The State Pattern

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

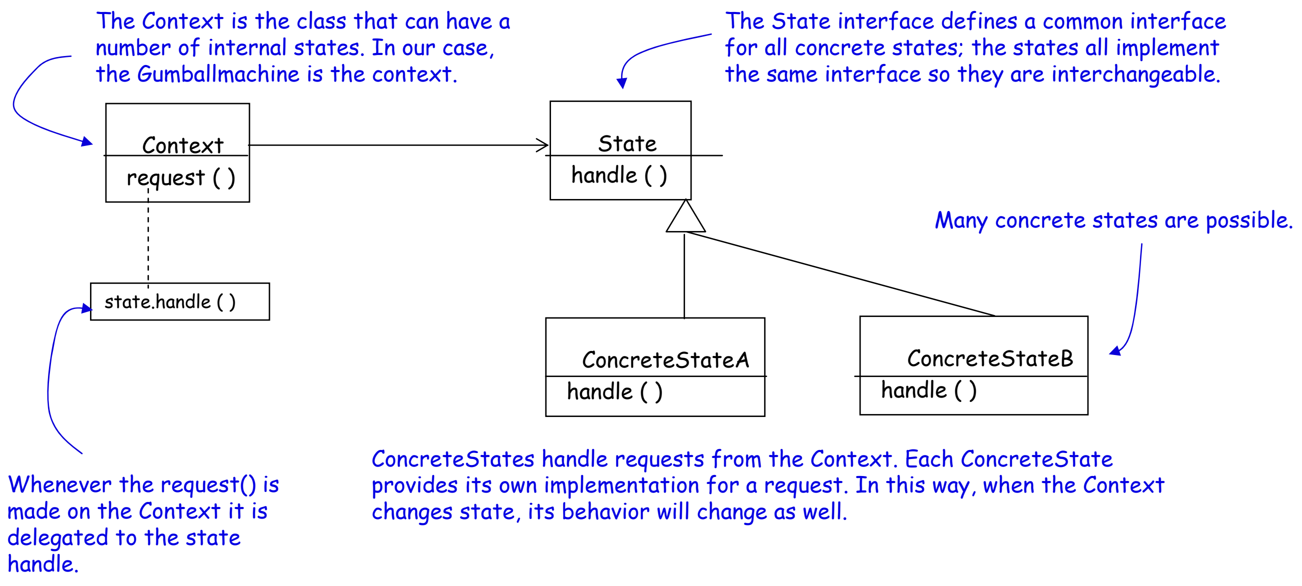
Toni Sellarès
Universitat de Girona

State Design Pattern: Motivation

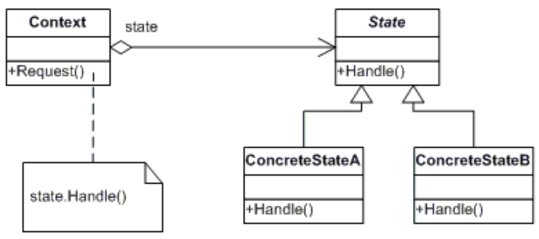
- The State pattern is useful when you want to have an object represent the state of an application, and you want to change the state by changing that object.
- The State pattern is intended to provide a mechanism to allow an object to alter its behavior in response to internal state changes. To the client, it appears as though the object has changed its class.
- The benefit of the State pattern is that state-specific logic is localized in classes that represent that 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.


The Context is the class that can have a number of internal states. In our case, 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.
Context
request ( )
State
handle ( )
Many concrete states are possible.
state.handle ( )
ConcreteStateA
handle ( )
ConcreteStateB
handle ( )
ConcreteStates handle requests from the Context. Each ConcreteState provides its own implementation for a request. In this way, when the Context changes state, its behavior will change as well.
Whenever the request() is made on the Context it is delegated to the state handle.

Participants



Context:

- defines the interface of interest to clients,
- maintains an instance of a ConcreteState subclass that defines the current state.

State: defines an interface for encapsulating the behavior associated with a particular state of the Context.

Concrete State: each subclass implements a behavior associated with a state of Context.

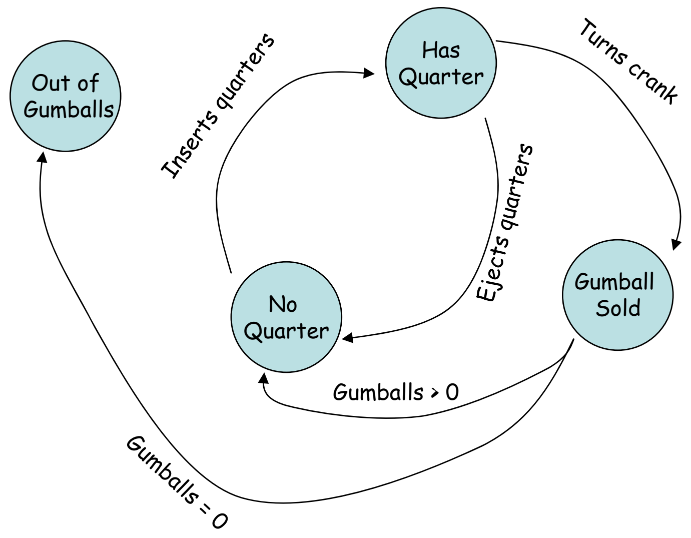
State Pattern: Structural Code

```
/**
 * Test driver for the pattern.
 */
public class Test {
          public static void main( String arg[] ) {
                    State state = new ConcreteStateA();
                    Context context = new Context();
                    context.setState( state );
                    context.request();
                    }
}

/**
 * Defines an interface for encapsulating the behavior
 * associated with a particular state of the Context.
 */
public interface Stat{
          void handle();
}
```

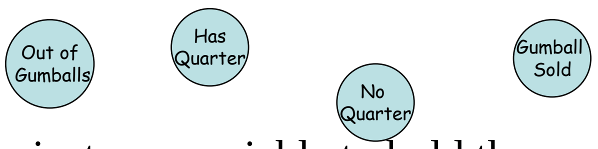
```
/**
 * Defines the interface of interest to clients. Maintains an instance of a ConcreteState
 * subclass that defines the current state.
 */
public class Context {
          private State state;
          public void setState( State state ) {
                    this.state = state; }
          public State getState() {
                    return state; }
          public void request(){
                    state.handle();}
}

/**
 * Each subclass implements a behavior associated with a state of the Context.
 */
public class ConcreteStateA implements State {
          public void handle() {}
}

public class ConcreteStateB implements State {
          public void handle() {}
}
```

State Pattern: Example - Gumball Machine

Here is one way that perhaps a gumball machine controller needs to work:



1. First, gather up your states:



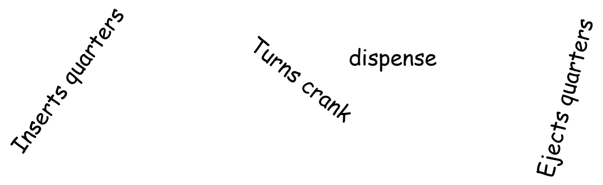
2. Create an instance variable to hold the current state and define values for each state:

```
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 by a unique integer

We hold the current state in an instance variable.

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



4. Now we create a method that acts as a state machine.We use conditionals to determine what behavior is appropriate in each state.

```
public void insertQuarter ( ) {
        if (state == HAS_QUARTER) {
            System.out.println(" Can't insert another quarter");
        } else if (state == SOLD_OUT) {
            System.out.println(" Can't insert a quarter, the machine is sold out");
        } else if (state == SOLD) {
            System.out.println (" Please wait we are getting you a gumball");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a quarter");
        }
}
```

You can exhibit the appropriate behavior for each state.

Or transition to another state.

Gumball Implementation

```
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(" Can't insert another quarter");
            } else if (state == SOLD_OUT) {
                  System.out.println(" Can't insert a quarter, the machine is sold out");
            } else if (state == SOLD) {
                  System.out.println (" Please wait we are getting you a gumball");
            } else if (state == NO_QUARTER) {
                  state = HAS_QUARTER;
                  System.out.println("You inserted a quarter");
            }
    }
    public void ejectQuarter ( ) {  }
    public void turnCrank ( ) { }
    public void dispense ( ) { }
    // other methods
```

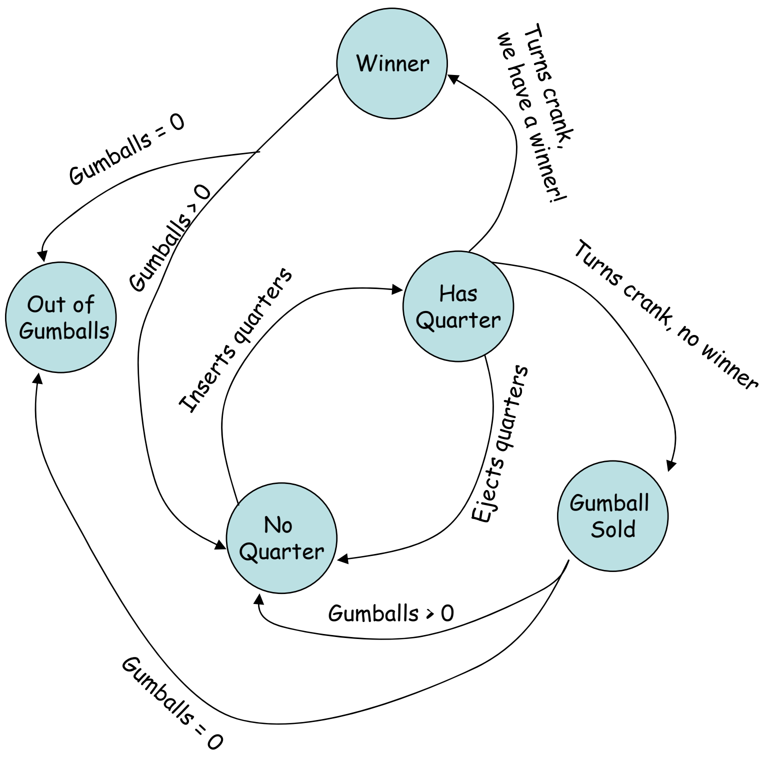
The insertQuarter method -- specifies what to do if a quarter is inserted.

Represent the methods for each action - customer tries to turn the crank etc.

Gumball Machine: a change request

- Gumball machine works great but need to take it to the next level:
  - Turn gumball buying into a game!
    - 10% of the time when the crank is turned, the customer gets two gumballs instead on one!
- 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.

1 in 10 Gumball Game



- Modifications to your well-thought out Gumball machine code:

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

First you need to add a new WINNER state here. That isn't too bad…..

```
public void insertQuarter ( ) { }
public void ejectQuarter ( ) {  }
public void turnCrank ( ) { }
public void dispense ( ) { }
// other methods
```

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

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

This isn't good. While the first design was "good", it isn' t going to hold up to modifications.

The new design

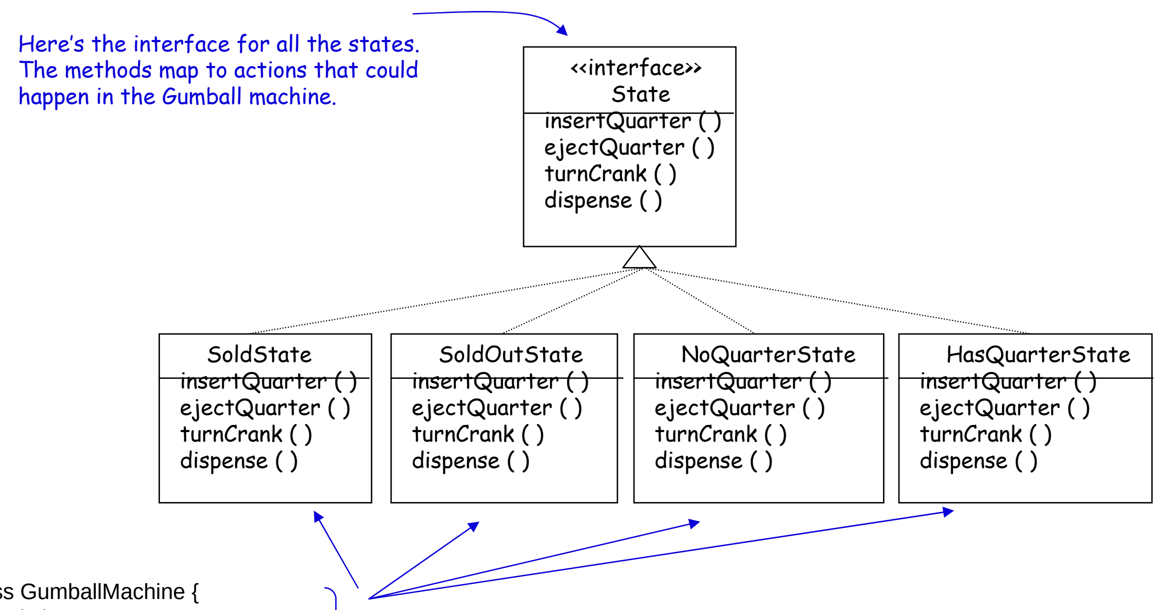
Instead of maintaining the code, we are going to rework the design to *encapsulate* the state objects in their own classes and then *delegate* to the current state when an action occurs:

1. Define a State interface that contains a method for every action in the Gumball Machine,

2. 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. We are going to get rid of all the conditional code and instead delegate to the state class to do all the work.

Defining the State Interfaces and Classes



```
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;
```

Implementing the State Classes

First, we implement the State interface

```
public class NoQuarterState implements State {
      Gumball Machine gumballMachine;

      public NoQuarterState (GumballMachine gumballMachine) {
            this.gumballMachine = gumballMachine;
      }
      public void insertQuarter ( )  {
            System.out.println("You have 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 is no quarter");
      }
      public void dispense ( ) {
            System.out.println("You have to pay first");
      }
}
```

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

Coming right up…….

Reworking the Gumball Machine

```
public class GumballMachine {
        State soldOutState;
        State noQuarterState;
        State hasQuarterState;
        State soldState;

        State state = soldOutState;
        int count = 0;
 …..
}
```

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….

This now holds a State object and not an integer.

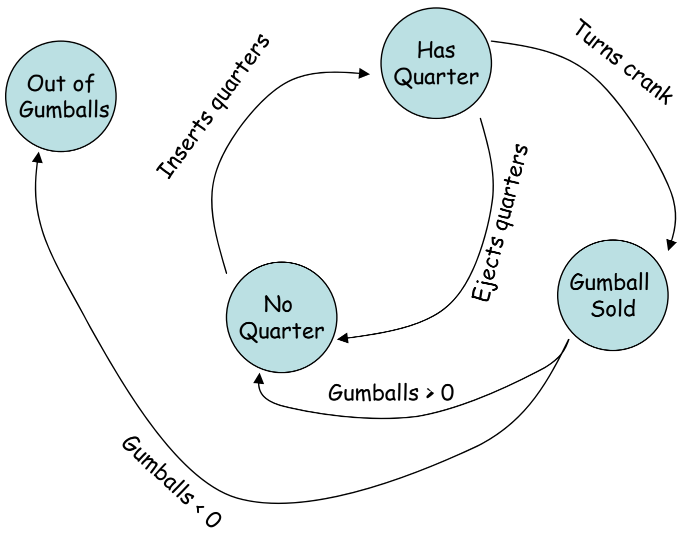
All the State objects are created and assigned in the constructor.

The Complete Gumball Machine Class

```
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 HasSoldState (this);
                this.count = numberGumballs;
                if (numberGumballs > 0) {
                        state = noQuarterState;
                }
        }
        public void insertQuarter () {
                state.insertQuarter ( );
        }
        public void ejectQuarter ( ) {
                state.ejectQuarter ( );
        }
        public void turnCrank ( ) {
                state.turnCrank ( );
                state.dispense( );
        }
        public void setState (State state) {
                this.state = state;
        }
        void releaseBall ( ) {
                System.out.println ("A gumball comes rolling out of the slot….");
                if  (count != 0 ) { count --; }  }
}
```

These methods are now VERY EASY to implement! We just delegate to the current state.

State Diagram



Check out the SoldState

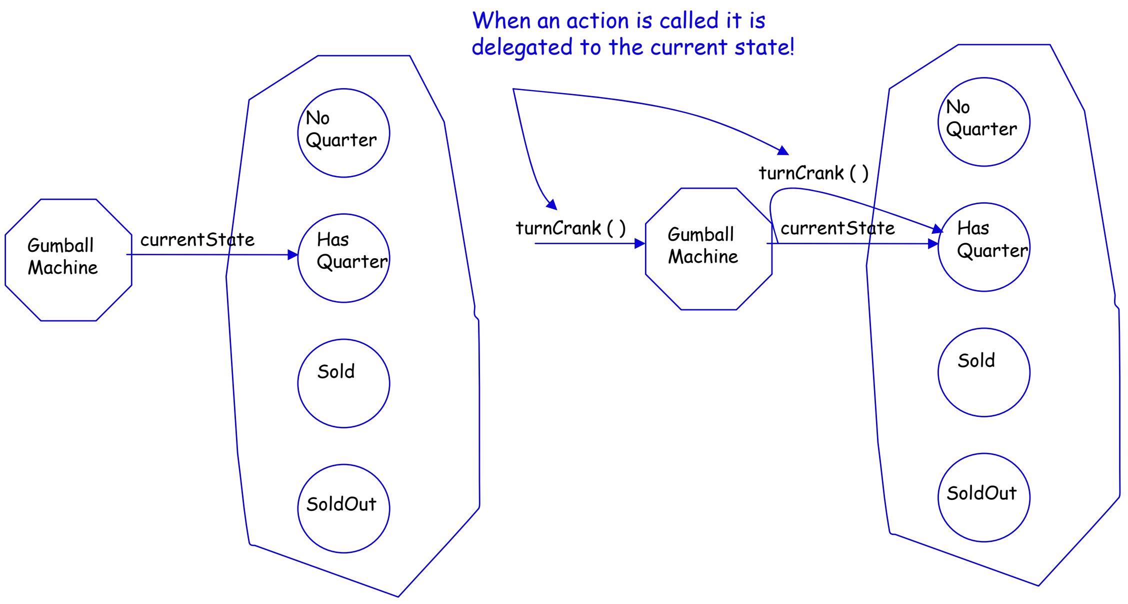
```
public class SoldState implements State {
    // constructor and instance variables here

    public void insertQuarter ( ) {
        System.out.println("Please wait…we are already getting 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 ( ) );
        }
    }
```

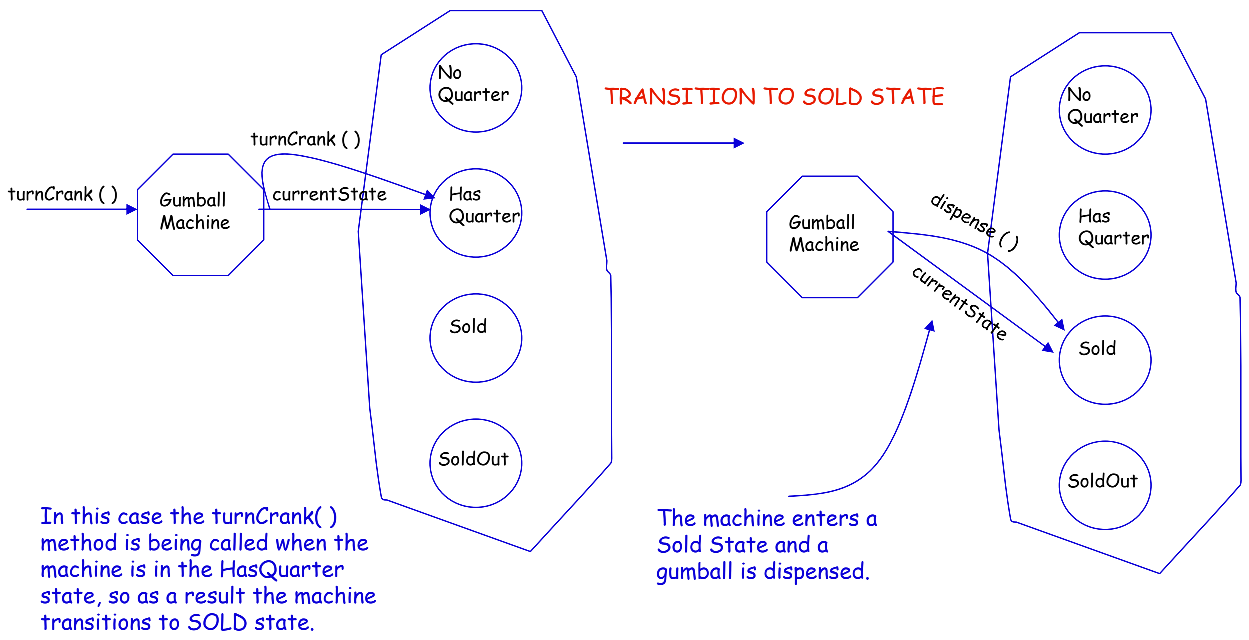
Here's where the work begins.

What have we done 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, yet left the Gumball Machine open for extension by adding new state classes,
- Created a code base and class structure that maps more closely to what is needed and is easier to read and understand.

The State Behavior….




turnCrank ( )
Gumball Machine
turnCrank ( )
currentState
No Quarter
Has Quarter
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 SOLD state.
TRANSITION TO SOLD STATE
Gumball Machine
dispense ( )
currentState
No Quarter
Has Quarter
Sold
SoldOut
The machine enters a Sold State and a gumball is dispensed.

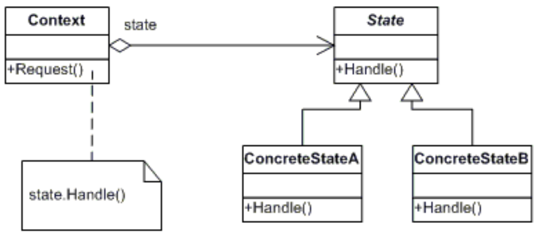
What do you need to do to now implement the 1 in 10 Gumball game?

Summary

- 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.
- 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.

Wait a sec….

What is this diagram familiar to?



In fact, the class diagram for the State is EXACTLY the same that for the Strategy pattern.

State vs Strategy

• The difference between the State and Strategy patterns is one of intent.

• They are both examples of Composition with Delegation.

State:

- Set of behaviors encapsulated in state objects; at any time the context is delegating to one of those states. Over time, the current states changes across the set of state objects to reflect the internal state of the context, so the context's behavior changes over time. Client knows very little, if anything, about the state objects.

- Alternative to putting lots of conditionals in your context: you can simply change the state object in the context to change its behavior!

Strategy:

- Client usually specifies the strategy object that the context is composed with. While the pattern provides the flexibility to change the strategy object at runtime, there is typically one strategy object that is most appropriate for a context object.

- Flexible alternative to subclassing: if you use inheritance to define the behavior of a class, you are stuck with it even if you need to change it. With Strategy you can change the behavior by composing with different objects!