* The state pattern is a behavioral object design pattern.
* The idea behind the state pattern is for an object to change its behavior depending on its state.
* In the state pattern,
  + We have a Context class, and this class has a State reference to a Concrete State instance.
  + The State interface declares particular methods that represent the behaviors of a particular state.
  + Concrete States implement these behaviors. By changing a Context's Concrete State, we change its behavior.
  + In essence, in the state pattern, a class (the Context) is supposed to behave like different classes depending on its state.
  + The state pattern avoids the use of switch and if statements to change behavior.

Let's look at an example of the state pattern. First off, we'll define the EmotionalState interface. It declares two methods, sayHello() and sayGoodbye().

// State

public interface EmotionalState {

public String sayHello();

public String sayGoodbye();

}

The HappyState class is a Concrete State that implements sayHello() and sayGoodbye() of EmotionalState. These messages are cheerful (representing a happy state).

// Concrete State

public class HappyState implements EmotionalState {

@Override

public String sayGoodbye() {

return "Bye, friend!";

}

@Override

public String sayHello() {

return "Hello, friend!";

}

}

The SadState class also implements the EmotionalState interface. The messages are sad (representing a sad state).

//Concrete State

public class SadState implements EmotionalState {

@Override

public String sayGoodbye() {

return "Bye. Sniff, sniff.";

}

@Override

public String sayHello() {

return "Hello. Sniff, sniff.";

}

}

The Person class is the Context class. It contains an EmotionalState reference to a concrete state. In this example, we have Person implement the EmotionalState reference, and we pass the calls to Person's sayHello() and sayGoodbye() methods on to the corresponding methods on the emotionalState reference. As a result of this, a Person object behaves differently depending on the state of Person (ie, the current EmotionalState reference).

// Context

public class Person implements EmotionalState {

EmotionalState emotionalState;

public Person(EmotionalState emotionalState)

{

this.emotionalState = emotionalState;

}

public void setEmotionalState(EmotionalState emotionalState)

{

this.emotionalState = emotionalState;

}

@Override

public String sayGoodbye()

{

return emotionalState.sayGoodbye();

}

@Override

public String sayHello()

{

return emotionalState.sayHello();

}

}

The Demo class demonstrates the state pattern. First, it creates a Person object with a HappyState object. We display the results of sayHello() and sayGoodbyte() when the person object is in the happy state. Next, we change the person object's state with a SadState object. We display the results of sayHello() and sayGoodbyte(), and we see that in the sad state, the person object's behavior is different.

public class Demo {

public static void main(String[] args) {

Person person = new Person(new HappyState());

System.out.println("Hello in happy state: " + person.sayHello());

System.out.println("Goodbye in happy state: " + person.sayGoodbye());

person.setEmotionalState(new SadState());

System.out.println("Hello in sad state: " + person.sayHello());

System.out.println("Goodbye in sad state: " + person.sayGoodbye());

}

}

OUTPUT:

Hello in happy state: Hello, friend!

Goodbye in happy state: Bye, friend!

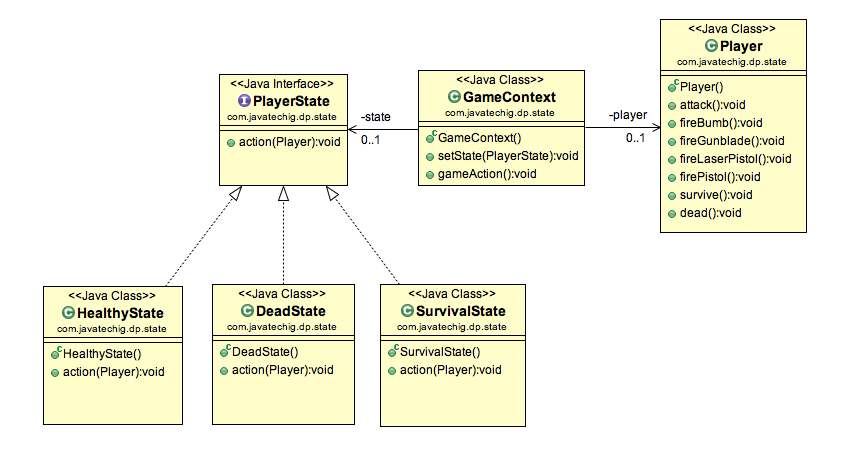
Hello in sad state: Hello. Sniff, sniff.

Goodbye in sad state: Bye. Sniff, sniff.

Example 4:

The game character can be in different states such as healthy, surviving and dead. When character is healthy, it allows user to fires at enemies with different weapons. When surviving state its health gets critical, and when its health reaches to 0, the character is said to be in dead state where the game is over.

Class Diagram:



Java Code

1. Define an interface named PlayerState that defines action method. The access() method takes the instance of Player class. This is required to perform player action.
2. Define three different classes that represents the different states. In this example, I have named them as HealthyState, SurvivalState, DeadState. All of three classes implements PlayerState interface and provides the specific action() method implementation.
3. The GameContxt class contains two setState() method composition. Now we will remove all of the code to conditional logic.
4. Execute code using below GameTest class.