State Pattern

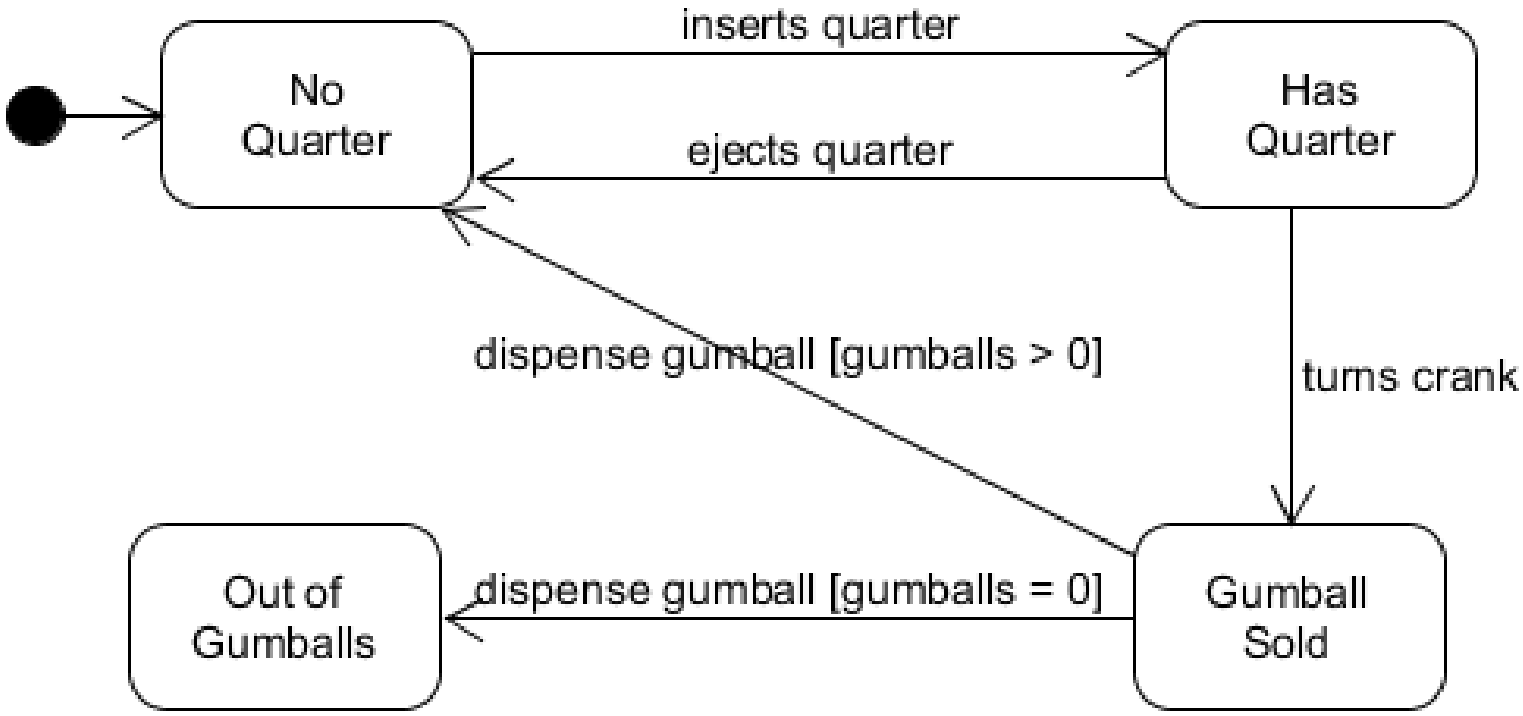
Contents

- State pattern discovery
- Structure and run-time mechanism of State pattern
- Similarity with Strategy pattern

State Pattern

- Purpose
  - Ties object circumstances to its behavior, allowing the object to behave in different ways based upon its internal state.
- Use When
  - The behavior of an object should be influenced by its state.
  - Complex conditions tie object behavior to its state.
  - Transitions between states need to be explicit.

States & Actions to Implement



Coding the States

```
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;
```

Writing the Code

```
public class GumballMachine {

    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;
    int count = 0;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) state = NO_QUARTER;
    }

    public void insertQuarter() {
        if (state == HAS_QUARTER) {
                ..
                ..

    }
    public void ejectQuarter() {
        if (state == HAS_QUARTER) {
                ..
                ..
```

Coding per Action

```
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");
        System.out.println("The machine is sold out");

    else if (state == SOLD)
        System.out.println("Please wait, already giving you a gumball");

    else if (state == NO_QUARTER) {
        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");
    }
}
```

Test Drive

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.ejectQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

                …
        }
}
```

A Change Request!

We think that by turning "gumball buying" into a game we can significantly increase our sales. We're going to put one of these stickers on every machine.
We're so glad we've got Java in the machines because this is going to be easy, right?

Be a Winner!
One in Ten
get a FREE
GUMBALL!

Messy state of things…

```
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() {
}

public void ejectQuarter() {
}

public void turnCrank() {
}

public void dispense() {
}
```

New Idea

- Encapsulate what varies
  - Put each state's behavior in its own class, then every state just implements its own actions
  - The Gumball Machine can delegate to the state object that represents the current state

- Favor composition over inheritance

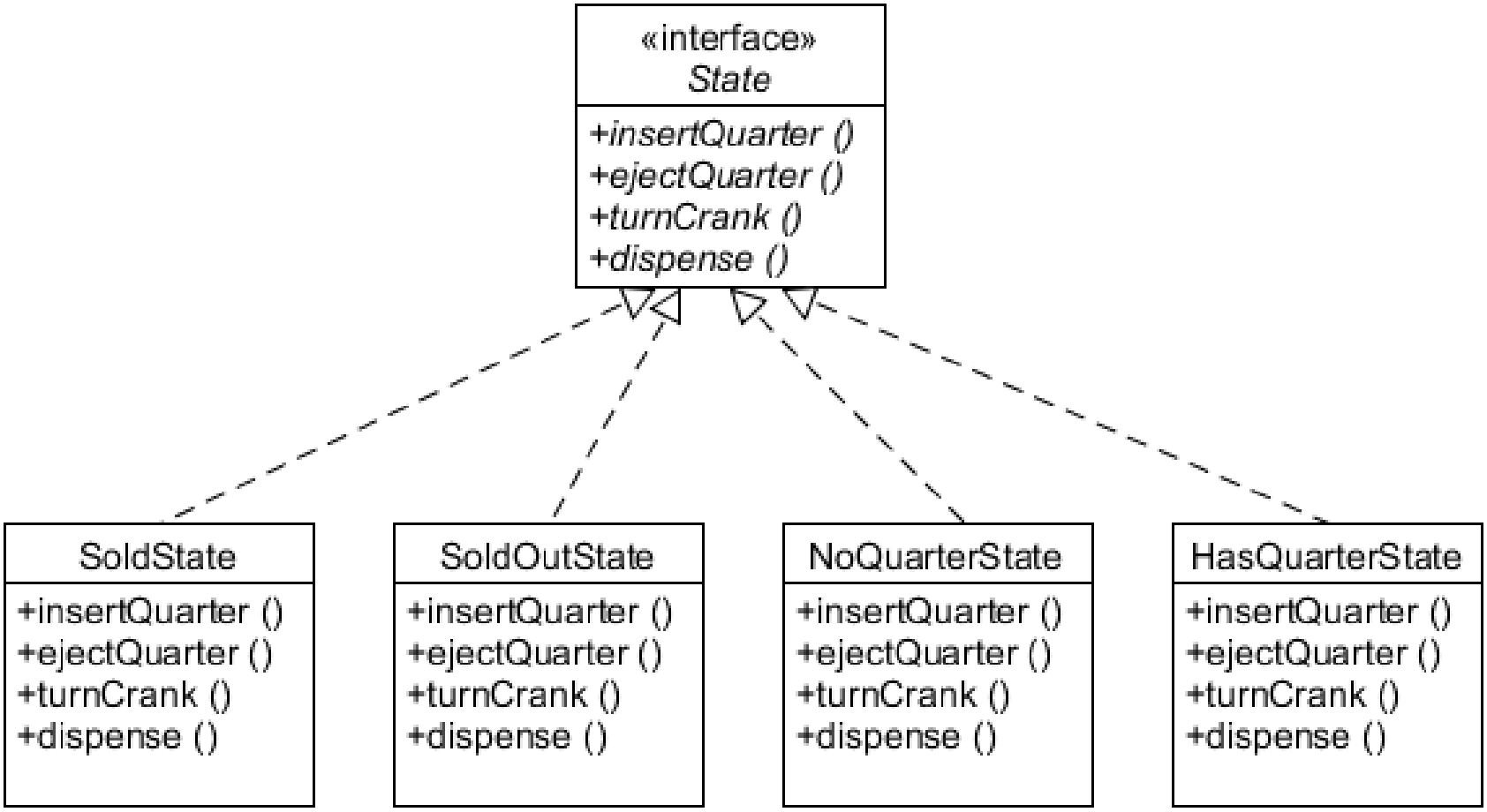
New Design

- Define a State interface that contains a method for every action in the Gumball Machine

- Implement a **State class** for every state of the machine.
  - responsible for behavior of the machine when it is in the corresponding state.

- Get rid of all of our conditional code and instead delegate to the state object to do the work for us

Defining State Interface and classes



Implementing 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(“No Gumball dispensed");
    }
}
```

Reworking the Gumball Machine

```
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;
        }
    }
}
```

Reworking the Gumball Machine

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

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

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

    public void dispense() {
        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;
    }
}
```

Implementing More States

```
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());
        gumballMachine.dispense();
    }
    public void dispense() {
        System.out.println(“No Gumball dispensed");
    }

}
```

Implementing More States

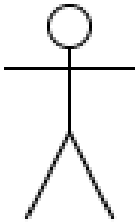
```
public class SoldState implements State {
    GumballMachine gumballMachine;

    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());
        }
    }
}
```

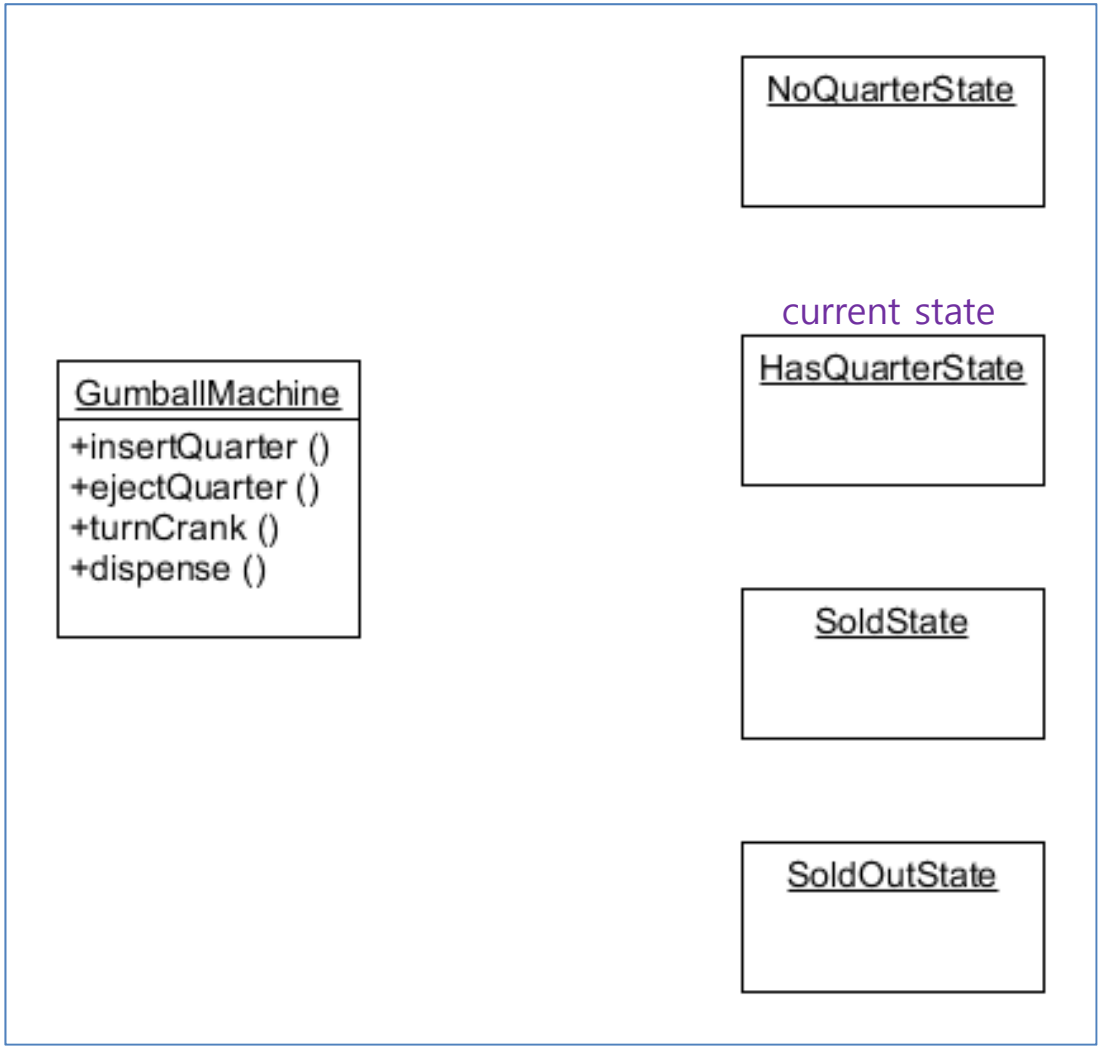
What We've Done So Far

- Structurally quite different from the original version, but functionally the same!

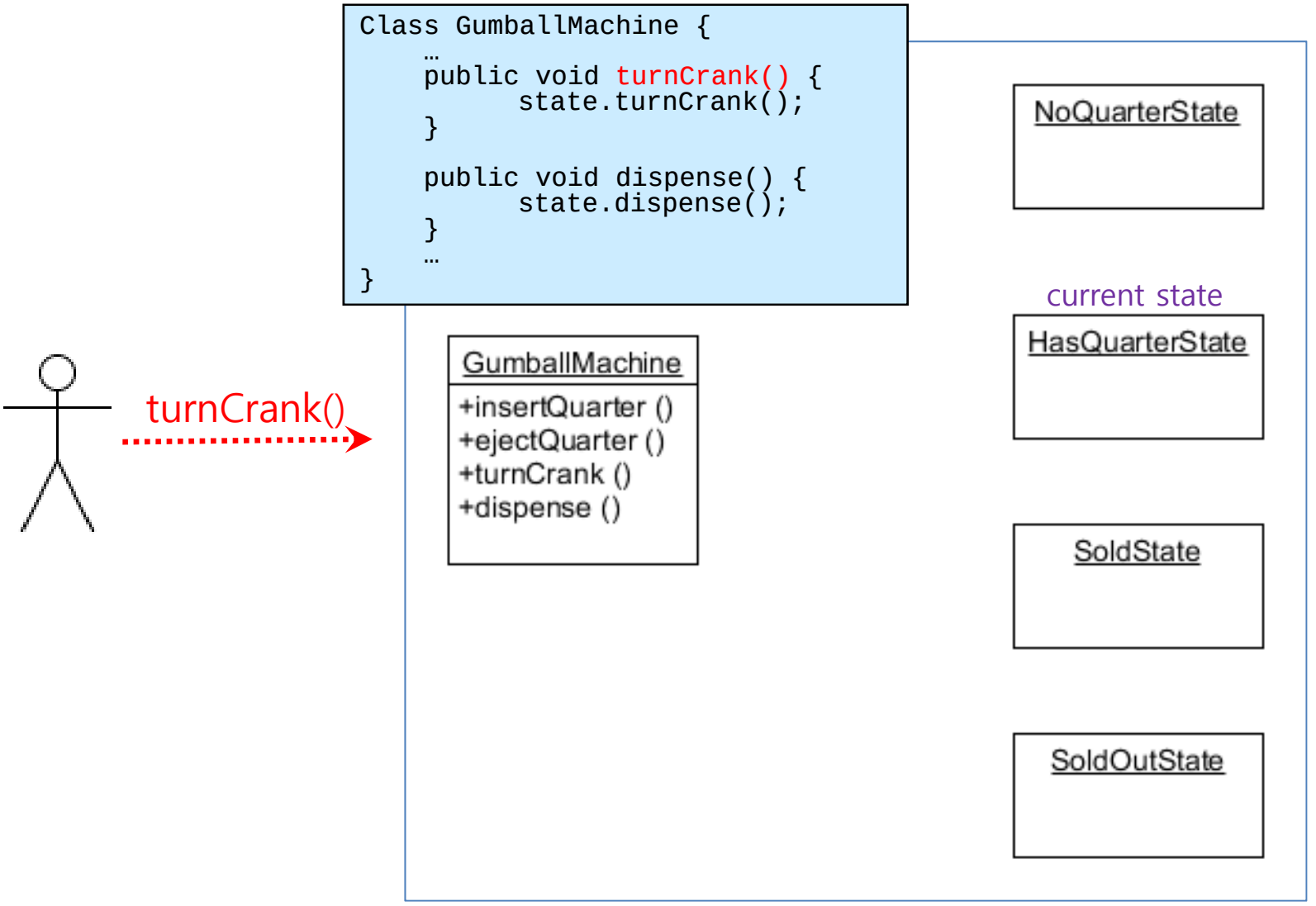
- Changes
  - Localized the **behavior of each state** into its own class
  - **Removed** all the troublesome **conditional statements** that would have been difficult to maintain
  - **Closed** each state **for modification**, and yet left the Gumball Machine **open to extension** by adding new state classes
  - Created a code base and class structure that maps much more closely to the Mighty Gumball diagram and is **easier to read and understand**

How It is Working

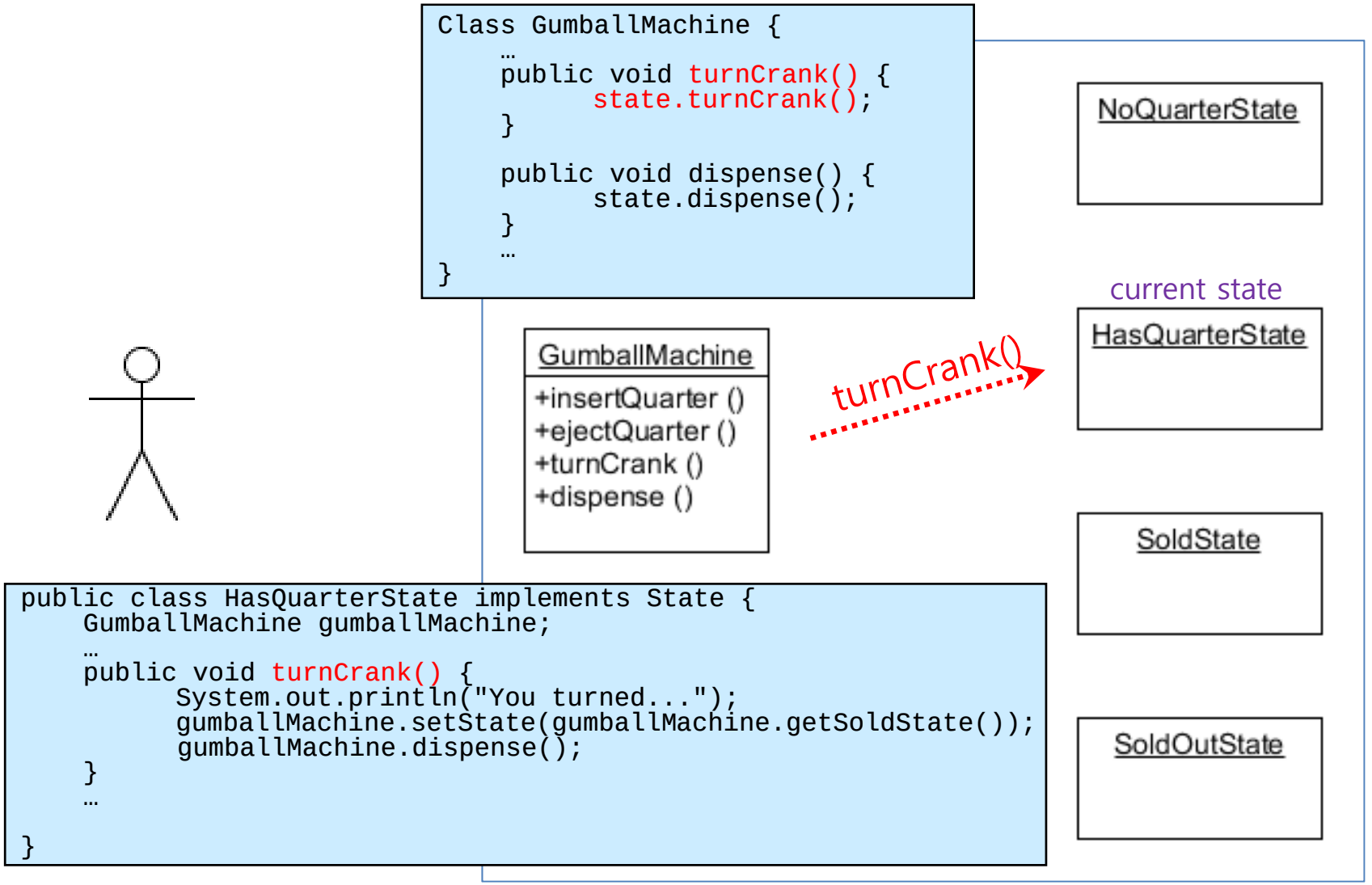




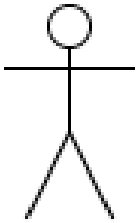
How It is Working

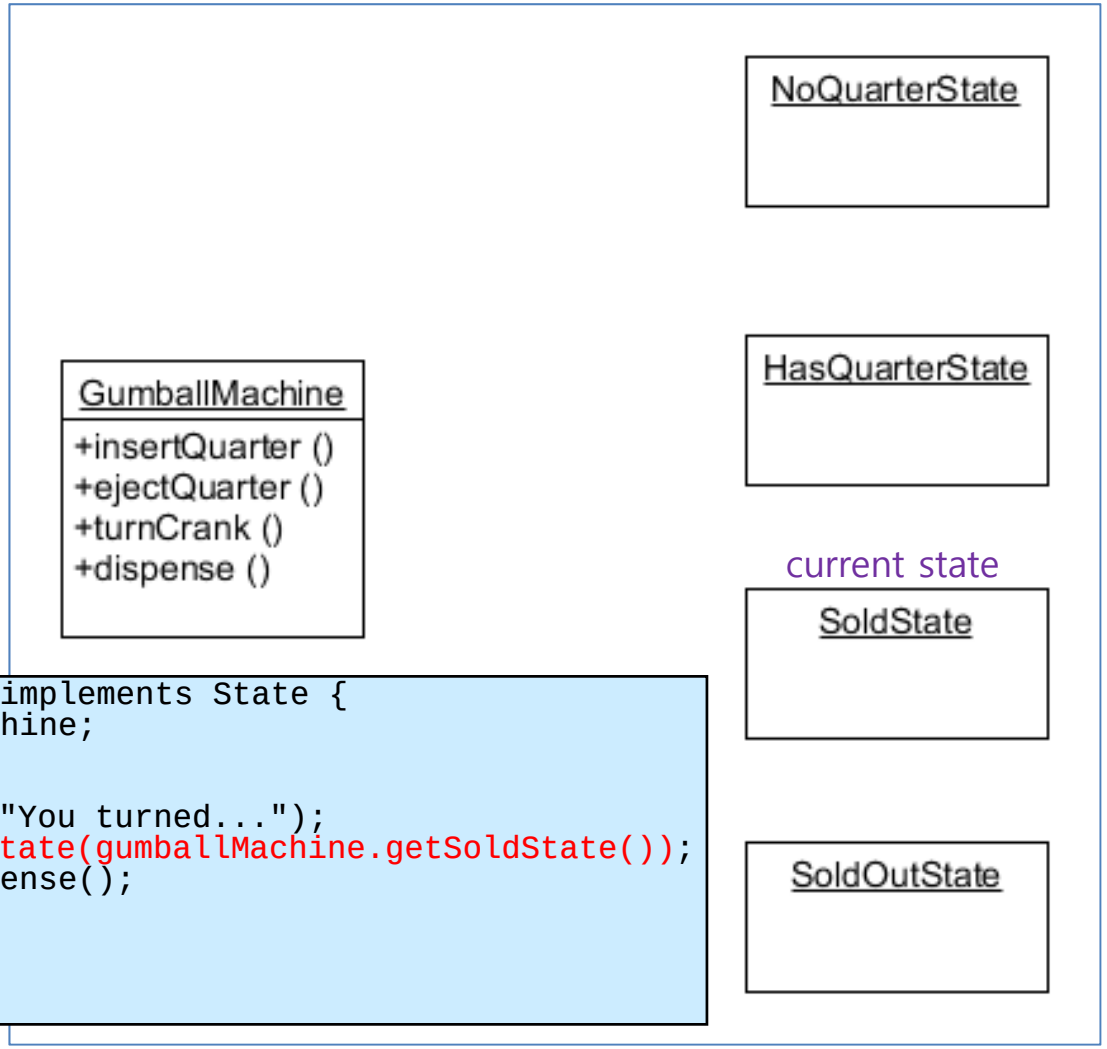


How It is Working



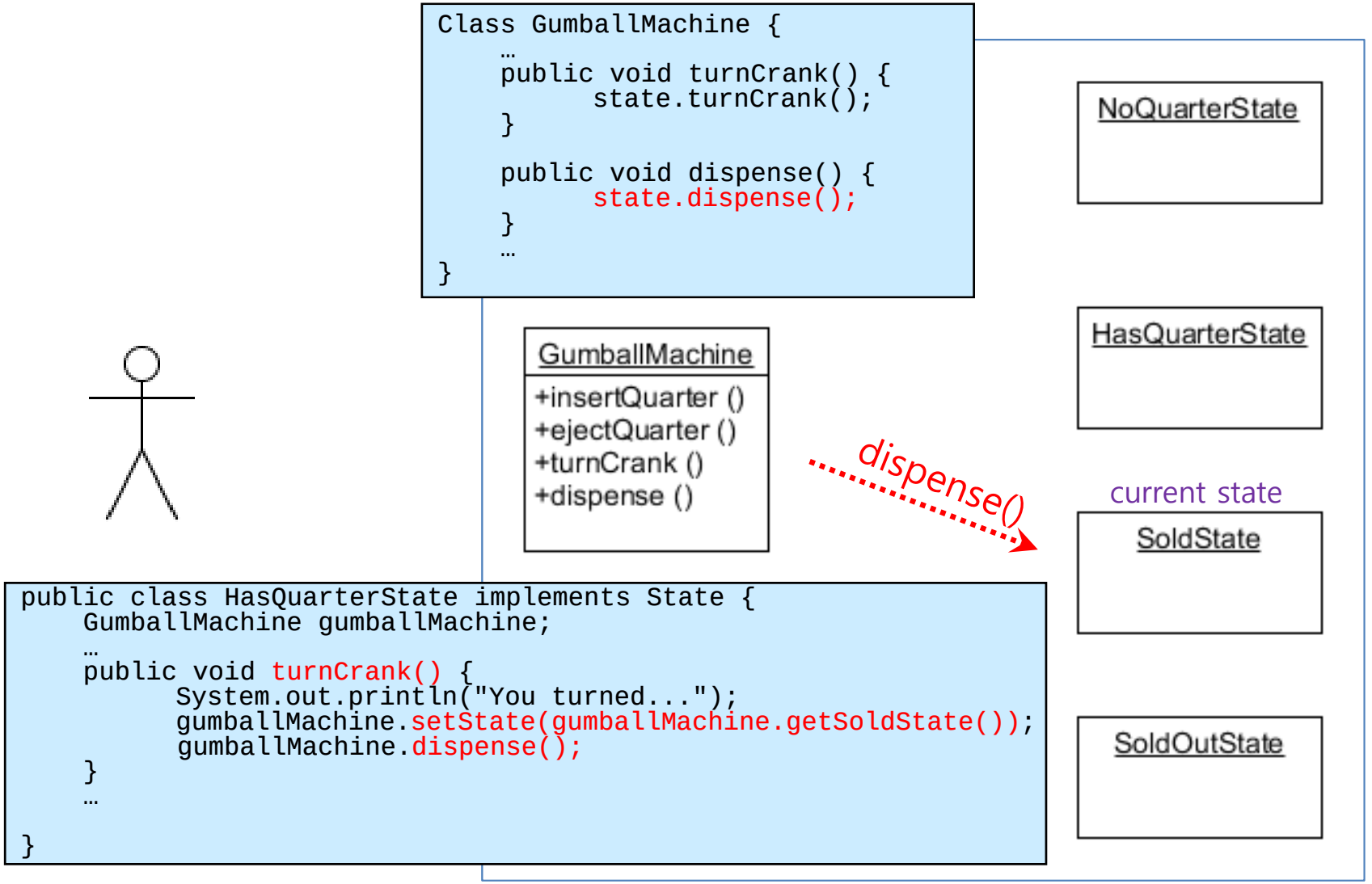
How It is Working





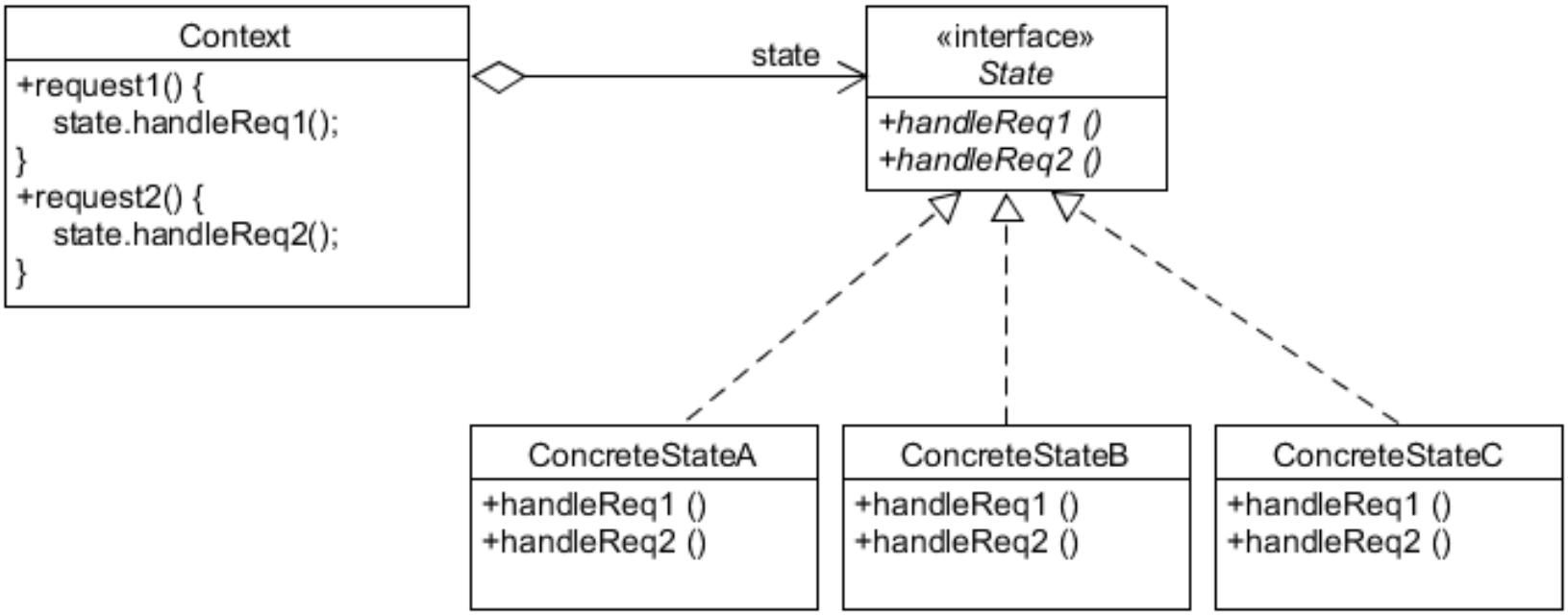

```
public class HasQuarterState implements State {
    GumballMachine gumballMachine;
    …
    public void turnCrank() {
          System.out.println("You turned...");
          gumballMachine.setState(gumballMachine.getSoldState());
          gumballMachine.dispense();
    }
    …

}
```

How It is Working



The 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.



Applicability of the State Pattern

- Use the State pattern when
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
  - Operations have large, multipart conditional statements that depend on the object's state. The State pattern puts each branch of the conditional in a separate class.

Consequences of the State Pattern

- Benefits
  - Puts all behavior associated with a state into one object
  - Allows state transition logic to be incorporated into a state object rather than in a monolithic if or switch statement
  - Helps avoid inconsistent states since state changes occur using just the one state object and not several objects or attributes
- Liabilities
  - Increased number of objects

State v.s. Strategy

- Note the similarities between the State and Strategy patterns!
  - The difference is one of intent.
- A State object encapsulates a state-dependent behavior (and possibly state transitions)
  - The context's behavior changes over time
  - An alternative to putting lots of conditionals in the context
- A Strategy object encapsulates an algorithm
  - Often, there is a strategy object that is most appropriate for a context object
  - A flexible alternative to subclassing
- They are both examples of Composition with Delegation!

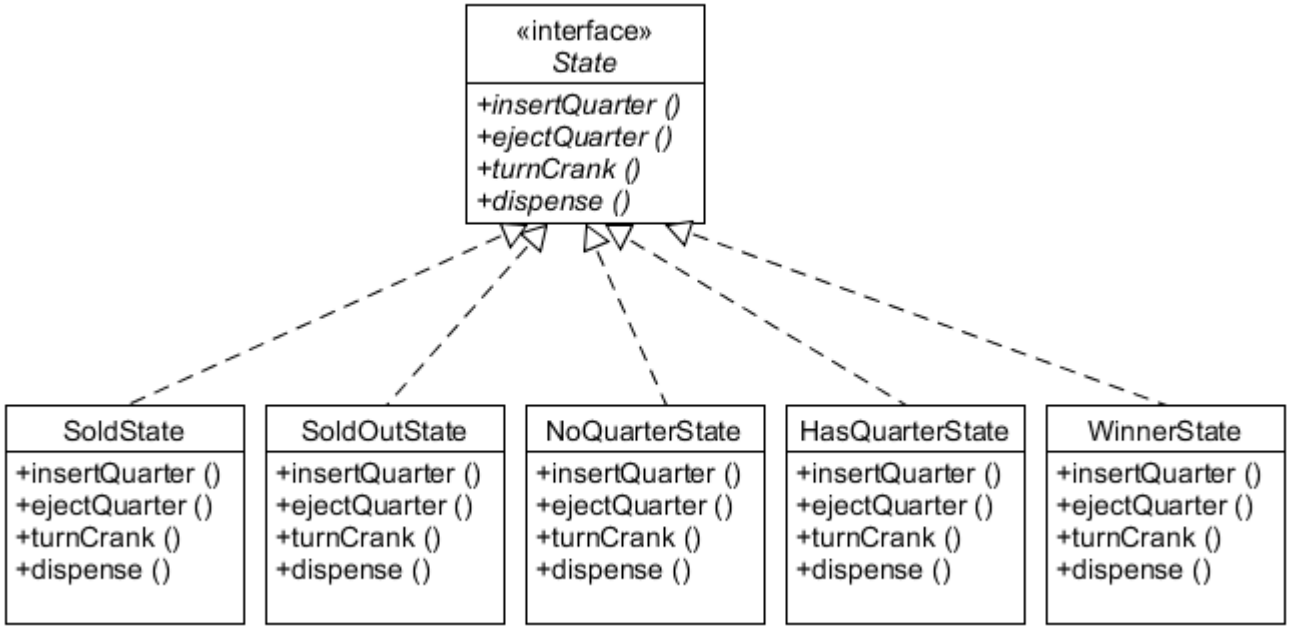
Implementation Issues

- Who defines the state transitions?
  - The Context class: okay for simple situations
  - The ConcreteState classes: generally more flexible, but causes implementation dependencies between the ConcreteState classes. In our example ConcreteState classes define the state transitions. Try to use getter methods.

- When are the ConcreteState objects created?
  - Create ConcreteState objects as needed
  - Create all ConcreteState objects once and have the Context object keep references to them

Finishing Gumball 1 in 10 game

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State winnerState;

    State state = soldOutState;
    int count = 0;
```



Coding the WinnerState

```
public class WinnerState implements State {
    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 dispense() {
        System.out.println("YOU'RE A WINNER! You get two gumballs!");
        gumballMachine.releaseBall();
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0)
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        else
            gumballMachine.setState(gumballMachine.getSoldOutState());
    }
}
```

Implementing 1 in 10 in the HasQuarterState

```
public class HasQuarterState implements State {
    Random randomWinner = new Random(System.currentTimeMillis());

    GumballMachine gumballMachine;

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

    public void turnCrank() {
        System.out.println("You turned...");
        int winner = randomWinner.nextInt(10);
        if ((winner == 0) && (gumballMachine.getCount() > 1))
            gumballMachine.setState(gumballMachine.getWinnerState());
        else
            gumballMachine.setState(gumballMachine.getSoldState());
        gumballMachine.dispense();
    }
    ...
}
```

Related Patterns

- The implementation of the State pattern builds on the Strategy pattern. The difference between State and Strategy is in the intent.
  - Strategy: the choice of algorithm is fairly stable.
  - State: a change in the state of the "context" object causes it to select from its "palette" of State objects.

Summary

- State Pattern
  - Encapsulate state-based behavior and delegate behavior to the current state
- Strategy Pattern
  - Encapsulate interchangeable behaviors and use delegation to decide which behavior to use
- Template Method
  - Subclasses decide how to implement steps in an algorithm