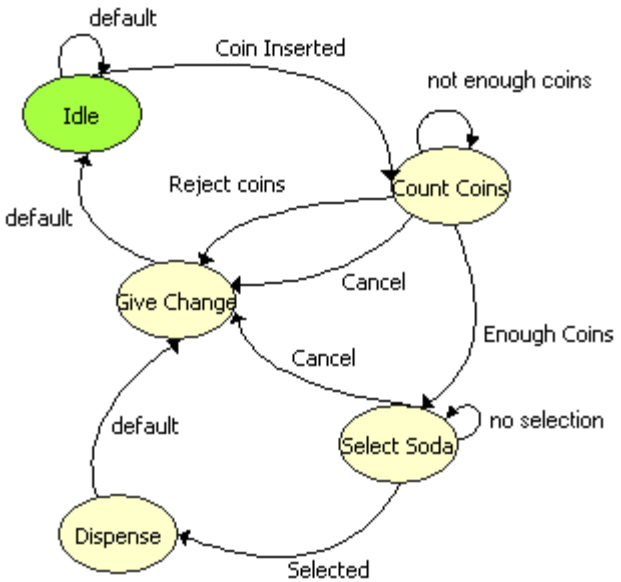
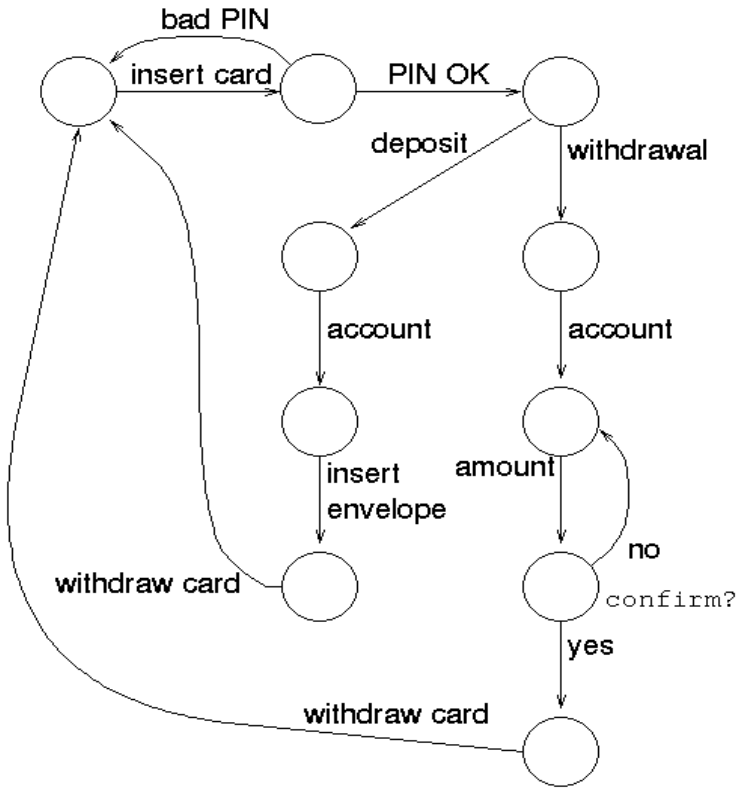
State Pattern



State Pattern



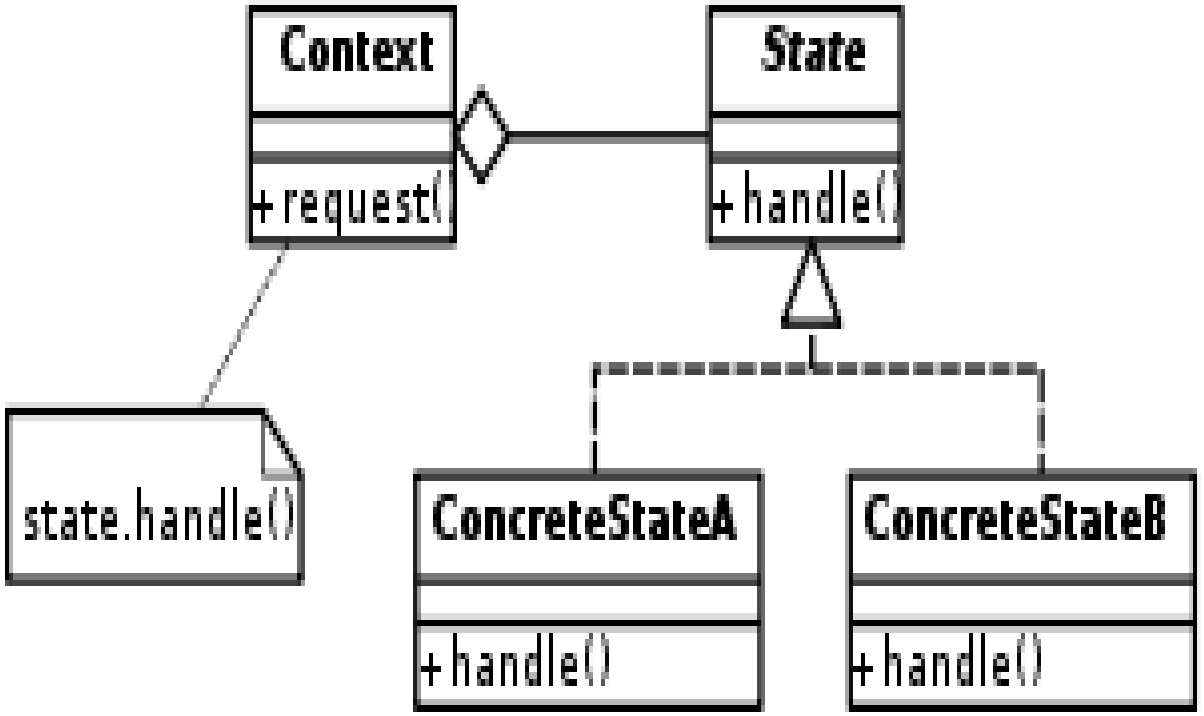
Problem

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

State Pattern

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

Class Diagram



State Pattern

- A behavioral pattern used to represent the state of an object
- A clean way for an object to partially change its type at runtime
- Typically involves an interface to represent state behaviors
- Concrete state classes implement interface (or extend abstract class)
- These classes **usually** are in charge of modifying state (sometimes context can do this)
- Context class *typically* contains references to all the different types of state objects as well as state object that represents current state

State is Similar to Strategy

- Class diagram is very similar
- State Pattern behavior changes are built in (as defined by state transitions)
- Strategy Pattern typically requires an object to be specified that contains desired behavior when creating a client object (QuackBehavior/FlyBehavior when a Duck was created from chapter 1 of HFDP)

Sample Code: State Interface

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

Concrete State Class

```
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() {
    }
}
```

State Context Class

```
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;
    }
}
```

State Pattern Summary

- State Pattern represents state as a full-blown class
- 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 relative to that state
- State transitions can be controlled by the State classes (most common) or by the Context class(es)
- State Pattern will increase the number of classes in your design, but avoids monolithic classes/methods
- State classes may be shared among Context instances for efficiency (thus they can be made static)

Gumball Example From HFDP

State Interface

```
import java.io.*;

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

HasQuarterState

```
import java.util.Random;

public class HasQuarterState implements State {
   Random randomWinner = new Random(System.currentTimeMillis());
   transient 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...");
      int winner = randomWinner.nextInt(10);
      if (winner == 0) {
         gumballMachine.setState(gumballMachine.getWinnerState());
      } else {
         gumballMachine.setState(gumballMachine.getSoldState());
      }    }

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

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

NoQuarterState

```
public class NoQuarterState implements State {
    transient 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";
   }
}
```

SoldOutState

```
public class SoldOutState implements State {
    transient 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";
    }
}
```

SoldState

```
public class SoldState implements State {
    transient 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";
   }}
```

WinnerState

```
public class WinnerState implements State {
    transient GumballMachine gumballMachine;

    public WinnerState(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("Please wait, we're already giving you a Gumball")    }

   public void turnCrank() {
      System.out.println("Turning again doesn't get you another gumball!");    }

  public void dispense() {
      System.out.println("YOU'RE A WINNER! You get 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());
         }      }   }

   public String toString() {
      return "despensing two gumballs for your quarter, because YOU'RE A WINNER!";
   }}
```

GumballMachine

```
import java.rmi.*;
import java.rmi.server.*;

public class GumballMachine
      extends UnicastRemoteObject implements GumballMachineRemote
{
   State soldOutState;
   State noQuarterState;
   State hasQuarterState;
   State soldState;
   State winnerState;

   State state = soldOutState;
   int count = 0;
   String location;

   public GumballMachine(String location, int numberGumballs) throws RemoteException {
      soldOutState = new SoldOutState(this);
      noQuarterState = new NoQuarterState(this);
      hasQuarterState = new HasQuarterState(this);
      soldState = new SoldState(this);
      winnerState = new WinnerState(this);

      this.count = numberGumballs;
      if (numberGumballs > 0) {
         state = noQuarterState;
      }
      this.location = location;
      System.out.println("gumball machine initialized");    }
```

GumballMachine

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

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

    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }


    public String toString() {
        StringBuffer result = new StringBuffer();
        result.append("\nMighty Gumball, Inc.");
        result.append("\nJava-enabled Standing Gumball Model #2004");
        result.append("\nInventory: " + count + " gumball");
        if (count != 1) {
            result.append("s");
        }
        result.append("\n");
        result.append("Machine is " + state + "\n");
        return result.toString();
    }
}
```

GumballMachine

```
void releaseBall() {
      System.out.println("A gumball comes rolling out the slot...");
      if (count != 0) {
         count = count - 1;      }   }

   public void refill(int count) {
      this.count = count;
      state = noQuarterState;   }

   public int getCount() {
      return count;  }

    public State getState() {
        return state;    }

    public String getLocation() {
        return location;    }

    public State getSoldOutState() {
        return soldOutState;    }

    public State getNoQuarterState() {
        return noQuarterState;    }

    public State getHasQuarterState() {
        return hasQuarterState;    }

    public State getSoldState() {
        return soldState;    }

    public State getWinnerState() {
        return winnerState;    }
```