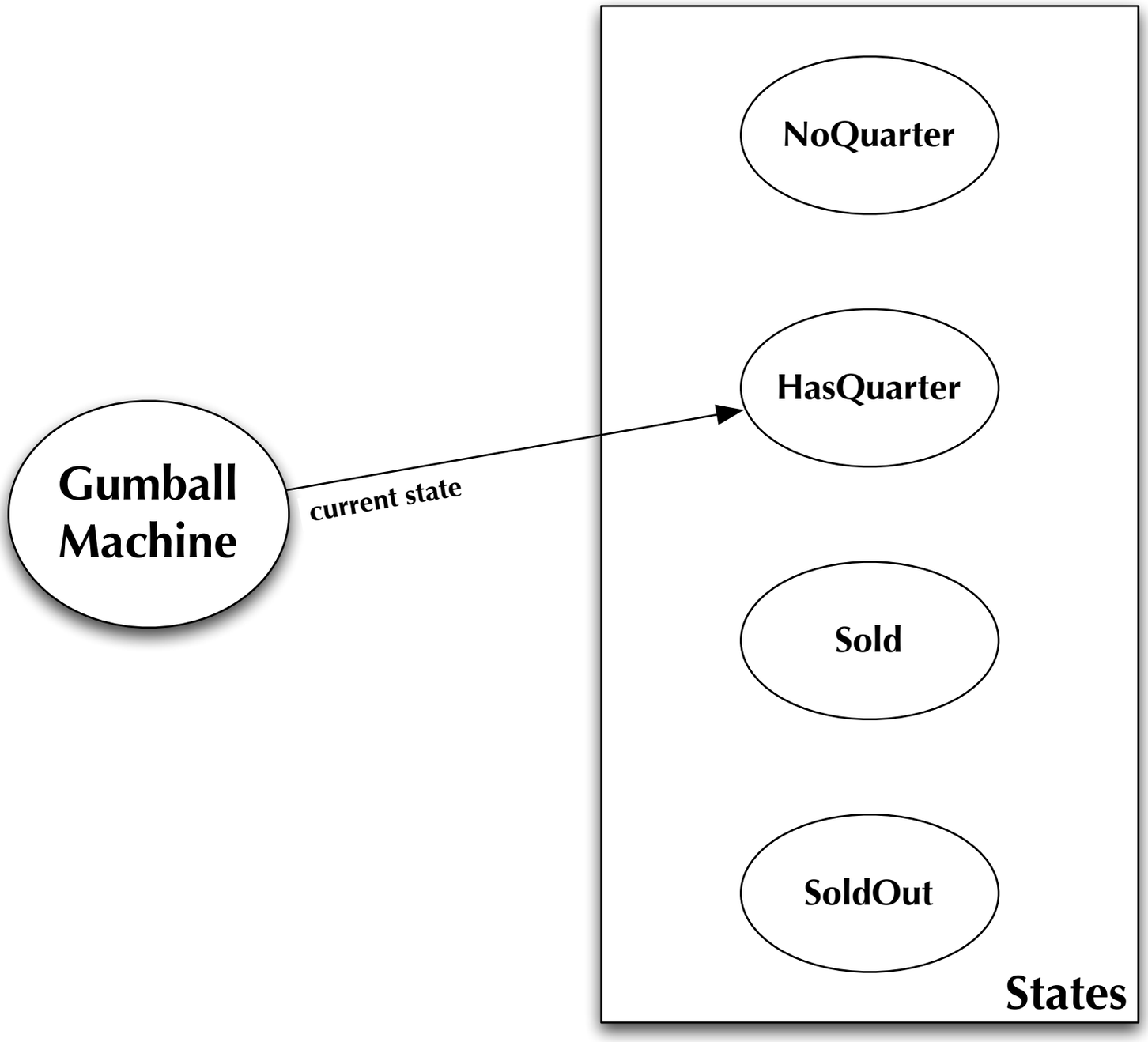
2nd Attempt: Use State Pattern

- 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

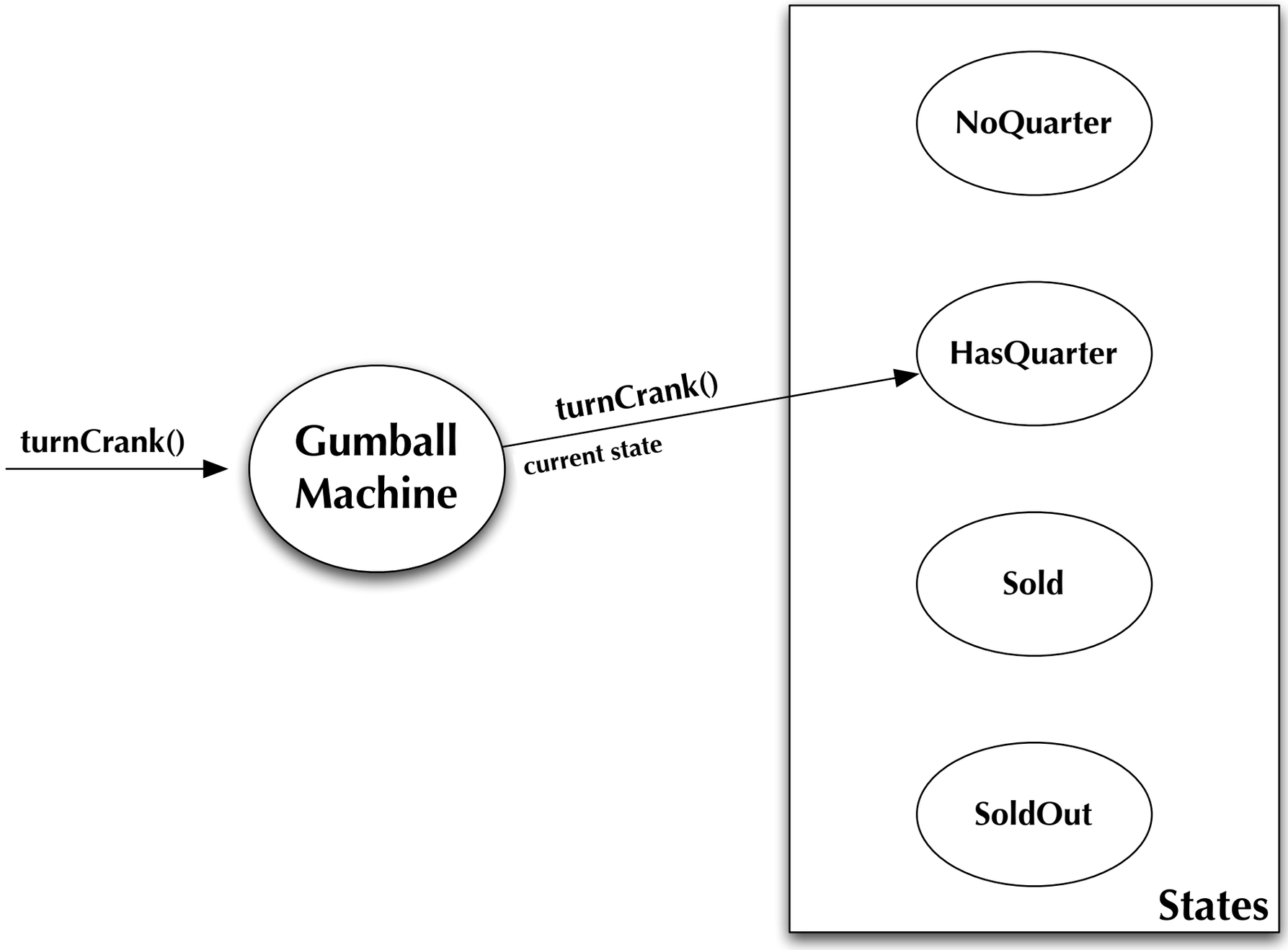
State Pattern in Action (I)



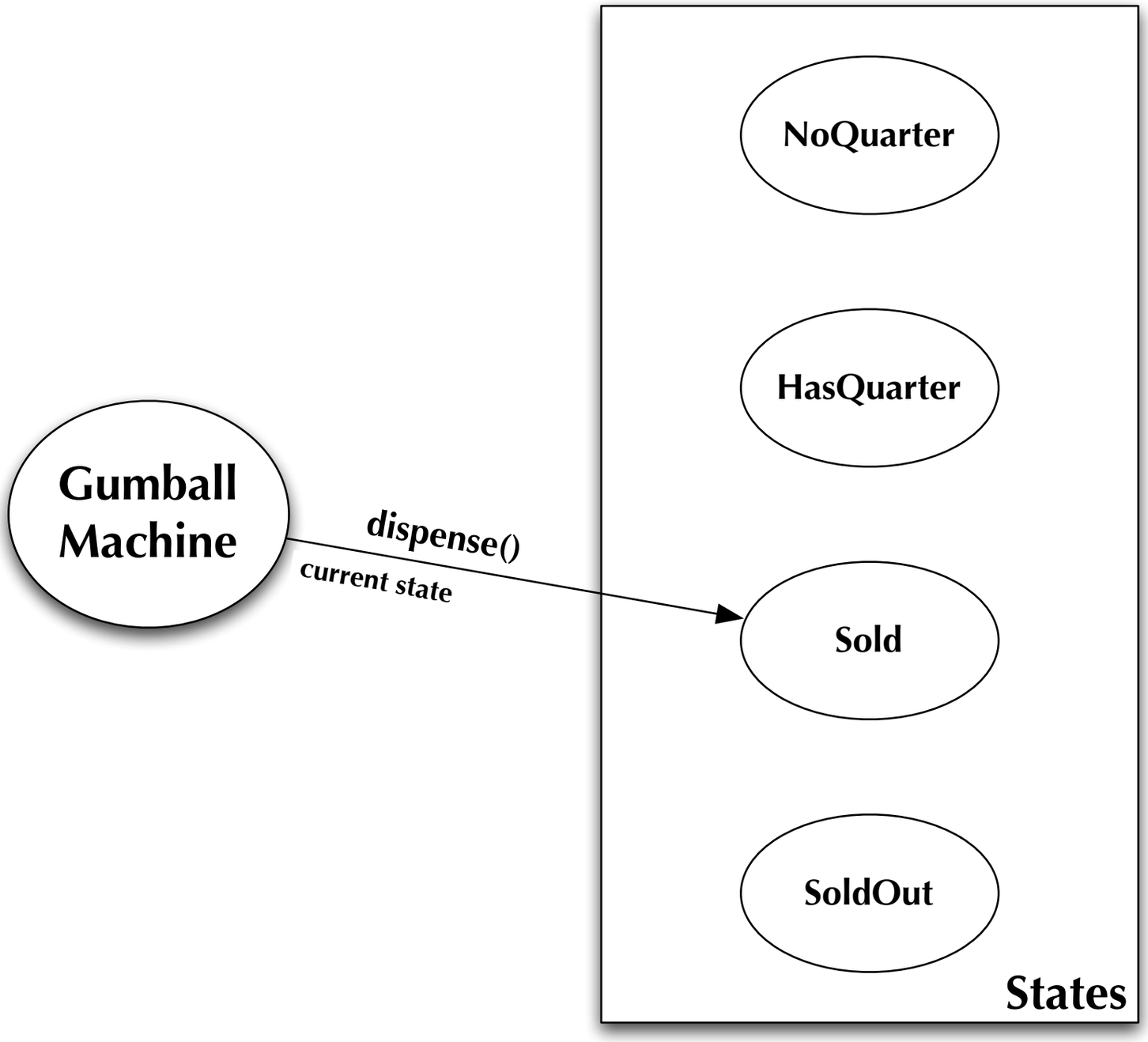
State Pattern in Action (II)



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

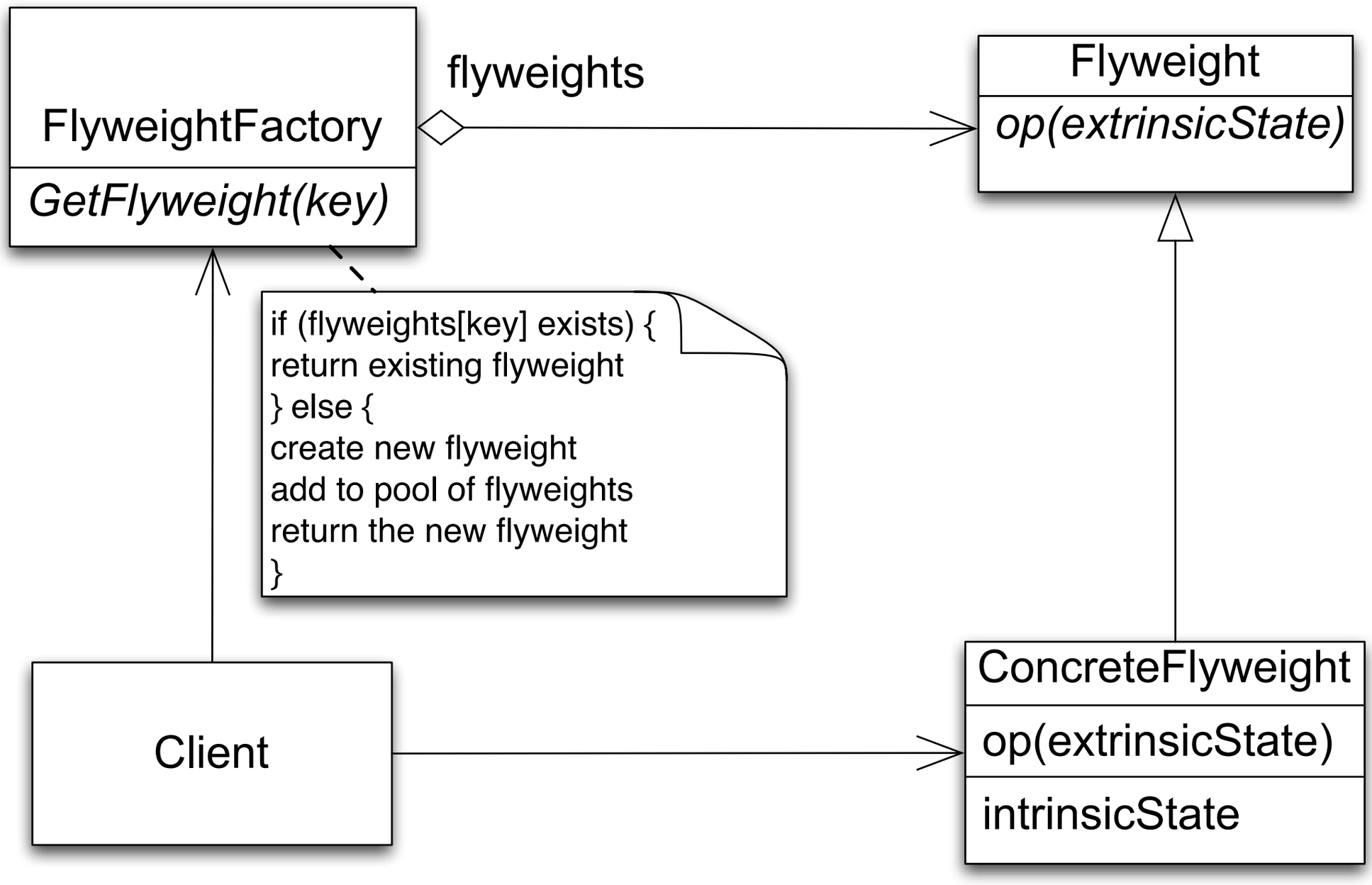
Flyweight, continued

- 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, 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

Wrapping Up

- 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

Coming Up Next

- Lecture 29: Proxy Pattern
    - Read Chapter 11 of the Design Patterns Textbook
- Lecture 30: Patterns of Patterns
    - Read Chapter 12 of the Design Patterns Textbook