The State Design Pattern

The intent of the **State** pattern is to distribute state-specific logic across classes that represent an object’s state.

- The state pattern provides a clean way for an object to partially change its type at runtime.
- When the behavior of an object can be modeled by using a UML statechart diagram, it may be a candidate for using the State design pattern.
- Every state in the statechart diagram becomes a subclass of a generic state class (which is either an interface or an abstract class).
- The states of an object are separated from the object itself.
- In the GoF classification this is a *behavioral* design pattern.
Acknowledgements

• Materials were borrowed from
  – *Design Patterns in Java* by Steven Metsker and William Wake (textbook for course)
  – *Head First Design Patterns* by Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra
A Carousel Door

• The carousel door control access to a storage rack for storing materials; it is operated by a single push button.
  – If the button is pressed when the door is opening or closing, it reverses direction
  – After fully open and a two second delay the door will start closing assuming the material has passed in, but if the button is pressed the door will remain open

• This behavior is modeled in the statechart diagram shown to the right
Challenge 22.1

• Here is a brain teaser, look at the statechart diagram before answering.

Suppose that you open the door and place a material bin in the doorway. Is there a way to make the door begin closing without waiting for it to time out?
As the state machine shows, when the door is open, touching the one-touch button will take the door to the StayOpen state, and a second touch will start the door closing.
Designing the Door Class

- The door class will extend Observable so clients (e.g. the GUI) can be notified of the current door state
- The constant values representing the five states are shown in the code below

```java
package com.oozinoz.carousel;
import java.util.Observable;

public class Door extends Observable {
    public final int CLOSED = -1;
    public final int OPENING = -2;
    public final int OPEN = -3;
    public final int CLOSING = -4;
    public final int STAYOPEN = -5;

    private int state = CLOSED;

    // ...
}
```
public String status() {
    switch (state) {
    case OPENING:
        return "Opening";
    case OPEN:
        return "Open";
    case CLOSING:
        return "Closing";
    case STAYOPEN:
        return "StayOpen";
    default:
        return "Closed";
    }
}

private void setState(int state) {
    this.state = state;
    setChanged();
    notifyObservers();
}

public void touch() {
    switch (state) {
    case OPENING:
        setState(CLOSING);
        break;
    case CLOSING:
    case CLOSED:
        setState(OPENING);
        break;
    case OPEN:
        setState(STAYOPEN);
        break;
    default:
        throw new Error("can't happen");
    }
}
Completing the Door Class

CHALLENGE 22.2

Write the code for the `complete()` and `timeout()` methods of the Door class.
Solution 22.2

```java
public void complete() {
    if (state == OPENING)
        setState(OPEN);
    else if (state == CLOSING)
        setState(CLOSED);
}

public void timeout() {
    setState(CLOSING);
}
```

- Our solution so far has not used a design pattern
- We will now refactor this solution and introduce the State design pattern
Refactoring to State

- We introduce an abstract DoorState class; the method touch() is abstract
- There are five subclasses each corresponding to the five possible states of the door
Code Snippets

• The DoorClosed class is typical
  – The constructor simply calls the parent constructor
  – The touch() method will change the state to opening

• The Door2 class
  – The constants are replaced with new instances of the various states
  – Each door must have a separate set of constants (we will improve this later)

```java
package com.oozinoz.carousel;
public class DoorClosed extends DoorState {
    public DoorClosed(Door2 door) {
        super(door);
    }
    public void touch() {
        door.setState(door.OPENING);
    }
}
```

```java
package com.oozinoz.carousel;
import java.util.Observable;

public class Door2 extends Observable {
    public final DoorState CLOSED = new DoorClosed(this);
    public final DoorState CLOSING = new DoorClosing(this);
    public final DoorState OPEN = new DoorOpen(this);
    public final DoorState OPENING = new DoorOpening(this);
    public final DoorState STAYOPEN = new DoorStayOpen(this);

    private DoorState state = CLOSED;
    // ...
}
```
Code Snippets

• The DoorState class
  – touch() is abstract so every subclass must define this method
  – complete() and timeout() are empty methods; they can be overridden as required
  – status() is shared
  – The constructor is passed a handle to the door itself

```java
class DoorState {
    protected Door2 door;

    @Override
    public abstract void touch();

    public void complete() {
    }

    public void timeout() {
    }

    public String status() {
        String s = getClass().getName();
        return s.substring(s.lastIndexOf('.') + 1);
    }

    public DoorState(Door2 door) {
        this.door = door;
    }
}
```
The Door2 class
   - Since its state field is a subclass of DoorState, the touch, complete, timeout, and status methods simply call the corresponding methods using the current state
   - The setState method calls the setChanged() method and notifies the observers

```java
package com.oozinoz.carousel;
import java.util.Observable;

public class Door2 extends Observable {
    // variables and constructor ...

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

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

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

    public String status() {
        return state.status();
    }

    protected void setState(DoorState state) {
        this.state = state;
        setChanged();
        notifyObservers();
    }
}```
Code Snippets

• The DoorOpen subclass
  – Touching the button during the 2 second delay will cause the door to stay open
  – If the button is not pressed during the delay, then the door starts closing

```java
package com.oozinoz.carousel;

public class DoorOpen extends DoorState {
    public DoorOpen(Door2 door) {
        super(door);
    }

    public void touch() {
        door.setState(door.STAYOPEN);
    }

    public void timeout() {
        door.setState(door.CLOSING);
    }
}
```

CHALLENGE 22.3

Write the code for DoorClosing.java.
Solution 22.3

```java
package com.oozinoz.carousel;

public class DoorClosing extends DoorState {
    public DoorClosing(Door2 door) {
        super(door);
    }
    public void touch() {
        door.setState(door.OPENING);
    }
    public void complete() {
        door.setState(door.CLOSED);
    }
}
```
Making States Constants

- Inefficiency in the previous solution
  - Each individual door needs a separate set of door state classes; it would be more efficient to have a single, static door state that can be shared
  - We can have the Door class provide the update of its state value, as seen here

```java
public void touch() {
    state = state.touch();
}
```

- The touch method in the DoorState class would return the state resulting from the user pressing the switch; here is the code for the DoorOpen touch method

```java
public DoorState touch() {
    return DoorState.STAYOPEN;
}
```
The new DoorState class

- In this new design the door state objects would be constants that pass around a Door object during state transitions

**CHALLENGE 22.4**

Complete the class diagram in Figure 22.4 to show a design that lets DoorState objects be constants and that passes around a Door object during state transitions.
Solution 22.4

```
Door

complete()
setState(
  state: DoorState)
timeout()
touch()
status()

DoorState

CLOSED: DoorClosed
OPENING: DoorOpening
OPEN: DoorOpen
STAYOPEN: DoorStayOpen
CLOSING: DoorClosing

complete(d: Door)
timeout(d: Door)
touch(d: Door)
status()
```
Why This is Simpler

- In this design the door state objects do not need to retain a reference to a door object
- Rather, as indicated, the door object is passed around during state transitions
- Therefore we only need one static instance of the DoorState class that can be shared by all doors
Our Gumball Machine

• “Normal” behavior
  – User put quarter into slot
  – Turns the crank
  – A gumball is dispensed

• Unusual situations
  – The machine has run out of gumballs
  – The quarter should be returned to the user
  – How would the machine be refilled?
public void insertQuarter() {
    if (state == HAS_QUATER) {
        System.out.println("You can’t insert another quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You can’t insert a quarter, the machine is sold out");
    } else if (state == SOLD) {
        System.out.println("Please wait, we’re already giving you a gumball");
    } else if (state == NO_QUARTER) {
        state = HAS_QUATER;
        System.out.println("You inserted a quarter");
    }
}

...but can also transition to other states, just as depicted in the diagram.

Each possible state is checked with a conditional statement...

...and exhibits the appropriate behavior for each possible state...
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) {
            System.out.println("You can’t insert another quarter");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You can’t insert a quarter, the machine is sold out");
        } else if (state == SOLD) {
            System.out.println("Please wait, we’re already giving you a gumball");
        } else {
            System.out.println("You haven’t inserted a quarter");
        }
    }

    public void turnCrank() {
        System.out.println("Turn the crank");
    }
}

Here are the four states; they match the states in Mighty Gumball’s state diagram.

Here’s the instance variable that is going to keep track of the current state we’re in. We start in the SOLD_OUT state.

We have a second instance variable that keeps track of the number of gumballs in the machine.

The constructor takes an initial inventory of gumballs. If the inventory isn’t zero, the machine enters state NO_QUARTER, meaning it is waiting for someone to insert a quarter, otherwise it stays in the SOLD_OUT state.

Now we start implementing the actions as methods...

When a quarter is inserted, if...

If the customer just bought a gumball he needs to wait until the transaction is complete before inserting another quarter.

and if the machine is sold out, we reject the quarter.
```java
public void ejectQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("Quarter returned");
        state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
    } else if (state == SOLD) {
        System.out.println("Sorry, you already turned the crank");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}

public void turnCrank() {
    if (state == SOLD) {
        System.out.println("Turning twice doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
        System.out.println("You turned but there's no quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
        System.out.println("You turned...");
        state = SOLD;
        dispense();
    } else if (state == NO_QUARTER) {
        System.out.println("You need to pay first");
    } else if (state == SOLD_OUT) {
        System.out.println("No gumball dispensed");
    } else if (state == HAS_QUARTER) {
        System.out.println("No gumball dispensed");
    }
}

public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0) {
            System.out.println("Oops, out of gumballs!");
            state = SOLD_OUT;
        } else {
            state = NO_QUARTER;
        }
    } else if (state == NO_QUARTER) {
        System.out.println("You need to pay first");
    } else if (state == SOLD_OUT) {
        System.out.println("No gumball dispensed");
    } else if (state == HAS_QUARTER) {
        System.out.println("No gumball dispensed");
    }
}
```
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);
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.ejectQuarter();
        System.out.println(gumballMachine);
        gumballMachine.insertQuarter();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        System.out.println(gumballMachine);
    }
}
%java GumballMachineTestDrive

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
Quarter returned
You turned but there’s no quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
You haven’t inserted a quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 2 gumballs
Machine is waiting for quarter

You inserted a quarter
You can’t insert another quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
Oops, out of gumballs!
You can’t insert a quarter, the machine is sold out
You turned, but there are no gumballs

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
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
• We first revise our statechart
• Then we introduce the State design pattern

1. First, we’re going to define a State interface that contains a method for every action in the Gumball Machine.

2. Then we’re going to 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, we’re going to get rid of all of our conditional code and instead delegate to the state class to do the work for us.
Then take each state in our design and encapsulate it in a class that implements the State interface.

To figure out what states we need, we look at our previous code...

```java
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;
}
```

...and we map each state directly to a class.

Don't forget, we need a new "winner" state too that implements the state interface. We'll come back to this after we reimplement the first version of the Gumball Machine.
First we need to implement the State interface.

```java
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");
    }
}
```

We get passed a reference to the Gumball Machine through the constructor. We’re just going to stash this in an instance variable.

If someone inserts a quarter, we print a message saying the quarter was accepted and then change the machine’s state to the HasQuarterState.

You’ll see how these work in just a sec...

You can’t get money back if you never gave it to us!

And, you can’t get a gumball if you don’t pay us.

We can’t be dispensing gumballs without payment.
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;

    private State state = soldOutState;
    private State noQuarterState;
    private State hasQuarterState;
    private State soldState;

    State state = soldOutState;
    int count = 0;

    public GumballMachine() {
        ....
    }

    public void insertQuarter() {
        ....
    }

    public void spinGumball() {
        ....
    }

    public void takeGumball() {
        ....
    }

    private void setState(State state) {
        ....
    }

    private void incrementCount() {
        ....
    }

    private void decrementCount() {
        ....
    }

    private void setSoldOut() {
        ....
    }

    private void setNoQuarter() {
        ....
    }

    private void setHasQuarter() {
        ....
    }

    private void setSold() {
        ....
    }

    private void setStateWithState() {
        ....
    }

    private void setStateWithInteger() {
        ....
    }

    private void setStateWithObject() {
        ....
    }

    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine();
        gumballMachine.insertQuarter();
        gumballMachine.spinGumball();
        gumballMachine.takeGumball();
    }
}
public class GumballMachine {
    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;

    public GumballMachine(int numberOfGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberOfGumballs;
        if (numberOfGumballs > 0) {
            state = noQuarterState;
        }
    }

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

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

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

    // More methods here including getters for each State...
}

// This includes methods like getNoQuarterState() for getting each state object, and getCount() for getting the gumball count...
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");
    }
}

When the state is instantiated we pass it a reference to the GumballMachine. This is used to transition the machine to a different state.

An inappropriate action for this state.

Return the customer’s quarter and transition back to the NoQuarterState.

When the crank is turned we transition the machine to the SoldState state by calling its setState() method and passing it the SoldState object. The SoldState object is retrieved by the getSoldState() getter method (there is one of these getter methods for each state).

Another inappropriate action for this state.
public class SoldState implements State {
    // constructor and instance variables here

    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());
        }
    }
}

And here’s where the real work begins...

We’re in the SoldState, which means the customer paid. So, we first need to ask the machine to release a gumball.

Then we ask the machine what the gumball count is, and either transition to the NoQuarterState or the SoldOutState.
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");
    }
}
The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The Context is the class that can have a number of internal states. In our example, the GumballMachine is the Context.

Whenever the request() is made on the Context it is delegated to the state to handle.

ConcreteStates handle requests from the Context. Each ConcreteState provides its own implementation for a request. In this way, when the Context changes state, its behavior will change as well.

The State interface defines a common interface for all concrete states; the states all implement the same interface, so they are interchangeable.

Many concrete states are possible.
public class GumballMachine {
    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State winnerState;
    State state = soldOutState;
    int count = 0;

    // methods here
}

Now let's implement the WinnerState class itself, it's remarkably similar to the SoldState class:

public class WinnerState implements State {

    // instance variables and constructor
    // insertQuarter error message
    // ejectQuarter error message
    // turnCrank error message

    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 class HasQuarterState implements State {
    Random randomWinner = new Random(System.currentTimeMillis());
    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.getCount() > 1)) {
            gumballMachine.setState(gumballMachine.getWinnerState());
        } else {
            gumballMachine.setState(gumballMachine.getSoldState());
        }
    }

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

First we add a random number generator to generate the 10% chance of winning...

...then we determine if this customer won.

If they won, and there's enough gumballs left for them to get two, we go to the WinnerState; otherwise, we go to the SoldState (just like we always did).
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.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        System.out.println(gumballMachine);
    }
}
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
YOU'RE A WINNER! You get two gumballs for your quarter
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 3 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
You inserted a quarter
You turned...
YOU'RE A WINNER! You get two gumballs for your quarter
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
Oops, out of gumballs!

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out