The State Design Pattern

Object Oriented Software Engineering
Lecture 8

Dr. Graham McDonald
graham.mcdonald@glasgow.ac.uk

Introduction

- State design pattern is one of the behavioral design patterns.
- This pattern is used when an object changes its behavior based on its state.

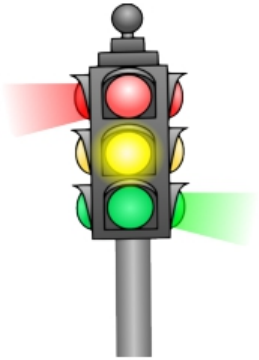
Learning outcomes

Managing Object States

The State Machine

When writing code for certain systems, we often need to build a state machine.

That is, we write code that's based on a set of states and transitions.

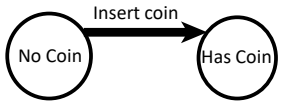


A Simple State Machine

A controller for a gumball machine

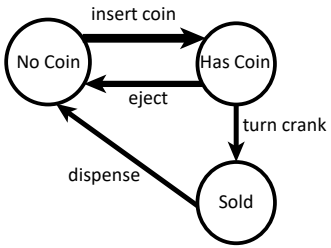
A Simple State Machine

A controller for a gumball machine



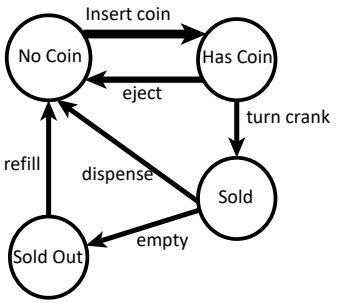
A Simple State Machine

A controller for a gumball machine



A Simple State Machine

A controller for a gumball machine



How do we implement this state machine?

States as Constants

A controller for a gumball machine

A common way is to implement the state machine as a set of state constants.

```
final static int SOLD_OUT =0;
final static int NO_COIN =1;
final static int HAS_COIN =3;
final static int SOLD =4;

int state = NO_COIN;
```

States as Constants

A controller for a gumball machine

Then for each transition you write a method.

```
public void insertCoin(){
  if(state == HAS_COIN)
     print("ERROR, already has coin");
  else if(state == SOLD_OUT)
     print("ERROR, machine sold out");
  else if(state == SOLD)
     print("ERROR, Already getting your gumball");
  else if(state == NO_COIN)
     state = HAS_COIN;
}
```

States as Constants

A controller for a gumball machine

Then for each transition you write a method.

```
public void turnCrank(){
  if(state == SOLD)
     print("ERROR, can't turn crank twice");
  else if(state == SOLD_OUT)
     print("ERROR, machine sold out");
  else if(state == NO_COIN)
     print("ERROR, insert coin first");
  else if(state == HAS_COIN){
     state = SOLD;
  dispense();
  }
}
```

States as Constants

A controller for a gumball machine

So we need one method per transition

```
public void turnCrank();
public void insertCoin();
public void ejectCoin();
public void dispense();
public void refill();
public void empty();
```

We can end up writing lots of code to handle states that either don't make sense or that cause a lot of problems. Implementing too many closely related transitions can also end up with code duplication and sometimes developer fatigue.

The State Design Pattern

Object Oriented Software Engineering
Lecture 8: Part 2

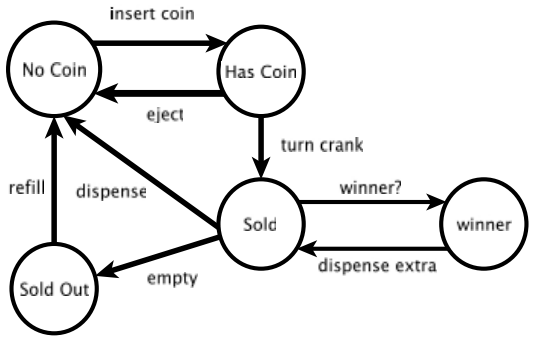
Dr. Graham McDonald
graham.mcdonald@glasgow.ac.uk

States as Constants

A controller for a gumball machine

Assume there is a new feature request from your stakeholder to add a promotion where every customer has a one in ten chance of winning a gumball.

To achieve this we need additional states and transitions



States as Constants

A controller for a gumball machine

But, we need to carry out three major code refactoring to accommodate this new feature request in our implementation

1. we'll first create another state as a constant.

```
final static int SOLD_OUT =0;
final static int NO_COIN =1;
final static int HAS_COIN =3;
final static int SOLD =4;
final static int WINNER =5;

int state = NO_COIN;
```

States as Constants

A controller for a gumball machine

But, we need to carry out three major code refactoring to accommodate this new feature request in our implementation

2. Then for the new state, we have to open up every existing method and to add a new conditional for that state.

```
public void turnCrank();
public void insertCoin();
public void ejectCoin();
public void dispense();
public void refill();
public void empty();
```

States as Constants

A controller for a gumball machine

But, we need to carry out three major code refactoring to accommodate this new feature request in our implementation

3. Finally, we have to write two new transition methods.

```
...
public void isWinner();
public void dispenseExtra();
```

States as Constants

A controller for a gumball machine

So with this design, our code is likely to get more complex with each additional feature request.

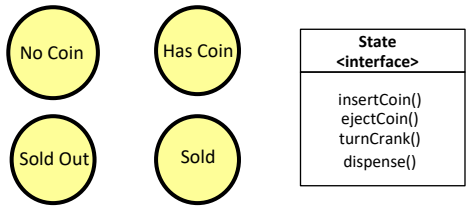
- The outcome is really not object oriented
- Any additions require many changes to code
- Difficult to understand all the states and transitions
- Violates the **open-closeed software engineering design principle:**
  - Any additions causes us to touch almost all our code.

Each State as an Object

A controller for a gumball machine

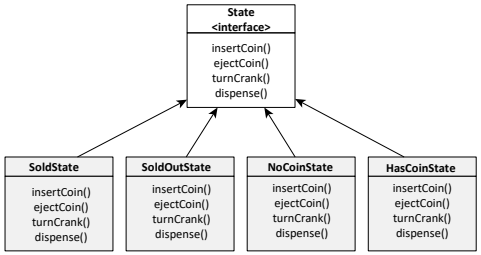
Alternative Design

1. Take all the states and convert them into objects.
2. Then make all the objects implement a common State interface



Each State as an Object

A controller for a gumball machine: An alternative Design



Advantages:

- The interface has a method for each state transition
- If we have a new state feature request, then all we have to do is to create a new class that implements the State interface. Also if there are new transitions, we just add new method across the classes.
- We are not touching existing code.

Each State as an Object

A controller for a gumball machine: An alternative Design

```
public class GumballMachine {
  final static int SOLD_OUT = 0;
  final static int NO_COIN = 1;
  final static int HAS_COIN = 2;
  final static int SOLD = 3;

  int state = SOLD_OUT;

  ...

  public void insertCoin() {
     if (state == HAS_COIN) {
       System.out.println("You can't insert another coin");
     }
     else if (state == NO_COIN) {
          state = HAS_COIN;
          System.out.println("You inserted a coin");
     }
     ...
  }
}
```

```
public interface State {

   public void insertCoin();
   public void ejectCoin();
   public void turnCrank();
   public void dispense();
   public void refill();
}
```

1

```
public class GumballMachine {
   State soldOutState;
   State noCoinState;
   State hasCoinState;
   State soldState;
   ...

   State state = noCoinState;

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

2

Download code from gumball_machine.zip on Moodle

State Pattern

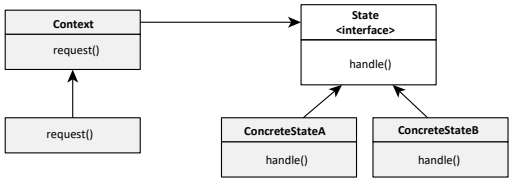


Figure: Class Diagram for the State Pattern

Context is the class that manages all the various states

Context delegates the request it receives to a state.

State Pattern

Definition:

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

Summary

State Pattern Key points:

- The pattern encapsulates state into separate classes.
- The context delegates to the current state to handle request.
- Once a request is handled, the current state may change.
- Each state has a different behaviour.
- Advantages:

  We can encapsulate what varies
  We favour composition over inheritance.
  Keep a class closed for modifications but open for extension