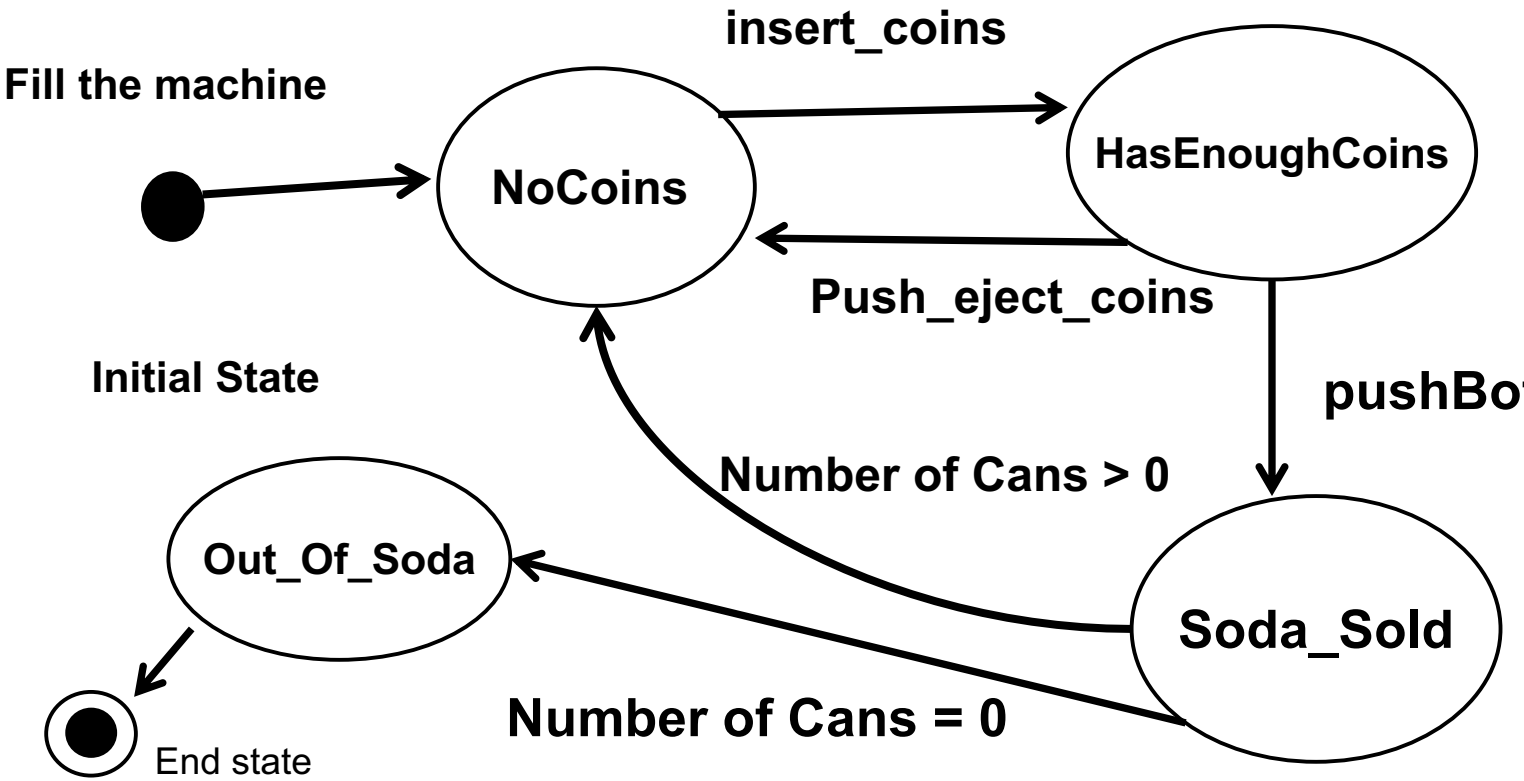
State Pattern

Problem

- You have objects with different states so that their behavior depends on their states.
- You need to implement State Machines to keep track of states and apply actions that change the states.
- **How to implement the "State Machines" ?**
  - Flexibility (be able to add or remove states, or change the transitions)
  - We do not want to have code duplication and hard coding of conditional code in one single class.

Example: Soda Vending Machine



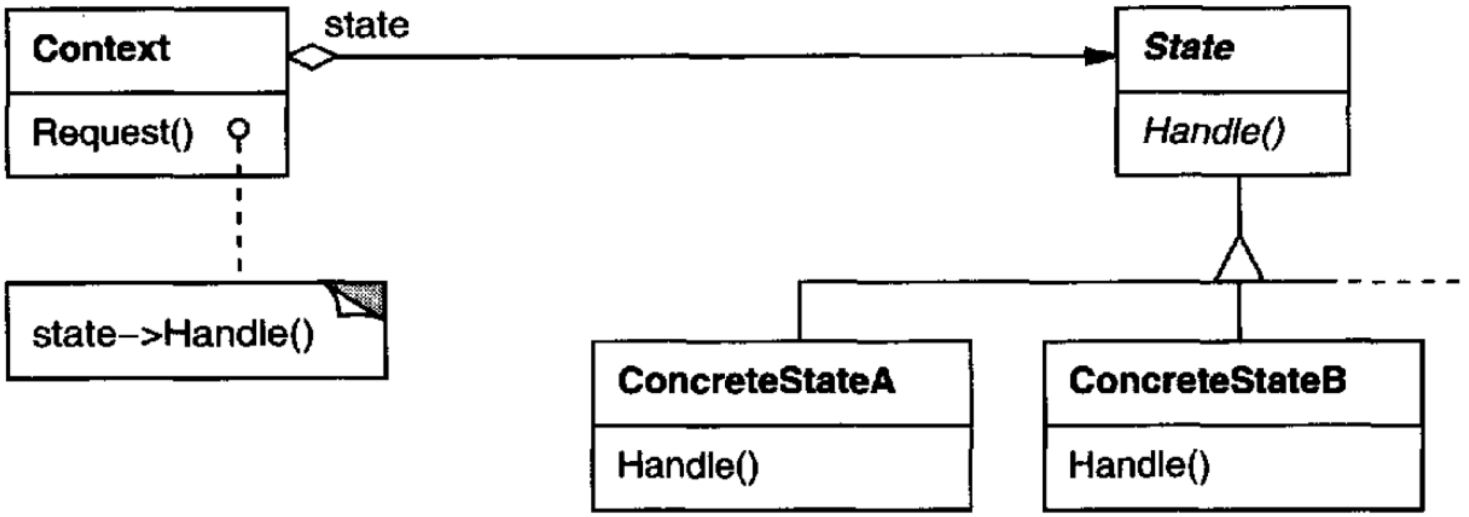
State Pattern

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

Solution: **3 main Steps**

1. First, define the **a State interface** that contains a method for every action in your state machine.
2. Then, 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.
3. Finally, get rid of all of our conditional code and instead **delegate to the state class to do the work**

Participants



- **Context**
  - defines the interface of interest to clients.
  - maintains an instance of a ConcreteState subclass that defines the current state.
- **State**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **ConcreteState** subclasses
  - each subclass implements a behavior associated with a state of the Context.

Implementation of Use Case – State Interface

```
public interface State {

        public void insertCoins();
        public void ejectCoins();
        public void pushButton();
        public void pushDispense();

        public void refill();
}
```

```
public class SodaMachine {

        State soldOutState;
        State noCointState;
        State hasEnoughCoinState;
        State soldState;

        State state;
        int count = 0;

        …
}
```

HasCoinState

```
public class HasCoinState implements State {

        SodaMachine sodaMachine;

        public HasCoinState(SodaMachine sodaMachine) {
                this. sodaMachine = sodaMachine;
        }

        public void insertCoin() {
                System.out.println("You can't insert another Dollar Coin");
        }
        public void ejectCoins() {
                System.out.println(”Coins returned");
                 sodaMachine.setState(sodaMachine.getNoQuarterState());
        }
        public void pushButton() {
                System.out.println("You pushed ...");
                 sodaMachine.setState(sodaMachine.getSoldState());
        }
        public void dispense() {
                  System.out.println("No Soda dispensed");
        }
  ....
  }
```

Soda Machine

```
public class SodaMachine {

        State soldOutState;
        State noCoinState;
        State hasCoinState;
        State soldState;

        State state;
        int count = 0;

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

        public void pushButton() {
                state.pushButton();
                state.dispense();
        }
        void releaseSodaCan() {
                System.out.println("A Soda Can comes out of the machine ...");
                if (count != 0) {
                        count = count - 1;
                }
        }
        void refill(int count) {
                this.count += count;
                System.out.println("The Soda Machine is just refilled;  It has now  " + this.count + " Soda Cans");
                state.refill();
        }
 }
```

TEST RUN

```
public class SodaMachine TestDrive {

public static void main(String[] args) {


SodaMachine sodaMachine = new SodaMachine(2);

                    System.out.println(sodaMachine);

                    sodaMachine.insertCoin();
                    sodaMachine.pushButton();
// one other Soda  can sold
                     sodaMachine. insertCoin();
                     sodaMachine. pushButton();
// one soda  can sold

                     sodaMachine. insertCoin();
                     sodaMachine. pushButton();
// out of Soda Cans
// Refill again with 5 cans
                     sodaMachine.refill(5);
                     sodaMachine. insertCoin();
                     sodaMachine. pushButton();

                    System.out.println(sodaMachine);
          }
}
```

When use the State Pattern

- You have an **object's behavior** that depends on its state, and it must change its behavior at run-time depending on that state.
- You have **operations** that *have large, multiple-part conditional statements that depend on the object's state*.
  - This state represented by one or more enumerated constants.
  - Often, several operations contain the **same conditional structure**.
  - You can use the state pattern to put each branch of the conditional in a separate class.
  - Using the state pattern you can treat the object's state as an object in its own right that can vary from other objects independently.

Consequences

- **The state pattern localizes state-specific behavior and partitions behavior** for different states.
  - All behavior associated with a particular state into one object
  - All state-specific code are in State subclasses, new states and transitions can be added easily by defining new subclasses.
- **State transitions are explicit.**
  - When an object defines its current state only in terms of internal data values, its state transitions have no explicit representation; they only show up as assignments to some variables.
  - When separate objects are used for different states, the transitions can be more explicit
  - **State objects can protect the Context from inconsistent internal states**, because state transitions are atomic from the Context's perspective

Consequences

- **State objects can be shared.**
  - If State objects have no instance variables — the case that the state they represent is **encoded entirely in their type** — then contexts can share a State object.
  - When states are shared, they are essentially flyweights with no intrinsic state, only behavior.

Summary

- The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- The solution of State pattern helps to implement state machines in a flexible manner.
- State Pattern is of type **Behavioral Patterns**

- **Related patterns:**
  - The Flyweight pattern explains when and how State objects can be shared.
  - State objects are often Singletons
  - Chain of Responsibility
  - Decorator