Remember your software should be open for extensions and closed for modifications (**open-closed principle)**

Answer the following questions:

1. Does this code adhere to Open Closed principle? Why or why not?
2. Is this design object-oriented? Why or why not?
3. Are the state transitions explicit? Or are they buried in the middle of a bunch of conditional statements?
4. One of the fundamental principles of design patterns is: *Find what varies and encapsulate it*. Did we do that in the current design? If not, what did we not encapsulate?
5. Do you think further additions will likely to cause bugs in the working code?

**Suggestions:**

Refactor the code so it is easy to maintain and modify

Localize the behavior of each state so that if we make changes to one state, we don’t run the risk of messing up the other code.

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

Then Gumball machine can just delegate to the state object that represents the current state – remember one of the principles of design patterns is: *favor composition over inheritance*

Will it make it easier to add new states? - we will 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 may be change a few transitions here and there.

**Outline:**

Instead of maintaining our existing code, we are going to rework it to encapsulate state objects in their own classes and then delegate to the current state when an action occurs. Let us first start with **GumballMachine\_Ver1 code and fix it.**

Step 1: First, define a State interface that contains a method for every action in the Gumball machine.

/\* Here we are putting all the behavior of a state in this interface. That way, the

concrete classes localize the behavior thus making things easier to change and

understand \*/

**public** **interface** State {

**public** **abstract** **void** insertQuarter();

**public** **abstract** **void** ejectQuarter();

**public** **abstract** **void** turnCrank();

**public** **abstract** **void** dispense();

}

Step 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. I have given you NoQuarterState.java code – you have to refactor the other classes from existing code

// Note that we are implementing the behavior that are

// appropriate for the NoQuarterState we are in. In some cases,

// this behavior includes moving the GumballMachine to

// a new state.

**public** **class** NoQuarterState **implements** State {

GumballMachine machine;

// we get passed a reference to the GumballMachine

// through the constructor. We just stash this

// in an instance variable.

**public** NoQuarterState(GumballMachine machine) {

**this**.machine = machine;

}

// if someone inserts a quarter, we print a message saying

// that the quarter was accepted, and then change the

// machine's state to HasQuarterState

**public** **void** insertQuarter() {

System.***out***.println("You inserted a quarter");

machine.setState(machine.getHasQuarterState() );

}

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

**public** **void** ejectQuarter() {

System.***out***.println("You haven't inserted a quarter");

}

// you can't get a gumball if you don't pay us

**public** **void** turnCrank() {

System.***out***.println("You turned, but there is no quarter");

}

// we can't be dispensing gumballs without payment

**public** **void** dispense() {

System.***out***.println("You need to pay first");

}

}

Step 3. Get rid of all our conditional code and instead delegate to the state class to do the work for us.

**public** **class** GumballMachine {

// Here are all the states again

**private** State soldOutState;

**private** State noQuarterState;

**private** State hasQuarterState;

**private** State soldState;

**private** State winnerState;

// and the current state of the machine

**private** State currentState = soldOutState;

// count holds the number of gumballs. Initially the

// machine is empty

**int** count = 0;

/\*\* constructor takes the initial number of gumballs

\* and stores it in the instance variable.

\* It also creates the State instances, one of each

\* **@param** numberGumballs the initial count of gumballs

\*/

**public** GumballMachine(**int** numberGumballs) {

// WRITE CODE HERE TO INITILIZE THE State instance variables

**this**.count = numberGumballs;

// if there are more than 0 gumballs, we set the state

// to the NoQuarterState

**if** (count > 0) {

currentState = noQuarterState;

}

}

// Actions are very easy to implement now. We just delegate

// to the current state

**public** **void** insertQuarter() {

currentState.insertQuarter();

}

**public** **void** ejectQuarter() {

currentState.ejectQuarter();

}

// Note that we don't need an action method for

// dispense() in GumballMachine because it is just an

// internal action; a user cannot ask the machine

// to dispense directly. But we do call dispense()

// on the State object from the turnCrank() method.

**public** **void** turnCrank() {

currentState.turnCrank();

currentState.dispense();

}

**public** **void** setState(State s) {

currentState = s;

}

// this is a helper method that releases the ball and

// decrements the count instance variable

**public** **void** releaseBall() {

System.***out***.println("A gumball comes rolling out of the slot...");

**if** (count != 0)

count--;

}

**public** **int** getCount() {

**return** count;

}

**public** State getNoQuarterState() {

**return** noQuarterState;

}

**public** State getSoldOutState() {

**return** soldOutState;

}

**public** State getSoldState() {

**return** soldState;

}

**public** State getHasQuarterState() {

**return** hasQuarterState;

}

**public** State getWinnerState() {

**return** winnerState;

}

**public** String toString() {

String s = "\nMighty Gumball, Inc.\n";

s += "Java enabled Standing Gumball Model #2014\n";

s += "Inventory: " + count + " gumballs\n";

**if** (count == 0)

s += "Machine is sold out\n";

**else**

s += "Machine is waiting for quarter\n";

**return** s;

}

}

Here is the UML diagram:

**State**

<<interface>>

insertQuarter()

ejectQuarter()

turnCrank()

dispense()

**SoldState NoQuarterState HasQuarterState SoldoutState**

insertQuarter() insertQuarter() insertQuarter() insertQuarter()

ejectQuarter() ejectQuarter() ejectQuarter() ejectQuarter()

turnCrank() turnCrank() turnCrank() turnCrank()

dispense() dispense() dispense() dispense()

Does this diagram look familiar? This is called **State design pattern**. Strategy pattern and State pattern are twins separated at birth.

Context State

request() handle()

state.handle() ConcreteStateA ConcreteStateB

handle() handle()

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

Whenever a request is made to the context it is delegated to the state to handle.

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

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.

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

Since the pattern encapsulates state into separate classes and delegates to the object representing the current state, we know that behavior changes along with internal state. In the Gumball machine 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 is in the HasQuarterState (the machine rejects the quarter).

What does it mean for an object to “appear to change its class?” Think about it from the perspective of the client: if an object you are 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.

The class diagrams for Strategy and State Patterns 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 a 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, 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.

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 are 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.