**Strategy Pattern**

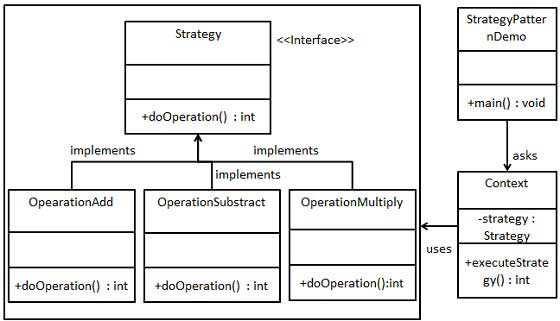
In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

Implementation

We are going to create a Strategy interface defining an action and concrete strategy classes implementing the Strategy interface. Context is a class which uses a Strategy.

StrategyPatternDemo, our demo class, will use Context and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses.



Step 1

Create an interface.

Strategy.java

public interface Strategy {

public int doOperation(int num1, int num2);

}

Step 2

Create concrete classes implementing the same interface.

OperationAdd.java

public class OperationAdd implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 + num2;

}

}

OperationSubstract.java

public class OperationSubstract implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 - num2;

}

}

OperationMultiply.java

public class OperationMultiply implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 \* num2;

}

}

Step 3

Create Context Class.

Context.java

public class Context {

private Strategy strategy;

public Context(Strategy strategy){

this.strategy = strategy;

}

public int executeStrategy(int num1, int num2){

return strategy.doOperation(num1, num2);

}

}

Step 4

Use the Context to see change in behaviour when it changes its Strategy.

StrategyPatternDemo.java

public class StrategyPatternDemo {

public static void main(String[] args) {

Context context = new Context(new OperationAdd());

System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationSubstract());

System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationMultiply());

System.out.println("10 \* 5 = " + context.executeStrategy(10, 5));

}

}

Step 5

Verify the output.

10 + 5 = 15

10 - 5 = 5

10 \* 5 = 50

**State Pattern**

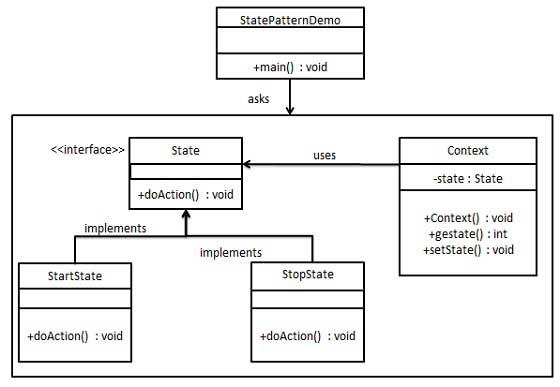
In State pattern a class behavior changes based on its state. This type of design pattern comes under behavior pattern.

In State pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.

Implementation

We are going to create a State interface defining an action and concrete state classes implementing the State interface. Context is a class which carries a State.

StatePatternDemo, our demo class, will use Context and state objects to demonstrate change in Context behavior based on type of state it is in.



Step 1

Create an interface.

State.java

public interface State {

public void doAction(Context context);

}

Step 2

Create concrete classes implementing the same interface.

StartState.java

public class StartState implements State {

public void doAction(Context context) {

System.out.println("Player is in start state");

context.setState(this);

}

public String toString(){

return "Start State";

}

}

StopState.java

public class StopState implements State {

public void doAction(Context context) {

System.out.println("Player is in stop state");

context.setState(this);

}

public String toString(){

return "Stop State";

}

}

Step 3

Create Context Class.

Context.java

public class Context {

private State state;

public Context(){

state = null;

}

public void setState(State state){

this.state = state;

}

public State getState(){

return state;

}

}

Step 4

Use the Context to see change in behaviour when State changes.

StatePatternDemo.java

public class StatePatternDemo {

public static void main(String[] args) {

Context context = new Context();

StartState startState = new StartState();

startState.doAction(context);

System.out.println(context.getState().toString());

StopState stopState = new StopState();

stopState.doAction(context);

System.out.println(context.getState().toString());

}

}

Step 5

Verify the output.

Player is in start state

Start State

Player is in stop state

Stop State

**Observer Pattern**

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

Implementation

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class Observer and a concrete class Subject that is extending class Observer.

ObserverPatternDemo, our demo class, will use Subject and concrete class object to show observer pattern in action.



Step 1

Create Subject class.

Subject.java

import java.util.ArrayList;

import java.util.List;

public class Subject {

private List<Observer> observers = new ArrayList<Observer>();

private int state;

public int getState() {

return state;

}

public void setState(int state) {

this.state = state;

notifyAllObservers();

}

public void attach(Observer observer){

observers.add(observer);

}

public void notifyAllObservers(){

for (Observer observer : observers) {

observer.update();

}

}

}

Step 2

Create Observer class.

Observer.java

public abstract class Observer {

protected Subject subject;

public abstract void update();

}

Step 3

Create concrete observer classes

BinaryObserver.java

public class BinaryObserver extends Observer{

public BinaryObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );

}

}

OctalObserver.java

public class OctalObserver extends Observer{

public OctalObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );

}

}

HexaObserver.java

public class HexaObserver extends Observer{

public HexaObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Hex String: " + Integer.toHexString( subject.getState() ).toUpperCase() );

}

}

Step 4

Use Subject and concrete observer objects.

ObserverPatternDemo.java

public class ObserverPatternDemo {

public static void main(String[] args) {

Subject subject = new Subject();

new HexaObserver(subject);

new OctalObserver(subject);

new BinaryObserver(subject);

System.out.println("First state change: 15");

subject.setState(15);

System.out.println("Second state change: 10");

subject.setState(10);

}

}

Step 5

Verify the output.

First state change: 15

Hex String: F

Octal String: 17

Binary String: 1111

Second state change: 10

Hex String: A

Octal String: 12

Binary String: 1010

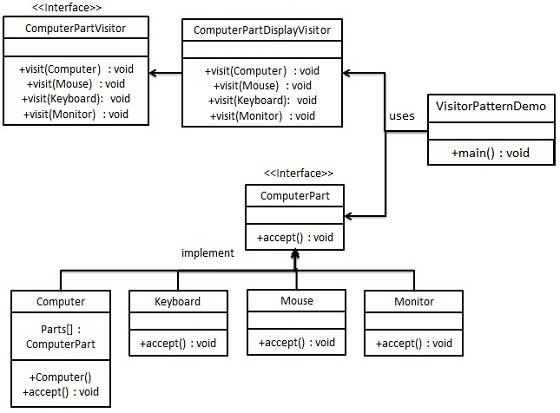
**Visitor Pattern**

In Visitor pattern, we use a visitor class which changes the executing algorithm of an element class. By this way, execution algorithm of element can vary as and when visitor varies. This pattern comes under behavior pattern category. As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object.

Implementation

We are going to create a ComputerPart interface defining accept opearation.Keyboard, Mouse, Monitor and Computer are concrete classes implementing ComputerPart interface. We will define another interface ComputerPartVisitor which will define a visitor class operations. Computer uses concrete visitor to do corresponding action.

VisitorPatternDemo, our demo class, will use Computer and ComputerPartVisitor classes to demonstrate use of visitor pattern.



Step 1

Define an interface to represent element.

ComputerPart.java

public interface ComputerPart {

public void accept(ComputerPartVisitor computerPartVisitor);

}

Step 2

Create concrete classes extending the above class.

Keyboard.java

public class Keyboard implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

Monitor.java

public class Monitor implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

Mouse.java

public class Mouse implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

Computer.java

public class Computer implements ComputerPart {

ComputerPart[] parts;

public Computer(){

parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};

}

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

for (int i = 0; i < parts.length; i++) {

parts[i].accept(computerPartVisitor);

}

computerPartVisitor.visit(this);

}

}

Step 3

Define an interface to represent visitor.

ComputerPartVisitor.java

public interface ComputerPartVisitor {

public void visit(Computer computer);

public void visit(Mouse mouse);

public void visit(Keyboard keyboard);

public void visit(Monitor monitor);

}

Step 4

Create concrete visitor implementing the above class.

ComputerPartDisplayVisitor.java

public class ComputerPartDisplayVisitor implements ComputerPartVisitor {

@Override

public void visit(Computer computer) {

System.out.println("Displaying Computer.");

}

@Override

public void visit(Mouse mouse) {

System.out.println("Displaying Mouse.");

}

@Override

public void visit(Keyboard keyboard) {

System.out.println("Displaying Keyboard.");

}

@Override

public void visit(Monitor monitor) {

System.out.println("Displaying Monitor.");

}

}

Step 5

Use the ComputerPartDisplayVisitor to display parts of Computer.

VisitorPatternDemo.java

public class VisitorPatternDemo {

public static void main(String[] args) {

ComputerPart computer = new Computer();

computer.accept(new ComputerPartDisplayVisitor());

}

}

Step 6

Verify the output.

Displaying Mouse.

Displaying Keyboard.

Displaying Monitor.

Displaying Computer.