Software Design and Architecture

State Pattern

Design principles

High-level principles

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution Principle
- **I**nterface Segregation
- **D**ependency Inversion

Low-level principles

- Encapsulate what varies
- Program to interfaces, not implementations
- Favor composition over inheritance
- Strive for loose coupling

State

Behavioral Patterns

- observer
- decorator
- strategy
- command
- template
- null object
- **state**

Creational Patterns

- factory method
- abstract factory
- singleton

Structural Patterns

- adapter
- facade

Problem

Methods that have large, multipart conditional statements that depend on the object's state are difficult to maintain and extend

Example

```
static int A=1;
static int B=2;
static int C=3;

private int state;

void DoSomething() {
    if (state==A) {
        // do this
    }
    else if (state==B) {
        state=A;
    }
    else if (state==C) {
        state=A;
    }
}
```

"Classes should be open for extension, but closed for modification"

```
void DoThis() {
        if (state==A) {
            // do this
        }
        else if (state==B) {
            state=A;
        }
        else if (state==C) {
            state=A;
        }
    }
```

works but difficult to extend

```
    static int D=4;
```

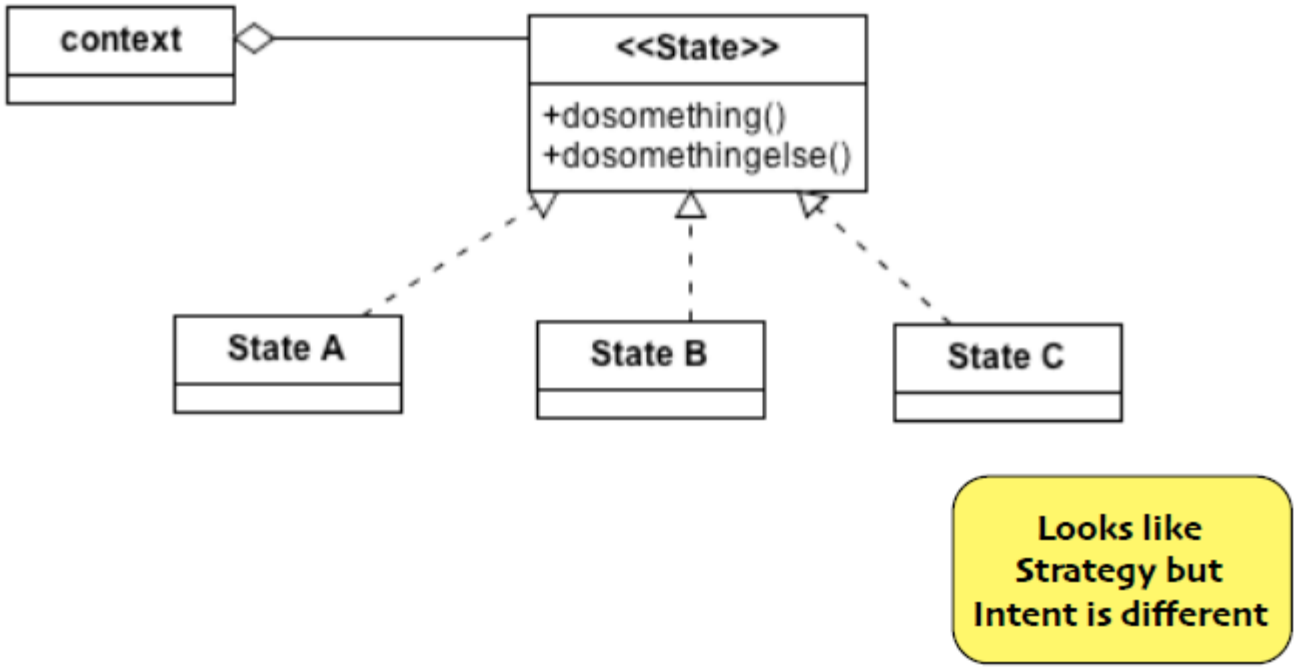
State Pattern

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

If we associate a class with behavior, then

- since the state pattern allows an object to change its behavior
- it will seem as if the object is an instance of a different class each time it changes state

Class Diagram



State Interface

```
public interface State {
    public void doThis();
    public void doThat();
}
```

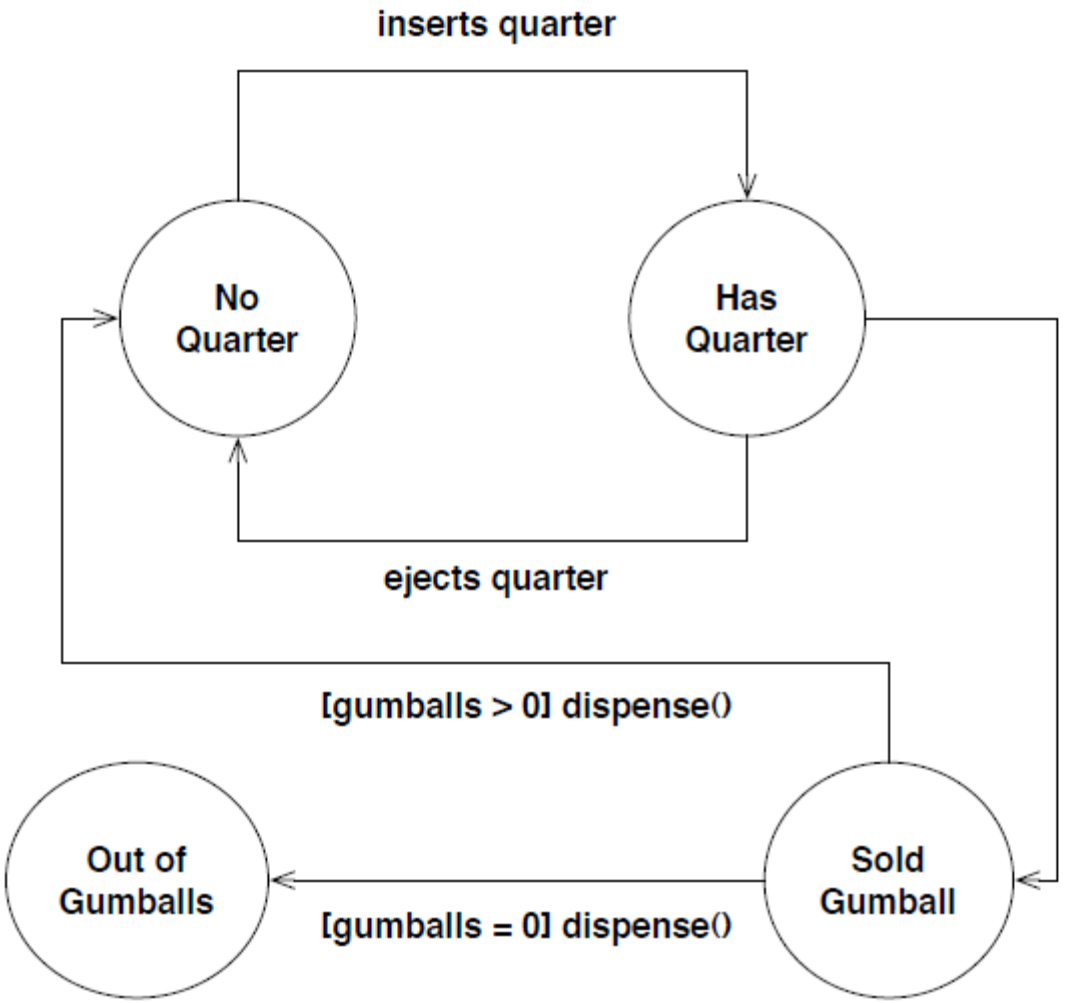
State_A

```
public class State_A implements State {

    StateContext statecontext;

    public State_A(StateContext statecontext) {
        this.statecontext=statecontext;
    }

    public void doThat() {
        statecontext.setState(statecontext.getState_B());
    }

    public void doThis() {
    }

}
```

context

```
public class StateContext {
     State State_A;
     State State_B;
     State myState;

public StateContext() {
          State_A = new State_A(this);
          State_B = new State_B(this);
          myState = State_A;
     }
 public void setState(State stateName) {
          this.myState = stateName;
     }

public State getState_A() {
   return State_A;
}

public State getState_B() {
        return State_B;
   }

}
```

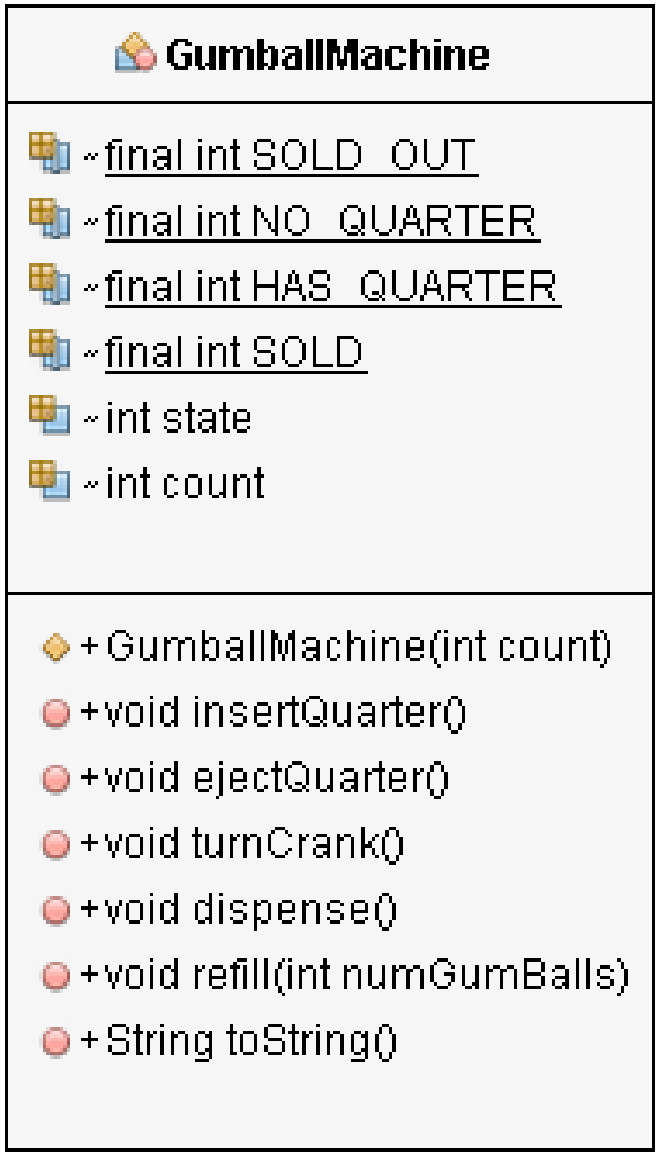
Example: State Machines for Gumball Machines



Each circle represents a state that the gumball machine can be in.

Each label corresponds to an event (method call) that can occur on the object

Example: State Machines for Gumball Machines



Problems with the First Design

- Does not support **Open Closed Principle**
  - A change to the state machine requires a change to the original class
    - You can't place new state machine behavior in an extension of the original class

- The design is not very object-oriented: indeed no objects at all except for the one that represents the state machine, in our case GumballMachine.

- State transitions are not explicit; they are hidden amongst a ton of conditional code

- We have not "**encapsulated what varies**"

Design with State Pattern

1. Create a State interface that has one method per state transition

2. Create one class per state in state machine. Each such class implements the State interface and provides the correct behavior for each action in that state

Design with State Pattern: State Interface

```
public interface State {
  public void insertQuarter();
  public void ejectQuarter();
  public void turnCrank();
  public void dispense();
}
```

Design with State Pattern: NoQuarterState Class

```
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");
  }

  public String toString() {
   return "waiting for quarter";
  }
}
```

Design with State Pattern: HasQuarterState Class

```
import java.util.Random;

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");
    }

  public String toString() {
    return "waiting for turn of crank";
  }
}
```

Design with State Pattern: SoldState Class

```
public class SoldState implements State {

    GumballMachine gumballMachine;

    public SoldState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

  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!");
  }

  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());
    }
  }

  public String toString() {
    return "dispensing a gumball";
  }
}
```

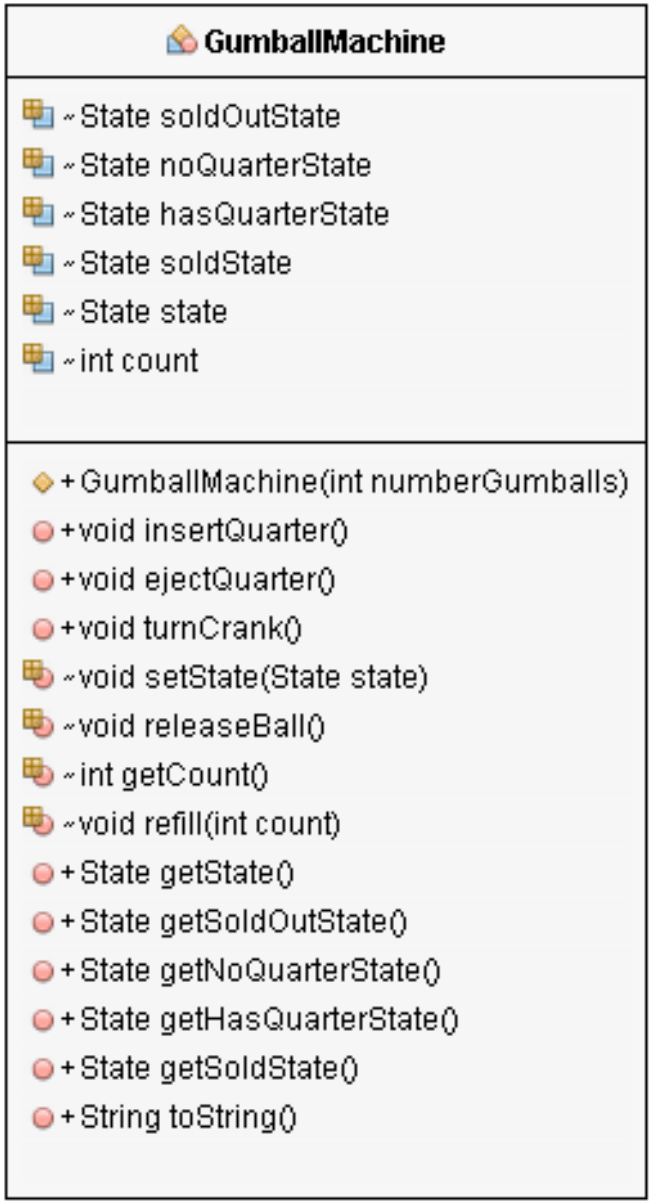
Design with State Pattern: SoldOutState Class

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

    public SoldOutState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

  public void insertQuarter() {
    System.out.println("You can't insert a quarter, the machine is sold out");
  }

  public void ejectQuarter() {
    System.out.println("You can't eject, you haven't inserted a quarter yet");
  }

  public void turnCrank() {
    System.out.println("You turned, but there are no gumballs");
  }

  public void dispense() {
    System.out.println("No gumball dispensed");
  }

  public String toString() {
    return "sold out";
  }
}
```

Design with State Pattern

3. Change GumballMachine class to point at an instance of one of the State implementations and delegate all calls to that class. An action may change the current state of the GumballMachine by making it point at a different State implementation

Design with State Pattern: GumballMachine Class



Implement a New State: WinnerState

- Add a new State implementation: WinnerState
  - Exactly like SoldState except that its dispense() method will dispense two gumballs from the machine, checking to make sure that the gumball machine has at least two gumballs
    - You can have WinnerState be a subclass of SoldState and just override the dispense() method

- Update HasQuarterState to generate random number between 1 and 10
  - if number == 1, then switch to an instance of WinnerState else an instance of SoldState

Design with State Pattern: WinnerState Class

```
public class WinnerState extends SoldState {

  public WinnerState(GumballMachine gumballMachine) {
    super(gumballMachine);
  }

  public void dispense() {
    System.out.println("YOU'RE A WINNER! Two gumballs for your quarter.");
    gumballMachine.releaseBall();
    if (gumballMachine.getCount() == 0) {
      gumballMachine.setState(gumballMachine.getSoldOutState());
    } else {
      gumballMachine.releaseBall();
      if (gumballMachine.getCount() > 0) {
        gumballMachine.setState(gumballMachine.getNoQuarterState());
      } else {
        System.out.println("Oops, out of gumballs!");
        gumballMachine.setState(gumballMachine.getSoldOutState());
      }
    }
  }
```

Design with State Pattern: GumballMachine Class