# Lesson 8 State Strategy and Memento

## The State Pattern

1. Intent

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

2. Motivation

   a. Objects have states and behavior (behavior as functions to process different data/states). Many times, the behavior and states are independent while other times behavior varies for different states the object is in.
   b. We achieve the above by factoring the states and their corresponding behavior into separate classes.

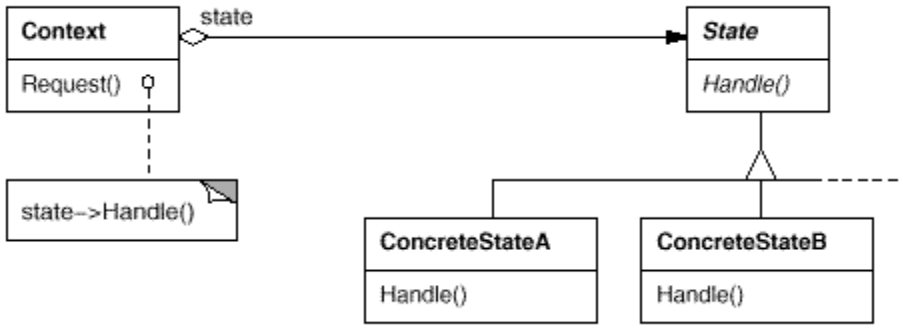
3. Applicability

   Use the State pattern in the following cases:

   a. An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
   b. Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

c. The number of all possible states is limited.
d. When we need to design state machines or vending machines.

## 4. Structure




## 5. Participants

a. Context
   - defines the interface of interest to clients.
   - maintains an instance of a ConcreteState subclass that defines the current state.

b. State
   - defines an interface for encapsulating the behavior associated with a particular state of the Context.

c. ConcreteState subclasses
   - each subclass implements a behavior associated with a state of the Context.

## 6. Consequences

a. It localizes state-specific behavior in each concrete state.
b. State information can be represented by an instance variable or by the type of the concrete state.

## 7. How to implement the State pattern?

Let's implement a Gumball machine with the State pattern

### a. State interface

```
public interface State {

	public void insertQuarter();
	public void ejectQuarter();
	public void turnCrank();
	public void dispense();
}
```

### b. Gumball machine (context)

```
public class GumballMachine {

	private State soldOutState;
	private State noQuarterState;
	private State hasQuarterState;
	private State soldState;

	private State state = soldOutState; //current state
	private int count = 0; //number of Gumballs available

	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;
		}
	}

	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;
		}
	}

	int getCount() {
		return count;
	}

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

    public State getState() {
        return state;
    }

    public State getSoldOutState() {
        return soldOutState;
    }

    public State getNoQuarterState() {
        return noQuarterState;
    }

    public State getHasQuarterState() {
        return hasQuarterState;
```

```
        }

        public State getSoldState() {
            return soldState;
        }

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

## c. Concrete state classes (one example given)

```
   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");
        }
```

## d. GumballMachineTestDrive (client)

```
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);
	}
}
```

# Lab 8-1

A multilevel computer game simulates driving a car. In the initial level, the road condition is "regular". As the player moves to higher levels, new conditions of the road appear, from regular to gravel, wet, and ice. The response to the car's controls, steering, braking, etc., varies with the road conditions. The game employs a State pattern to react to the player's use of the car controls.

For the purpose of this exercise, design a small GUI with four buttons (labeled left, accel, right and brake), a choice of conditions (labeled regular, gravel, wet and ice), and a feedback textfield showing the effect (in an arbitrary scale) of pressing a button in a given road condition as follows:

| | Regular | Gravel | Wet | Ice |
|---|---|---|---|---|
| left | 5 | 3 | 4 | 1 |
| accel | 9 | 7 | 9 | 3 |
| right | 5 | 3 | 4 | 1 |
| brake | 8 | 6 | 7 | 2 |

## The Strategy Pattern

1. Intent

   Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

2. Motivation

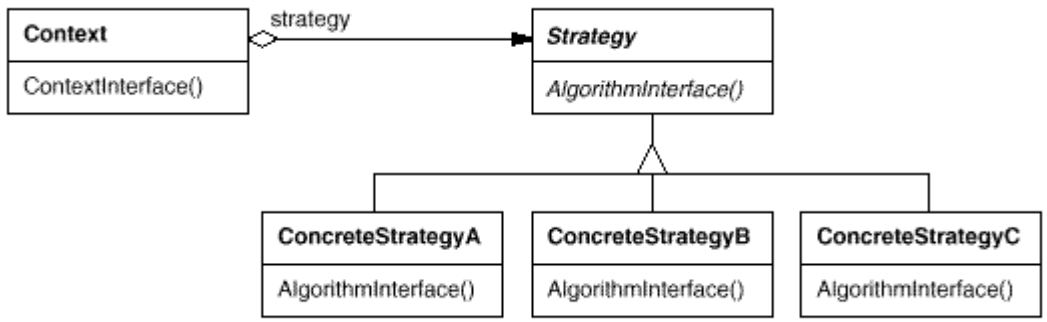
   a. The client and a strategy it uses to perform a function should not be hard-wired together
      - to keep the client lean;
      - or make the change of strategy easier;
      - or add new strategies without affecting client code.
   b. Your application may need different strategies for different situations during runtime.

3. Applicability

   Use the Strategy pattern when
   a. Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
   b. You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
   c. An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.

d. A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

## 4. Structure




## 5. Participants

a. Strategy - declares an interface common to all supported algorithms.
b. Concrete Strategy - implements the algorithm using the Strategy interface.
c. Context - is configured with one or more Concrete Strategy objects. In multi-layered enterprise applications, a Context often uses a Hash Map that stores server-side information. (For example, compression strategies for transporting different files at runtime).

## 6. Consequences

a. Families of related algorithms are defined by hierarchies of Strategy classes with inheritance.

b. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.
c. Strategies eliminate conditional statements (from client) by encapsulating the behavior in separate Strategy classes.
d. Strategies can provide different implementations of the same behavior. The client can choose among strategies with different time and space trade-offs.
e. You should use the Strategy pattern only when the variation in behavior is relevant to clients.
f. Strategies are stateless objects across invocations.

## 7. Implementation

### a. Strategy interface

```
public interface CompressionStrategy {
	public Archive compressFiles(List<File> files);
}
```

### b. Concrete strategy classes

```
public class ZipCompressionStrategy implements CompressionStrategy {

	@Override
	public Archive compressFiles(List<File> files) {
		// TODO Auto-generated method stub

	}

}

public class RarCompressionStrategy implements CompressionStrategy {

	@Override
	public Archive compressFiles(List<File> files) {
		// TODO Auto-generated method stub

	}

}
```

## c. Context class (FileCompressor)

```
public class FileCompressor {
	private CompressionStrategy strategy;
	private Archive archive;

	public void setStrategy(CompressionStrategy strategy) {
		this.strategy = strategy;
	}

	public Archive createArchive(List<File> files){
		return strategy.compressFiles(files);
	}

}
```

## d. Client

```
public class StrategyClient {

	public static void main(String[] args) {
		List<File> filesToCompress =
getFilesToCompress("C:\files");
		FileCompressor fc = new FileCompressor();
		//choose the best strategy for compression
		//fc.setStrategy(…);
		Archive archive = fc.createArchive(filesToCompress);
		//...
	}

	public static List<File> getFilesToCompress(String location){
		//get all files from a location
		List<File> files = new ArrayList<File>();
		//…
		return files;
	}

}
```

# Lab 8-2

You are going to implement three different pricing models for an airline, and calculate the revenue generated by each strategy (model) for computing the total revenue of the airline's flights.

The Strategy pattern is a good example of Subtyping. In other words, we are programming to an interface, not Subclassing, in which case we would be using an abstract class. So create a generic Strategy interface, called Model. It will have one method, long getRevenue(List flights) which takes a List of flights as its only parameter and returns a long of whole dollars.

Then create 3 different concrete strategies, SinglePrice, TwoClasses, and MultiClass. These will all implement the Model interface. The Model interface should define a constant for the base ticket price ($300), and the fixed cost to fly a plane ($50,000). For now, we'll ignore distance and different plane sizes.

The three models will behave as follows:

SinglePrice will just take the number of passengers, multiply by the constant ticket price to get a total revenue for a flight, then subtract the fixed cost. It will then total all the flights and return the revenue total.
TwoClasses will take the number of passengers, and price 1/3 of them as Business Class, where the cost is 1.5 times the ticket price. 2/3 will be in Coach at 0.75 times the ticket price. The fixed cost will be 1.1 times higher (for the business class meals and lost seats).

MultiClass will take the number of passengers, and price 1/10 of them as First Class, where the cost is 4 times the ticket price. 1/5 will be in Business class at 1.5 times the ticket price. The rest (7/10) will be in Coach at 0.75 times the base ticket price. The fixed cost will be 1.2 times higher (for the first and business class meals and lost seats).

## The Memento Pattern

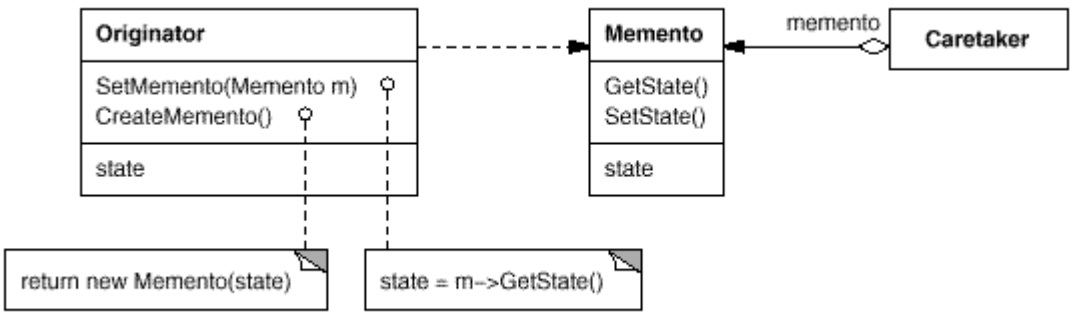
1. Intent

   Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

2. Motivation

   c. Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.
   d. You must save state information somewhere so that you can restore objects to their previous states.

3. Structure




4. Participants

   a. Memento
      - stores internal state of the Originator object. The memento may store as much or as little of the

originator's internal state as necessary at its originator's discretion.

- protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento—it can only pass the memento to other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.

b. Originator
   - creates a memento containing a snapshot of its current internal state.
   - uses the memento to restore its internal state.

c. Caretaker
   - is responsible for the memento's safekeeping.
   - never operates on or examines the contents of a memento.

## 5. Applicability

Use the Memento pattern when

e. A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
f. A direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

## 6. Implementation

### e. Memento (a previous state or 'snapshot')

```
class Memento {
	private String state;

	public Memento(String state) {
		this.state = state;
	}

	String getState() {
		return state;
	}
}
```

### f. Originator (its state gets externalized to be stored in a Memento object)

```
public class Originator {
	private String externalizedState;

	String getExternalizedState() {
		return externalizedState;
	}

	void setExternalizedState(String externalizedState) {
		this.externalizedState = externalizedState;
	}

	public Memento saveStateToMemento() {
		return new Memento(externalizedState);
	}

	public void getStateFromMemento(Memento Memento) {
		externalizedState = Memento.getState();
	}
}
```

### g. CareTaker class (keeps a collection of previous states in case client needs to restore them or undo something)

```
class CareTaker {
	private List<Memento> mementoList = new ArrayList<Memento>();

	void add(Memento state) {
		mementoList.add(state);
	}

	Memento get(int index) {
		return mementoList.get(index);
	}
}
```

**Lab 8-3**

Develop a GUI that edits user information. You will need at least 3 buttons – load (from a database), undo, save. When you start, you use the ‘load’ button to read an existing profile for the user, if it does exist. If not, then create a new profile for the user. At any time during the edit work, you can always click on ‘undo’ to restore from errors, like what the MS Word allows you to do. Once you are done with the user profile, click ‘save’ button to store it (in the database).