Patrón Factory

El patrón factory es uno de los patrones de diseño más utilizados en Java. Este tipo de patrón de diseño se incluye en el patrón de creación, ya que proporciona una de las mejores formas de crear un objeto.

En el patrón Factory, creamos un objeto sin exponer la lógica de creación al cliente y nos referimos a un objeto recién creado usando una interfaz común.

Implementación

Vamos a crear una interfaz *Shape y clases concretas implementando la interfaz Shape* . Una clase de fábrica *ShapeFactory* se define como el siguiente paso.

*FactoryPatternDemo* , nuestra clase de demostración usará *ShapeFactory* para obtener un objeto *Shape* . Pasará información ( *CÍRCULO / RECTÁNGULO / CUADRADO* ) a *ShapeFactory* para obtener el tipo de objeto que necesita.



Paso 1

Crea una interfaz.

*Forma.java*

public interface Shape {

void draw();

}

Paso 2

Crear clases concretas implementando la misma interfaz.

*Rectángulo.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.");

}

}

*círculo.java*

public class Circle implements Shape {

@Override

public void draw() {

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

}

}

Paso 3

Cree una fábrica para generar objetos de clase concreta en función de la información dada.

*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;

}

}

Etapa 4

Use Factory para obtener un objeto de una clase concreta pasando información como el tipo.

*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 square

shape3.draw();

}

}

Paso 5

Verifique la salida.

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Patrón Abstract Factory

Los patrones de Abstract Factory funcionan en torno a una superfábrica que crea otras fábricas. Esta fábrica también se denomina fábrica de fábricas. Este tipo de patrón de diseño se incluye en el patrón de creación, ya que proporciona una de las mejores formas de crear un objeto.

En el patrón Abstract Factory, una interfaz es responsable de crear una fábrica de objetos relacionados sin especificar explícitamente sus clases. Cada fábrica generada puede dar los objetos según el patrón de fábrica.

Implementación

Vamos a crear una interfaz Shape y una clase concreta que la implemente. Creamos una clase de fábrica abstracta AbstractFactory como siguiente paso. Se define la clase de fábrica ShapeFactory, que amplía AbstractFactory. Se crea una clase de creador/generador de fábrica FactoryProducer.

AbstractFactoryPatternDemo, nuestra clase de demostración usa FactoryProducer para obtener un objeto AbstractFactory. Pasará información (CÍRCULO / RECTÁNGULO / CUADRADO para Forma) a AbstractFactory para obtener el tipo de objeto que necesita.



Paso 1

Cree una interfaz para Formas.

*Forma.java*

public interface Shape {

void draw();

}

Paso 2

Crear clases concretas implementando la misma interfaz.

*RectánguloRedondeado.java*

public class RoundedRectangle implements Shape {

@Override

public void draw() {

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

}

}

*RoundedSquare.java*

public class RoundedSquare implements Shape {

@Override

public void draw() {

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

}

}

*Rectángulo.java*

public class Rectangle implements Shape {

@Override

public void draw() {

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

}

}

Paso 3

Cree una clase abstracta para obtener fábricas para objetos de forma normal y redondeada.

*AbstractFactory.java*

public abstract class AbstractFactory {

abstract Shape getShape(String shapeType) ;

}

Etapa 4

Cree clases de fábrica que extiendan AbstractFactory para generar objetos de clases concretas en función de la información dada.

*ShapeFactory.java*

public class ShapeFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

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

return new Rectangle();

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

return new Square();

}

return null;

}

}

*RoundedShapeFactory.java*

public class RoundedShapeFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

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

return new RoundedRectangle();

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

return new RoundedSquare();

}

return null;

}

}

Paso 5

Cree una clase de generador/productor de fábrica para obtener fábricas pasando información como Forma

*FactoryProducer.java*

public class FactoryProducer {

public static AbstractFactory getFactory(boolean rounded){

if(rounded){

return new RoundedShapeFactory();

}else{

return new ShapeFactory();

}

}

}

Paso 6

Use FactoryProducer para obtener AbstractFactory para obtener fábricas de clases concretas pasando información como tipo.

*AbstractFactoryPatternDemo.java*

public class AbstractFactoryPatternDemo {

public static void main(String[] args) {

//get shape factory

AbstractFactory shapeFactory = FactoryProducer.getFactory(false);

//get an object of Shape Rectangle

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

//call draw method of Shape Rectangle

shape1.draw();

//get an object of Shape Square

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

//call draw method of Shape Square

shape2.draw();

//get shape factory

AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);

//get an object of Shape Rectangle

Shape shape3 = shapeFactory1.getShape("RECTANGLE");

//call draw method of Shape Rectangle

shape3.draw();

//get an object of Shape Square

Shape shape4 = shapeFactory1.getShape("SQUARE");

//call draw method of Shape Square

shape4.draw();

}

}

Paso 7

Verifique la salida.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Inside RoundedRectangle::draw() method.

Inside RoundedSquare::draw() method.

Patrón Singleton

El patrón Singleton es uno de los patrones de diseño más simples en Java. Este tipo de patrón de diseño se incluye en el patrón de creación, ya que proporciona una de las mejores formas de crear un objeto.

Este patrón involucra una sola clase que es responsable de crear un objeto mientras se asegura de que solo se cree un único objeto. Esta clase proporciona una forma de acceder a su único objeto al que se puede acceder directamente sin necesidad de instanciar el objeto de la clase.

Implementación

Vamos a crear una clase *SingleObject* . *La clase SingleObject* tiene su constructor como privado y tiene una instancia estática de sí mismo.

*La clase SingleObject* proporciona un método estático para llevar su instancia estática al mundo exterior. *SingletonPatternDemo* , nuestra clase de demostración usará la clase *SingleObject* para obtener un objeto *SingleObject* .



Paso 1

Crear una clase Singleton.

*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!");

}

}

Paso 2

Obtenga el único objeto de la clase singleton.

*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();

}

}

Paso 3

Verifique la salida.

Hello World!

Patrón Builder

El patrón Builder construye un objeto complejo usando objetos simples y usando un enfoque paso a paso. Este tipo de patrón de diseño se incluye en el patrón de creación, ya que proporciona una de las mejores formas de crear un objeto.

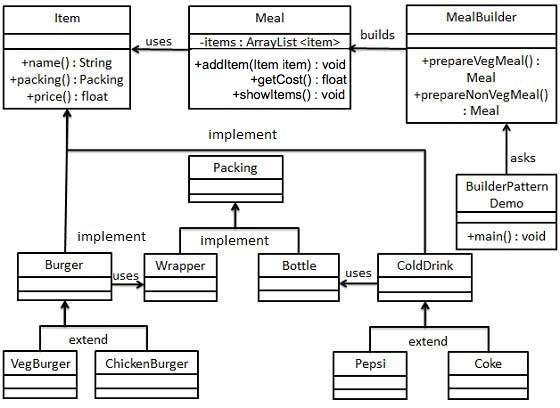
Una clase Builder construye el objeto final paso a paso. Este constructor es independiente de otros objetos.

Implementación

Hemos considerado un caso de negocio de restaurante de comida rápida donde una comida típica podría ser una hamburguesa y una bebida fría. La hamburguesa puede ser una hamburguesa vegetariana o una hamburguesa de pollo y se empaquetará en un envoltorio. La bebida fría puede ser una coca cola o una pepsi y se envasa en una botella.

Vamos a crear una interfaz *de elementos* que represente alimentos como hamburguesas y bebidas frías y clases concretas que implementen la interfaz *de elementos* y una interfaz *de empaque* que represente el empaque de alimentos y clases concretas que implementen la interfaz *de empaque* , ya que la hamburguesa se envasaría en un envoltorio y una bebida fría. sería embalado como botella.

Luego creamos una clase *Meal que tiene ArrayList* of *Item* y *MealBuilder* para crear diferentes tipos de objetos *Meal* combinando *Item* . *BuilderPatternDemo* , nuestra clase de demostración utilizará *MealBuilder* para crear una *comida* .



Paso 1

Cree un elemento de interfaz que represente el alimento y el empaque.

*Elemento.java*

public interface Item {

public String name();

public Packing packing();

public float price();

}

*Embalaje.java*

public interface Packing {

public String pack();

}

Paso 2

Crear clases concretas implementando la interfaz de Packing.

*Envoltorio.java*

public class Wrapper implements Packing {

@Override

public String pack() {

return "Wrapper";

}

}

*Botella.java*

public class Bottle implements Packing {

@Override

public String pack() {

return "Bottle";

}

}

Paso 3

Cree clases abstractas que implementen la interfaz del elemento proporcionando funcionalidades predeterminadas.

*Hamburguesa.java*

public abstract class Burger implements Item {

@Override

public Packing packing() {

return new Wrapper();

}

@Override

public abstract float price();

}

*BebidaFría.java*

public abstract class ColdDrink implements Item {

@Override

public Packing packing() {

return new Bottle();

}

@Override

public abstract float price();

}

Etapa 4

Cree clases concretas que amplíen las clases Burger y ColdDrink

*VegBurger.java*

public class VegBurger extends Burger {

@Override

public float price() {

return 25.0f;

}

@Override

public String name() {

return "Veg Burger";

}

}

*ChickenBurger.java*

public class ChickenBurger extends Burger {

@Override

public float price() {

return 50.5f;

}

@Override

public String name() {

return "Chicken Burger";

}

}

*Coca-Cola.java*

public class Coke extends ColdDrink {

@Override

public float price() {

return 30.0f;

}

@Override

public String name() {

return "Coke";

}

}

*Pepsi.java*

public class Pepsi extends ColdDrink {

@Override

public float price() {

return 35.0f;

}

@Override

public String name() {

return "Pepsi";

}

}

Paso 5

Cree una clase de comida que tenga los objetos Item definidos anteriormente.

*comida.java*

import java.util.ArrayList;

import java.util.List;

public class Meal {

private List<Item> items = new ArrayList<Item>();

public void addItem(Item item){

items.add(item);

}

public float getCost(){

float cost = 0.0f;

for (Item item : items) {

cost += item.price();

}

return cost;

}

public void showItems(){

for (Item item : items) {

System.out.print("Item : " + item.name());

System.out.print(", Packing : " + item.packing().pack());

System.out.println(", Price : " + item.price());

}

}

}

Paso 6

Cree una clase MealBuilder, la clase constructora real responsable de crear objetos Meal.

*MealBuilder.java*

public class MealBuilder {

public Meal prepareVegMeal (){

Meal meal = new Meal();

meal.addItem(new VegBurger());

meal.addItem(new Coke());

return meal;

}

public Meal prepareNonVegMeal (){

Meal meal = new Meal();

meal.addItem(new ChickenBurger());

meal.addItem(new Pepsi());

return meal;

}

}

Paso 7

BuiderPatternDemo usa MealBuider para demostrar el patrón de construcción.

*BuilderPatternDemo.java*

public class BuilderPatternDemo {

public static void main(String[] args) {

MealBuilder mealBuilder = new MealBuilder();

Meal vegMeal = mealBuilder.prepareVegMeal();

System.out.println("Veg Meal");

vegMeal.showItems();

System.out.println("Total Cost: " + vegMeal.getCost());

Meal nonVegMeal = mealBuilder.prepareNonVegMeal();

System.out.println("\n\nNon-Veg Meal");

nonVegMeal.showItems();

System.out.println("Total Cost: " + nonVegMeal.getCost());

}

}

Paso 8

Verifique la salida.

Veg Meal

Item : Veg Burger, Packing : Wrapper, Price : 25.0

Item : Coke, Packing : Bottle, Price : 30.0

Total Cost: 55.0

Non-Veg Meal

Item : Chicken Burger, Packing : Wrapper, Price : 50.5

Item : Pepsi, Packing : Bottle, Price : 35.0

Total Cost: 85.5

Patrón Prototype

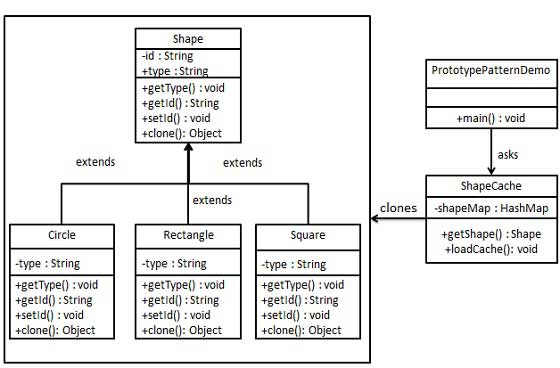
El patrón de prototipo se refiere a la creación de objetos duplicados teniendo en cuenta el rendimiento. Este tipo de patrón de diseño se incluye en el patrón de creación, ya que proporciona una de las mejores formas de crear un objeto.

Este patrón implica implementar una interfaz de prototipo que indica que se cree un clon del objeto actual. Este patrón se utiliza cuando la creación de objetos directamente es costosa. Por ejemplo, se va a crear un objeto después de una costosa operación de base de datos. Podemos almacenar en caché el objeto, devolver su clon en la próxima solicitud y actualizar la base de datos cuando sea necesario, reduciendo así las llamadas a la base de datos.

Implementación

Vamos a crear una clase abstracta *Shape* y clases concretas que extiendan la clase *Shape* . Una clase *ShapeCache* se define como el siguiente paso que almacena objetos de forma en un *Hashtable* y devuelve su clon cuando se le solicita.

*PrototypPatternDemo* , nuestra clase de demostración usará la clase *ShapeCache* para obtener un objeto *Shape* .



Paso 1

Cree una clase abstracta que implemente la interfaz *Clonable* .

*Forma.java*

public abstract class Shape implements Cloneable {

private String id;

protected String type;

abstract void draw();

public String getType(){

return type;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public Object clone() {

Object clone = null;

try {

clone = super.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return clone;

}

}

Paso 2

Cree clases concretas que extiendan la clase anterior.

*Rectángulo.java*

public class Rectangle extends Shape {

public Rectangle(){

type = "Rectangle";

}

@Override

public void draw() {

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

}

}

*Square.java*

public class Square extends Shape {

public Square(){

type = "Square";

}

@Override

public void draw() {

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

}

}

*círculo.java*

public class Circle extends Shape {

public Circle(){

type = "Circle";

}

@Override

public void draw() {

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

}

}

Paso 3

Cree una clase para obtener clases concretas de la base de datos y guárdelas en un *Hashtable* .

*ShapeCache.java*

import java.util.Hashtable;

public class ShapeCache {

private static Hashtable<String, Shape> shapeMap = new Hashtable<String, Shape>();

public static Shape getShape(String shapeId) {

Shape cachedShape = shapeMap.get(shapeId);

return (Shape) cachedShape.clone();

}

// for each shape run database query and create shape

// shapeMap.put(shapeKey, shape);

// for example, we are adding three shapes

public static void loadCache() {

Circle circle = new Circle();

circle.setId("1");

shapeMap.put(circle.getId(),circle);

Square square = new Square();

square.setId("2");

shapeMap.put(square.getId(),square);

Rectangle rectangle = new Rectangle();

rectangle.setId("3");

shapeMap.put(rectangle.getId(), rectangle);

}

}

Etapa 4

*PrototypePatternDemo* usa la clase *ShapeCache* para obtener clones de formas almacenadas en una *tabla Hash* .

*PrototypePatternDemo.java*

public class PrototypePatternDemo {

public static void main(String[] args) {

ShapeCache.loadCache();

Shape clonedShape = (Shape) ShapeCache.getShape("1");

System.out.println("Shape : " + clonedShape.getType());

Shape clonedShape2 = (Shape) ShapeCache.getShape("2");

System.out.println("Shape : " + clonedShape2.getType());

Shape clonedShape3 = (Shape) ShapeCache.getShape("3");

System.out.println("Shape : " + clonedShape3.getType());

}

}

Paso 5

Verifique la salida.

Shape : Circle

Shape : Square

Shape : Rectangle

Patrón Adapter

El patrón Adapter funciona como un puente entre dos interfaces incompatibles. Este tipo de patrón de diseño se incluye en el patrón estructural, ya que este patrón combina la capacidad de dos interfaces independientes.

Este patrón involucra una sola clase que es responsable de unir funcionalidades de interfaces independientes o incompatibles. Un ejemplo de la vida real podría ser el caso de un lector de tarjetas que actúa como un adaptador entre la tarjeta de memoria y una computadora portátil. Usted conecta la tarjeta de memoria en el lector de tarjetas y el lector de tarjetas en la computadora portátil para que la tarjeta de memoria se pueda leer a través de la computadora portátil.

Estamos demostrando el uso del patrón Adapter a través del siguiente ejemplo en el que un dispositivo reproductor de audio solo puede reproducir archivos mp3 y quiere usar un reproductor de audio avanzado capaz de reproducir archivos vlc y mp4.

Implementación

Tenemos una interfaz *MediaPlayer* y una clase concreta *AudioPlayer* implementando la interfaz *MediaPlayer* . *AudioPlayer* puede reproducir archivos de audio en formato mp3 de forma predeterminada.

Tenemos otra interfaz *AdvancedMediaPlayer* y clases concretas que implementan la interfaz *AdvancedMediaPlayer* . Estas clases pueden reproducir archivos en formato vlc y mp4.

Queremos que *AudioPlayer* también reproduzca otros formatos. Para lograr esto, hemos creado una clase de adaptador *MediaAdapter* que implementa la interfaz *MediaPlayer* y usa objetos *AdvancedMediaPlayer* para reproducir el formato requerido.

*AudioPlayer* utiliza la clase de adaptador *MediaAdapter* pasándole el tipo de audio deseado sin conocer la clase real que puede reproducir el formato deseado. *AdapterPatternDemo* , nuestra clase de demostración usará la clase *AudioPlayer* para reproducir varios formatos.



Paso 1

Cree interfaces para Media Player y Advanced Media Player.

*MediaPlayer.java*

public interface MediaPlayer {

public void play(String audioType, String fileName);

}

*AdvancedMediaPlayer.java*

public interface AdvancedMediaPlayer {

public void playVlc(String fileName);

public void playMp4(String fileName);

}

Paso 2

Cree clases concretas implementando la interfaz *AdvancedMediaPlayer .*

*VlcPlayer.java*

public class VlcPlayer implements AdvancedMediaPlayer{

@Override

public void playVlc(String fileName) {

System.out.println("Playing vlc file. Name: "+ fileName);

}

@Override

public void playMp4(String fileName) {

//do nothing

}

}

*Mp4Player.java*

public class Mp4Player implements AdvancedMediaPlayer{

@Override

public void playVlc(String fileName) {

//do nothing

}

@Override

public void playMp4(String fileName) {

System.out.println("Playing mp4 file. Name: "+ fileName);

}

}

Paso 3

Cree una clase de adaptador que implemente la interfaz *MediaPlayer .*

*MediaAdapter.java*

public class MediaAdapter implements MediaPlayer {

AdvancedMediaPlayer advancedMusicPlayer;

public MediaAdapter(String audioType){

if(audioType.equalsIgnoreCase("vlc") ){

advancedMusicPlayer = new VlcPlayer();

}else if (audioType.equalsIgnoreCase("mp4")){

advancedMusicPlayer = new Mp4Player();

}

}

@Override

public void play(String audioType, String fileName) {

if(audioType.equalsIgnoreCase("vlc")){

advancedMusicPlayer.playVlc(fileName);

}

else if(audioType.equalsIgnoreCase("mp4")){

advancedMusicPlayer.playMp4(fileName);

}

}

}

Etapa 4

Cree una clase concreta que implemente la interfaz *MediaPlayer .*

*reproductor de audio.java*

public class AudioPlayer implements MediaPlayer {

MediaAdapter mediaAdapter;

@Override

public void play(String audioType, String fileName) {

//inbuilt support to play mp3 music files

if(audioType.equalsIgnoreCase("mp3")){

System.out.println("Playing mp3 file. Name: " + fileName);

}

//mediaAdapter is providing support to play other file formats

else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){

mediaAdapter = new MediaAdapter(audioType);

mediaAdapter.play(audioType, fileName);

}

else{

System.out.println("Invalid media. " + audioType + " format not supported");

}

}

}

Paso 5

Use AudioPlayer para reproducir diferentes tipos de formatos de audio.

*AdapterPatternDemo.java*

public class AdapterPatternDemo {

public static void main(String[] args) {

AudioPlayer audioPlayer = new AudioPlayer();

audioPlayer.play("mp3", "beyond the horizon.mp3");

audioPlayer.play("mp4", "alone.mp4");

audioPlayer.play("vlc", "far far away.vlc");

audioPlayer.play("avi", "mind me.avi");

}

}

Paso 6

Verifique la salida.

Playing mp3 file. Name: beyond the horizon.mp3

Playing mp4 file. Name: alone.mp4

Playing vlc file. Name: far far away.vlc

Invalid media. avi format not supported

Patrón Observer

El patrón de observador se utiliza cuando existe una relación de uno a muchos entre los objetos, por ejemplo, si se modifica un objeto, sus objetos dependientes se notificarán automáticamente. El patrón del observador cae en la categoría de patrón de comportamiento.

Implementación

El patrón Observer utiliza tres clases de actores. Sujeto, observador y cliente. El sujeto es un objeto que tiene métodos para adjuntar y separar observadores de un objeto cliente. Hemos creado una clase abstracta *Observer* y una clase concreta *Subject* que es una extensión de la clase *Observer* .

*ObserverPatternDemo* , nuestra clase de demostración, utilizará el objeto de clase *Subject* y concreto para mostrar el patrón de observador en acción.



Paso 1

Crear clase Asunto.

*Asunto.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();

}

}

}

Paso 2

Crear clase de observador.

*observador.java*

public abstract class Observer {

protected Subject subject;

public abstract void update();

}

Paso 3

Crear clases de observadores concretos

*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() );

}

}

Etapa 4

Utilice *Sujeto* y objetos observadores concretos.

*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);

}

}

Paso 5

Verifique la salida.

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

Patrón Bridge

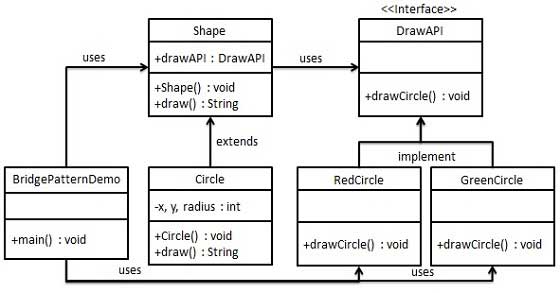
Bridge se usa cuando necesitamos desacoplar una abstracción de su implementación para que los dos puedan variar de forma independiente. Este tipo de patrón de diseño se incluye en el patrón estructural, ya que este patrón desacopla la clase de implementación y la clase abstracta al proporcionar una estructura de puente entre ellas.

Este patrón involucra una interfaz que actúa como un puente que hace que la funcionalidad de las clases concretas sea independiente de las clases del implementador de la interfaz. Ambos tipos de clases se pueden alterar estructuralmente sin afectarse entre sí.

Estamos demostrando el uso del patrón Bridge a través del siguiente ejemplo en el que se puede dibujar un círculo en diferentes colores usando el mismo método de clase abstracta pero diferentes clases de implementadores de puente.

Implementación

Tenemos una interfaz *DrawAPI* que actúa como un puente implementador y clases concretas *RedCircle* , *GreenCircle* implementando la interfaz *DrawAPI . Shape* es una clase abstracta y utilizará el objeto de *DrawAPI* . *BridgePatternDemo* , nuestra clase de demostración usará la clase *Shape* para dibujar círculos de diferentes colores.



Paso 1

Crear interfaz de implementador de puente.

*DrawAPI.java*

public interface DrawAPI {

public void drawCircle(int radius, int x, int y);

}

Paso 2

Cree clases de implementadores de puentes concretos que implementen la interfaz *DrawAPI .*

*RedCircle.java*

public class RedCircle implements DrawAPI {

@Override

public void drawCircle(int radius, int x, int y) {

System.out.println("Drawing Circle[ color: red, radius: " + radius + ", x: " + x + ", " + y + "]");

}

}

*GreenCircle.java*

public class GreenCircle implements DrawAPI {

@Override

public void drawCircle(int radius, int x, int y) {

System.out.println("Drawing Circle[ color: green, radius: " + radius + ", x: " + x + ", " + y + "]");

}

}

Paso 3

Cree una *forma* de clase abstracta utilizando la interfaz *DrawAPI* .

*Forma.java*

public abstract class Shape {

protected DrawAPI drawAPI;

protected Shape(DrawAPI drawAPI){

this.drawAPI = drawAPI;

}

public abstract void draw();

}

Etapa 4

Crea una clase concreta implementando la interfaz *Shape* .

*círculo.java*

public class Circle extends Shape {

private int x, y, radius;

public Circle(int x, int y, int radius, DrawAPI drawAPI) {

super(drawAPI);

this.x = x;

this.y = y;

this.radius = radius;

}

public void draw() {

drawAPI.drawCircle(radius,x,y);

}

}

Paso 5

Use las clases *Shape* y *DrawAPI* para dibujar círculos de diferentes colores.

*BridgePatternDemo.java*

public class BridgePatternDemo {

public static void main(String[] args) {

Shape redCircle = new Circle(100,100, 10, new RedCircle());

Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

redCircle.draw();

greenCircle.draw();

}

}

Paso 6

Verifique la salida.

Drawing Circle[ color: red, radius: 10, x: 100, 100]

Drawing Circle[ color: green, radius: 10, x: 100, 100]

Patrón MVC

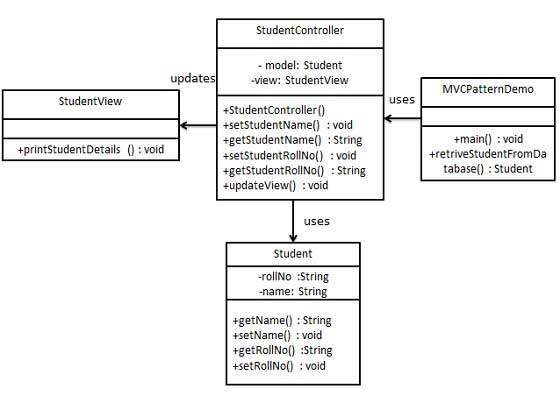
MVC Pattern significa Modelo-Vista-Controlador Patrón. Este patrón se utiliza para separar las preocupaciones de la aplicación.

* **Modelo** : el modelo representa un objeto o JAVA POJO que transporta datos. También puede tener lógica para actualizar el controlador si sus datos cambian.
* **Vista** : representa la visualización de los datos que contiene el modelo.
* **Controlador** : el controlador actúa tanto en el modelo como en la vista. Controla el flujo de datos en el objeto modelo y actualiza la vista cada vez que cambian los datos. Mantiene la vista y el modelo separados.

Implementación

Vamos a crear un objeto *Student* que actúe como modelo. *StudentView* será una clase de vista que puede imprimir los detalles de los estudiantes en la consola y *StudentController* es la clase de controlador responsable de almacenar datos en el objeto *Student* y actualizar la vista *StudentView* en consecuencia.

*MVCPatternDemo* , nuestra clase de demostración, usará *StudentController* para demostrar el uso del patrón MVC.



Paso 1

Crear modelo.

*Estudiante.java*

public class Student {

private String rollNo;

private String name;

public String getRollNo() {

return rollNo;

}

public void setRollNo(String rollNo) {

this.rollNo = rollNo;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

Paso 2

Crear vista.

*StudentView.java*

public class StudentView {

public void printStudentDetails(String studentName, String studentRollNo){

System.out.println("Student: ");

System.out.println("Name: " + studentName);

System.out.println("Roll No: " + studentRollNo);

}

}

Paso 3

Crear controlador.

*StudentController.java*

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view){

this.model = model;

this.view = view;

}

public void setStudentName(String name){

model.setName(name);

}

public String getStudentName(){

return model.getName();

}

public void setStudentRollNo(String rollNo){

model.setRollNo(rollNo);

}

public String getStudentRollNo(){

return model.getRollNo();

}

public void updateView(){

view.printStudentDetails(model.getName(), model.getRollNo());

}

}

Etapa 4

Use los métodos *de StudentController* para demostrar el uso del patrón de diseño de MVC.

*MVCPatternDemo.java*

public class MVCPatternDemo {

public static void main(String[] args) {

//fetch student record based on his roll no from the database

Student model = retriveStudentFromDatabase();

//Create a view : to write student details on console

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

//update model data

controller.setStudentName("John");

controller.updateView();

}

private static Student retriveStudentFromDatabase(){

Student student = new Student();

student.setName("Robert");

student.setRollNo("10");

return student;

}

}

Paso 5

Verifique la salida.

Student:

Name: Robert

Roll No: 10

Student:

Name: John

Roll No: 10

Patrón Strategy

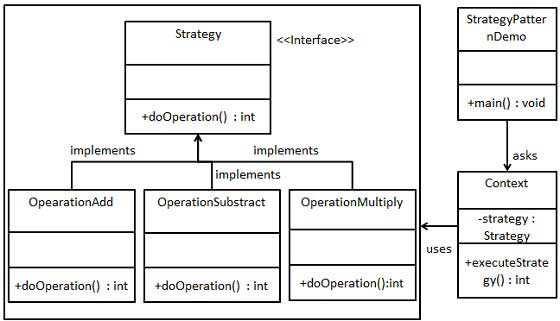
En el patrón de estrategia, el comportamiento de una clase o su algoritmo se puede cambiar en tiempo de ejecución. Este tipo de patrón de diseño se incluye en el patrón de comportamiento.

En el patrón de estrategia, creamos objetos que representan varias estrategias y un objeto de contexto cuyo comportamiento varía según su objeto de estrategia. El objeto de estrategia cambia el algoritmo de ejecución del objeto de contexto.

Implementación

Vamos a crear una interfaz *de estrategia que defina una acción y clases de estrategia concretas que implementen la interfaz de estrategia* . *El contexto* es una clase que utiliza una estrategia.

*StrategyPatternDemo* , nuestra clase de demostración, usará objetos *de contexto* y estrategia para demostrar el cambio en el comportamiento del contexto en función de la estrategia que implementa o usa.



Paso 1

Crea una interfaz.

*Estrategia.java*

public interface Strategy {

public int doOperation(int num1, int num2);

}

Paso 2

Crear clases concretas implementando la misma interfaz.

*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;

}

}

Paso 3

Crear clase *de contexto* .

*Contexto.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);

}

}

Etapa 4

Utilice el *contexto* para ver el cambio de comportamiento cuando cambia su *estrategia* .

*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));

}

}

Paso 5

Verifique la salida.

10 + 5 = 15

10 - 5 = 5

10 \* 5 = 50

Patrón DAO

El patrón de objeto de acceso a datos o patrón DAO se utiliza para separar las operaciones o la API de acceso a datos de bajo nivel de los servicios comerciales de alto nivel. Los siguientes son los participantes en el patrón de objetos de acceso a datos.

* **Interfaz de objeto de acceso a datos** : esta interfaz define las operaciones estándar que se realizarán en un objeto modelo.
* **Clase concreta de objeto de acceso a datos** : esta clase implementa la interfaz anterior. Esta clase es responsable de obtener datos de una fuente de datos que puede ser una base de datos/xml o cualquier otro mecanismo de almacenamiento.
* **Objeto modelo u objeto de valor** : este objeto es un POJO simple que contiene métodos get/set para almacenar datos recuperados mediante la clase DAO.

Implementación

Vamos a crear un objeto *de Estudiante* que actúe como un Modelo o un Objeto de Valor. *StudentDao* es la interfaz de objetos de acceso a datos. *StudentDaoImpl* es una clase concreta que implementa la interfaz de objetos de acceso a datos. *DaoPatternDemo* , nuestra clase de demostración, utilizará *StudentDao* para demostrar el uso del patrón de objetos de acceso a datos.



Paso 1

Crear objeto de valor.

*Estudiante.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;

}

}

Paso 2

Crear interfaz de objetos de acceso a datos.

*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);

}

Paso 3

Cree una clase concreta que implemente la interfaz anterior.

*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");

}

}

Etapa 4

Use *StudentDao* para demostrar el uso del patrón de objetos de acceso a datos.

*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() + " ]");

}

}

Paso 5

Verifique la salida.

Student: [RollNo : 0, Name : Robert ]

Student: [RollNo : 1, Name : John ]

Student: Roll No 0, updated in the database

Student: [RollNo : 0, Name : Michael ]