Design Patterns

Kris.Hermans@pxl.be

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

1

State

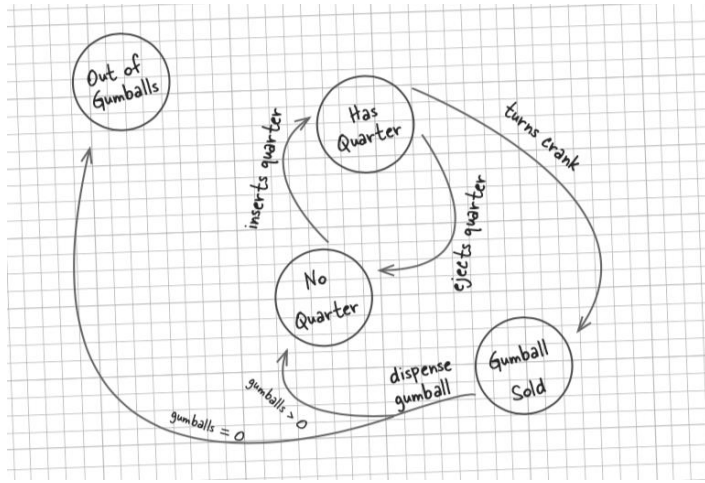
Twin brother of the
Strategy pattern …

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

2

Write a gumball machine



PXL IT

3

GumballMachine: states (naive)

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.

PXL IT

4

GumballMachine: insertQuarter

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

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.

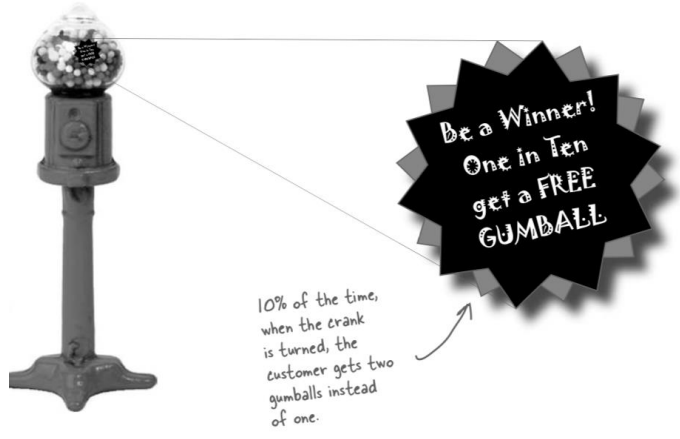
5

Observations

- Using constants (numbers) for all possible states
- Define one state variable (a number)
- insertQuarter, ejectQuarter, turnCrank, dispense, … have a lot of if(switch)-statements to check its state a behave accordingly
- Not ready for changes

6

New feature



PXL IT

7

Implementation hell...

```
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...

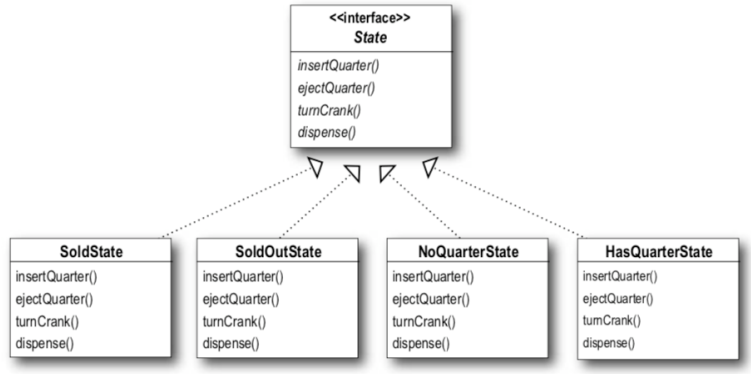
... but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

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.

PXL IT

8

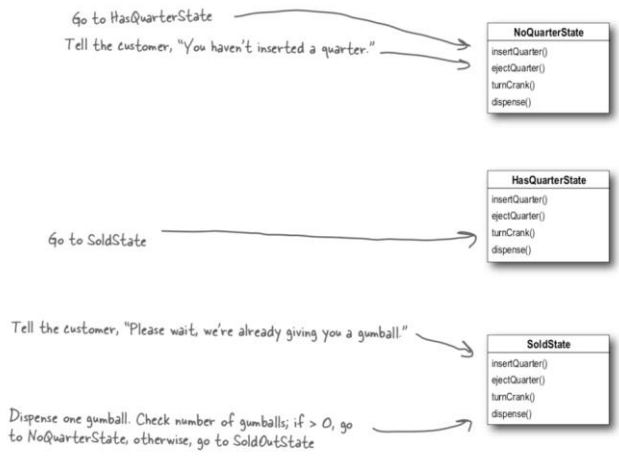
State introduced



9

Implement each state

- Improves testability



10

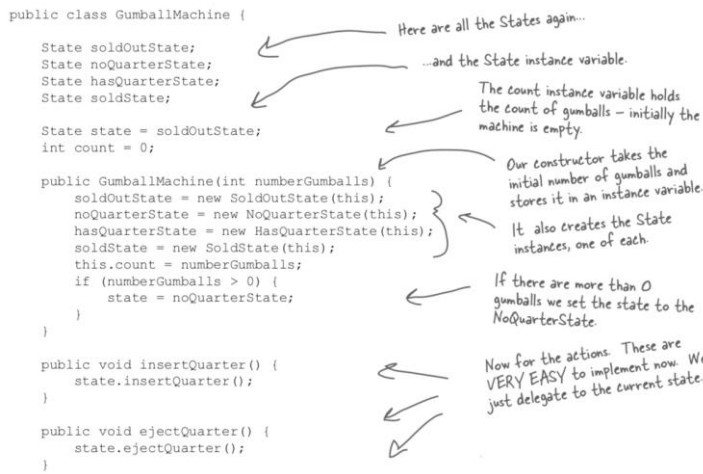
NoQuarterState


```
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");
    }
}
```

PXL IT

11

GumballMachine reworked




```
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 SoldState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        }
    }

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

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

PXL IT

12

GumballMachine reworked

```
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...
}
```

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.

13

New Implementation

- Each behavior of each state in a separate class
- All troublesome if-statements are removed
- Each state is closed for modification, but GumballMachine is open for extension by adding new state classes (eg WinnerState)
  - Open-Closed Principle (SOLID)
- Easier to understand because each class has its own reponsibility
  - Single Responsibility (SOLID)

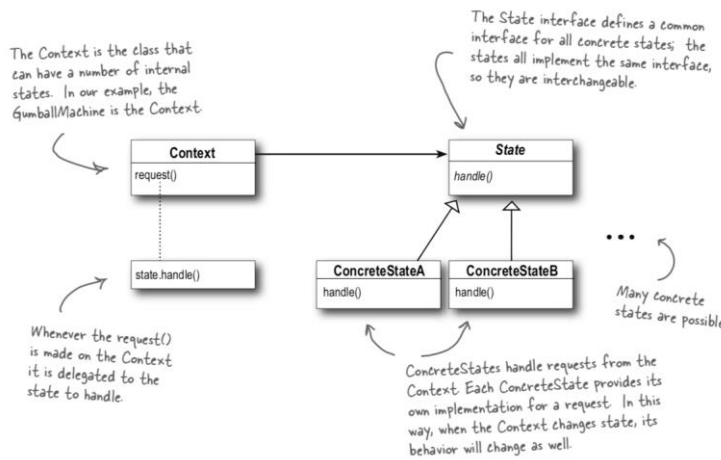
14

The State Pattern

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

15

The State Pattern



16

Shares with Strategy exactly the same UML diagram!

- Difference in INTENT
  - Strategy: client specifies a strategy object that is most appropriate. It is not changed often at run-time.
  - State: delegate behavior to state objects and change them internally at run-time. The client knows very little about state objects themself.

17