

CS525

Advanced Software

Development

Lesson 10 – The State Pattern

Design Patterns

Elements of Reusable Object-Oriented Software

Payman Salek, M.S.

March 2022

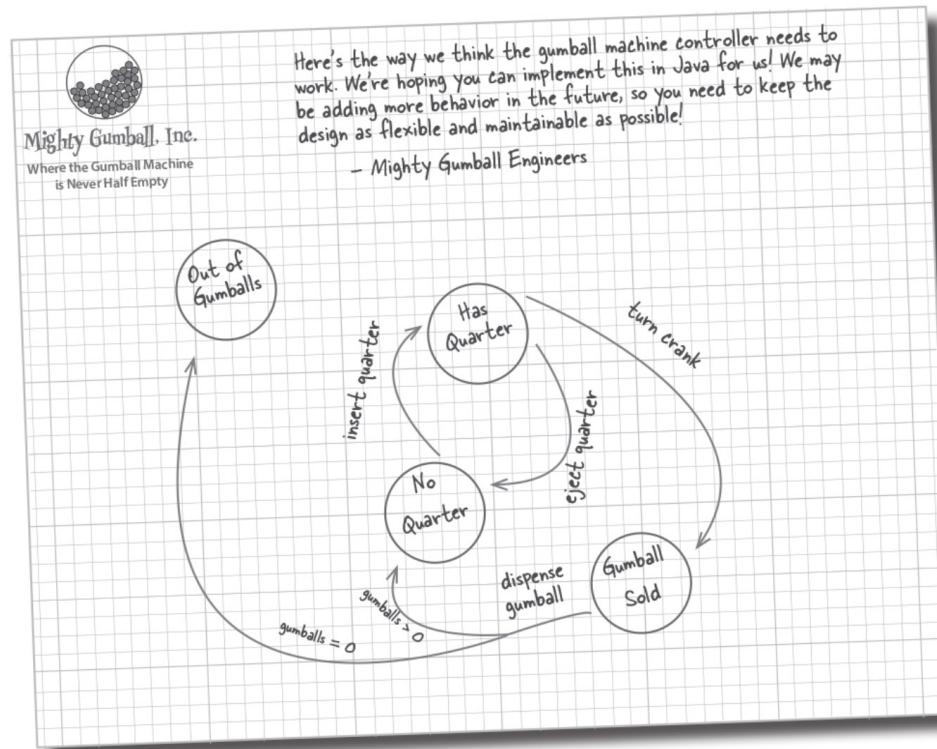
© 2022 Maharishi International University



A little Known Fact

The Strategy and State Patterns are twins separated at birth. You'd think they'd live similar lives, but the Strategy Pattern went on to create a wildly successful business around interchangeable algorithms, while State took the perhaps more noble path of helping objects to control their behavior by changing their internal state.

Setting the stage (Java Breakers)



State Machine 101 (define states)



Instance Variable to Hold the State

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

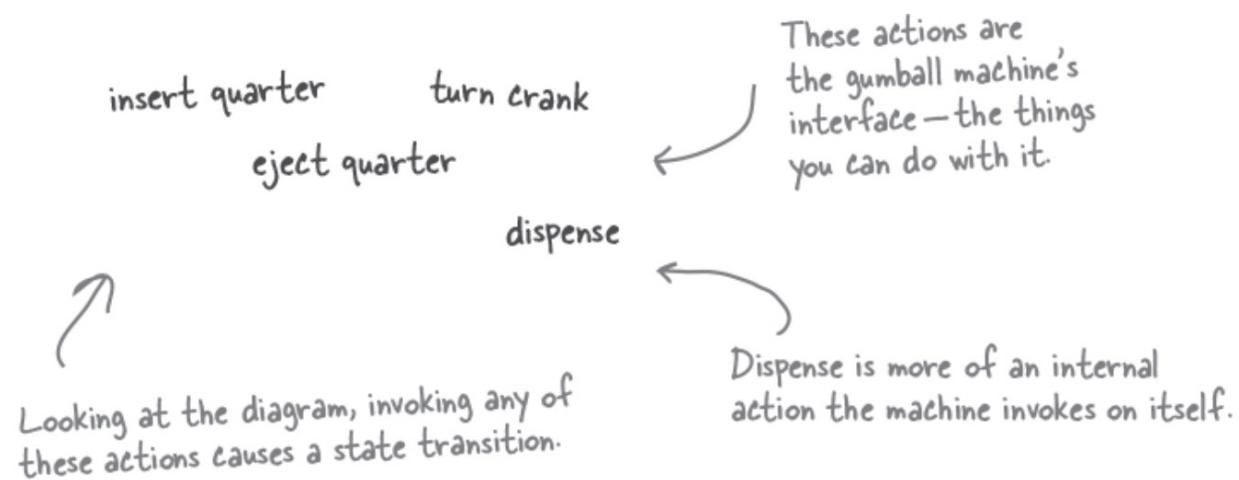
```
final static int SOLD_OUT = 0;  
final static int NO_QUARTER = 1;  
final static int HAS_QUARTER = 2;  
final static int SOLD = 3;
```

Here's each state represented
as a unique integer...

```
int state = SOLD_OUT;
```

...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.

Identify All Actions that Change State



Implement Each Action

3/9/22

```
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.

The Gumball Machine

3/9/22

```
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;  
        }  
        Now we start implementing  
        the actions as methods....  
    }  
  
    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");  
        }  
        If the customer just bought a  
        gumball, he needs to wait until the  
        transaction is complete before  
        inserting another quarter.  
    }  
  
    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.  
  
    And if the machine is sold  
    out, we reject the quarter.  
}
```

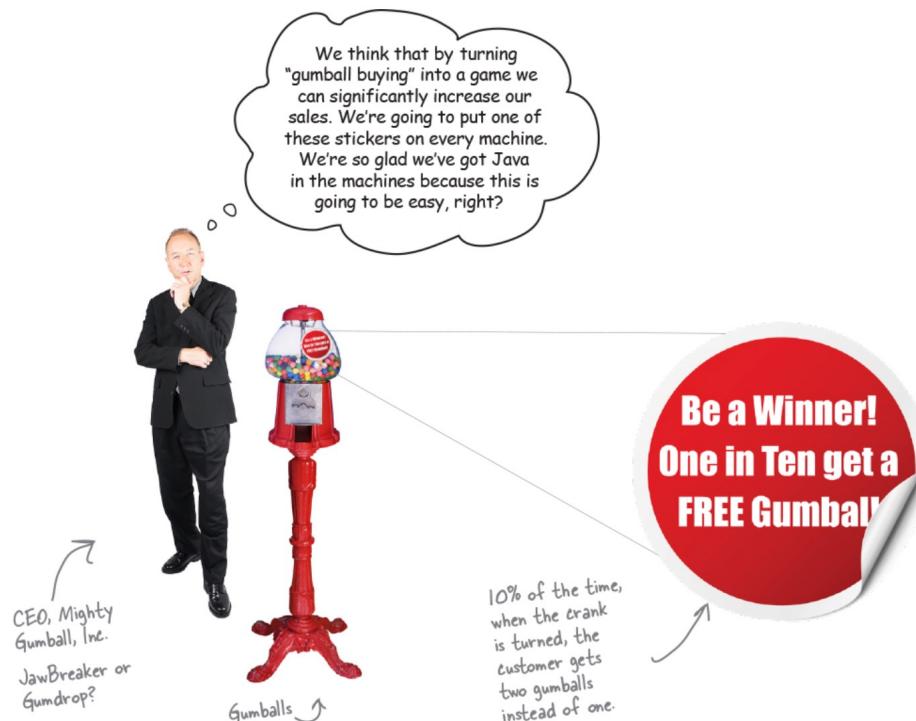
Here are the four states; they match the states in Mighty Gumball's state diagram.

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.

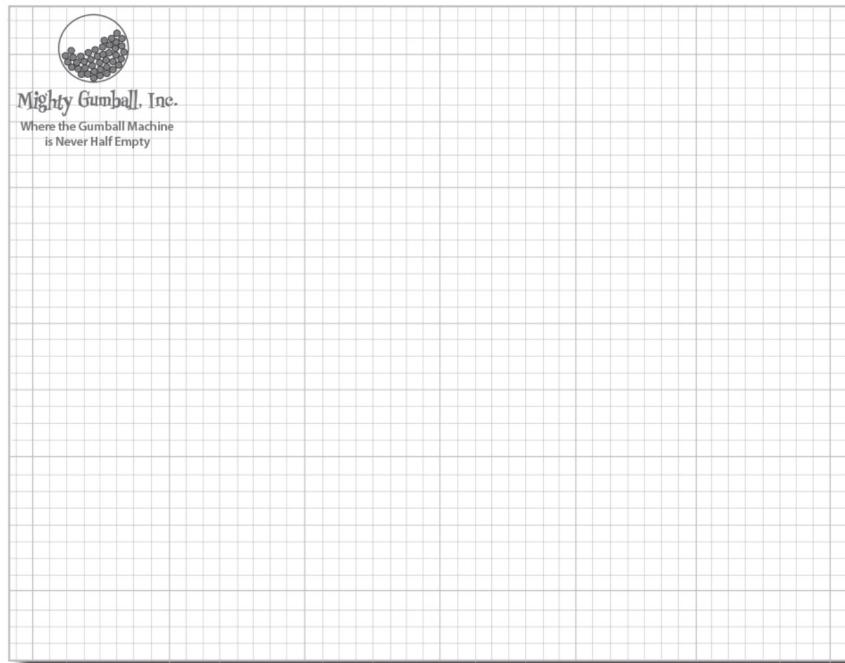
We have a second instance variable that keeps track of the number of gumballs in the machine.

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.

A Change Request



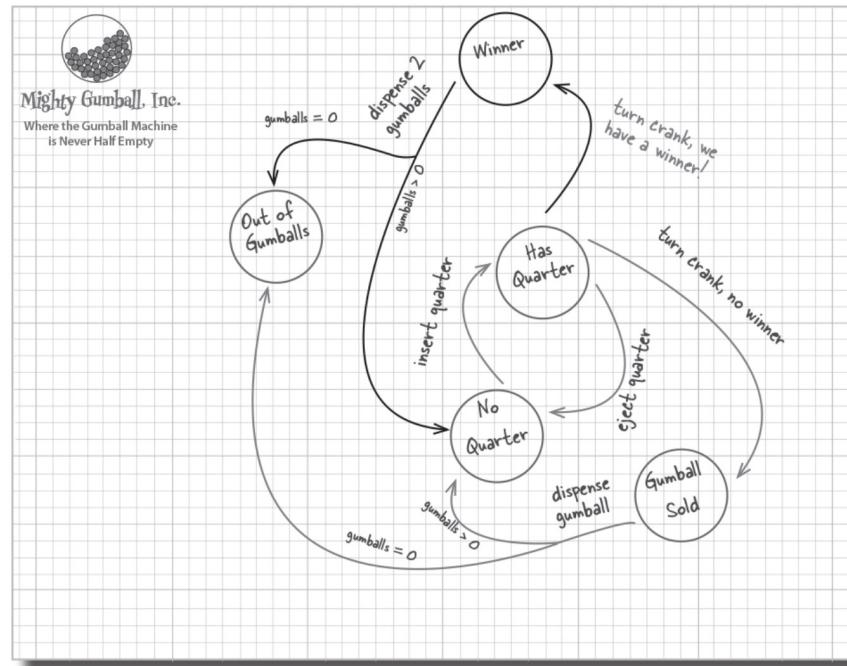
Add the New State



Use Mighty Gumball's stationery to draw your state diagram.



Solution



The Messy State of Things!

```
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.

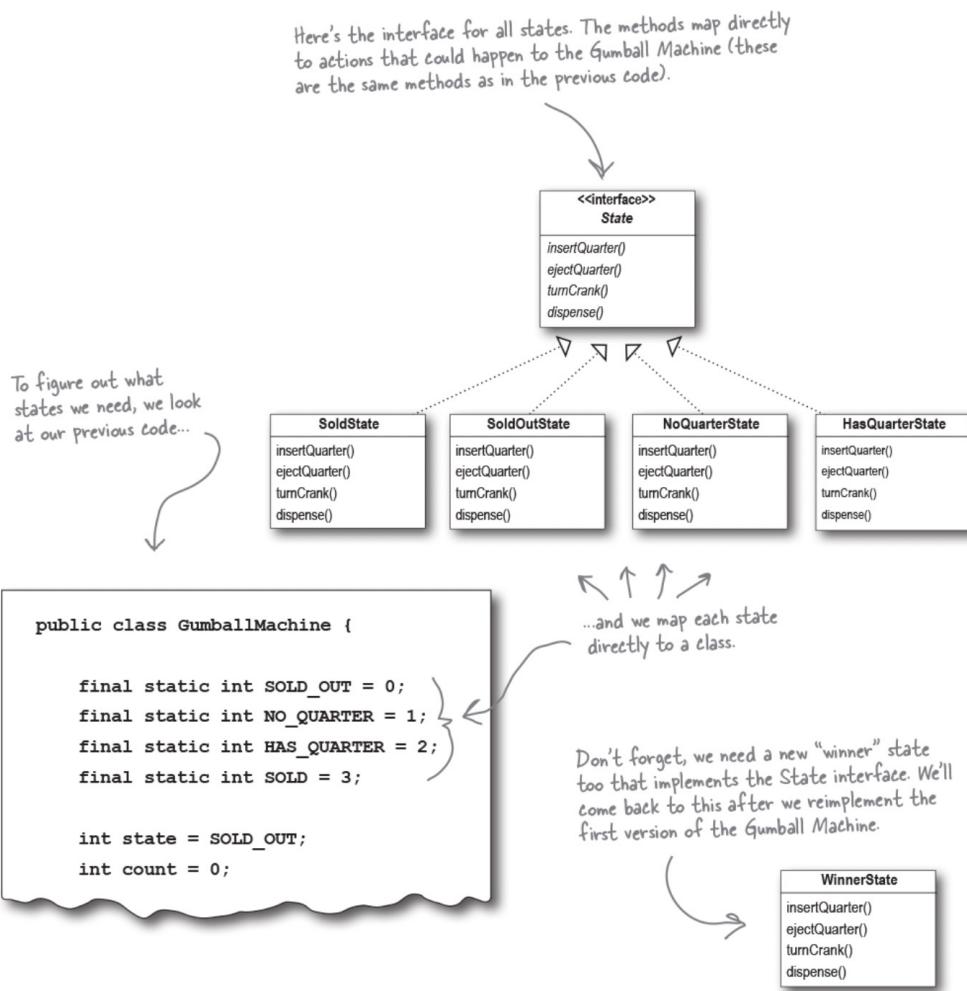
turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.

The New Design

- 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 the work to the State class.

Defining the State interface and classes

3/9/22



Implementing our State classes

3/9/22

```
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

3/9/22

```
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...

```
public class GumballMachine {  
  
    state soldOutState;  
    state noQuarterState;  
    state hasQuarterState;  
    state soldState;  
  
    state state = soldOutState;  
    int count = 0;
```

New code

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

This now holds a State object, not an integer.

Reworking the Gumball Machine

3/9/22

```
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;  
        }  
    }  
  
    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...  
}
```

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.

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.

Implementing More States

3/9/22

```
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.

Implementing More States

3/9/22

```
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!");  
    }  
  
    And here's where the  
    real work begins...  
  
    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.

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.

Can you write one of the States?

3/9/22

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

    public SoldOutState(GumballMachine gumballMachine) {

    }

    public void insertQuarter() {

    }

    public void ejectQuarter() {

    }

    public void turnCrank() {

    }

    public void dispense() {

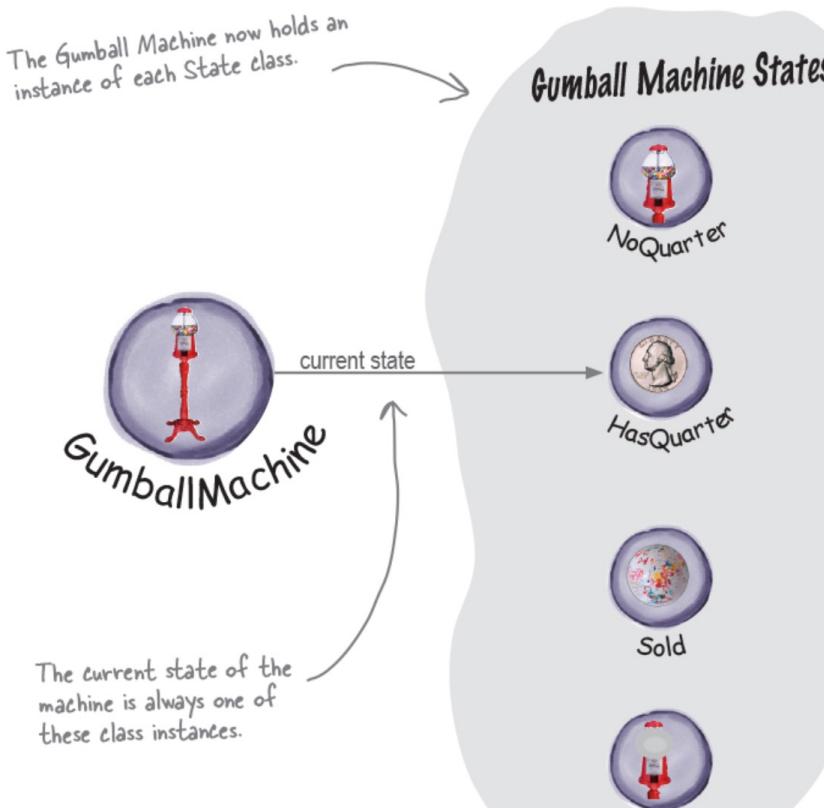
    }
}
```

So far...

- 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.

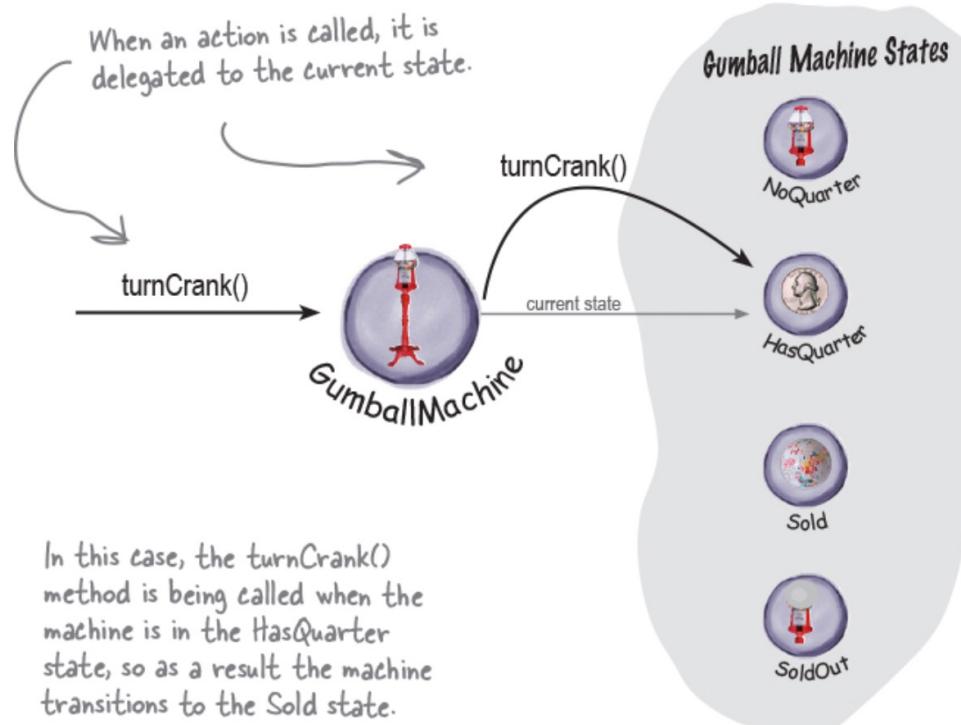
How does it work?

3/9/22



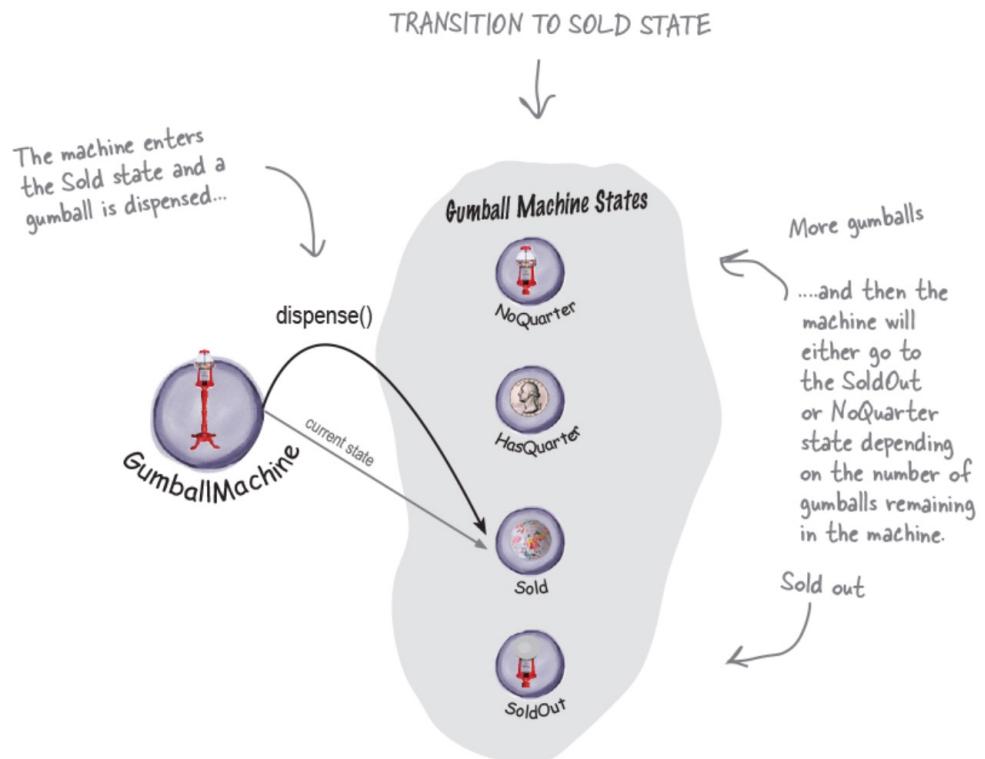
How does it work?

3/9/22



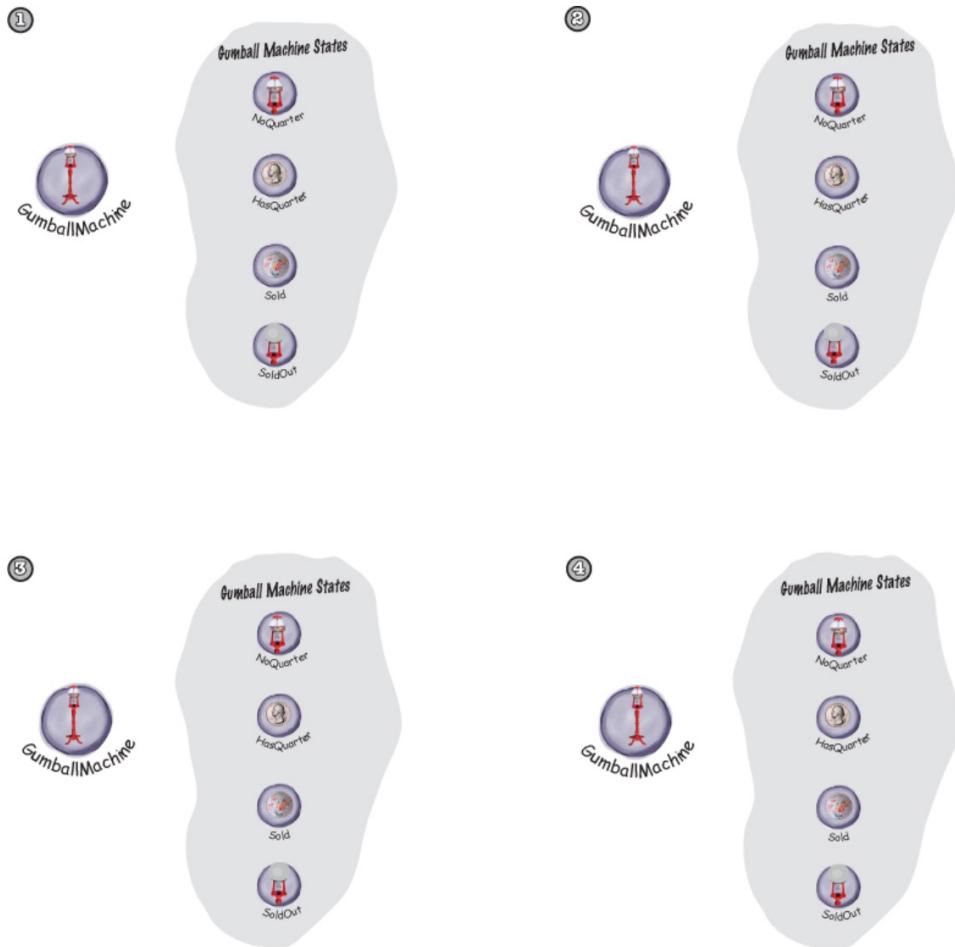
How does it work?

3/9/22



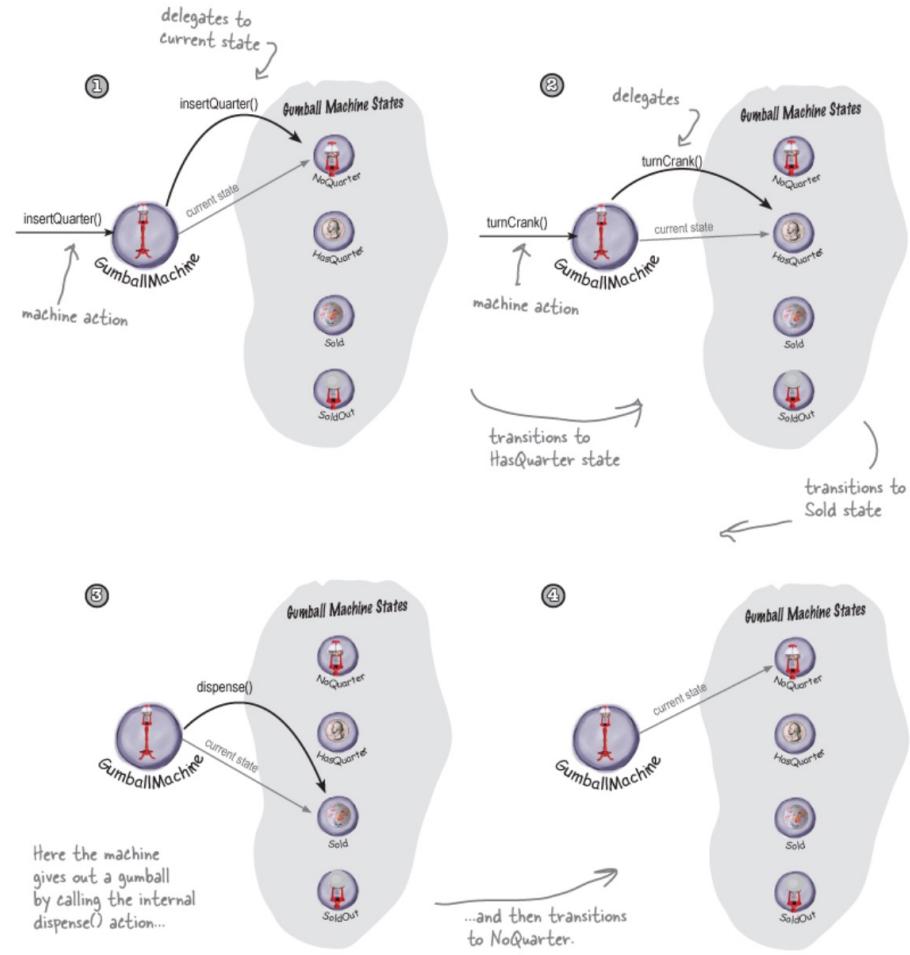
Trace the Steps of the Gumball Machine

3/9/22



The Solution

3/9/22

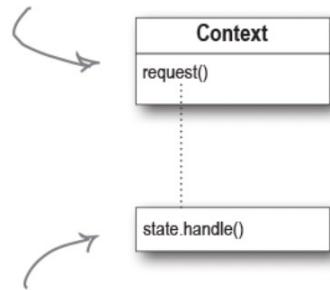


The State Pattern Defined

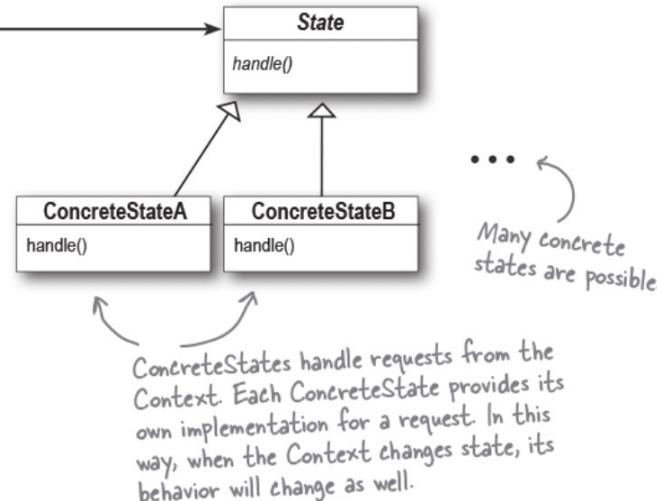
Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The State Pattern (UML)

The Context is the class that can have a number of internal states. In our example, the GumballMachine is the Context.



The State interface defines a common interface for all concrete states; the states all implement the same interface, so they are interchangeable.



The 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.

The Gumball 1 in 10 Game

```
public class WinnerState implements State {  
  
    // instance variables and constructor  
    // insertQuarter error message  
    // ejectQuarter error message  
    // turnCrank error message  
  
    public void dispense() {  
        gumballMachine.releaseBall();  
        if (gumballMachine.getCount() == 0) {  
            gumballMachine.setState(gumballMachine.getSoldOutState());  
        } else {  
            gumballMachine.releaseBall(); ← If we have a second gumball, we release it.  
            System.out.println("YOU'RE A WINNER! You got two gumballs for your quarter");  
            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.

If we were able to release two gumballs, we let the user know he was a winner.

Finishing the Game...

3/9/22

```
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 WinnerState; otherwise, we go to SoldState (just like we always did).

The Demo

3/9/22

```
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...

Summary

3/9/22

