State

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 26

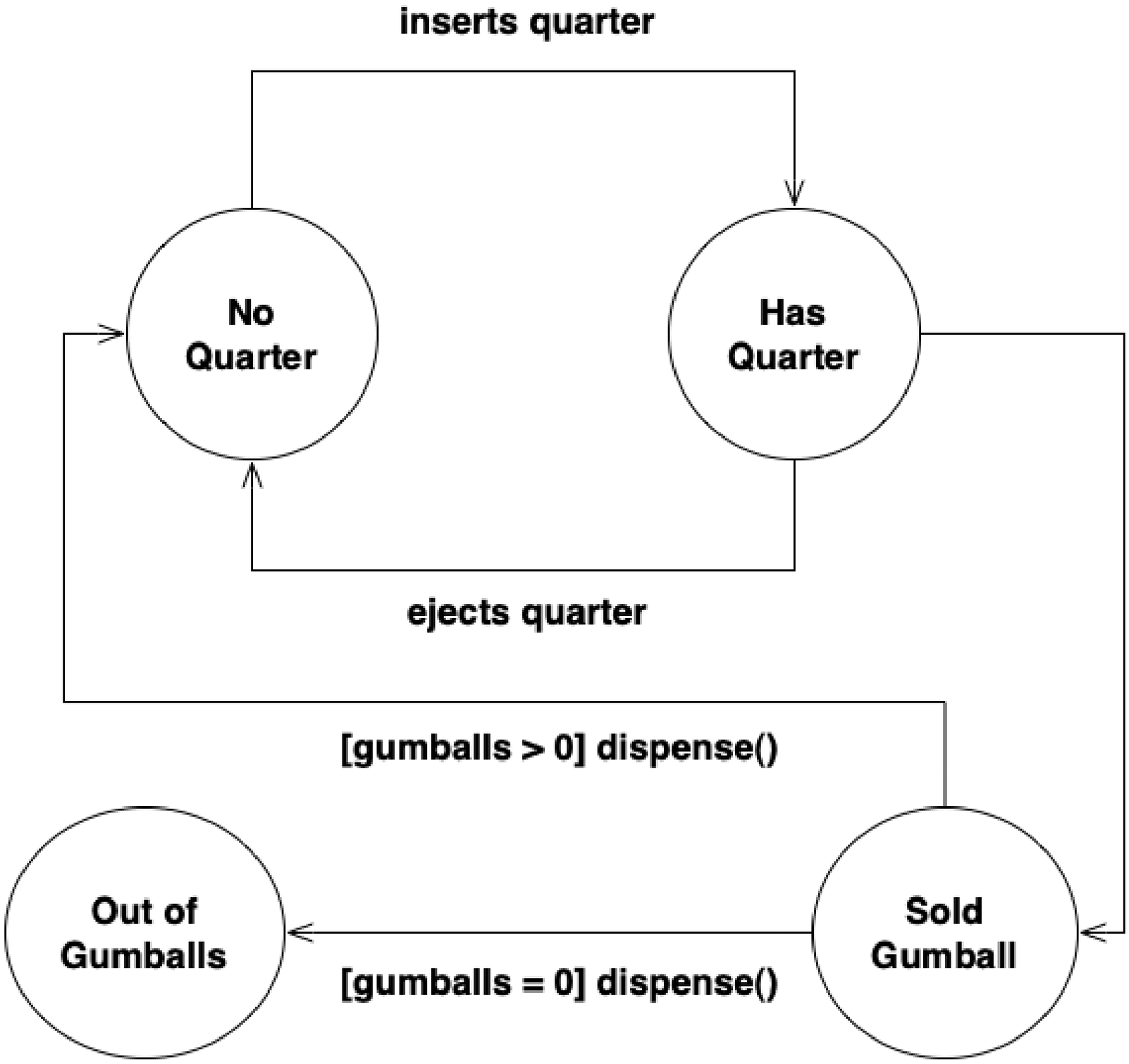
Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Head First Design Patterns

- This material gets us back in the book chapter order – Chapter 10 on State Pattern
- The example in the book uses a Gumball Machine (relatively duck free)…

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

- Common to use state machines in C for embedded system behavior
- State-Centric State Machines
- State-Centric State Machines with Hidden Transitions
- Event-Centric State Machines
- Table-Driven State Machines
  - From the book Making Embedded Systems, White, 2011, O'Reilly

- By the way, what's a Finite State Machine?
  - a state machine where all states and transitions are defined
    - from Embedded Systems Fundamentals with Arm Cortex-M based Microcontrollers, Dean, 2017, ARM Education Media

State-Centric State Machines

- This form of a state machine is just a big if-else or switch
- If I get an unhandled state or an improper event, do I just stay in the state I'm in or raise an error – depends on the system…

```
while (1) {
        look for event
        switch (state) {
                // one case for each unique state
                case (state)
                        if event valid for this state
                                handle event, prepare for new state, set new state
                // should consider error cases
                default (unhandled state)
                        error
        }
}
```

State-Centric State Machine with Hidden Transitions

- This version of the code has another function to determine what state you go to based on the state you're in

```
while (1) {
        look for event
        switch (state) {
        // one case for each unique state
        case (state)
                make sure this state is doing what it should
                if event valid for this state
                        handle event, prepare for new state
                        call the next state function
        default (unhandled state)
                error
}
```

Using a next state function separates actions from state transitions – better encapsulated, fewer dependencies – but in some cases may reduce readability

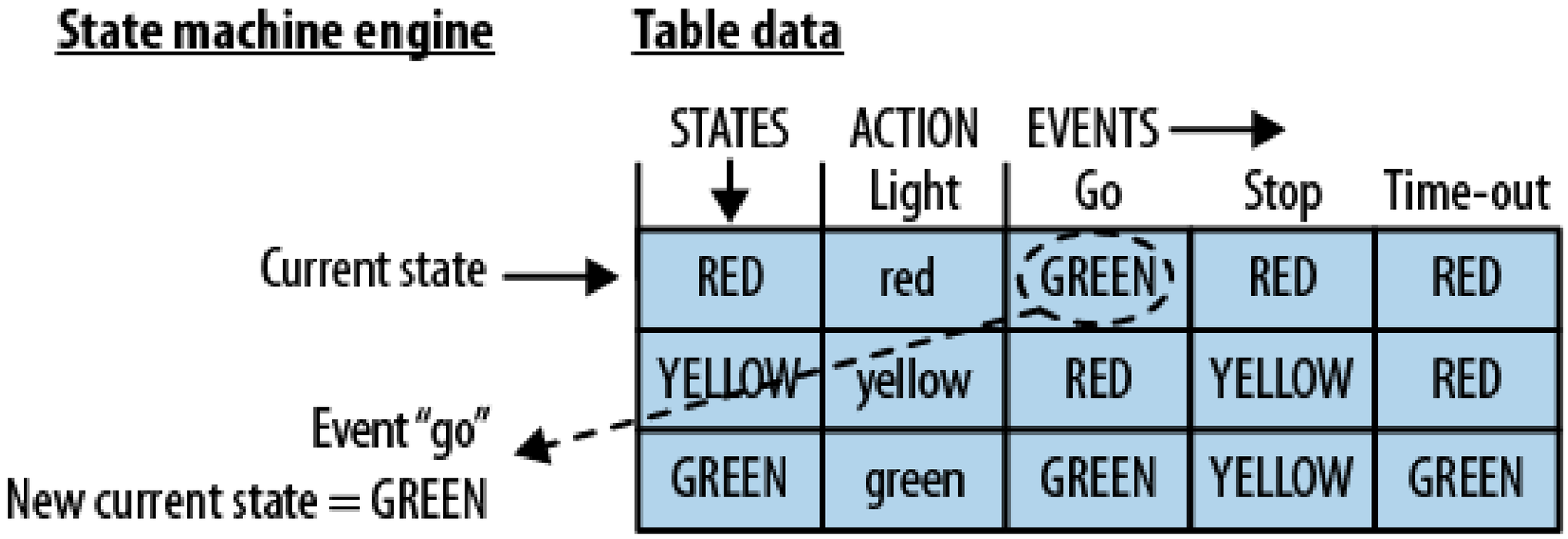
Event-Centric State Machines

- This version of the state machine switches based on transition events, not states

```
while (1) {
        look for event
    switch (event) {
        // one case for each unique event
        case (event)
            if for active state, there is a state transition for this event
                handle event, prepare for new state
                go to the next state
        default (unhandled state)
            error
}
```

Table-Driven State Machines

- All states, actions, and events are represented in a table
- Data structures and program logic are used to apply the tabular transitions
- This table is for a traffic light controller



Modeling State without State Pattern, revisited

- I'll take a swing at a custom state machine
- 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 or event (Event-centric)
  - Inside each method, we code the behavior that transition would have given the current state; we do this using conditional statements

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

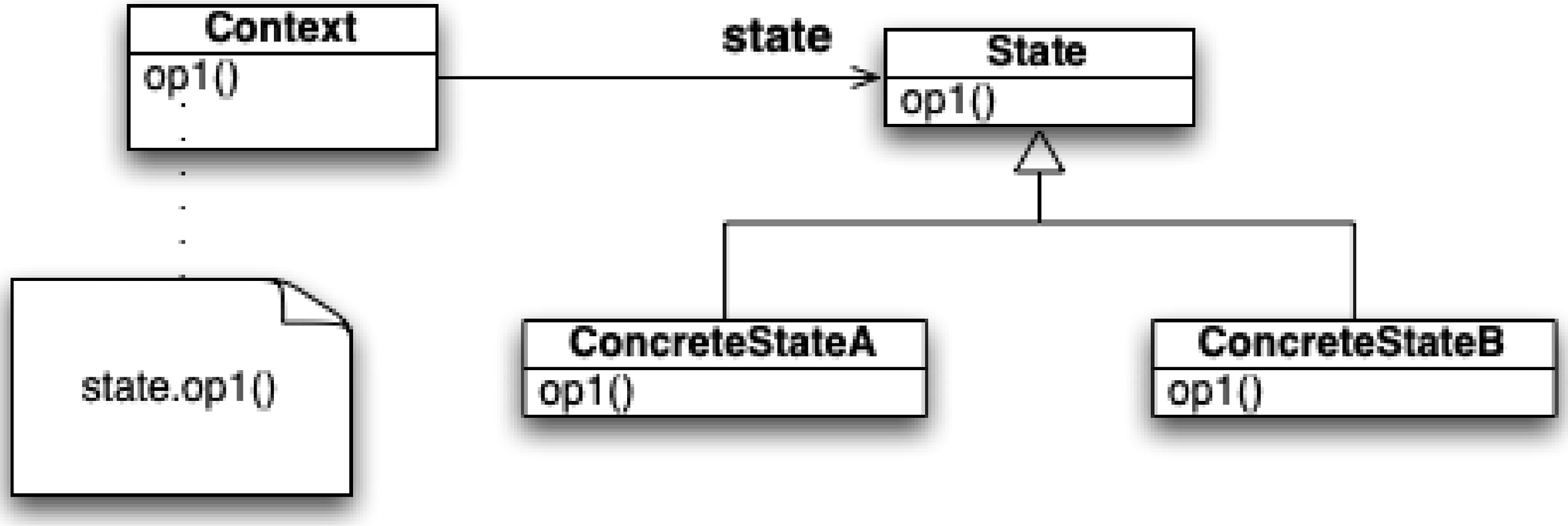
Design Problems with First Attempt

- 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"

The OO State Pattern: Definition

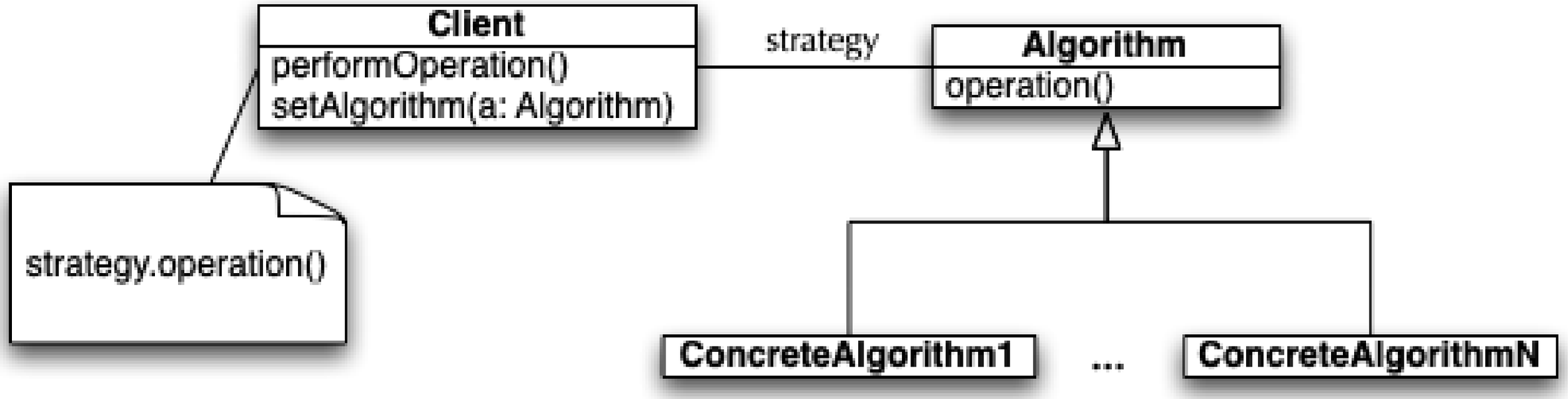
- The state pattern provides a clean way for an object to vary its behavior based on its current “state”
  - That is, the object’s public interface doesn’t change but each method’s behavior may be different as the object’s internal state changes
- **Definition**: The State Pattern allows an object to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



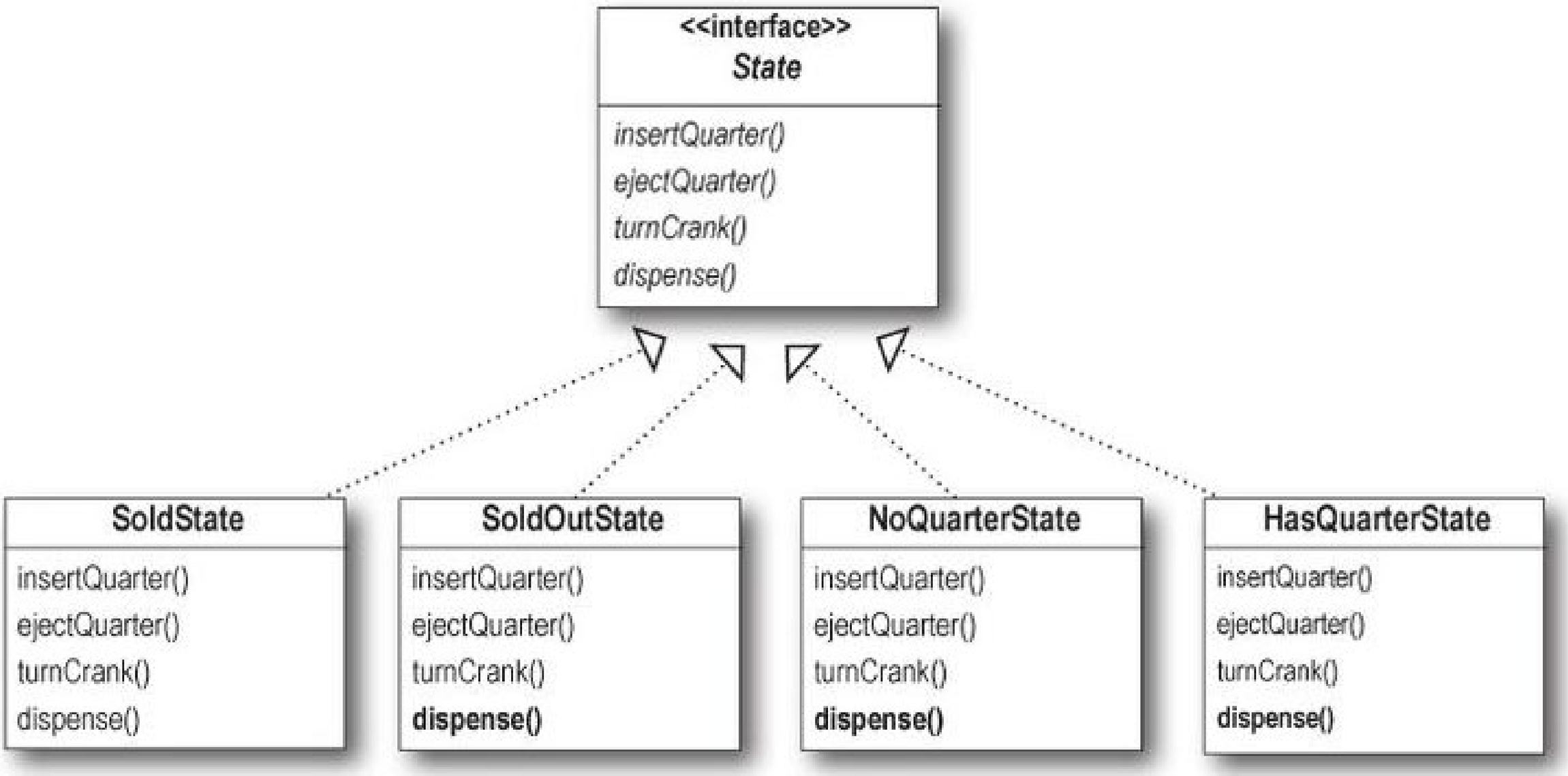
Strategy and State are structurally equivalent; their intent however is different.

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.

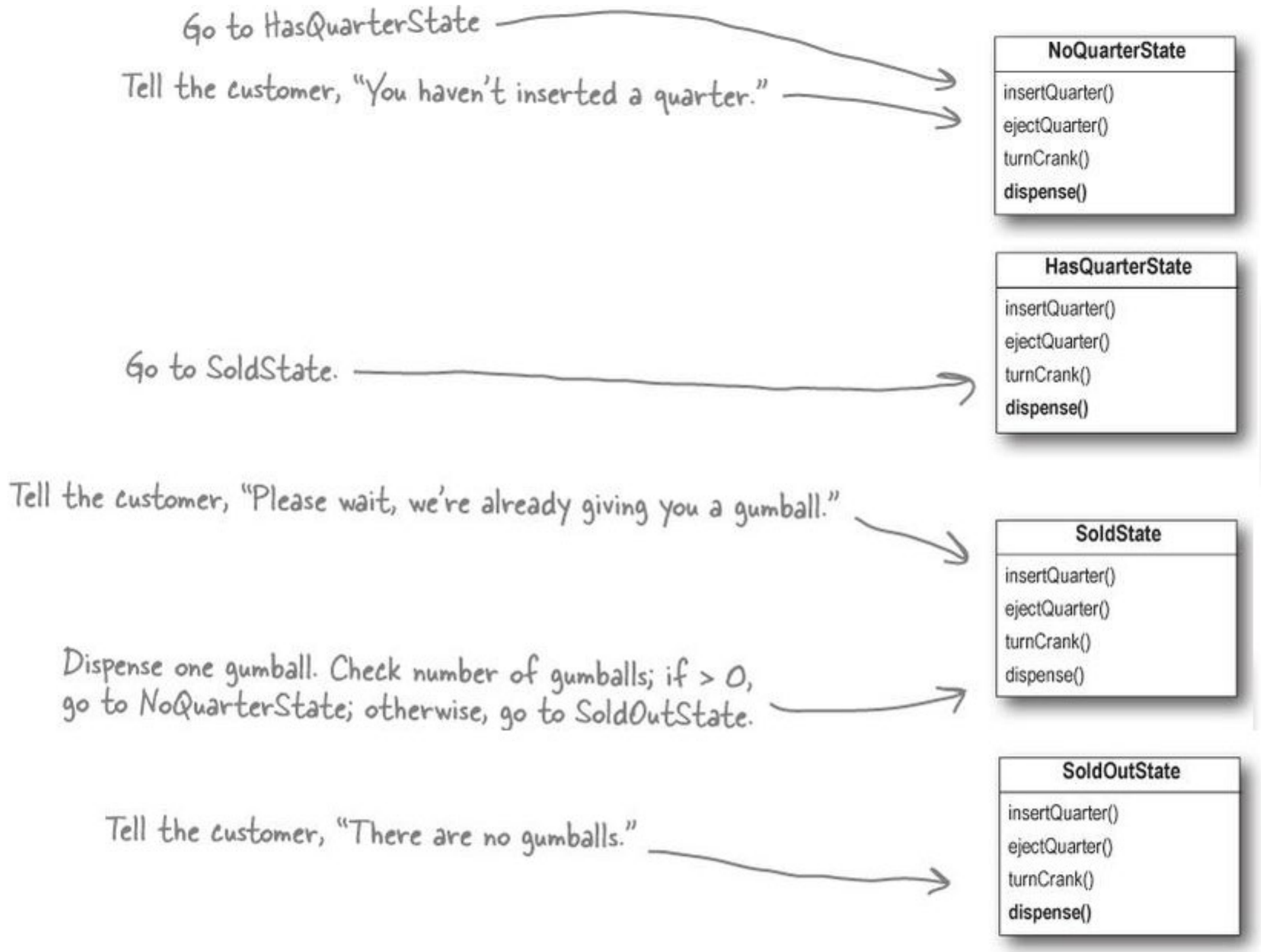
2nd Attempt: Use State Pattern

- 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

Define State Interface and Classes



What are the behaviors of the classes when an action occurs?



Typical State Code

- Fairly lengthy implementation…
- Check out the full implementation in the book
- A Python example can be found here: https://refactoring.guru/design-patterns/state/python/example

```
public class SoldState implements State {
    //constructor and instance variables here

    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }

    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }

    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }

    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}
```

Here are all the inappropriate actions for this state.

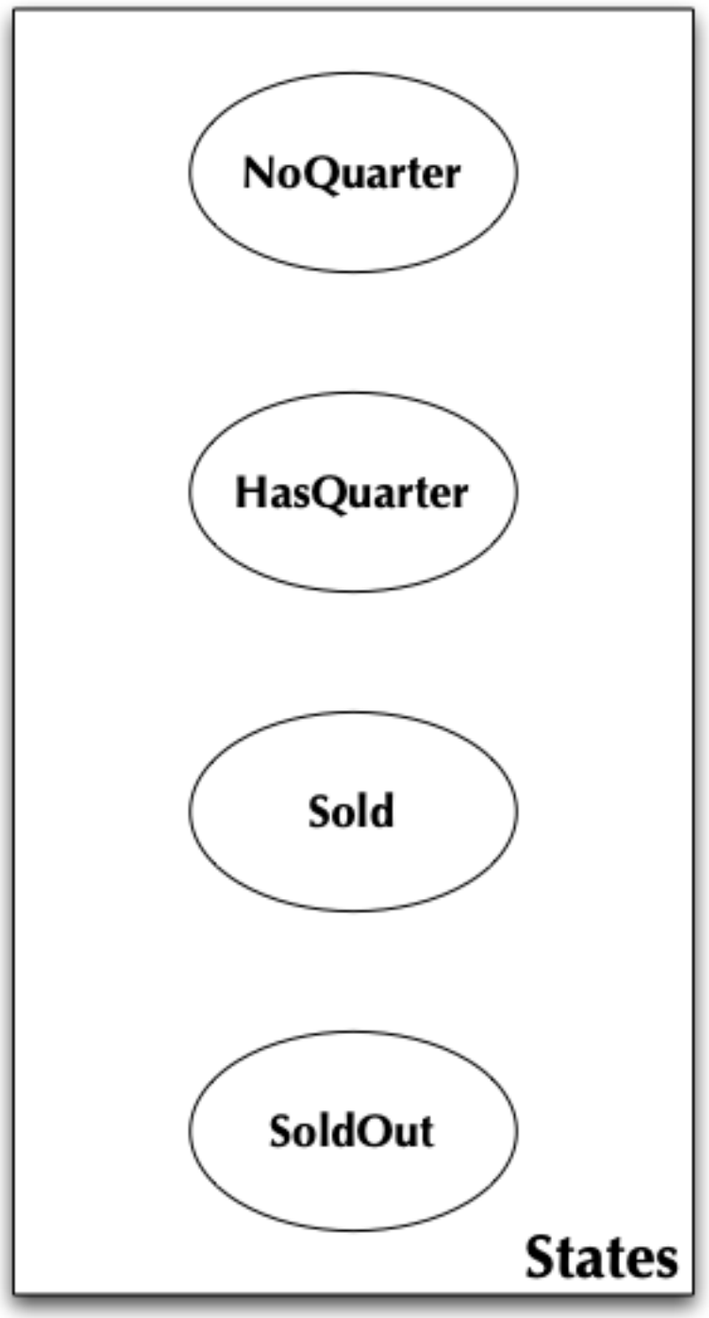
And here's where the real work begins...

We're in the SoldState, which means the customer paid. So, we first need to ask the machine to release a gumball.

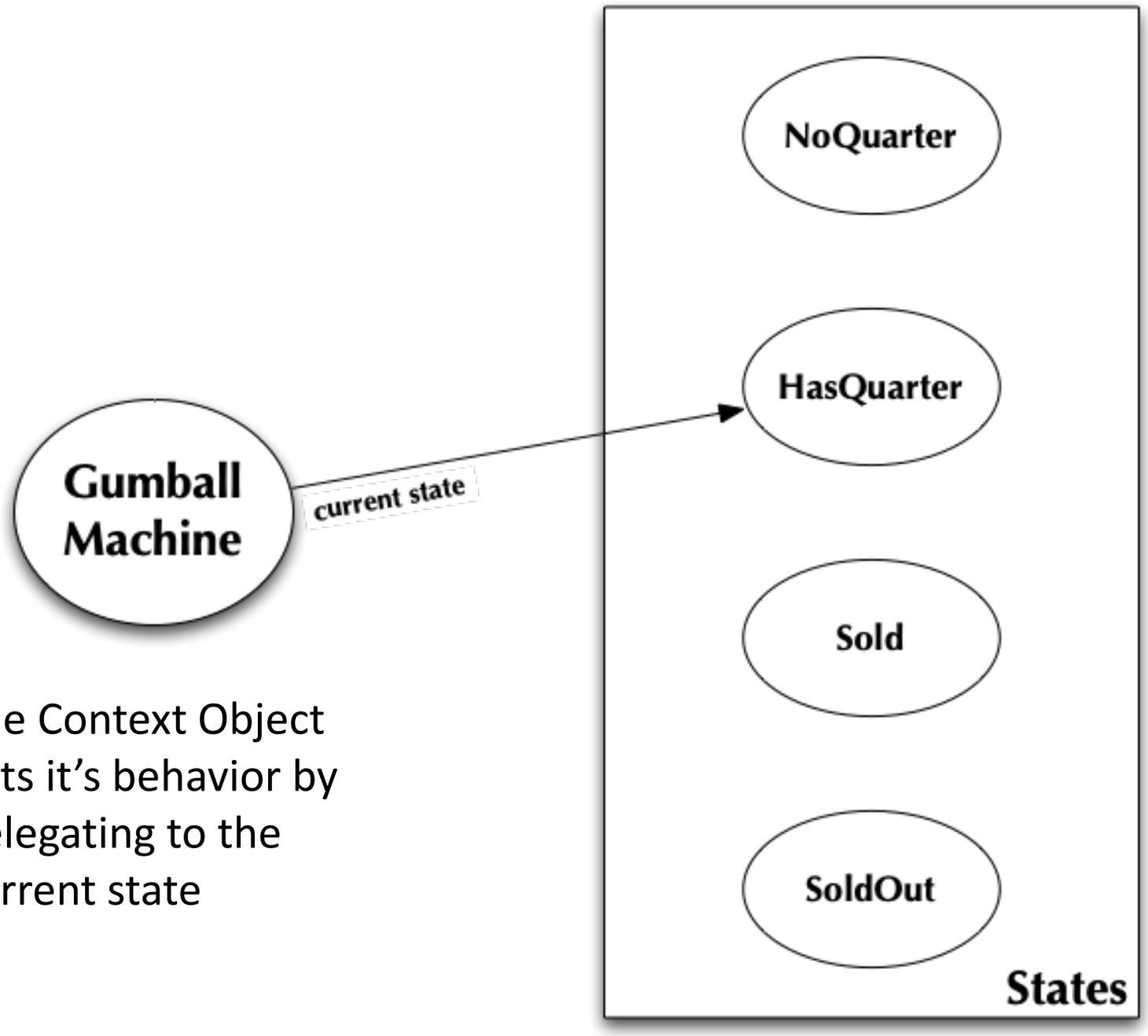
Then we ask the machine what the gumball count is, and either transition to the NoQuarterState or the SoldOutState.

State Pattern in Action (I)

The Context Object – Gumball Machine – knows the current state

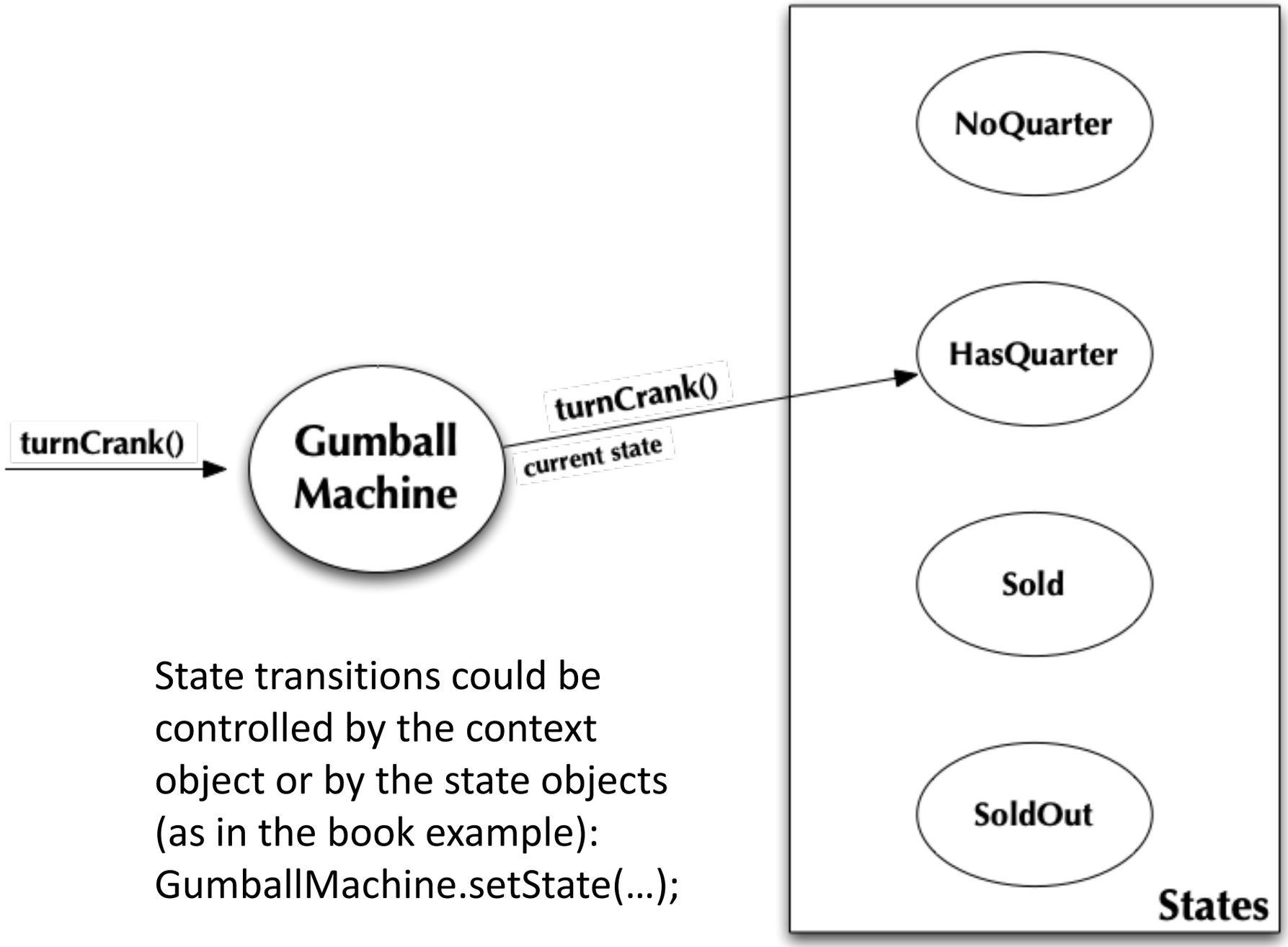


State Pattern in Action (II)



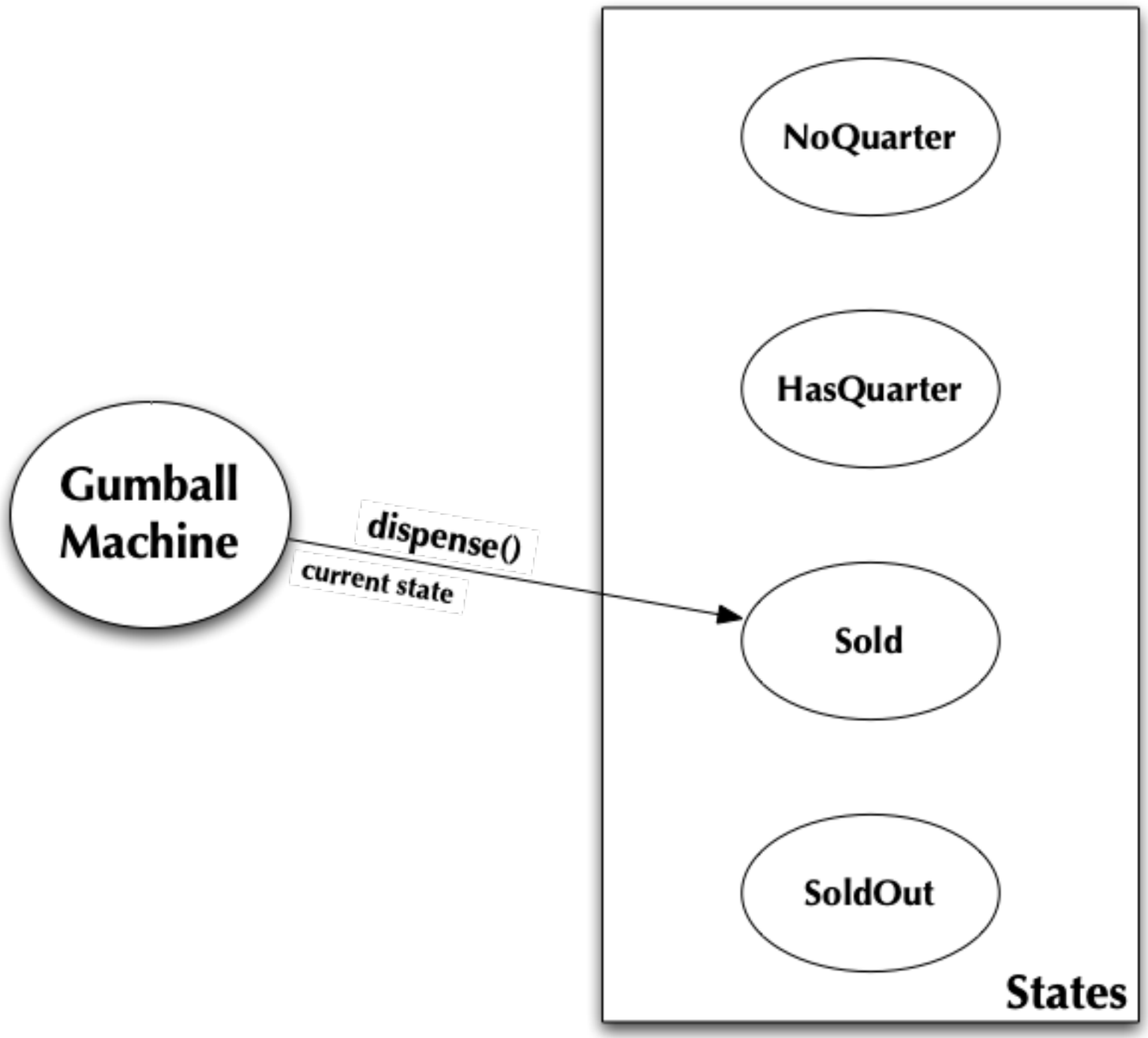
The Context Object gets it’s behavior by delegating to the current state

State Pattern in Action (III)



State transitions could be controlled by the context object or by the state objects (as in the book example):
GumballMachine.setState(…);

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

Final Points

- 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 Context gets its behavior by delegating to the current state object it is composed with.
- By encapsulating each state into a class, we localize any changes that will need to be made.
- The State and Strategy Patterns have the same class diagram, but they differ in intent.
  - Strategy Pattern typically configures Context classes with a behavior or algorithm.
  - State Pattern allows a Context to change its behavior as the state of the Context changes.
- State transitions can be controlled by the State classes or by the Context classes.
- Using the State Pattern will typically result in a greater number of classes in your design.
- State classes may be shared among Context instances.