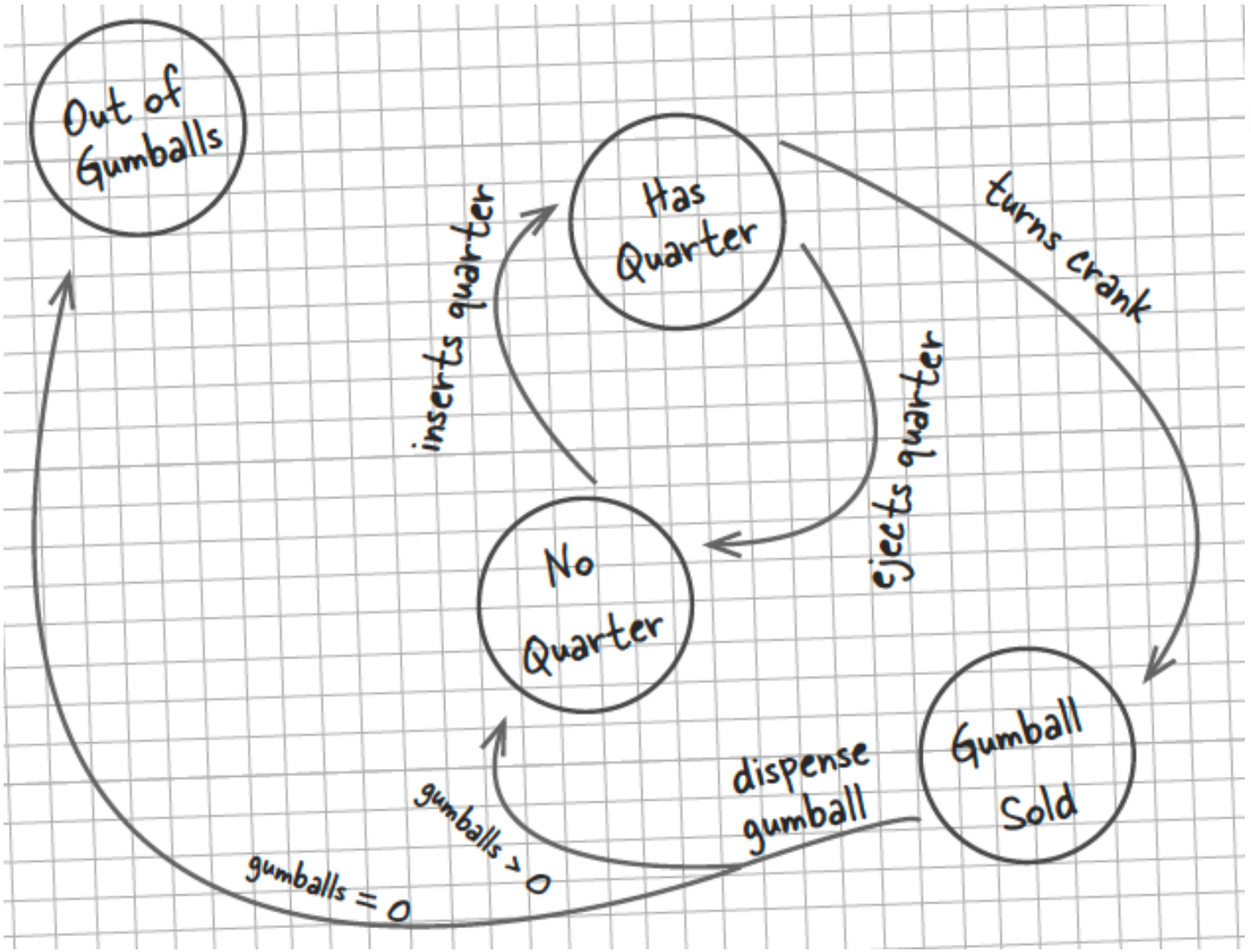
The State Pattern

Gumball machines have gone high tech

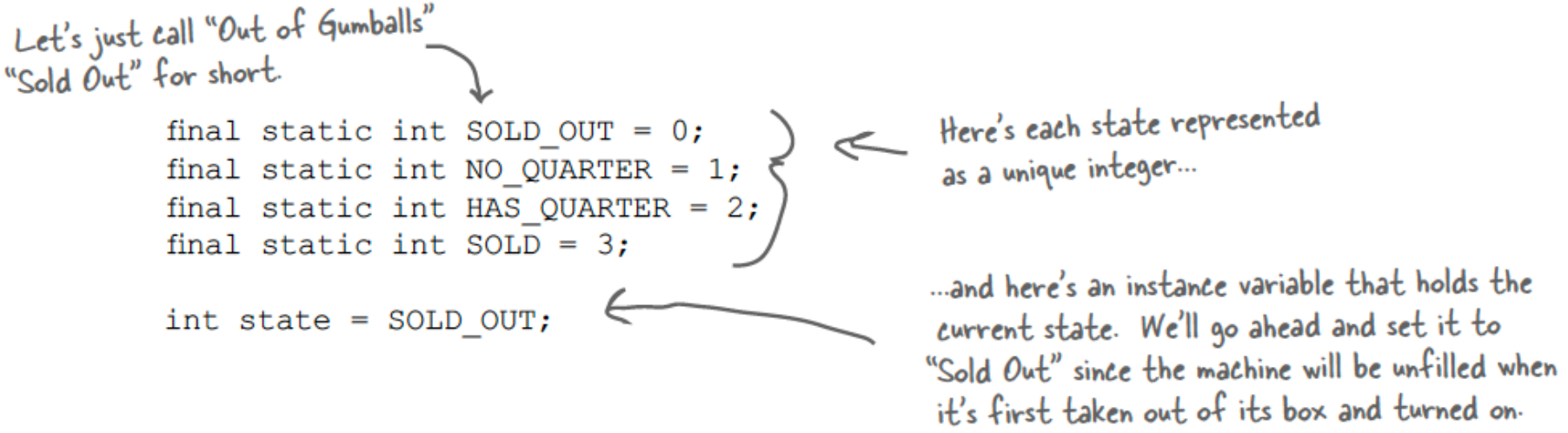


State machine

- gather up your states:

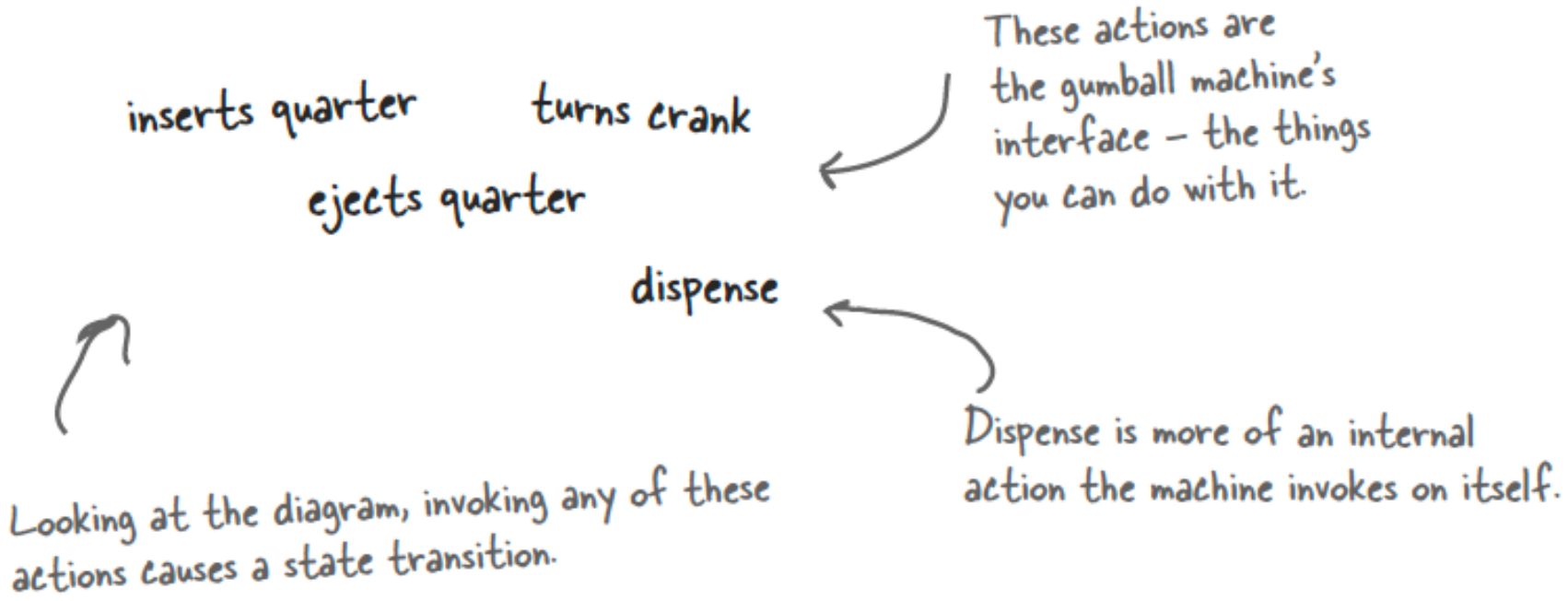
values for each of the states

- create an instance variable to hold the current state, and define values for each of the states:



gather up actions

- gather up all the actions that can happen in the system:



create a class that acts as the state machine

- For each action, we create a method that uses conditional statements to determine what behavior is appropriate in each state. For instance, for the insert quarter action:

```
public void insertQuarter() {

    if (state == HAS_QUARTER) {

        System.out.println("You can't insert another quarter");

    } else if (state == SOLD_OUT) {

        System.out.println("You can't insert a quarter, the machine is sold out");

    } else if (state == SOLD) {

        System.out.println("Please wait, we're already giving you a gumball");

    } else if (state == NO_QUARTER) {

        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");

    }
}
```

Each possible state is checked with a conditional statement...

...and exhibits the appropriate behavior for each possible state...

a common technique

- Here we're talking about a common technique: modeling state within an object by creating an instance variable to hold the state values and writing conditional code within our methods to handle the various states.

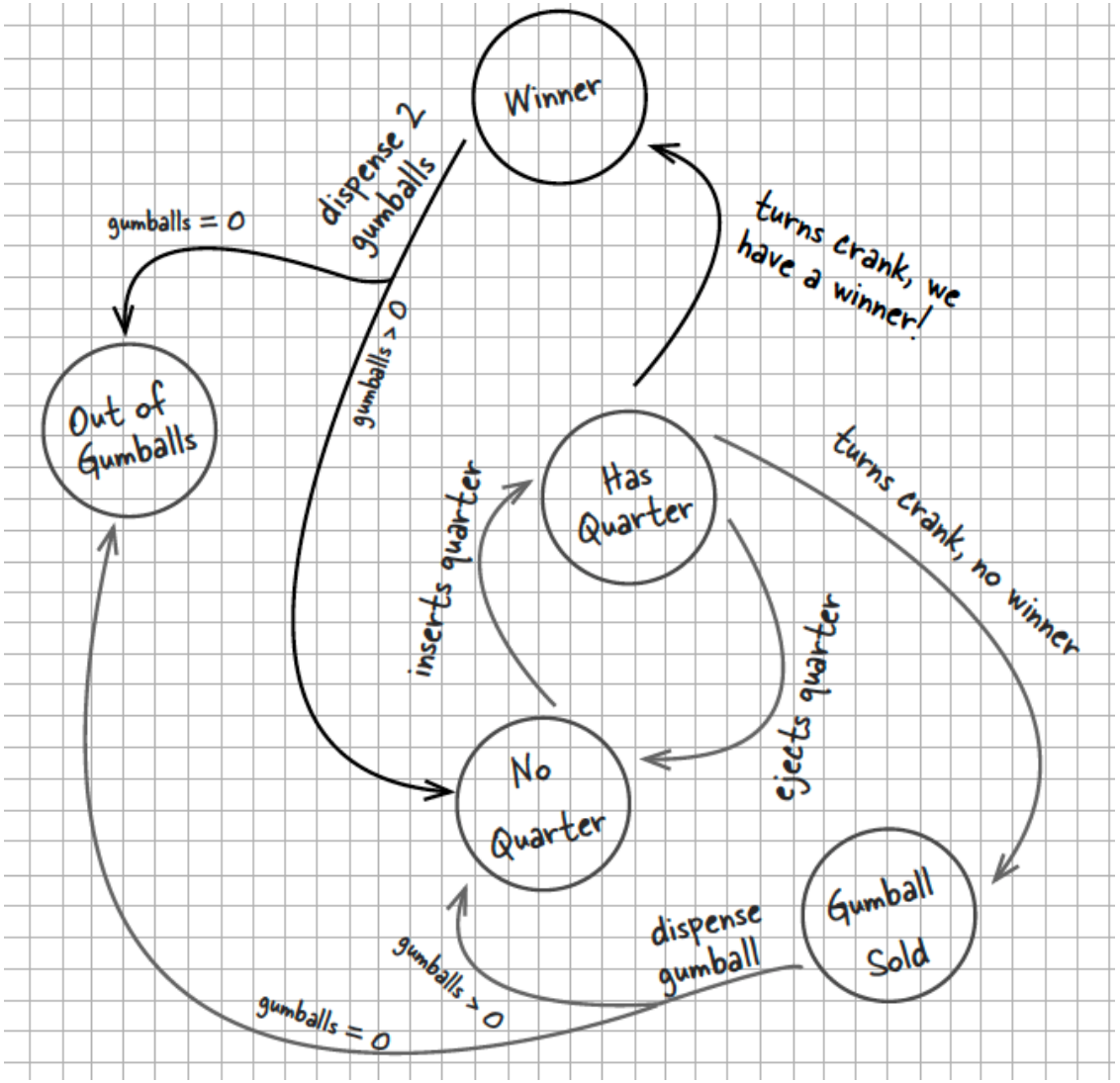
A change request

- We think that by turning "gumball buying" into a game we can significantly increase our sales. We're going to put one of these stickers on every machine.

Design puzzle

- Draw a state diagram for a Gumball Machine that handles the 1 in 10 contest. In this contest, 10% of the time the Sold state leads to two balls being released, not one.

New design


Winner
dispense 2 gumballs
gumballs = 0
gumballs > 0
turns crank, we have a winner!
Out of Gumballs
Has Quarter
turns crank, no winner
inserts quarter
ejects quarter
No Quarter
Gumball Sold
dispense gumball
gumballs > 0
gumballs = 0

The messy STATE of things

- Just because you've written your gumball machine using a well-thought out methodology doesn't mean it's going to be easy to extend.

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;


public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

public void dispense() {
    // dispense code here
}
```

First, you'd have to add a new WINNER state here. That isn't too bad...

... but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.

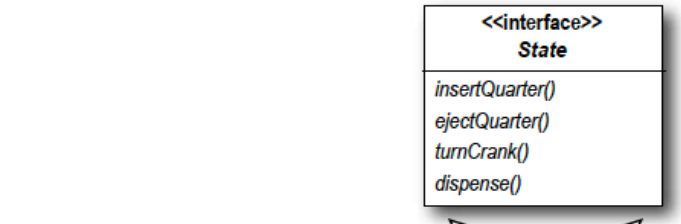
Problems

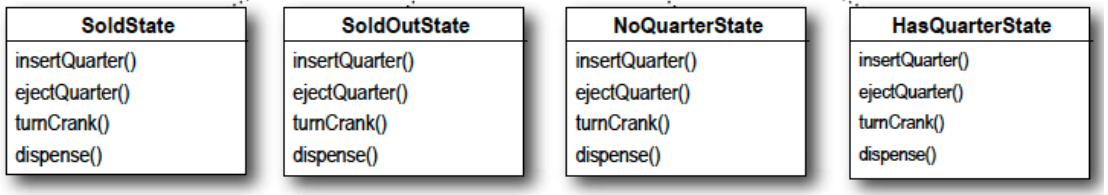
- This design isn't even very object-oriented.
- State transitions aren't explicit; they are buried in the middle of a bunch of conditional statements.
- We haven't encapsulated anything that varies here.
- This code certainly isn't adhering to the Open Closed Principle.
- Further additions are likely to cause bugs in working code.

The new design

- Instead of maintaining our existing code, we're going to rework it to encapsulate state objects in their own classes and then delegate to the current state when an action occurs.
  - First, we're going to define a State interface that contains a method for every action in the Gumball Machine.
  - Then we're going to implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.
  - Finally, we're going to get rid of all of our conditional code and instead delegate to the state class to do the work for us.

Here's the interface for all states. The methods map directly to actions that could happen to the Gumball Machine (these are the same methods as in the previous code).



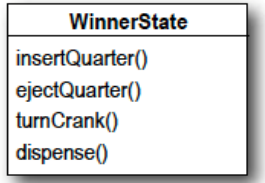


To figure out what states we need, we look at our previous code...

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
```

... and we map each state directly to a class.

Don't forget, we need a new "winner" state too that implements the state interface. We'll come back to this after we reimplement the first version of the Gumball Machine.



Implementing the State classes

First we need to implement the State interface.

We get passed a reference to the Gumball Machine through the constructor. We're just going to stash this in an instance variable.

```
public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }

    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
     }

    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

If someone inserts a quarter, we print a message saying the quarter was accepted and then change the machine's state to the HasQuarterState.

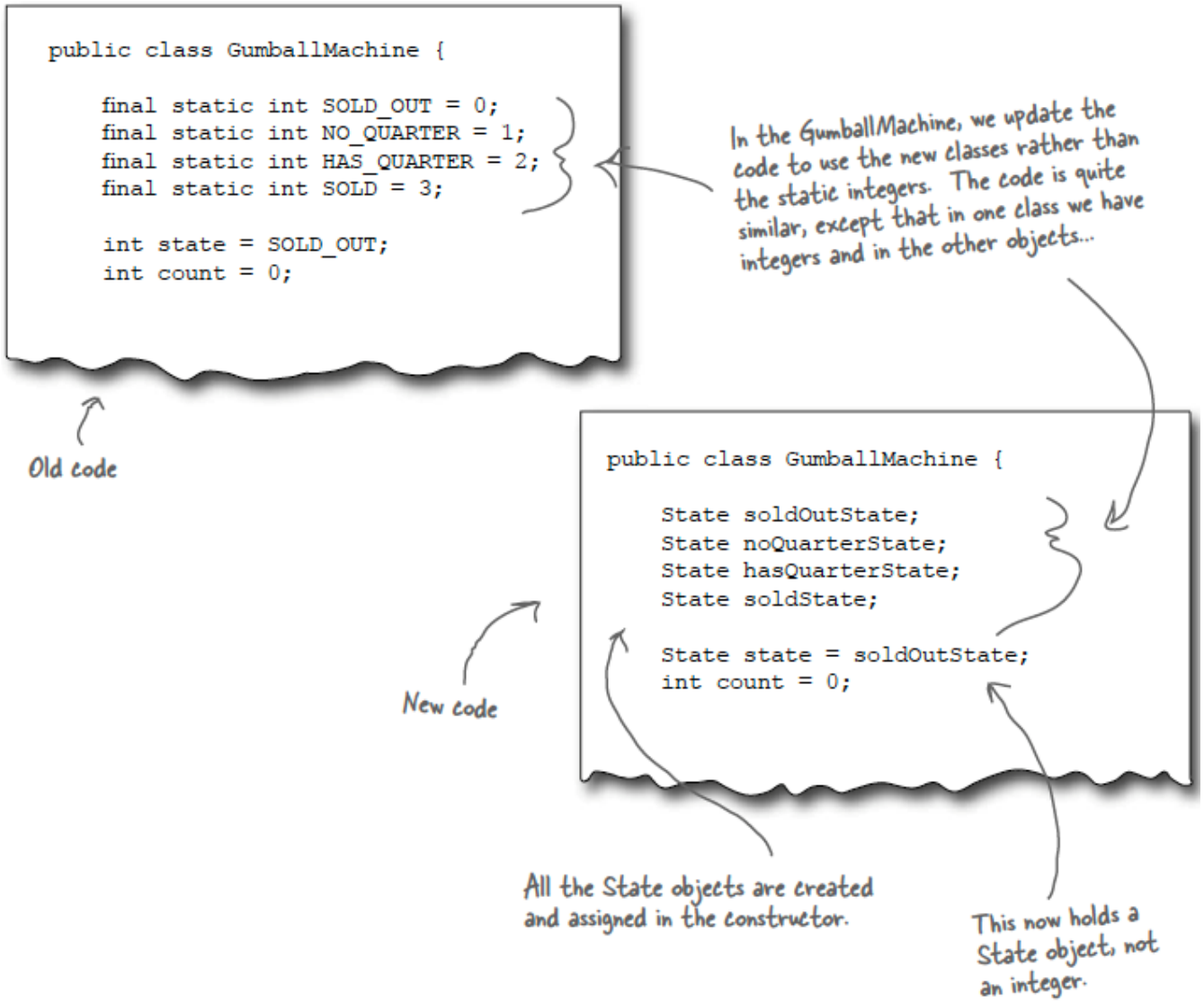
You'll see how these work in just a sec...

You can't get money back if you never gave it to us!

And, you can't get a gumball if you don't pay us.

We can't be dispensing gumballs without payment.

Reworking the Gumball Machine



```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        }
    }

    public void insertQuarter() {
        state.insertQuarter();
    }

    public void ejectQuarter() {
        state.ejectQuarter();
    }
```

Here are all the States again...

...and the State instance variable.

The count instance variable holds the count of gumballs – initially the machine is empty.

Our constructor takes the initial number of gumballs and stores it in an instance variable.

It also creates the State instances, one of each.

If there are more than 0 gumballs we set the state to the NoQuarterState.

Now for the actions. These are VERY EASY to implement now. We just delegate to the current state.

```
    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }

    void releaseBall() {
        System.out.println("A gumball comes rolling out the slot...");
        if (count != 0) {
            count = count - 1;
        }
    }

    // More methods here including getters for each State...
}
```

Note that we don't need an action method for dispense() in GumballMachine because it's just an internal action; a user can't ask the machine to dispense directly. But we do call dispense() on the State object from the turnCrank() method.

This method allows other objects (like our State objects) to transition the machine to a different state.

The machine supports a releaseBall() helper method that releases the ball and decrements the count instance variable.

This includes methods like getNoQuarterState() for getting each state object, and getCount() for getting the gumball count.

```
public class HasQuarterState implements State {
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }
    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

When the state is instantiated we pass it a reference to the GumballMachine. This is used to transition the machine to a different state.

An inappropriate action for this state.

Return the customer's quarter and transition back to the NoQuarterState.

When the crank is turned we transition the machine to the SoldState state by calling its setState() method and passing it the SoldState object. The SoldState object is retrieved by the getSoldState() getter method (there is one of these getter methods for each state).

Another inappropriate action for this state.

```
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.

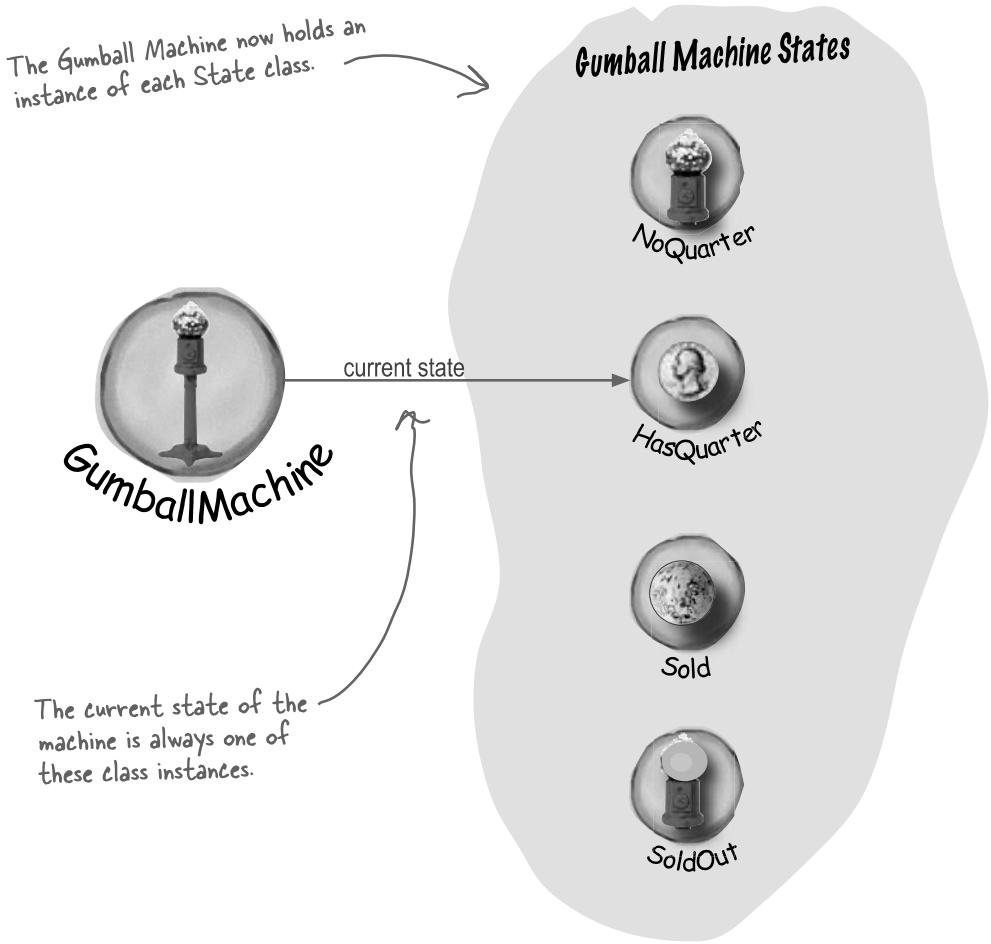
Then we ask the machine what the gumball count is, and either transition to the NoQuarterState or the SoldOutState.

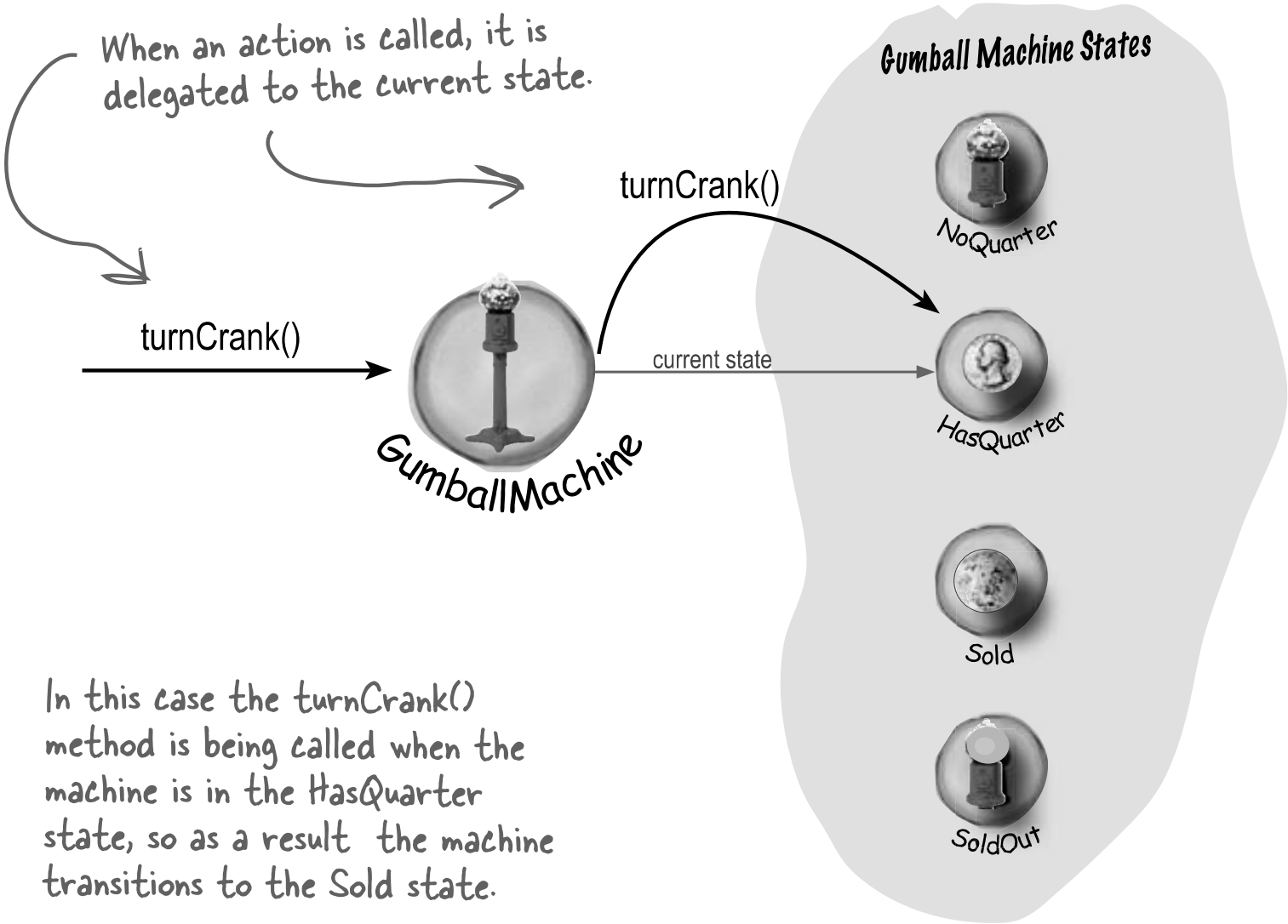
```
public class SoldOutState implements [          ] {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {



    }

    public void insertQuarter() {



    }

    public void ejectQuarter() {



    }

    public void turnCrank() {



    }

    public void dispense() {



    }

}
```

In the Sold Out state, we really can't do anything until someone refills the Gumball Machine.

```
public class SoldOutState implements State {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert a quarter, the machine is sold out");
    }

    public void ejectQuarter() {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }

    public void turnCrank() {
        System.out.println("You turned, but there are no gumballs");
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }

}
```

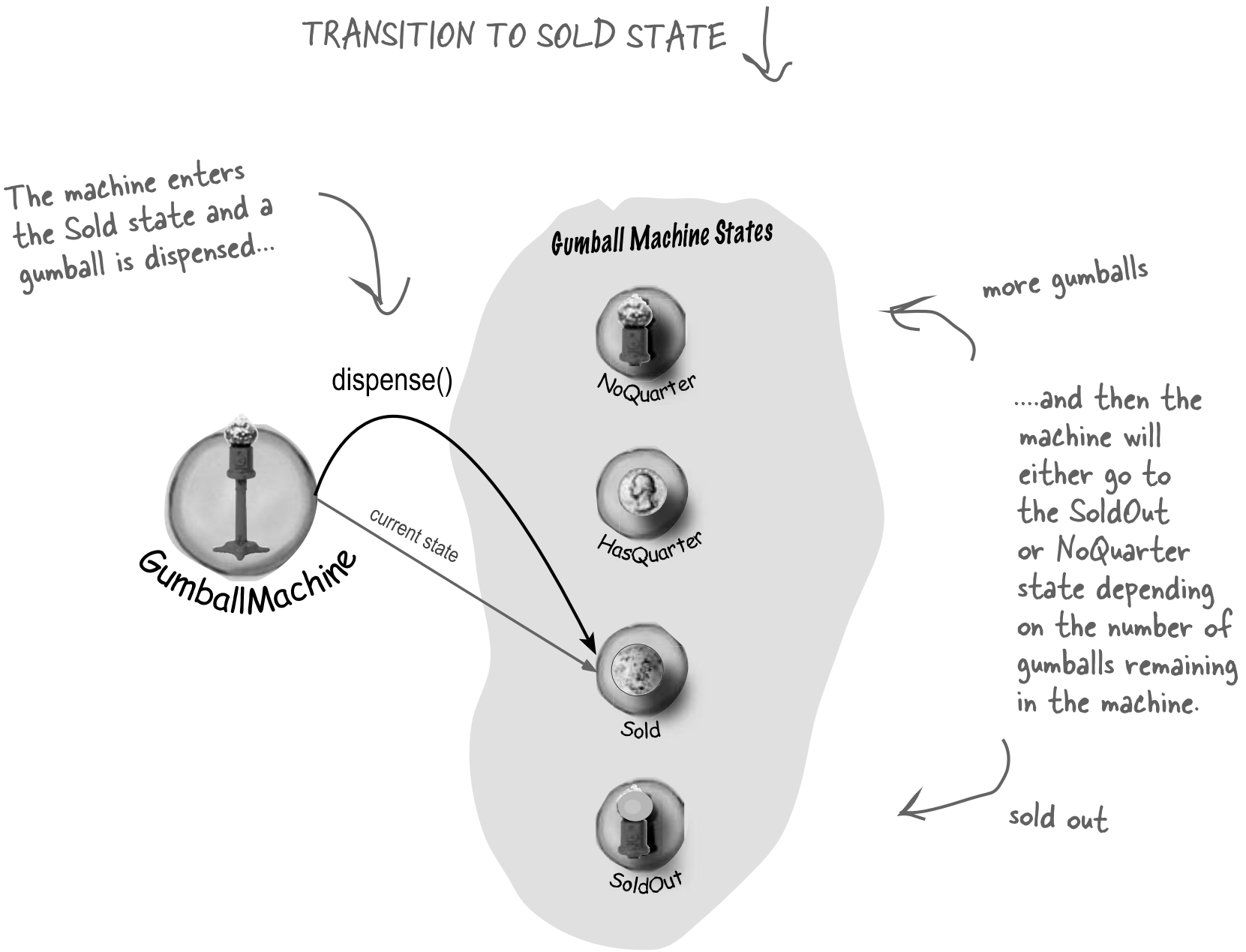
What we’ve done so far…





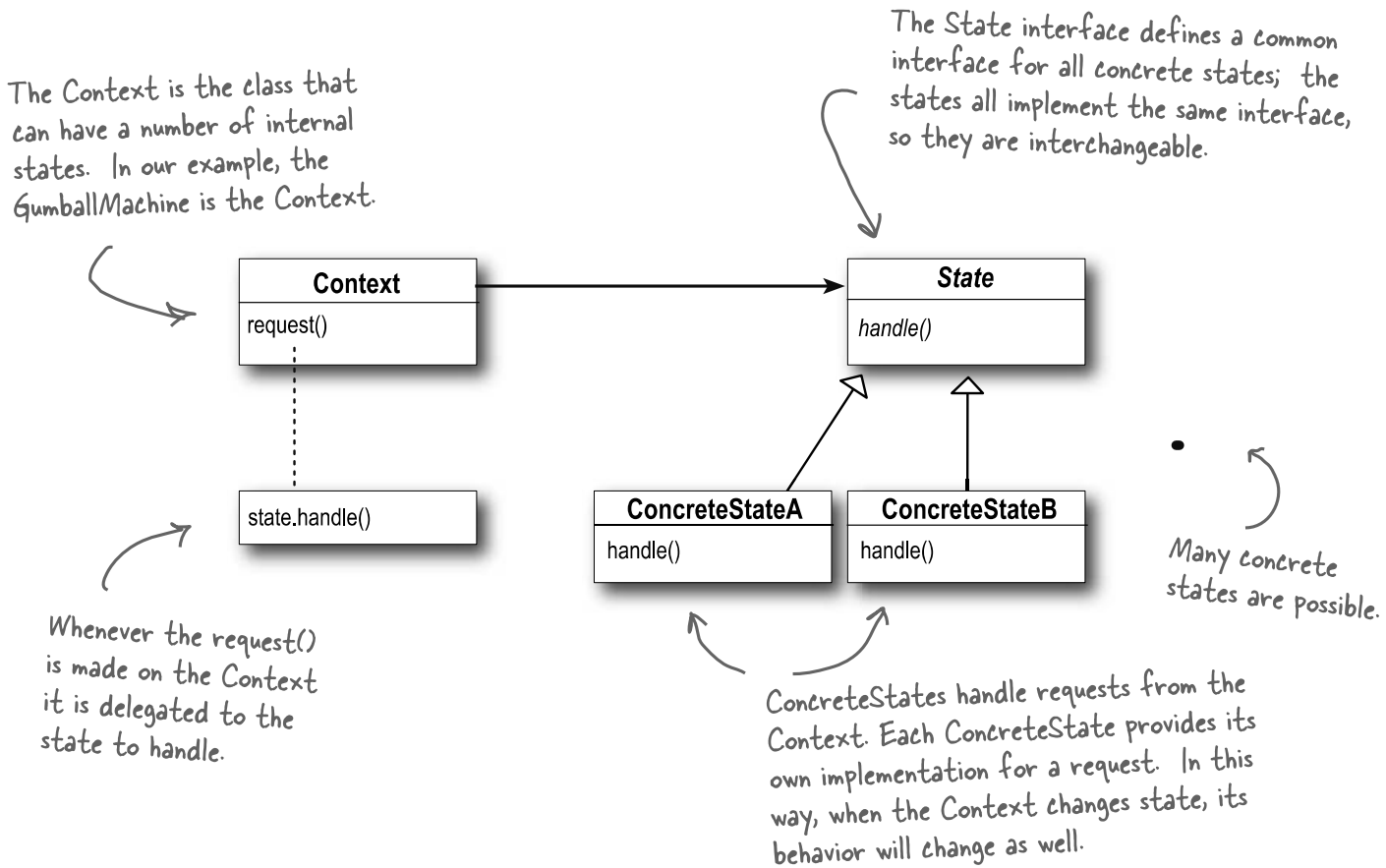
In this case the turnCrank() method is being called when the machine is in the HasQuarter state, so as a result the machine transitions to the Sold state.

TRANSITION TO SOLD STATE



The State Pattern defined

- The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.



Same with Strategy pattern?

- Yes, the class diagrams are essentially the same, but the two patterns differ in their intent.
- With the State Pattern, we have a set of behaviors encapsulated in state objects; at any time the context is delegating to one of those state. Over time, the current state changes across the set of state objects to reflect the internal state of the context, so the context's behavior changes over time as well. The client usually knows very little, if anything, about the state objects.

- With Strategy, the client usually specifies the strategy object that the context is composed with. Now, while the pattern provides the flexibility to change the strategy object at runtime, often there is a strategy that is most appropriate for a context object.

- In general, think of the Strategy Pattern as a flexible alternative to subclassing; if you use inheritance to define the behavior of a class, then you're stuck with that behavior even if you need to change it. With Strategy you can change the behavior by composing with a different object.
- Think of the State Pattern as an alternative to putting lots of conditionals in your context; by encapsulating the behavior within state objects, you can simply change the state object in context to change its behavior.

Gumball 1 in 10 game

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State winnerState;

    State state = soldOutState;
    int count = 0;

    // methods here
}
```

All you need to add here is the new WinnerState and initialize it in the constructor.

Don't forget you also have to add a getter method for WinnerState too.

```
public class WinnerState implements State {

    // instance variables and constructor

    // insertQuarter error message

    // ejectQuarter error message

    // turnCrank error message

    public void dispense() {
        System.out.println("YOU'RE A WINNER! You get two gumballs for your quarter");
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() == 0) {
            gumballMachine.setState(gumballMachine.getSoldOutState());
        } else {
            gumballMachine.releaseBall();
            if (gumballMachine.getCount() > 0) {
                gumballMachine.setState(gumballMachine.getNoQuarterState());
            } else {
                System.out.println("Oops, out of gumballs!");
                gumballMachine.setState(gumballMachine.getSoldOutState());
            }
        }
    }
}
```

Just like SoldState.

Here we release two gumballs and then either go to the NoQuarterState or the SoldOutState.

As long as we have a second gumball we release it.

```
public class HasQuarterState implements State {
    Random randomWinner = new Random(System.currentTimeMillis());
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        int winner = randomWinner.nextInt(10);
        if ((winner == 0) && (gumballMachine.getCount() > 1)) {
            gumballMachine.setState(gumballMachine.getWinnerState());
        } else {
            gumballMachine.setState(gumballMachine.getSoldState());
        }
    }
    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

First we add a random number generator to generate the 10% chance of winning...

...then we determine if this customer won.

If they won, and there's enough gumballs left for them to get two, we go to the WinnerState; otherwise, we go to the SoldState (just like we always did).

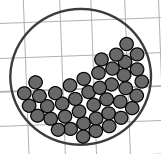
Test drive

This code really hasn't changed at all; we just shortened it a bit.

Once, again, start with a gumball machine with 5 gumballs.

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

We want to get a winning state, so we just keep pumping in those quarters and turning the crank. We print out the state of the gumball machine every so often...

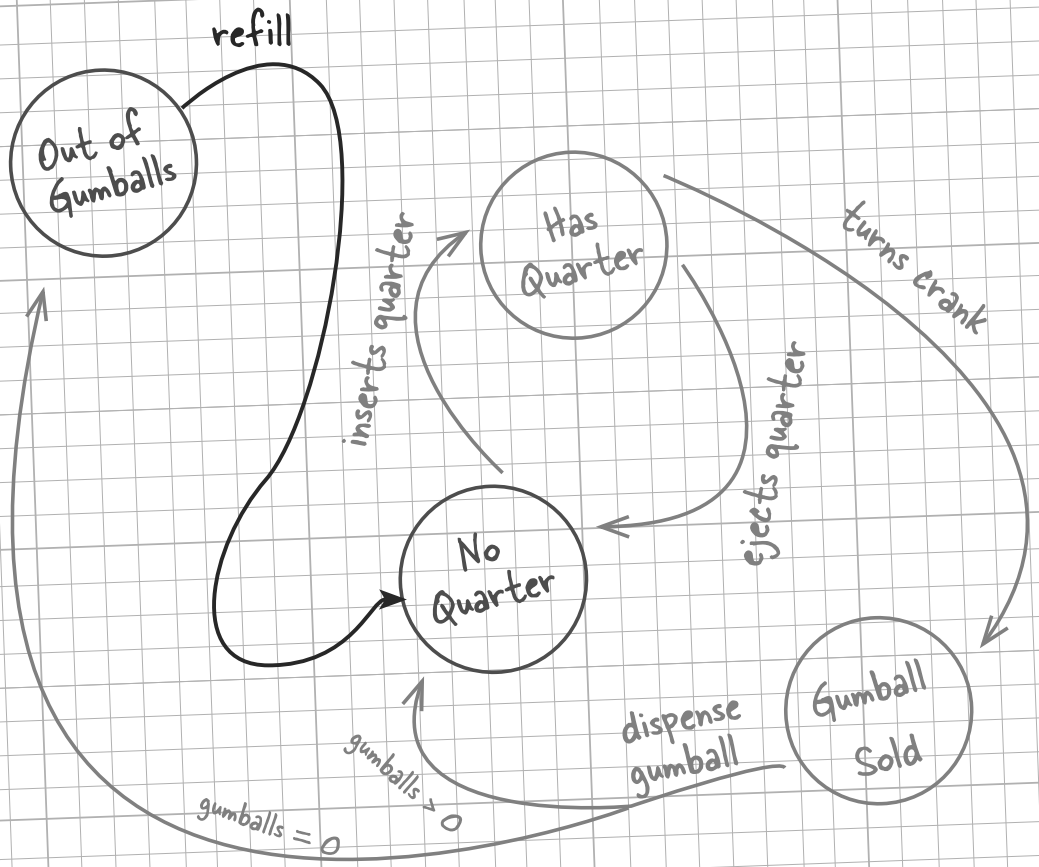


Mighty Gumball, Inc.

Where the Gumball Machine is Never Half Empty

There's one transition we forgot to put in the original spec... we need a way to refill the gumball machine when it's out of gumballs! Here's the new diagram – can you implement it for us? You did such a good job on the rest of the gumball machine we have no doubt you can add this in a jiffy!

– The Mighty Gumball Engineers



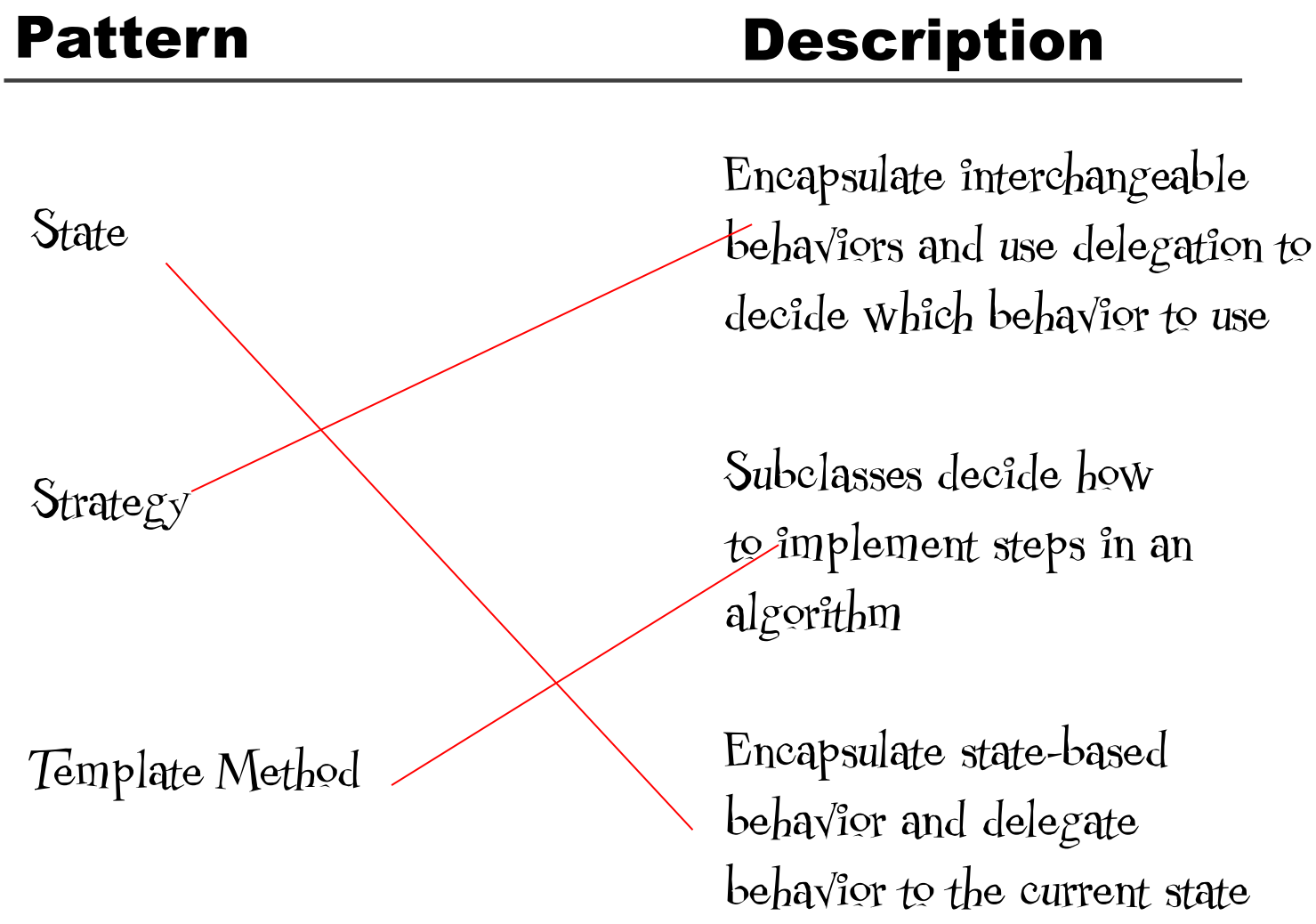
Exercise

- We need you to write the refill() method for the Gumball machine. It has one argument – the number of gumballs you're adding to the machine – and should update the gumball machine count and reset the machine's state.

We need you to write the refill() method for the Gumball machine. It has one argument, the number of gumballs you're adding to the machine, and should update the gumball machine count and reset the machine's state.

```
void refill(int count) {
    this.count = count;
    state = noQuarterState;
}
```

Match each pattern with its description:



Reviews

- 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.