**Design Pattern**

A design patterns are well-proved solution for solving the specific problem/task.

1. **Creational design Patterns**
2. **Singleton**

**Need :**

Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers

**Definition:**

The singleton pattern is a design pattern that restricts the instantiation of a class to one object.

Implementation :

class Singleton

{

private static Singleton obj;

private Singleton() {}

public static Singleton getInstance()

{

if (obj==null)

obj = new Singleton();

return obj;

}

}

1. **Factory**

Need :

* creates objects without exposing the instantiation logic to the client.
* refers to the newly created object through a common interface
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

Usage of Factory Design Pattern

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.

Implementation :

Electricity Bill Calculation

Step 1: Create a Plan abstract class.

import java.io.\*;

abstract class Plan{

         protected double rate;

         abstract void getRate();

         public void calculateBill(int units){

              System.out.println(units\*rate);

          }

}/

Step 2: Create the concrete classes that extends Plan abstract class.

class  DomesticPlan extends Plan{

        //@override

         public void getRate(){

             rate=3.50;

        }

   }.

class  CommercialPlan extends Plan{

   //@override

    public void getRate(){

        rate=7.50;

   }

/end of CommercialPlan class.

class  InstitutionalPlan extends Plan{

   //@override

    public void getRate(){

        rate=5.50;

   }

/end of InstitutionalPlan class.

Step 3: Create a GetPlanFactory to generate object of concrete classes based on given information

class GetPlanFactory{

       public Plan getPlan(String planType){

            if(planType == null){

             return null;

            }

          if(planType.equalsIgnoreCase("DOMESTICPLAN")) {

                 return new DomesticPlan();

               }

           else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){

                return new CommercialPlan();

            }

          else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {

                return new InstitutionalPlan();

          }

      return null;

   }

}

Step 4:Generate Bill by using the GetPlanFactory to get the object of concrete classes by passing an information such as type of plan DOMESTICPLAN or COMMERCIALPLAN or INSTITUTIONALPLAN.

import java.io.\*;

class GenerateBill{

    public static void main(String args[])throws IOException{

      GetPlanFactory planFactory = new GetPlanFactory();

      System.out.print("Enter the name of plan for which the bill will be generated: ");

      BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

      String planName=br.readLine();

      System.out.print("Enter the number of units for bill will be calculated: ");

      int units=Integer.parseInt(br.readLine());

      Plan p = planFactory.getPlan(planName);

      //call getRate() method and calculateBill()method of DomesticPaln.

       System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");

           p.getRate();

           p.calculateBill(units);

            }

    }

1. **Abstract Factory**

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories

Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

Implementation :

Step 1

Create an interface for Shapes and Colors.

*Shape.java* public class OperationAdd implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 + num2;

}

}

Step 7

Verify the output.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Inside RoundedRectangle::draw() method.

Inside RoundedSquare::draw() method.

1. **Builder**

Builder pattern builds a complex object using simple objects and using a step by step approach

A Builder class builds the final object step by step. This builder is independent of other objects.

Intent

Defines an instance for creating an object but letting subclasses decide which class to instantiate

Refers to the newly created object through a common interface

1. **Behavioral design patterns**
   1. **Strategy**

* In Strategy pattern, a class behavior or its algorithm can be changed at run time.
* 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 :

## 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

* 1. 5 = 50

**ii) Command**

*It encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command*"

Intent

encapsulate a request in an object

- allows the parameterization of clients with different requests

- allows saving the requests in a queue

Implementation of above UML:

#### Step 1

#### Create a ActionListernerCommand interface that will act as a Command.

#### public interface ActionListenerCommand {

#### public void execute();

#### }

#### Step 2

#### Create a Document class that will act as a Receiver.

#### File: Document.java

#### public class Document {

#### public void open(){

#### System.out.println("Document Opened");

#### }

#### public void save(){

#### System.out.println("Document Saved");

#### }

#### }

#### Step 3

#### Create a ActionOpen class that will act as an ConcreteCommand.

#### File: ActionOpen.java

#### public class ActionOpen implements ActionListenerCommand{

#### private Document doc;

#### public ActionOpen(Document doc) {

#### this.doc = doc;

#### }

#### @Override

#### public void execute() {

#### doc.open();

#### }

#### }

#### Step 4

#### Create a ActionSave class that will act as an ConcreteCommand.

#### File: AdapterPatternDemo.java

#### public class ActionSave implements ActionListenerCommand{

#### private Document doc;

#### public ActionSave(Document doc) {

#### this.doc = doc;

#### }

#### @Override

#### public void execute() {

#### doc.save();

#### }

#### }

#### Step 5

#### Create a MenuOptions class that will act as an Invoker.

#### File: ActionSave.java

#### public class ActionSave implements ActionListenerCommand{

#### private Document doc;

#### public ActionSave(Document doc) {

#### this.doc = doc;

#### }

#### @Override

#### public void execute() {

#### doc.save();

#### }

#### }

#### Step 6

#### Create a CommanPatternClient class that will act as a Client.

#### File: AdapterPatternDemo.java

#### public class CommandPatternClient {

#### public static void main(String[] args) {

#### Document doc = new Document();

#### 

#### ActionListenerCommand clickOpen = new ActionOpen(doc);

#### ActionListenerCommand clickSave = new ActionSave(doc);

#### 

#### MenuOptions menu = new MenuOptions(clickOpen, clickSave);

#### 

#### menu.clickOpen();

#### menu.clickSave();

#### }

#### }

**iii)Chain of Responsibility**

Intent:

It avoids attaching the sender of a request to its receiver, giving this way other objects the possibility of handling the request too.

The objects become parts of a chain and the request is sent from one object to another across the chain until one of the objects will handle it.

Implementation

Step 1

Create a Logger abstract class.

public abstract class Logger {

public static int OUTPUTINFO=1;

public static int ERRORINFO=2;

public static int DEBUGINFO=3;

protected int levels;

protected Logger nextLevelLogger;

public void setNextLevelLogger(Logger nextLevelLogger) {

this.nextLevelLogger = nextLevelLogger;

}

public void logMessage(int levels, String msg){

if(this.levels<=levels){

displayLogInfo(msg);

}

if (nextLevelLogger!=null) {

nextLevelLogger.logMessage(levels, msg);

}

}

protected abstract void displayLogInfo(String msg);

}

Step 2

Create a ConsoleBasedLogger class.

File: ConsoleBasedLogger.java

public class ConsoleBasedLogger extends Logger {

public ConsoleBasedLogger(int levels) {

this.levels=levels;

}

@Override

protected void displayLogInfo(String msg) {

System.out.println("CONSOLE LOGGER INFO: "+msg);

}

}

Step 3

Create a DebugBasedLogger class.

File: DebugBasedLogger.java

public class DebugBasedLogger extends Logger {

public DebugBasedLogger(int levels) {

this.levels=levels;

}

@Override

protected void displayLogInfo(String msg) {

System.out.println("DEBUG LOGGER INFO: "+msg);

}

}// End of the DebugBasedLogger class.

Step 4

Create a ErrorBasedLogger class.

File: ErrorBasedLogger.java

public class ErrorBasedLogger extends Logger {

public ErrorBasedLogger(int levels) {

this.levels=levels;

}

@Override

protected void displayLogInfo(String msg) {

System.out.println("ERROR LOGGER INFO: "+msg);

}

}// End of the ErrorBasedLogger class.

Step 5

Create a ChainOfResponsibilityClient class.

File: ChainofResponsibilityClient.java

public class ChainofResponsibilityClient {

private static Logger doChaining(){

Logger consoleLogger = new ConsoleBasedLogger(Logger.OUTPUTINFO);

Logger errorLogger = new ErrorBasedLogger(Logger.ERRORINFO);

consoleLogger.setNextLevelLogger(errorLogger);

Logger debugLogger = new DebugBasedLogger(Logger.DEBUGINFO);

errorLogger.setNextLevelLogger(debugLogger);

return consoleLogger;

}

public static void main(String args[]){

Logger chainLogger= doChaining();

chainLogger.logMessage(Logger.OUTPUTINFO, "Enter the sequence of values ");

chainLogger.logMessage(Logger.ERRORINFO, "An error is occured now");

chainLogger.logMessage(Logger.DEBUGINFO, "This was the error now debugging is compeled");

}

}

**iv) Observer**

It says that "just define a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically".

The Memento pattern is also known as Dependents or Publish-Subscribe.

Need:

It describes the coupling between the objects and the observer.

It provides the support for broadcast-type communication.

Usage:

When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.

When the framework we writes and needs to be enhanced in future with new observers with minimal chamges.

Implementation of Observer Pattern

Step 1:

Create a ResponseHandler1 class the will implement the java.util.Observer interface.

import java.util.Observable;

import java.util.Observer;

public class ResponseHandler1 implements Observer {

private String resp;

public void update(Observable obj, Object arg) {

if (arg instanceof String) {

resp = (String) arg;

System.out.println("\nReceived Response: " + resp );

}

}

}

Step 2:

Create a ResponseHandler2 class the will implement the java.util.Observer interface.

import java.util.Observable;

import java.util.Observer;

public class ResponseHandler2 implements Observer {

private String resp;

public void update(Observable obj, Object arg) {

if (arg instanceof String) {

resp = (String) arg;

System.out.println("\nReceived Response: " + resp );

}

}

}

Step 3:

Create an EventSource class that will extend the java.util.Observable class .

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.Observable;

public class EventSource extends Observable implements Runnable {

@Override

public void run() {

try {

final InputStreamReader isr = new InputStreamReader(System.in);

final BufferedReader br = new BufferedReader(isr);

while (true) {

String response = br.readLine();

setChanged();

notifyObservers(response);

}

}

catch (IOException e) {

e.printStackTrace();

}

}

}

**3 Structural design patterns**

**i) Adaptor**

Intent

Convert the interface of a class into another interface clients expect.

Adapter lets classes work together, that could not otherwise because of incompatible interfaces**.**

**ii) Decorator**

Definition:

The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Intent

The intent of this pattern is to add additional responsibilities dynamically to an object.

Advantages:

The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime.

The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.

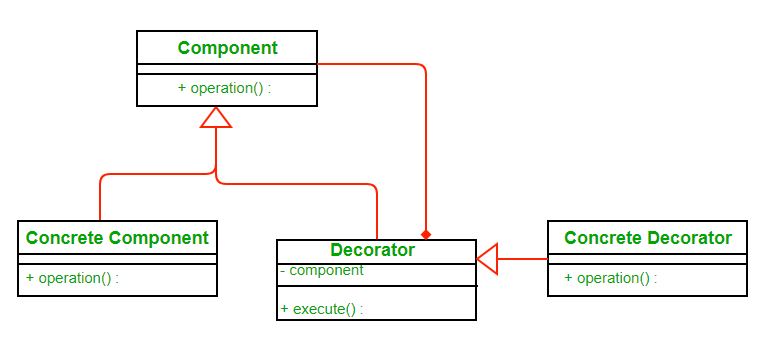
Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.

Disadvantages:

Decorators can complicate the process of instantiating the component because you not only have to instantiate the component, but wrap it in a number of decorators.

It can be complicated to have decorators keep track of other decorators, because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent.

Implementation :



**iii) Flyweight**

A Flyweight Pattern says that just "to reuse already existing similar kind of objects by storing them and create new object when no matching object is found".

Intent

The intent of this pattern is to use sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary.

Advantage of Flyweight Pattern

It reduces the number of objects.

It reduces the amount of memory and storage devices required if the objects are persisted

Implementation :

Step 1

Create an interface.

Shape.java

public interface Shape {

void draw();

}

Step 2

Create concrete class implementing the same interface.

Circle.java

public class Circle implements Shape {

private String color;

private int x;

private int y;

private int radius;

public Circle(String color){

this.color = color;

}

public void setX(int x) {

this.x = x;

}

public void setY(int y) {

this.y = y;

}

public void setRadius(int radius) {

this.radius = radius;

}

@Override

public void draw() {

System.out.println("Circle: Draw() [Color : " + color + ", x : " + x + ", y :" + y + ", radius :" + radius);

}

}

Step 3

Create a factory to generate object of concrete class based on given information.

ShapeFactory.java

import java.util.HashMap;

public class ShapeFactory {

private static final HashMap circleMap = new HashMap();

public static Shape getCircle(String color) {

Circle circle = (Circle)circleMap.get(color);

if(circle == null) {

circle = new Circle(color);

circleMap.put(color, circle);

System.out.println("Creating circle of color : " + color);

}

return circle;

}

}

Step 4

Use the factory to get object of concrete class by passing an information such as color.

FlyweightPatternDemo.java

public class FlyweightPatternDemo {

private static final String colors[] = { "Red", "Green", "Blue", "White", "Black" };

public static void main(String[] args) {

for(int i=0; i < 20; ++i) {

Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());

circle.setX(getRandomX());

circle.setY(getRandomY());

circle.setRadius(100);

circle.draw();

}

}

private static String getRandomColor() {

return colors[(int)(Math.random()\*colors.length)];

}

private static int getRandomX() {

return (int)(Math.random()\*100 );

}

private static int getRandomY() {

return (int)(Math.random()\*100);

}

}

Step 5

Verify the output.

Creating circle of color : Black

Circle: Draw() [Color : Black, x : 36, y :71, radius :100

Creating circle of color : Green

Circle: Draw() [Color : Green, x : 27, y :27, radius :100

Creating circle of color : White

Circle: Draw() [Color : White, x : 64, y :10, radius :100

Creating circle of color : Red

Circle: Draw() [Color : Red, x : 15, y :44, radius :100

Circle: Draw() [Color : Green, x : 19, y :10, radius :100

Circle: Draw() [Color : Green, x : 94, y :32, radius :100

Circle: Draw() [Color : White, x : 69, y :98, radius :100

Creating circle of color : Blue

Circle: Draw() [Color : Blue, x : 13, y :4, radius :100

Circle: Draw() [Color : Green, x : 21, y :21, radius :100

Circle: Draw() [Color : Blue, x : 55, y :86, radius :100

Circle: Draw() [Color : White, x : 90, y :70, radius :100

Circle: Draw() [Color : Green, x : 78, y :3, radius :100

Circle: Draw() [Color : Green, x : 64, y :89, radius :100

Circle: Draw() [Color : Blue, x : 3, y :91, radius :100

Circle: Draw() [Color : Blue, x : 62, y :82, radius :100

Circle: Draw() [Color : Green, x : 97, y :61, radius :100

Circle: Draw() [Color : Green, x : 86, y :12, radius :100

Circle: Draw() [Color : Green, x : 38, y :93, radius :100

Circle: Draw() [Color : Red, x : 76, y :82, radius :100

Circle: Draw() [Color : Blue, x : 95, y :82, radius :100