Chapter 10. The State Pattern: The State of Things

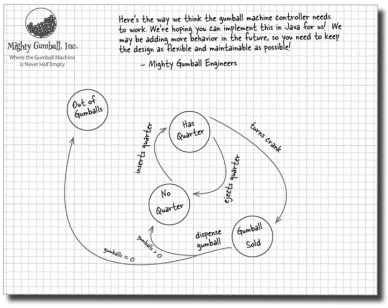
A little known fact: the Strategy and State Patterns were twins separated at birth. As you know, the Strategy Pattern went on to create a wildly successful business around interchangeable algorithms. State, however, took the perhaps more noble path of helping objects to control their behavior by changing their internal state. He's often overheard telling his object clients, "Just repeat after me: I'm good enough, I'm smart enough, and doggonit..."

Jawva Breakers

Java toasters are so '90s. Today people are building Java into *real* devices, like gumball machines. That's right, gumball machines have gone high tech; the major manufacturers have found that by putting CPUs into their machines, they can increase sales, monitor inventory over the network and measure customer satisfaction more accurately.

> **NOTE**
>
> At least that's their story – we think they just got bored with the circa 1800's technology and needed to find a way to make their jobs more exciting.

But these manufacturers are gumball machine experts, not software developers, and they've asked for your help:



Cubicle Conversation

Anne: This diagram looks like a state diagram.

Joe: Right, each of those circles is a state...

Anne: ... and each of the arrows is a state transition.

Frank: Slow down, you two, it's been too long since I studied state diagrams. Can you remind me what they're all about?

Anne: Sure, Frank. Look at the circles; those are states. "No Quarter" is probably the starting state for the gumball machine because it's just sitting there waiting for you to put your quarter in. All states are just different configurations of the machine that behave in a certain way and need some action to take them to another state.

Joe: Right. See, to go to another state, you need to do something like put a quarter in the machine. See the arrow from "No Quarter" to "Has Quarter?"

Frank: Yes...

Joe: That just means that if the gumball machine is in the "No Quarter" state and you put a quarter in, it will change to the "Has Quarter" state. That's the state transition.

Frank: Oh, I see! And if I'm in the "Has Quarter" state, I can turn the crank and change to the "Gumball Sold" state, or eject the quarter and change back to the "No Quarter" state.

Anne: You got it!

Frank: This doesn't look too bad then. We've obviously got four states, and I think we also have four actions: "inserts quarter," "ejects quarter," "turns crank" and "dispense." But... when we dispense, we test for zero or more gumballs in the "Gumball Sold" state, and then either go to the "Out of Gumballs" state or the "No Quarter" state. So we actually have five transitions from one state to another.

Anne: That test for zero or more gumballs also implies we've got to keep track of the number of gumballs too. Any time the machine gives you a gumball, it might be the last one, and if it is, we need to transition to the "Out of Gumballs" state.

Joe: Also, don't forget that you could do nonsensical things, like try to eject the quarter when the gumball machine is in the "No Quarter" state, or insert two quarters.

Frank: Oh, I didn't think of that; we'll have to take care of those too.

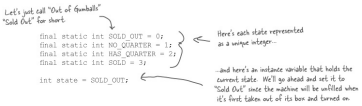
Joe: For every possible action we'll just have to check to see which state we're in and act appropriately. We can do this! Let's start mapping the state diagram to code...

State machines 101

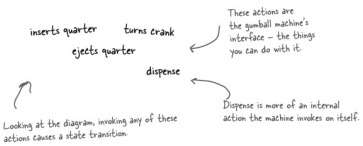
How are we going to get from that state diagram to actual code? Here's a quick introduction to implementing state machines:

1. First, gather up your states:

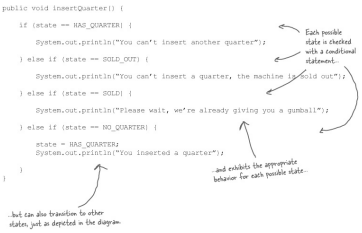
2. Next, create an instance variable to hold the current state, and define values for each of the states:



3. Now we gather up all the actions that can happen in the system:



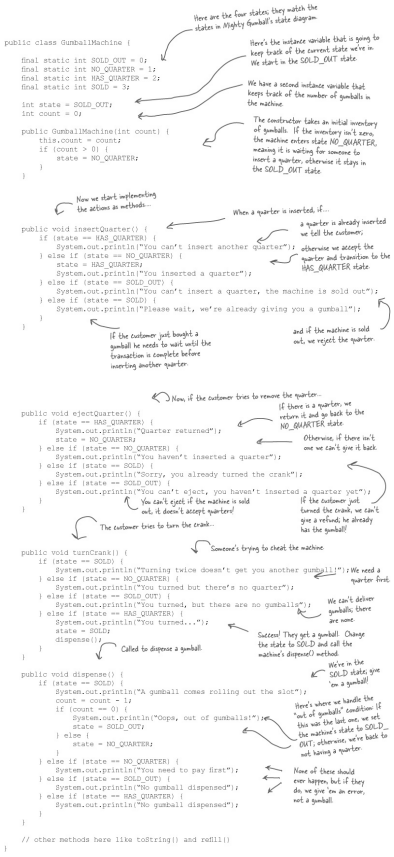
4. Now we create a class that acts as the state machine. For each action, we create a method that uses conditional statements to determine what behavior is appropriate in each state. For instance, for the insert quarter action, we might write a method like this:



With that quick review, let's go implement the Gumball Machine!

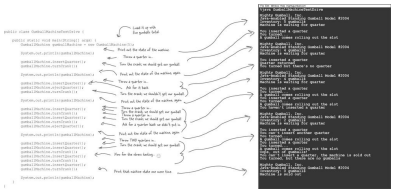
Writing the code

It's time to implement the Gumball Machine. We know we're going to have an instance variable that holds the current state. From there, we just need to handle all the actions, behaviors and state transitions that can happen. For actions, we need to implement inserting a quarter, removing a quarter, turning the crank and dispensing a gumball; we also have the empty gumball condition to implement as well.



In-house testing

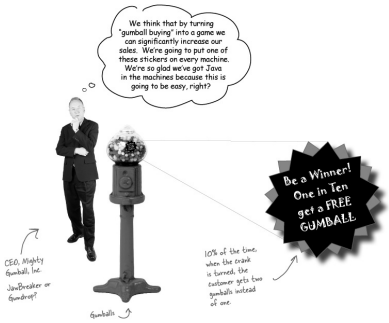
That feels like a nice solid design using a well-thought out methodology, doesn't it? Let's do a little in-house testing before we hand it off to Mighty Gumball to be loaded into their actual gumball machines. Here's our test harness:

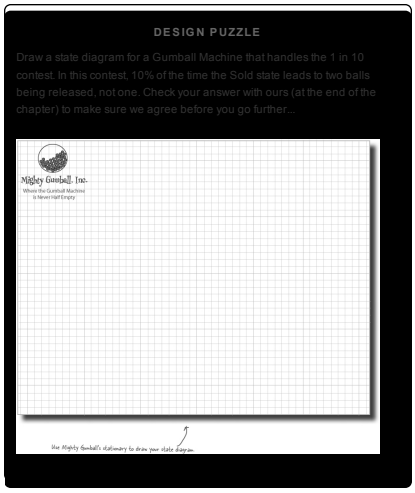


You knew it was coming... a change request!

Mighty Gumball, Inc. has loaded your code into their newest machine and their quality assurance experts are putting it through its paces. So far, everything's looking great from their perspective.

In fact, things have gone so smoothly they'd like to take things to the next level...





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. In fact, when you go back and look at your code and think about what you'll have to do to modify it, well...

```
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.

SHARPEN YOUR PENCIL

Which of the following describe the state of our implementation? (Choose all that apply.)

	A.	This code certainly isn't adhering to the Open Closed Principle.
	B.	This code would make a FORTRAN programmer proud.
	C.	This design isn't even very object oriented.
	D.	State transitions aren't explicit; they are buried in the middle of a bunch of conditional statements.
	E.	We haven't encapsulated anything that varies here.
	F.	Further additions are likely to cause bugs in working code.

Joe: You're right about that! We need to refactor this code so that it's easy to maintain and modify.

Anne: We really should try to localize the behavior for each state so that if we make changes to one state, we don't run the risk of messing up the other code.

Joe: Right; in other words, follow that ol' "encapsulate what varies" principle.

Anne: Exactly.

Joe: If we put each state's behavior in its own class, then every state just implements its own actions.

Anne: Right. And maybe the Gumball Machine can just delegate to the state object that represents the current state.

Joe: Ah, you're good: favor composition... more principles at work.

Anne: Cute. Well, I'm not 100% sure how this is going to work, but I think we're on to something.

Joe: I wonder if this will make it easier to add new states?

Anne: I think so... We'll still have to change code, but the changes will be much more limited in scope because adding a new state will mean we just have to add a new class and maybe change a few transitions here and there.

Joe: I like the sound of that. Let's start hashing out this new design!

The new design

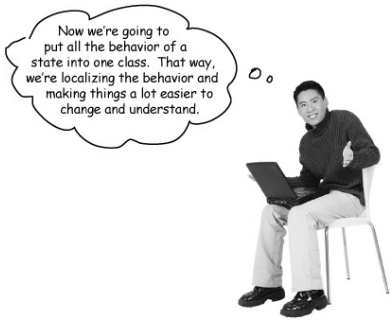
It looks like we've got a new plan: instead of maintaining our existing code, we're going to rework it to encapsulate state objects in their own classes and

then delegate to the current state when an action occurs.

We're following our design principles here, so we should end up with a design that is easier to maintain down the road. Here's how we're going to do it:

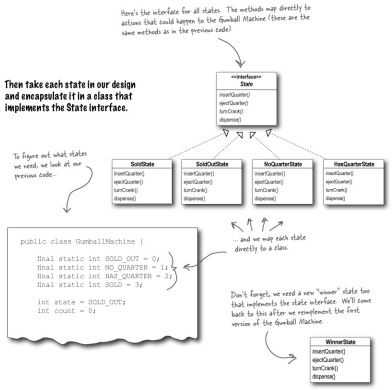
1. **First, we're going to define a State interface that contains a method for every action in the Gumball Machine.**
2. **Then we're going to implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.**
3. **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.**

Not only are we following design principles, as you'll see, we're actually implementing the State Pattern. But we'll get to all the official State Pattern stuff after we rework our code...



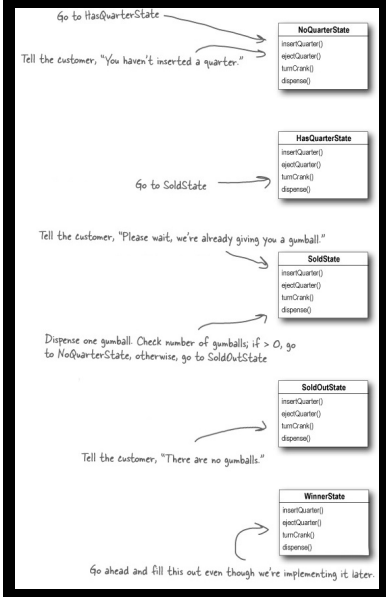
Defining the State interfaces and classes

First let's create an interface for State, which all our states implement:



SHARPEN YOUR PENCIL

To implement our states, we first need to specify the behavior of the classes when each action is called. Annotate the diagram below with the behavior of each action in each class; we've already filled in a few for you.



Implementing our State classes

Time to implement a state: we know what behaviors we want; we just need to get it down in code. We're going to closely follow the state machine code we wrote, but this time everything is broken out into different classes.

Let's start with the NoQuarterState:

```
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");
    }
}
```

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.

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...

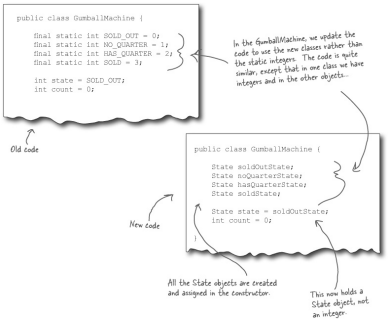
You can't get money back if you never gave it to us!

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

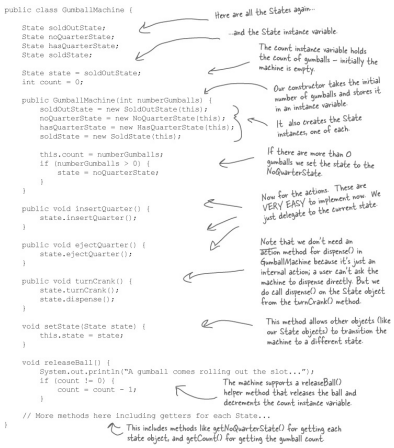
We can't be dispensing gumballs without payment.

Reworking the Gumball Machine

Before we finish the State classes, we're going to rework the Gumball Machine – that way you can see how it all fits together. We'll start with the state-related instance variables and switch the code from using integers to using state objects:

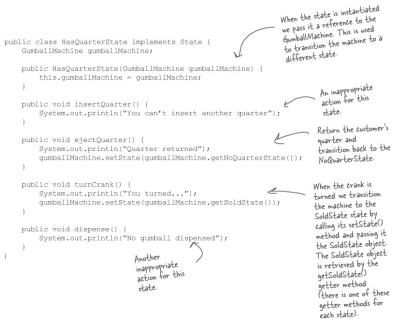


Now, let's look at the complete GumballMachine class...

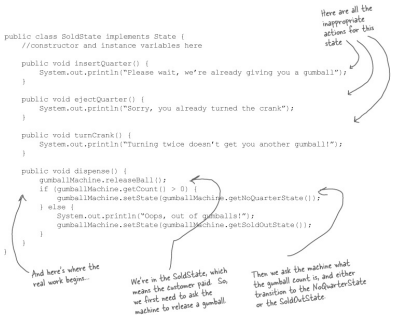


Implementing more states

Now that you're starting to get a feel for how the Gumball Machine and the states fit together, let's implement the HasQuarterState and the SoldState classes...



Now, let's check out the SoldState class...



BRAIN POWER

Look back at the GumballMachine implementation. If the crank is turned and not successful (say the customer didn't insert a quarter first), we call dispense anyway, even though it's unnecessary. How might you fix this?

SHARPEN YOUR PENCIL

We have one remaining class we haven't implemented: SoldOutState. Why don't you implement it? To do this, carefully think through how the Gumball Machine should behave in each situation. Check your answer before moving on...

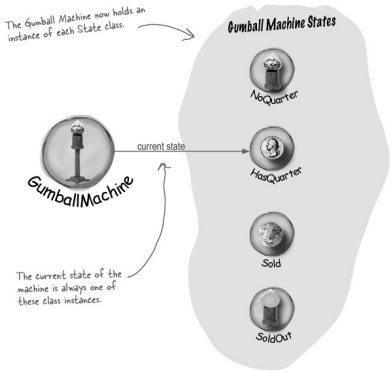
```
public class SoldOutState implements State      {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {


    }

    public void insertQuarter() {


    }

    public void ejectQuarter() {


    }

    public void turnCrank() {


    }

    public void dispense() {


    }
}
```

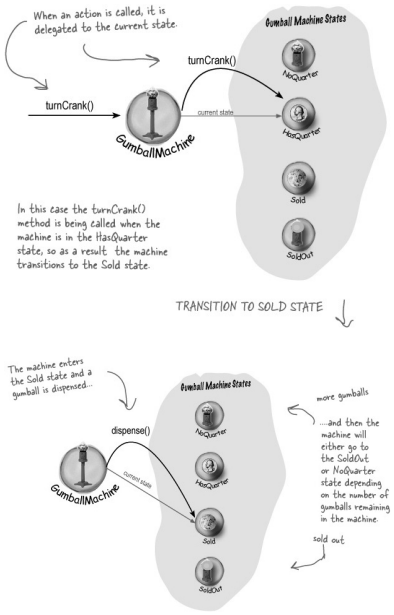
Let's take a look at what we've done so far...

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.

Now let's look a little more at the functional aspect of what we did:

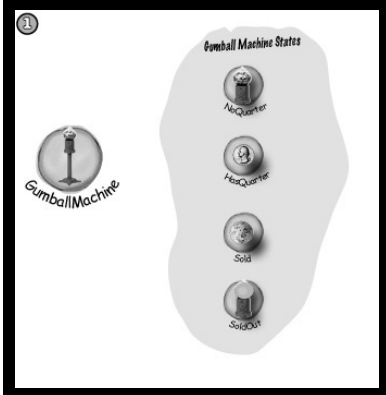


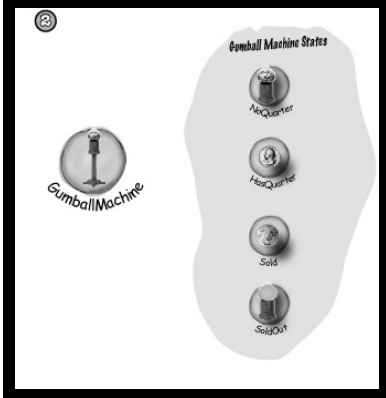

When an action is called, it is delegated to the current state.
turnCrank()
turnCrank()
current state
GumballMachine
Gumball Machine States
NoQuarter
HasQuarter
Sold
SoldOut
In this case the turnCrank() method is being called when the machine is in the HasQuarter state, so as a result the machine transitions to the Sold state.
TRANSITION TO SOLD STATE
The machine enters the Sold state and a gumball is dispensed...
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

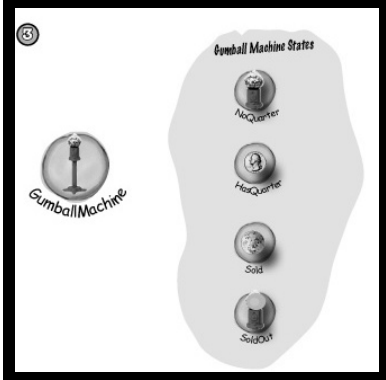
SHARPEN YOUR PENCIL

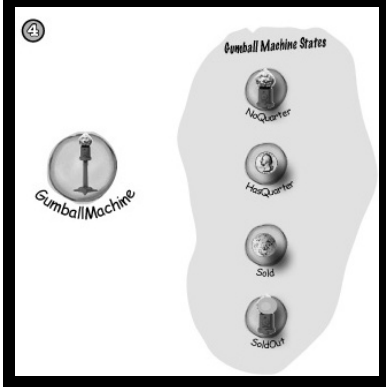
Behind the Scenes: Self-Guided Tour

Trace the steps of the Gumball Machine starting with the NoQuarter state. Also annotate the diagram with actions and output of the machine. For this exercise you can assume there are plenty of gumballs in the machine.









The State Pattern defined

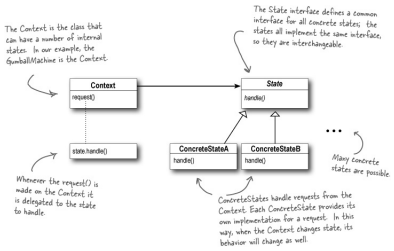
Yes, it's true, we just implemented the State Pattern! So now, let's take a look at what it's all about:

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

The first part of this description makes a lot of sense, right? Because the pattern encapsulates state into separate classes and delegates to the object representing the current state, we know that behavior changes along with the internal state. The Gumball Machine provides a good example: when the gumball machine is in the NoQuarterState and you insert a quarter, you get different behavior (the machine accepts the quarter) than if you insert a quarter when it's in the HasQuarterState (the machine rejects the quarter).

What about the second part of the definition? What does it mean for an object to "appear to change its class?" Think about it from the perspective of a client: if an object you're using can completely change its behavior, then it appears to you that the object is actually instantiated from another class. In reality, however, you know that we are using composition to give the appearance of a class change by simply referencing different state objects.

Okay, now it's time to check out the State Pattern class diagram:



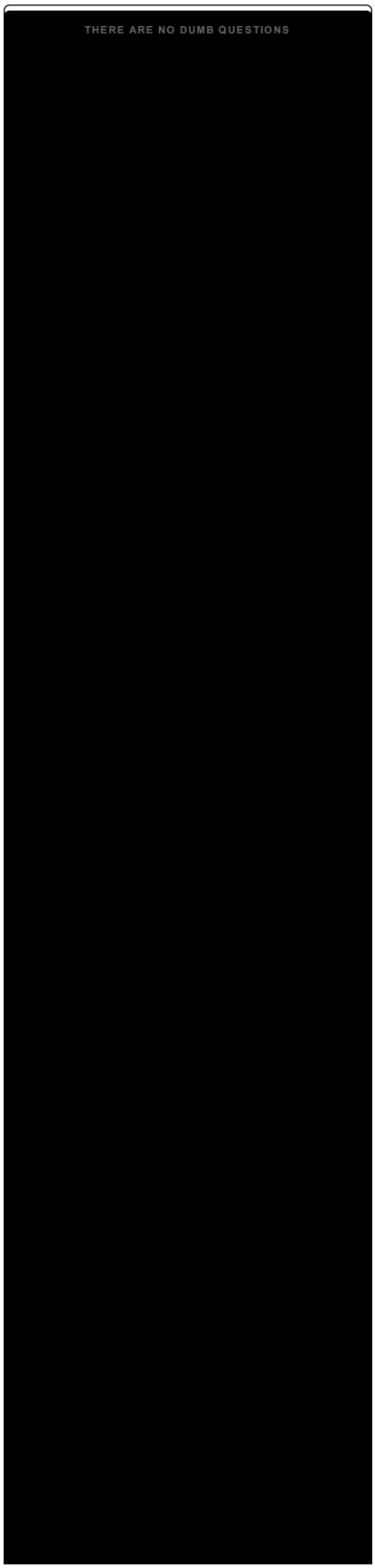
You've got a good eye! Yes, the class diagrams are essentially the same, but the two patterns differ in their *intent*.

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 states. 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 object that is most appropriate for a context object. For instance, in Chapter 1, some of our ducks were configured to fly with typical flying behavior (like mallard ducks), while others were configured with a fly behavior that kept them grounded (like rubber ducks and decoy ducks).

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 behaviors within state objects, you can simply change the state object in context to change its behavior.


THERE ARE NO DUMB QUESTIONS

Q: **Q: In the GumballMachine, the states decide what the next state should be. Do the ConcreteStates always decide what state to go to next?**

A: **A:** *No, not always. The alternative is to let the Context decide on the flow of state transitions.*

As a general guideline, when the state transitions are fixed they are appropriate for putting in the Context; however, when the transitions are more dynamic, they are typically placed in the state classes themselves (for instance, in the GumballMachine the choice of the transition to NoQuarter or SoldOut depended on the runtime count of gumballs).

The disadvantage of having state transitions in the state classes is that we create dependencies between the state classes. In our implementation of the GumballMachine we tried to minimize this by using getter methods on the Context, rather than hardcoding explicit concrete state classes.

Notice that by making this decision, you are making a decision as to which classes are closed for modification – the Context or the state classes – as the system evolves.

Q: **Q: Do clients ever interact directly with the states?**

A: **A:** *No. The states are used by the Context to represent its internal state and behavior, so all requests to the states come from the Context. Clients don't directly change the state of the Context. It is the Context's job to oversee its state, and you don't usually want a client changing the state of a Context without that Context's knowledge.*

Q: **Q: If I have lots of instances of the Context in my application, is it possible to share the state objects across them?**

A: **A:** *Yes, absolutely, and in fact this is a very common scenario. The only requirement is that your state objects do not keep their own internal state; otherwise, you'd need a unique instance per context.*

To share your states, you'll typically assign each state to a static instance variable. If your state needs to make use of methods or instance variables in your Context, you'll also have to give it a reference to the Context in each handler() method.

Q: **Q: It seems like using the State Pattern always increases the number of classes in our designs. Look how many more classes our GumballMachine had than the original design!**

A: **A:** *You're right, by encapsulating state behavior into separate state classes, you'll always end up with more classes in your design. That's often the price you pay for flexibility. Unless your code is some "one off" implementation you're going to throw away (yeah, right), consider building it with the additional classes and you'll probably thank yourself down the road. Note that often what is important is the number of classes that you expose to your clients, and there are ways to hide these extra classes from your clients (say, by declaring them package visible).*

Also, consider the alternative: if you have an application that has a lot of state and you decide not to use separate objects, you'll instead end up with very large, monolithic conditional statements. This makes your code hard to maintain and understand. By using objects, you make states explicit and reduce the effort needed to understand and maintain your code.

Q: **Q: The State Pattern class diagram shows that State is an abstract class. But didn't you use an interface in the implementation of the gumball machine's state?**

A: **A:** *Yes. Given we had no common functionality to put into an abstract class, we went with an interface. In your own implementation, you might want to consider an abstract class. Doing so has the benefit of allowing you to add*

methods to the abstract class later, without breaking the concrete state implementations.

We still need to finish the Gumball 1 in 10 game

Remember, we're not done yet. We've got a game to implement; but now that we've got the State Pattern implemented, it should be a breeze. First, we need to add a state to the GumballMachine class:

```
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.

Now let's implement the WinnerState class itself, it's remarkably similar to the SoldState class:

```
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.

Finishing the game

We've just got one more change to make: we need to implement the random chance game and add a transition to the WinnerState. We're going to add both to the HasQuarterState since that is where the customer turns the crank:

```
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).

Wow, that was pretty simple to implement! We just added a new state to the GumballMachine and then implemented it. All we had to do from there was to implement our chance game and transition to the correct state. It looks like our new code strategy is paying off...

Demo for the CEO of Mighty Gumball, Inc.

The CEO of Mighty Gumball has dropped by for a demo of your new gumball game code. Let's hope those states are all in order! We'll keep the demo short and sweet (the short attention span of CEOs is well documented), but hopefully long enough so that we'll win at least once.

```
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);
    }
}
```

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

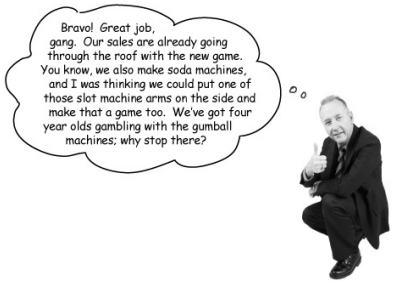
Once, again, start with a gumball machine with 5 gumballs.

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...

THERE ARE NO DUMB QUESTIONS

Q: Why do we need the WinnerState? Couldn't we just have the SoldState dispense two gumballs?

A: *That's a great question. SoldState and WinnerState are almost identical, except that WinnerState dispenses two gumballs instead of one. You certainly could put the code to dispense two gumballs into the SoldState. The downside is, of course, that now you've got TWO states represented in one State class: the state in which you're a winner, and the state in which you're not. So you are sacrificing clarity in your State class to reduce code duplication. Another thing to consider is the principle you learned in the previous chapter: One class, One responsibility. By putting the WinnerState responsibility into the SoldState, you've just given the SoldState TWO responsibilities. What happens when the promotion ends? Or the stakes of the contest change? So, it's a tradeoff and comes down to a design decision.*



Sanity check...

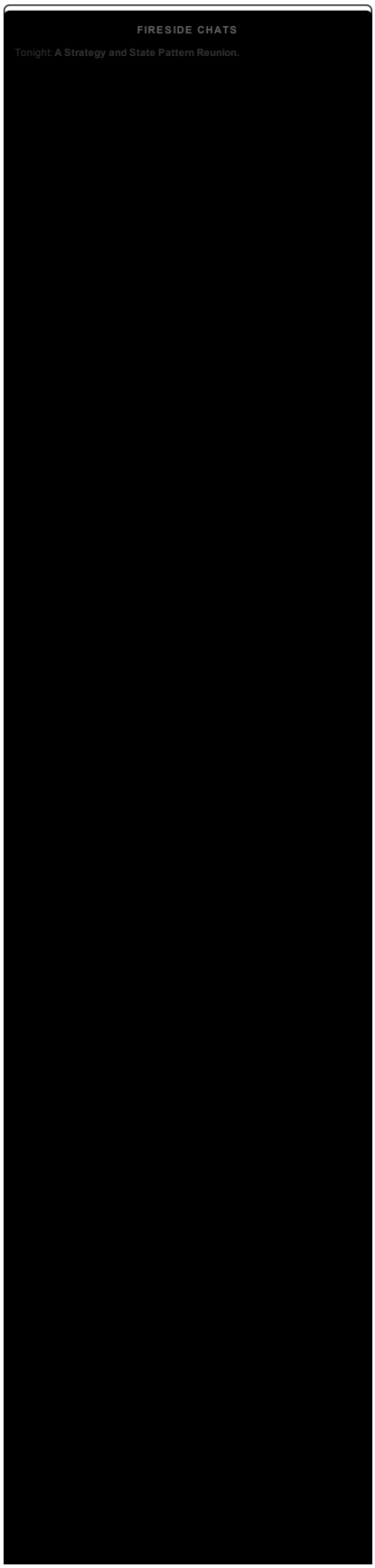
Yes, the CEO of Mighty Gumball probably needs a sanity check, but that's not what we're talking about here. Let's think through some aspects of the GumballMachine that we might want to shore up before we ship the gold version:

- We've got a lot of duplicate code in the Sold and Winning states and we might want to clean those up. How would we do it? We could make State into an abstract class and build in some default behavior for the methods; after all, error messages like, "You already inserted a quarter," aren't going to be seen by the customer. So all "error response" behavior could be generic and inherited from the abstract State class.

NOTE

Dammit Jim, I'm a gumball machine, not a computer!

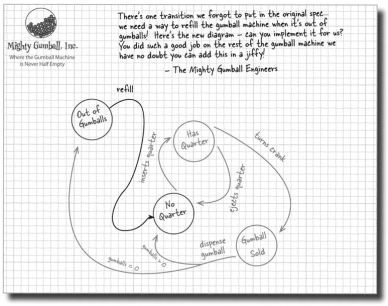
- The dispense() method always gets called, even if the crank is turned when there is no quarter. While the machine operates correctly and doesn't dispense unless it's in the right state, we could easily fix this by having turnCrank() return a boolean, or by introducing exceptions. Which do you think is a better solution?
- All of the intelligence for the state transitions is in the State classes. What problems might this cause? Would we want to move that logic into the Gumball Machine? What would be the advantages and disadvantages of that?
- Will you be instantiating a lot of GumballMachine objects? If so, you may want to move the state instances into static instance variables and share them. What changes would this require to the GumballMachine and the States?


FIRESIDE CHATS
Tonight: A Strategy and State Pattern Reunion.

Strategy	State
Hey bro. Did you hear I was in Chapter 1?	
	Yeah, word is definitely getting around.
I was just over giving the Template Method guys a hand – they needed me to help them finish off their chapter. So, anyway, what is my noble brother up to?	
	Same as always – helping classes to exhibit different behaviors in different states.
I don't know, you always sound like you've just copied what I do and you're using different words to describe it. Think about it: I allow objects to incorporate different behaviors or algorithms through composition and delegation. You're just copying me.	
	I admit that what we do is definitely related, but my intent is totally different than yours. And, the way I teach my clients to use composition and delegation is totally different.
Oh yeah? How so? I don't get it.	
	Well if you spent a little more time thinking about something other than *yourself*, you might. Anyway, think about how you work: you have a class you're instantiating and you usually give it a strategy object that implements some behavior. Like, in Chapter 1 you were handing out quack behaviors, right? Real ducks got a real quack, rubber ducks got a quack that squeaked.
Yeah, that was some *fine* work... and I'm sure you can see how that's more powerful than inheriting your behavior, right?	
	Yes, of course. Now, think about how I work; it's totally different.
Sorry, you're going to have to explain that.	
	Okay, when my Context objects get created, I may tell them the state to start in, but then they change their own state over time.
Hey, come on, I can change behavior at	

runtime too; that's what composition is all about!	
	Sure you can, but the way I work is built around discrete states; my Context objects change state over time according to some well defined state transitions. In other words, changing behavior is built in to my scheme – it's how I work!
Well, I admit, I don't encourage my objects to have a well-defined set of transitions between states. In fact, I typically like to control what strategy my objects are using.	
	Look, we've already said we're alike in structure, but what we do is quite different in intent. Face it, the world has uses for both of us.
Yeah, yeah, keep living your pipe dreams brother. You act like you're a big pattern like me, but check it out: I'm in Chapter 1; they stuck you way out in Chapter 10. I mean, how many people are actually going to read this far?	
	Are you kidding? This is a Head First book and Head First readers rock. Of course they're going to get to Chapter 10!
That's my brother, always the dreamer.	

We almost forgot!



SHARPEN YOUR PENCIL

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.

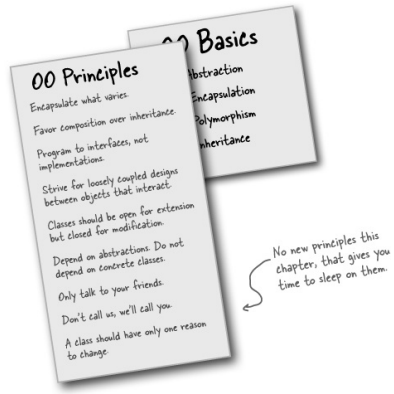
WHO DOES WHAT?

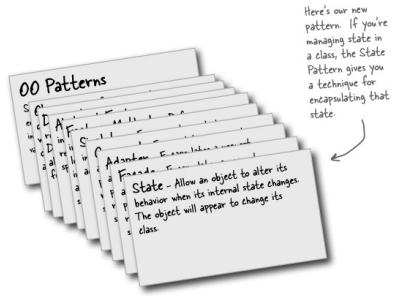
Match each pattern with its description:

State	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use
Strategy	Subclasses decide how to implement steps in an algorithm
Template Method	Encapsulate state-based behavior and delegate behavior to the current state

Tools for your Design Toolbox

It's the end of another chapter; you've got enough patterns here to breeze through any job interview!

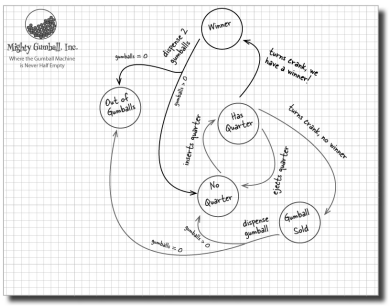




BULLET POINTS

- 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.

Exercise Solutions



SHARPEN YOUR PENCIL

Based on our first implementation, which of the following apply? (Choose all that apply.)

	A.	This code certainly isn't adhering to the Open Closed Principle!
	B.	This code would make a FORTRAN programmer proud.
	C.	This design isn't even very object oriented.
	D.	State transitions aren't explicit; they are buried in the middle of a bunch of conditional code.
	E.	We haven't encapsulated anything that varies here.
	F.	Further additions are likely to cause bugs in working code.

SHARPEN YOUR PENCIL

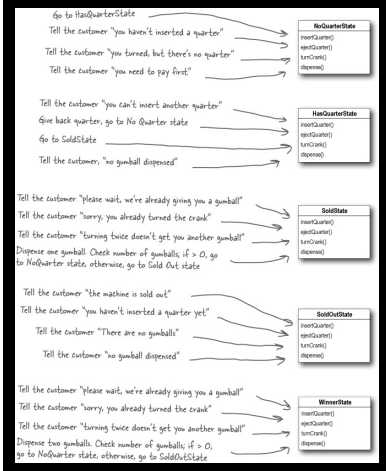
We have one remaining class we haven't implemented: SoldOutState. Why don't you implement it? To do this, carefully think through how the Gumball Machine should behave in each situation. Check your answer before moving on...

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

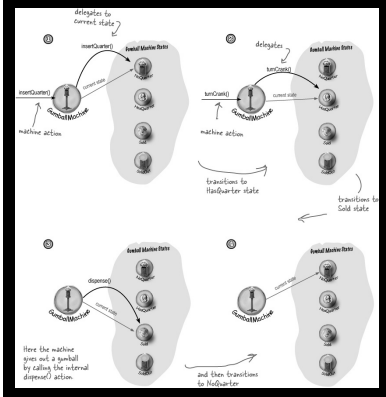
```
public class SoldOutState implements State {
    GumballMachine gumballMachine;

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

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

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

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

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

SHARPEN YOUR PENCIL

To implement the states, we first need to define what the behavior will be when the corresponding action is called. Annotate the diagram below with the behavior of each action in each class; we've already filled in a few for you.



BEHIND THE SCENES: SELF-GUIDED TOUR SOLUTION



WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
State	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use
Strategy	Subclasses decide how to implement steps in an algorithm
Template Method	Encapsulate state-based behavior and delegate behavior to the current state

SHARPEN YOUR PENCIL

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;
}
```

PREV
9. The Iterator and Composite Patterns: Well-Managed Collecti...

NEXT
11. The Proxy Pattern: Controlling Object Access