State Pattern

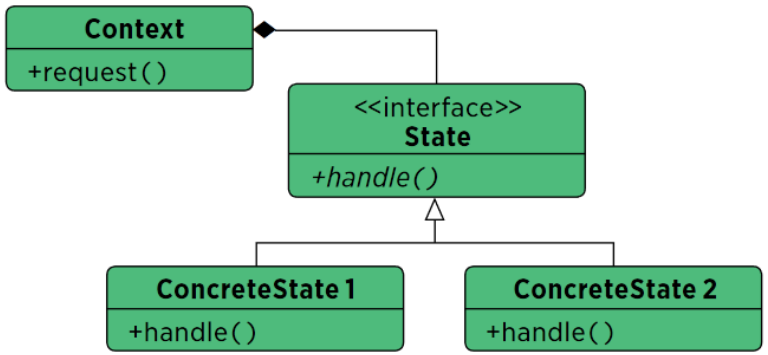
For the Complete Code, See the "Official" Head-First Design Patterns GitHub Repo:

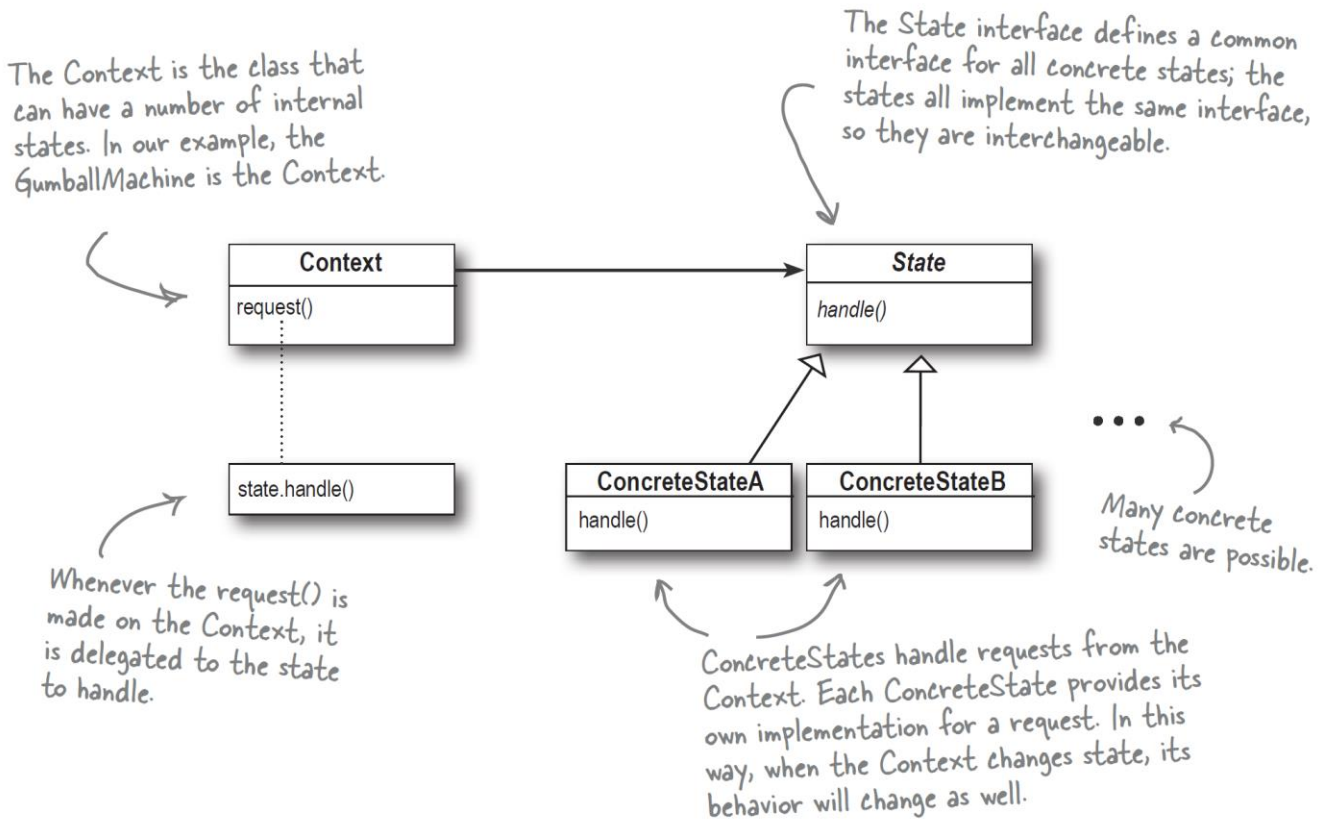
https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/

And the course SVN repo:

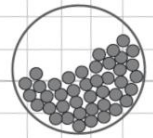
svn://cosc436.net:65436/Examples/trunk

STATE — Object Behavioral





Purpose	Ties object circumstances to its behavior, allowing the object to behave in different ways based upon its internal state.
Use When	• The behavior of an object should be influenced by its state. • Complex conditions tie object behavior to its state. • Transitions between states need to be explicit.
Example	An email object can have various states, all of which will change how the object handles different functions. If the state is “not sent” then the call to send() is going to send the message while a call to recallMessage() will either throw an error or do nothing. However, if the state is “sent” then the call to send() would either throw an error or do nothing while the call to recallMessage() would attempt to send a recall notification to recipients. To avoid conditional statements in most or all methods there would be multiple state objects that handle the implementation with respect to their particular state. The calls within the Email object would then be delegated down to the appropriate state object for handling.

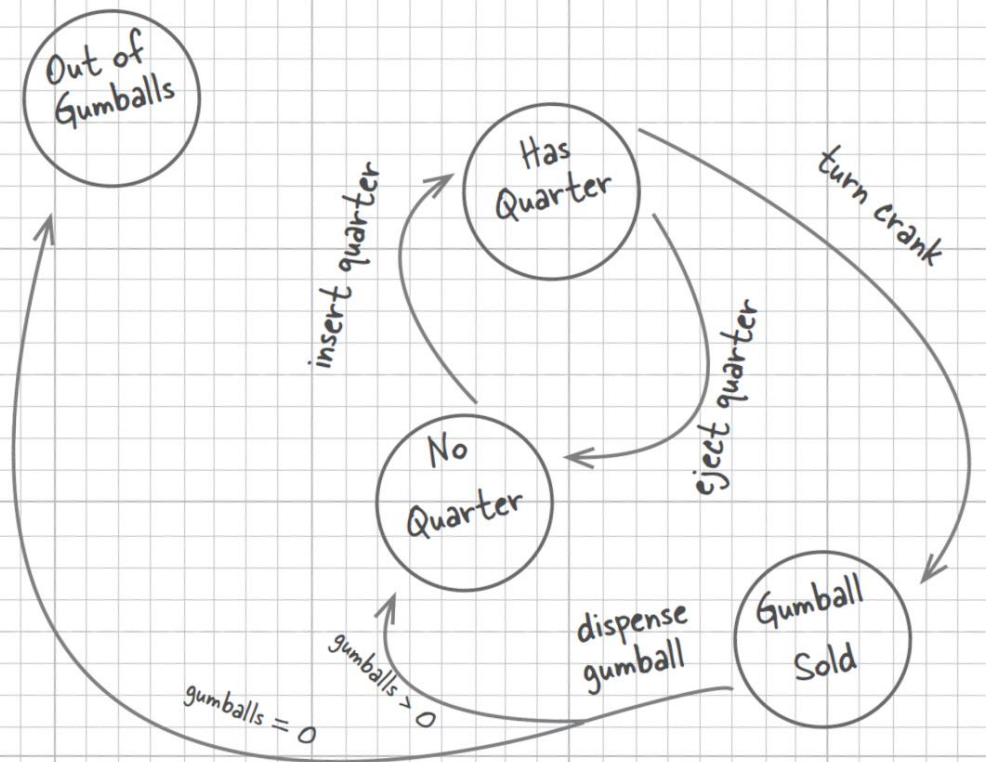


Mighty Gumball, Inc.

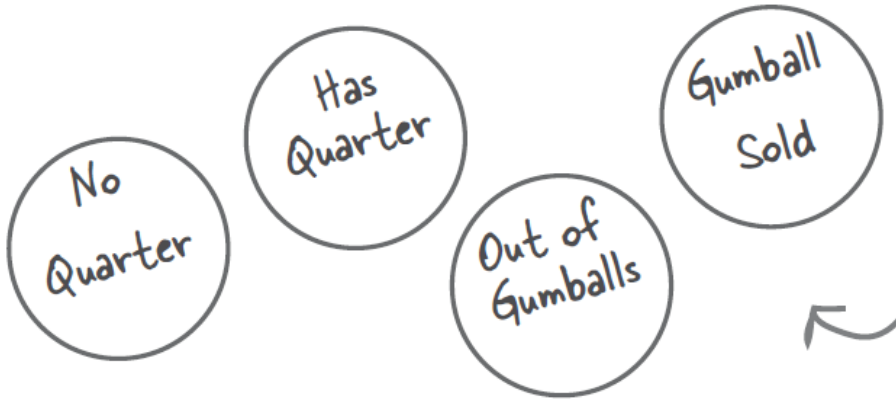
Where the Gumball Machine is Never Half Empty

Here's the way we think the gumball machine controller needs to work. We're hoping you can implement this in Java for us! We may be adding more behavior in the future, so you need to keep the design as flexible and maintainable as possible!

– Mighty Gumball Engineers



What we know…



But perhaps <u>not</u> the best way of representing this…

Let's just call "Out of Gumballs" "Sold Out" for short.

```
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;
```

Here's each state represented as a unique integer…

…and here's an instance variable that holds the current state. We'll go ahead and set it to "Sold Out" since the machine will be unfilled when it's first taken out of its box and turned on.

Since user actions drive state changes, we may be tempted to do this, for example!

```
public void insertQuarter() {

    if (state == HAS_QUARTER) {

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

    } else if (state == NO_QUARTER) {

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

    }
}
```

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

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

...but can also transition to other states, just as depicted in the diagram.

Already starting to look messy!!!

```
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;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }

    public void insertQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("You can't insert another quarter");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a 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");
        }
    }
```

Here are the four states; they match the states in Mighty Gumball's state diagram.

Here's the instance variable that is going to keep track of the current state we're in. We start in the SOLD_OUT state.

We have a second instance variable that keeps track of the number of gumballs in the machine.

The constructor takes an initial inventory of gumballs. If the inventory isn't zero, the machine enters state NO_QUARTER, meaning it is waiting for someone to insert a quarter; otherwise, it stays in the SOLD_OUT state.

Now we start implementing the actions as methods....

When a quarter is inserted...

...if a quarter is already inserted, we tell the customer...

...otherwise, we accept the quarter and transition to the HAS_QUARTER state.

If the customer just bought a gumball, he needs to wait until the transaction is complete before inserting another quarter.

And if the machine is sold out, we reject the quarter.

...and even more messy!!!

```
public void ejectQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("Quarter returned");
        state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
    } else if (state == SOLD) {
        System.out.println("Sorry, you already turned the crank");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}
```

Now, if the customer tries to remove the quarter...

...if there is a quarter, we return it and go back to the NO_QUARTER state...

...otherwise, if there isn't one we can't give it back.

You can't eject if the machine is sold out, it doesn't accept quarters!

If the customer just turned the crank, we can't give a refund; he already has the gumball!

```
public void turnCrank() {
    if (state == SOLD) {
        System.out.println("Turning twice doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
        System.out.println("You turned but there's no quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
        System.out.println("You turned...");
        state = SOLD;
        dispense();
    }
}
```

The customer tries to turn the crank...

Someone's trying to cheat the machine.

We need a quarter first.

We can't deliver gumballs; there are none.

Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.

...what have we done!?!?

Called to dispense a gumball.

```
public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0) {
            System.out.println("Oops, out of gumballs!");
            state = SOLD_OUT;
        } else {
            state = NO_QUARTER;
        }
    } else if (state == NO_QUARTER) {
        System.out.println("You need to pay first");
    } else if (state == SOLD_OUT) {
        System.out.println("No gumball dispensed");
    } else if (state == HAS_QUARTER) {
        System.out.println("You need to turn the crank");
    }
}

    // other methods here like toString() and refill()
}
```

We're in the SOLD state; give 'em a gumball!

Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.

None of these should ever happen, but if they do, we give 'em an error, not a gumball.

...Rough Outline..

```
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
}
```

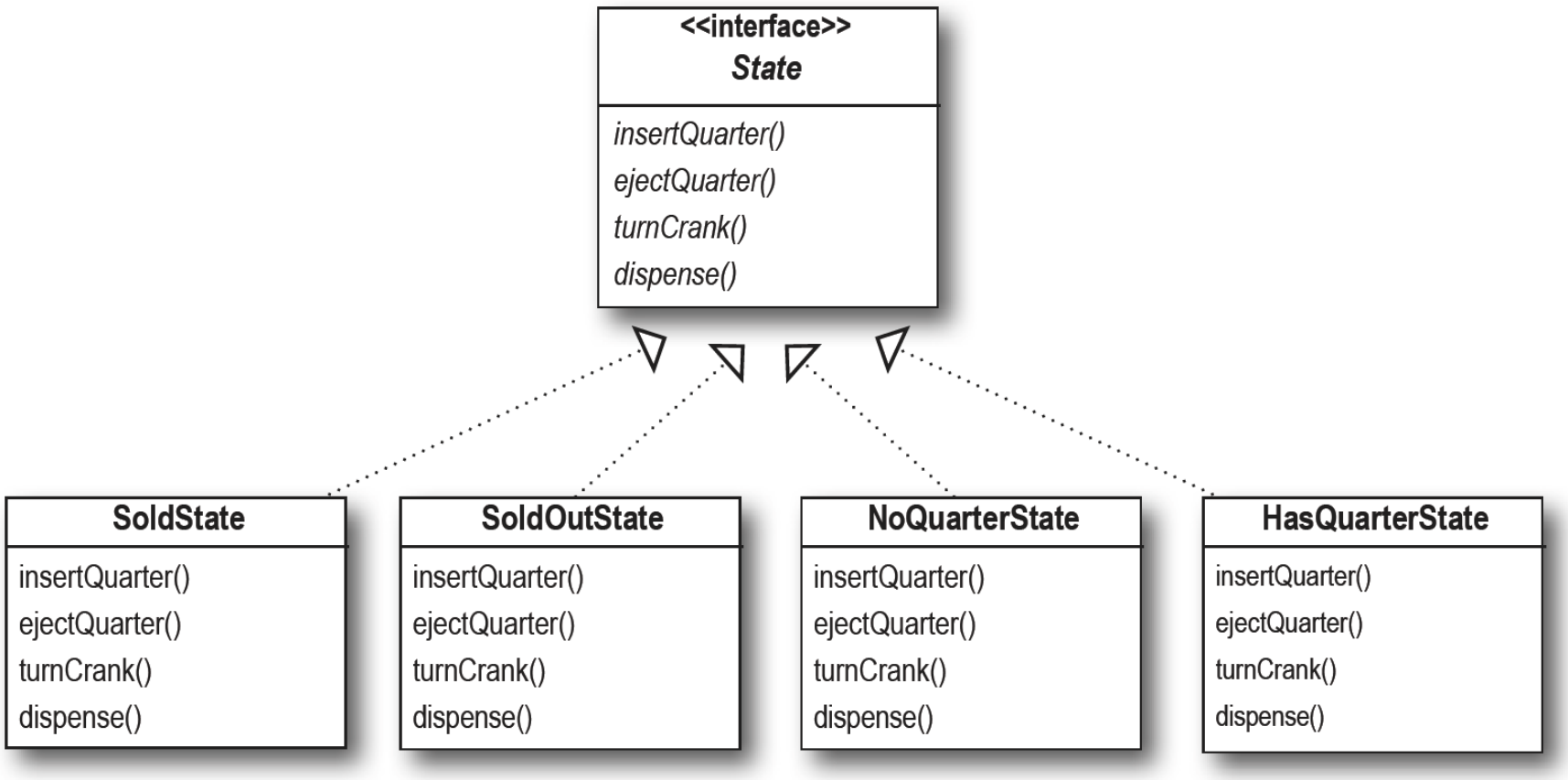
First, you'd have to add a new WINNER state here. That isn't too bad...

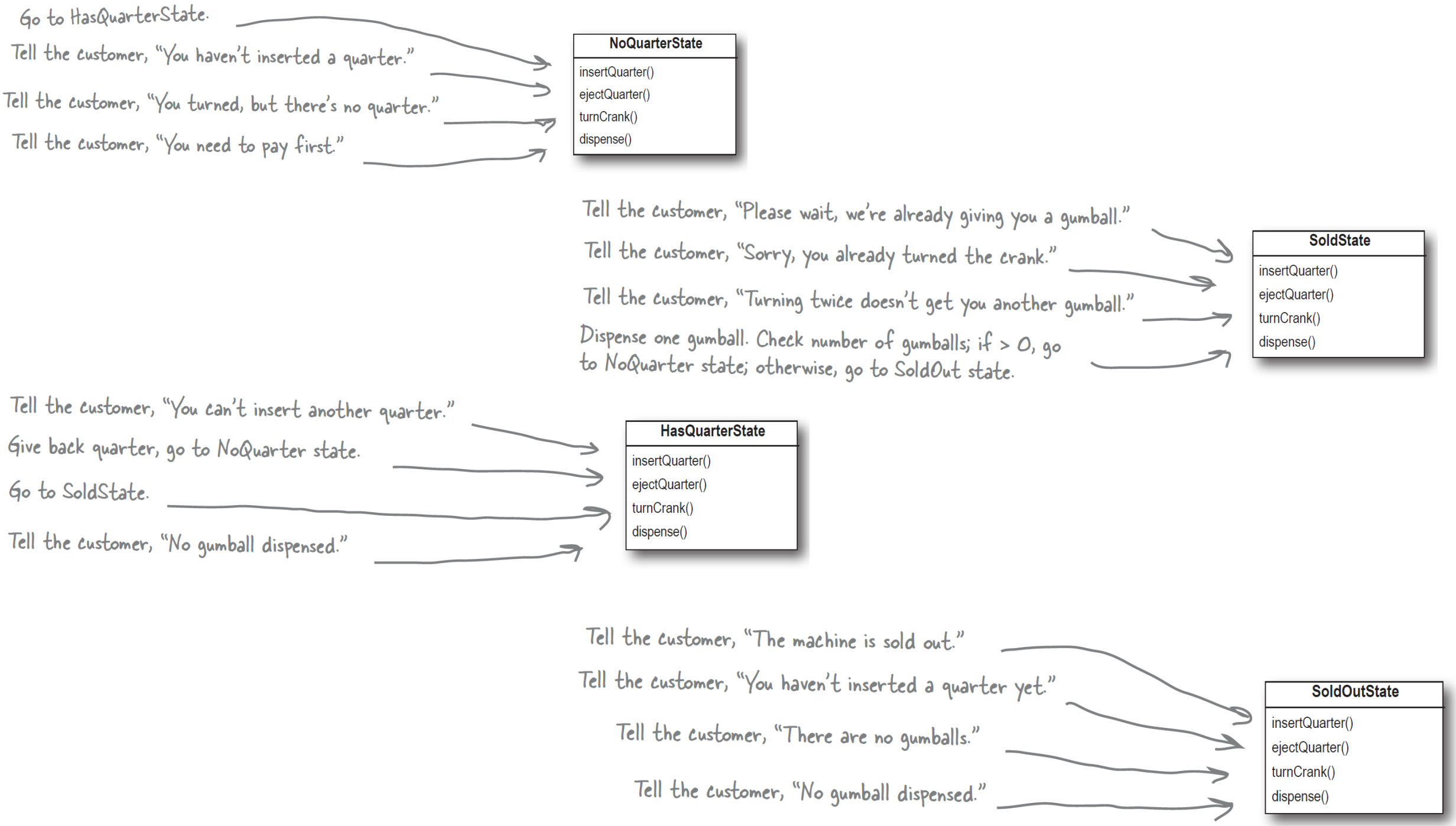
...but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

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.

Of course, there is a better way!

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).




Go to HasQuarterState.
Tell the customer, "You haven't inserted a quarter."
Tell the customer, "You turned, but there's no quarter."
Tell the customer, "You need to pay first."
NoQuarterState
insertQuarter()
ejectQuarter()
turnCrank()
dispense()
Tell the customer, "Please wait, we're already giving you a gumball."
Tell the customer, "Sorry, you already turned the crank."
Tell the customer, "Turning twice doesn't get you another gumball."
Dispense one gumball. Check number of gumballs; if > 0, go to NoQuarter state; otherwise, go to SoldOut state.
SoldState
insertQuarter()
ejectQuarter()
turnCrank()
dispense()
Tell the customer, "You can't insert another quarter."
Give back quarter, go to NoQuarter state.
Go to SoldState.
Tell the customer, "No gumball dispensed."
HasQuarterState
insertQuarter()
ejectQuarter()
turnCrank()
dispense()
Tell the customer, "The machine is sold out."
Tell the customer, "You haven't inserted a quarter yet."
Tell the customer, "There are no gumballs."
Tell the customer, "No gumball dispensed."
SoldOutState
insertQuarter()
ejectQuarter()
turnCrank()
dispense()

An Example of one "State Class"

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.

Well use meaningful objects instead of arbitrary constants.

```
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;
```

Old code

In the GumballMachine, we update the code to use the new classes rather than the static integers. The code is quite similar, except that in one class we have integers and in the other objects...

New code

```
public class GumballMachine {

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

    State state = soldOutState;
    int count = 0;
```

All the State objects are created and assigned in the constructor.

This now holds a State object, not an integer.

Construct our Object

```
public class GumballMachine {

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

    State state;
    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;
        } else {
            state = soldOutState;
        }
    }
```

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; otherwise, we start in the SoldOutState

~~Altered~~ Altering States.

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

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

Another State Defined.

```
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.

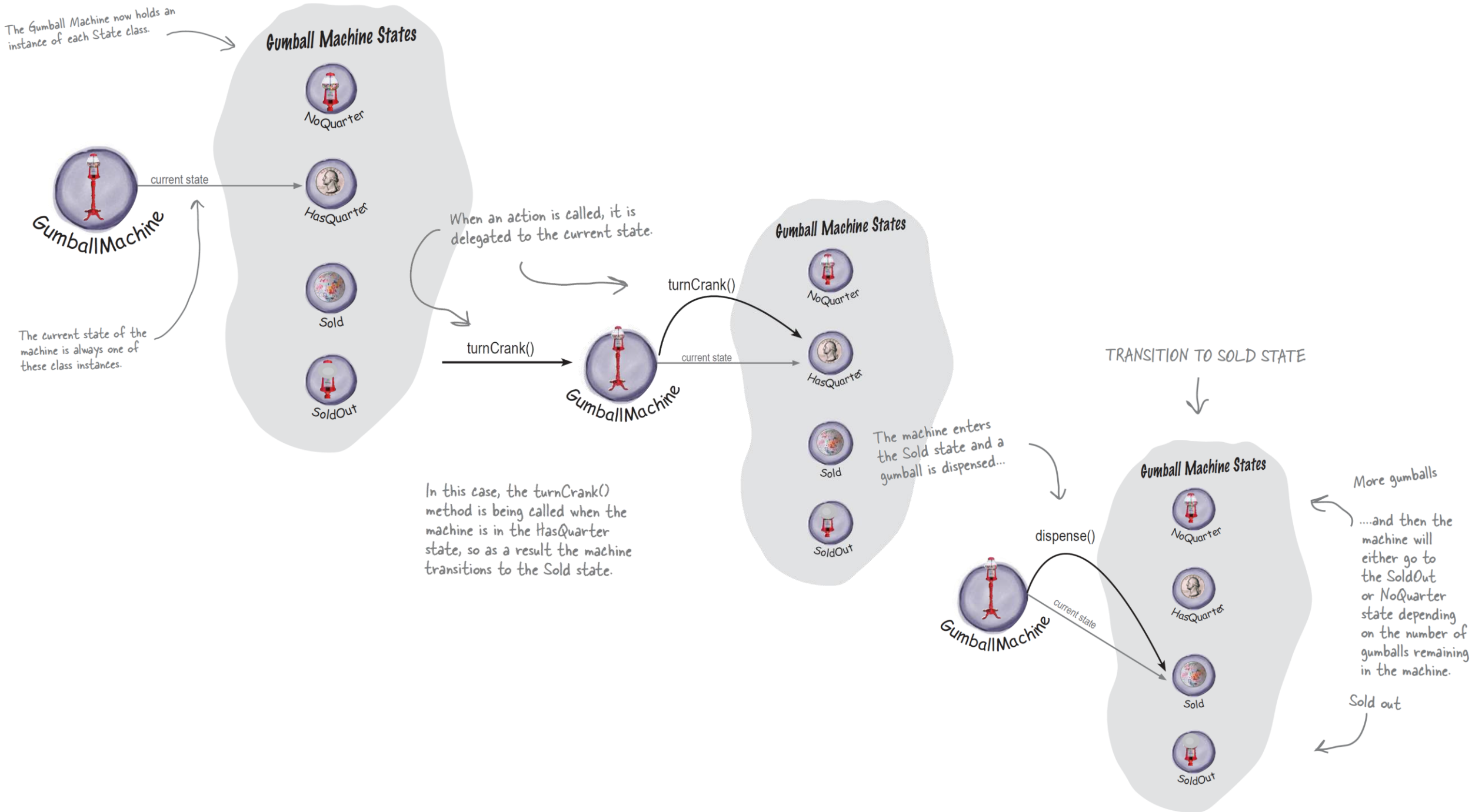
When the crank is turned we transition the machine to the SoldState state by calling its setState() method and passing it the SoldState object. The SoldState object is retrieved by the getSoldState() getter method (there is one of these getter methods for each state).

Another inappropriate action for this state.

Let's take a look at what we've done so far...

For starters, you now have a Gumball Machine implementation that is *structurally* quite different from your first version, and yet *functionally it is exactly the same*. By structurally changing the implemention, you've:

- Localized the behavior of each state into its own class.
- Removed all the troublesome if statements that would have been difficult to maintain.
- Closed each state for modification, and yet left the Gumball Machine open to extension by adding new state classes (and we'll do this in a second).
- Created a code base and class structure that maps much more closely to the Mighty Gumball diagram and is easier to read and understand.


The Gumball Machine now holds an instance of each State class.
Gumball Machine States
NoQuarter
HasQuarter
Sold
SoldOut
current state
GumballMachine
The current state of the machine is always one of these class instances.
When an action is called, it is delegated to the current state.
turnCrank()
turnCrank()
current state
GumballMachine
Gumball Machine States
NoQuarter
HasQuarter
Sold
SoldOut
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.
The machine enters the Sold state and a gumball is dispensed...
TRANSITION TO SOLD STATE
dispense()
current state
GumballMachine
Gumball Machine States
NoQuarter
HasQuarter
Sold
SoldOut
More gumballs
....and then the machine will either go to the SoldOut or NoQuarter state depending on the number of gumballs remaining in the machine.
Sold out