**Design Pattern**

|  |  |
| --- | --- |
| 1 | Creational Patterns These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case. |
| 2 | Structural Patterns These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities. |
| 3 | Behavioral Patterns These design patterns are specifically concerned with communication between objects. |
| 4 | J2EE Patterns These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center. |

**Factory Pattern**

Factory pattern is one of most used design pattern in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the*Shape* interface. A factory class *ShapeFactory* is defined as a next step.

*FactoryPatternDemo*, our demo class will use *ShapeFactory* to get a *Shape*object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to*ShapeFactory* to get the type of object it needs.



Step 1

Create an interface.

*Shape.java*

public interface Shape {

void draw();

}

Step 2

Create concrete classes implementing the same interface.

*Rectangle.java*

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

*Square.java*

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Inside Square::draw() method.");

}

}

*Circle.java*

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Inside Circle::draw() method.");

}

}

Step 3

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

*ShapeFactory.java*

public class ShapeFactory {

//use getShape method to get object of type shape

public Shape getShape(String shapeType){

if(shapeType == null){

return null;

}

if(shapeType.equalsIgnoreCase("CIRCLE")){

return new Circle();

} else if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

} else if(shapeType.equalsIgnoreCase("SQUARE")){

return new Square();

}

return null;

}

}

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

*FactoryPatternDemo.java*

public class FactoryPatternDemo {

public static void main(String[] args) {

ShapeFactory shapeFactory = new ShapeFactory();

//get an object of Circle and call its draw method.

Shape shape1 = shapeFactory.getShape("CIRCLE");

//call draw method of Circle

shape1.draw();

//get an object of Rectangle and call its draw method.

Shape shape2 = shapeFactory.getShape("RECTANGLE");

//call draw method of Rectangle

shape2.draw();

//get an object of Square and call its draw method.

Shape shape3 = shapeFactory.getShape("SQUARE");

//call draw method of circle

shape3.draw();

}

}

Step 5

Verify the output.

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

**Abstract Factory Pattern**

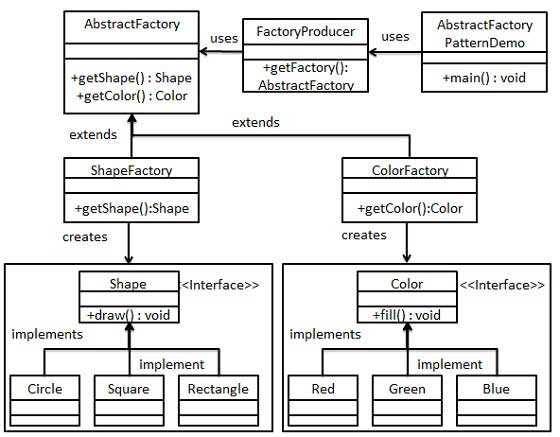
Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In 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**

We are going to create a *Shape* and *Color* interfaces and concrete classes implementing these interfaces. We create an abstract factory class*AbstractFactory* as next step. Factory classes *ShapeFactory* and *ColorFactory*are defined where each factory extends *AbstractFactory*. A factory creator/generator class *FactoryProducer* is created.

*AbstractFactoryPatternDemo*, our demo class uses *FactoryProducer* to get a*AbstractFactory* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*for *Shape*) to *AbstractFactory* to get the type of object it needs. It also passes information (*RED / GREEN / BLUE* for *Color*) to *AbstractFactory* to get the type of object it needs.



Step 1

Create an interface for Shapes.

*Shape.java*

public interface Shape {

void draw();

}

Step 2

Create concrete classes implementing the same interface.

*Rectangle.java*

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

*Square.java*

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Inside Square::draw() method.");

}

}

*Circle.java*

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Inside Circle::draw() method.");

}

}

Step 3

Create an interface for Colors.

*Color.java*

public interface Color {

void fill();

}

Step4

Create concrete classes implementing the same interface.

*Red.java*

public class Red implements Color {

@Override

public void fill() {

System.out.println("Inside Red::fill() method.");

}

}

*Green.java*

public class Green implements Color {

@Override

public void fill() {

System.out.println("Inside Green::fill() method.");

}

}

*Blue.java*

public class Blue implements Color {

@Override

public void fill() {

System.out.println("Inside Blue::fill() method.");

}

}

Step 5

Create an Abstract class to get factories for Color and Shape Objects.

*AbstractFactory.java*

public abstract class AbstractFactory {

abstract Color getColor(String color);

abstract Shape getShape(String shape) ;

}

Step 6

Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

*ShapeFactory.java*

public class ShapeFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

if(shapeType == null){

return null;

}

if(shapeType.equalsIgnoreCase("CIRCLE")){

return new Circle();

}else if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

}else if(shapeType.equalsIgnoreCase("SQUARE")){

return new Square();

}

return null;

}

@Override

Color getColor(String color) {

return null;

}

}

*ColorFactory.java*

public class ColorFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

return null;

}

@Override

Color getColor(String color) {

if(color == null){

return null;

}

if(color.equalsIgnoreCase("RED")){

return new Red();

}else if(color.equalsIgnoreCase("GREEN")){

return new Green();

}else if(color.equalsIgnoreCase("BLUE")){

return new Blue();

}

return null;

}

}

Step 7

Create a Factory generator/producer class to get factories by passing an information such as Shape or Color

*FactoryProducer.java*

public class FactoryProducer {

public static AbstractFactory getFactory(String choice){

if(choice.equalsIgnoreCase("SHAPE")){

return new ShapeFactory();

}else if(choice.equalsIgnoreCase("COLOR")){

return new ColorFactory();

}

return null;

}

}

Step 8

Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

*AbstractFactoryPatternDemo.java*

public class AbstractFactoryPatternDemo {

public static void main(String[] args) {

//get shape factory

AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

//get an object of Shape Circle

Shape shape1 = shapeFactory.getShape("CIRCLE");

//call draw method of Shape Circle

shape1.draw();

//get an object of Shape Rectangle

Shape shape2 = shapeFactory.getShape("RECTANGLE");

//call draw method of Shape Rectangle

shape2.draw();

//get an object of Shape Square

Shape shape3 = shapeFactory.getShape("SQUARE");

//call draw method of Shape Square

shape3.draw();

//get color factory

AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");

//get an object of Color Red

Color color1 = colorFactory.getColor("RED");

//call fill method of Red

color1.fill();

//get an object of Color Green

Color color2 = colorFactory.getColor("Green");

//call fill method of Green

color2.fill();

//get an object of Color Blue

Color color3 = colorFactory.getColor("BLUE");

//call fill method of Color Blue

color3.fill();

}

}

Step 9

Verify the output.

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Inside Red::fill() method.

Inside Green::fill() method.

Inside Blue::fill() method.

# Singleton Pattern

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

**Implementation**

We're going to create a *Single Object* class. *Single Object* class has its constructor as private and has a static instance of itself. *Single Object* class provides a static method to get its static instance to outside world.  *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.



**Step 1**

Create a Singleton Class.

*SingleObject.java*

public class SingleObject {

//create an object of SingleObject

private static SingleObject instance = new SingleObject();

//make the constructor private so that this class cannot be

//instantiated

private SingleObject(){}

//Get the only object available

public static SingleObject getInstance(){

return instance;

}

public void showMessage(){

System.out.println("Hello World!");

}

}

**Step 2**

Get the only object from the singleton class.

*SingletonPatternDemo.java*

public class SingletonPatternDemo {

public static void main(String[] args) {

//illegal construct

//Compile Time Error: The constructor SingleObject() is not visible

//SingleObject object = new SingleObject();

//Get the only object available

SingleObject object = SingleObject.getInstance();

//show the message

object.showMessage();

}

}

**Step 3**

Verify the output.

Hello World!

# Facade Pattern

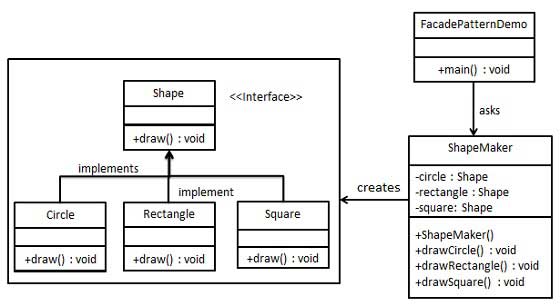
Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

**Implementation**

We are going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A facade class *ShapeMaker* is defined as a next step.

*ShapeMaker* class uses the concrete classes to delegate user calls to these classes. *FacadePatternDemo*, our demo class, will use *ShapeMaker* class to show the results.



**Step 1**

Create an interface.

*Shape.java*

public interface Shape {

void draw();

}

**Step 2**

Create concrete classes implementing the same interface.

*Rectangle.java*

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Rectangle::draw()");

}

}

*Square.java*

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Square::draw()");

}

}

*Circle.java*

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Circle::draw()");

}

}

**Step 3**

Create a facade class.

*ShapeMaker.java*

public class ShapeMaker {

private Shape circle;

private Shape rectangle;

private Shape square;

public ShapeMaker() {

circle = new Circle();

rectangle = new Rectangle();

square = new Square();

}

public void drawCircle(){

circle.draw();

}

public void drawRectangle(){

rectangle.draw();

}

public void drawSquare(){

square.draw();

}

}

**Step 4**

Use the facade to draw various types of shapes.

*FacadePatternDemo.java*

public class FacadePatternDemo {

public static void main(String[] args) {

ShapeMaker shapeMaker = new ShapeMaker();

shapeMaker.drawCircle();

shapeMaker.drawRectangle();

shapeMaker.drawSquare();

}

}

**Step 5**

Verify the output.

Circle::draw()

Rectangle::draw()

Square::draw()

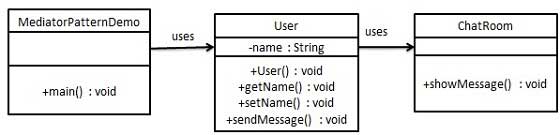
# Mediator Pattern

Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling. Mediator pattern falls under behavioral pattern category.

**Implementation**

We are demonstrating mediator pattern by example of a chat room where multiple users can send message to chat room and it is the responsibility of chat room to show the messages to all users. We have created two classes*ChatRoom* and *User*. *User* objects will use *ChatRoom* method to share their messages.

*MediatorPatternDemo*, our demo class, will use *User* objects to show communication between them.



**Step 1**

Create mediator class.

*ChatRoom.java*

import java.util.Date;

public class ChatRoom {

public static void showMessage(User user, String message){

System.out.println(new Date().toString() + " [" + user.getName() + "] : " + message);

}

}

**Step 2**

Create user class

*User.java*

public class User {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public User(String name){

this.name = name;

}

public void sendMessage(String message){

ChatRoom.showMessage(this,message);

}

}

**Step 3**

Use the *User* object to show communications between them.

*MediatorPatternDemo.java*

public class MediatorPatternDemo {

public static void main(String[] args) {

User robert = new User("Robert");

User john = new User("John");

robert.sendMessage("Hi! John!");

john.sendMessage("Hello! Robert!");

}

}

**Step 4**

Verify the output.

Thu Jan 31 16:05:46 IST 2013 [Robert] : Hi! John!

Thu Jan 31 16:05:46 IST 2013 [John] : Hello! Robert!

**Data Access Object Pattern**

Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.

* Data Access Object Interface - This interface defines the standard operations to be performed on a model object(s).
* Data Access Object concrete class - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
* Model Object or Value Object - This object is simple POJO containing get/set methods to store data retrieved using DAO class.

**Implementation**

We are going to create a *Student* object acting as a Model or Value Object.*StudentDao* is Data Access Object Interface.*StudentDaoImpl* is concrete class implementing Data Access Object Interface. *DaoPatternDemo*, our demo class, will use *StudentDao* to demonstrate the use of Data Access Object pattern.



Step 1

Create Value Object.

*Student.java*

public class Student {

private String name;

private int rollNo;

Student(String name, int rollNo){

this.name = name;

this.rollNo = rollNo;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getRollNo() {

return rollNo;

}

public void setRollNo(int rollNo) {

this.rollNo = rollNo;

}

}

Step 2

Create Data Access Object Interface.

*StudentDao.java*

import java.util.List;

public interface StudentDao {

public List<Student> getAllStudents();

public Student getStudent(int rollNo);

public void updateStudent(Student student);

public void deleteStudent(Student student);

}

Step 3

Create concrete class implementing above interface.

*StudentDaoImpl.java*

import java.util.ArrayList;

import java.util.List;

public class StudentDaoImpl implements StudentDao {

//list is working as a database

List<Student> students;

public StudentDaoImpl(){

students = new ArrayList<Student>();

Student student1 = new Student("Robert",0);

Student student2 = new Student("John",1);

students.add(student1);

students.add(student2);

}

@Override

public void deleteStudent(Student student) {

students.remove(student.getRollNo());

System.out.println("Student: Roll No " + student.getRollNo() + ", deleted from database");

}

//retrive list of students from the database

@Override

public List<Student> getAllStudents() {

return students;

}

@Override

public Student getStudent(int rollNo) {

return students.get(rollNo);

}

@Override

public void updateStudent(Student student) {

students.get(student.getRollNo()).setName(student.getName());

System.out.println("Student: Roll No " + student.getRollNo() + ", updated in the database");

}

}

Step 4

Use the *StudentDao* to demonstrate Data Access Object pattern usage.

*DaoPatternDemo.java*

public class DaoPatternDemo {

public static void main(String[] args) {

StudentDao studentDao = new StudentDaoImpl();

//print all students

for (Student student : studentDao.getAllStudents()) {

System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");

}

//update student

Student student =studentDao.getAllStudents().get(0);

student.setName("Michael");

studentDao.updateStudent(student);

//get the student

studentDao.getStudent(0);

System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");

}

}

Step 5

Verify the output.

Student: [RollNo : 0, Name : Robert ]

Student: [RollNo : 1, Name : John ]

Student: Roll No 0, updated in the database

Student: [RollNo : 0, Name : Michael ]

**Front Controller Pattern**

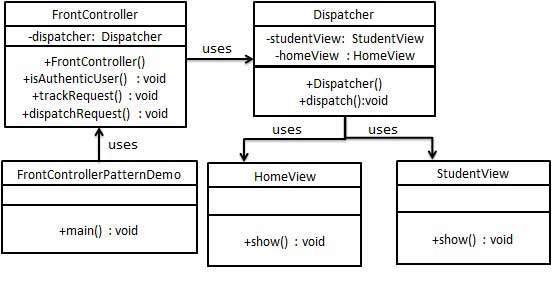
The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. This handler can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers. Following are the entities of this type of design pattern.

* Front Controller - Single handler for all kinds of requests coming to the application (either web based/ desktop based).
* Dispatcher - Front Controller may use a dispatcher object which can dispatch the request to corresponding specific handler.
* View - Views are the object for which the requests are made.

**Implementation**

We are going to create a *FrontController* and *Dispatcher* to act as Front Controller and Dispatcher correspondingly. *HomeView* and *StudentView*represent various views for which requests can come to front controller.

*FrontControllerPatternDemo*, our demo class, will use *FrontController* to demonstrate Front Controller Design Pattern.



Step 1

Create Views.

*HomeView.java*

public class HomeView {

public void show(){

System.out.println("Displaying Home Page");

}

}

*StudentView.java*

public class StudentView {

public void show(){

System.out.println("Displaying Student Page");

}

}

Step 2

Create Dispatcher.

*Dispatcher.java*

public class Dispatcher {

private StudentView studentView;

private HomeView homeView;

public Dispatcher(){

studentView = new StudentView();

homeView = new HomeView();

}

public void dispatch(String request){

if(request.equalsIgnoreCase("STUDENT")){

studentView.show();

}

else{

homeView.show();

}

}

}

Step 3

Create FrontController

*FrontController.java*

public class FrontController {

private Dispatcher dispatcher;

public FrontController(){

dispatcher = new Dispatcher();

}

private boolean isAuthenticUser(){

System.out.println("User is authenticated successfully.");

return true;

}

private void trackRequest(String request){

System.out.println("Page requested: " + request);

}

public void dispatchRequest(String request){

//log each request

trackRequest(request);

//authenticate the user

if(isAuthenticUser()){

dispatcher.dispatch(request);

}

}

}

Step 4

Use the *FrontController* to demonstrate Front Controller Design Pattern.

*FrontControllerPatternDemo.java*

public class FrontControllerPatternDemo {

public static void main(String[] args) {

FrontController frontController = new FrontController();

frontController.dispatchRequest("HOME");

frontController.dispatchRequest("STUDENT");

}

}

Step 5

Verify the output.

Page requested: HOME

User is authenticated successfully.

Displaying Home Page

Page requested: STUDENT

User is authenticated successfully.

Displaying Student Page