**Patrones de Diseño en Java**

Como analistas y programadores vamos desarrollando a diario nuestras habilidades para resolver problemas usuales que se presentan en el ***desarrollo del software***. Por cada problema que se nos presenta pensamos distintas formas de resolverlo, incluyendo soluciones exitosas que ya hemos usado anteriormente en problemas similares. Es así que a mayor experiencia que tengamos, nuestro abanico de posibilidades para resolver un problema crece, pero al final siempre habrá una sola solución que mejor se adapte a nuestra aplicación. Si documentamos esta solución, podemos reutilizarla y compartir esa información que hemos aprendido para resolver de la mejor manera un problema específico.

Los **patrones del diseño** tratan los problemas del diseño que se repiten y que se presentan en situaciones particulares del diseño, con el fin de proponer soluciones a ellas. Por lo tanto, los patrones de diseño son soluciones exitosas a problemas comunes. Existen muchas formas de implementar patrones de diseño. Los detalles de las implementaciones son llamadas estrategias.

En este artículo, veremos algunos patrones de diseño para las aplicaciones basadas en **J2EE**.

Breve historia de los patrones de diseño

Un *patrón de diseño* es una abstracción de una solución en un nivel alto. Los patrones solucionan problemas que existen en muchos niveles de abstracción. Hay patrones que abarcan las distintas etapas del desarrollo; desde el **análisis hasta el diseño** y desde la **arquitectura** hasta la **implementación**.

Muchos diseñadores y arquitectos de *software* han definido el término de ***patrón de diseño*** de varias formas que corresponden al ámbito a la cual se aplican los patrones. Luego, se dividió los patrones en diferentes categorías de acuerdo a su uso.

Los diseñadores de software extendieron la idea de **patrones de diseño** al proceso de desarrollo de software. Debido a las características que proporcionaron los *lenguajes orientados a objetos* (como herencia, abstracción y encapsulamiento) les permitieron relacionar entidades de los lenguajes de programación a entidades del mundo real fácilmente, los diseñadores empezaron a aplicar esas características para crear soluciones comunes y reutilizables para problemas frecuentes que exhibían patrones similares.

Fue por los años 1994, que apareció el libro "*Design Patterns: Elements of Reusable Object Oriented Sofware*" escrito por los ahora famosos **Gang of Four** (GoF, que en español es la pandilla de los cuatro) formada por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Ellos recopilaron y documentaron 23 patrones de diseño aplicados usualmente por expertos diseñadores de software orientado a objetos. Desde luego que ellos no son los inventores ni los únicos involucrados, pero ese fue luego de la publicación de ese libro que empezó a difundirse con más fuerza la idea de patrones de diseño.

El grupo de **GoF** clasificaron los patrones en 3 grandes categorías basadas en su **PROPÓSITO**: creacionales, estructurales y de comportamiento.

* **Creacionales**: Patrones creacionales tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.
* **Estructurales**: Los patrones estructurales describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionados pueden ser incluso objetos simples u objetos compuestos.
* **Comportamiento**: Los patrones de comportamiento nos ayudan a definir la comunicación e iteración entre los objetos de un sistema. El propósito de este patrón es reducir el acoplamiento entre los objetos.

En el segundo nivel, ellos clasificaron los patrones en 2 ámbitos: Clases y objetos. Es así que, tenemos 6 tipos de patrones:

|  |  |
| --- | --- |
| **Creacionales** | * **Creacional de la Clase** Los patrones creacionales de Clases usan la herencia como un mecanismo para lograr la instanciación de la Clase. Por ejemplo el método Factoría. * **Creacional del objeto** Los patrones creacionales de objetos son más escalables y dinámicos comparados de los patrones creacionales de Clases. Por ejemplo la Factoría abstracta y el patrón Singleton. |
| **Estructurales** | * **Estructural de la Clase** Los patrones estructurales de Clases usan la herencia para proporcionar interfaces más útiles combinando la funcionalidad de múltiples Clases. Por ejemplo el patrón Adaptador (Clase). * **Estructural de Objetos** Los patrones estructurales de objetos crean objetos complejos agregando objetos individuales para construir grandes estructuras. La composición de l patrón estructural del objeto puede ser cambiado en tiempo de ejecución, el cual nos da flexibilidad adicional sobre los patrones estructurales de Clases. Por ejemplo el Adaptador (Objeto), Facade, Bridge, Composite. |
| **Comportamiento** | * **Comportamiento de Clase** Los patrones de comportamiento de Clases usan la herencia para distribuir el comportamiento entre Clases. Por ejemplo Interpreter. * **Comportamiento de Objeto** Los patrones de comportamiento de objetos nos permite analizar los patrones de comunicación entre objetos interconectados, como objetos incluidos en un objeto complejo. Ejemplo Iterator, Observer, Visitor. |

# Patrones de Diseño

Los patrones de diseño consisten en dar soluciones a problemas generales.

Un patrón es la descripción de la solución a un problema en cierto contexto

## Factory(Fábrica)

El patrón **Factory** es uno de los patrones más usados en Java. Este patrón de diseño cabe dentro de la categoria de **creational pattern** ya que nos proporciona una de las mejores formas de crear un objeto

En este patrón nosotros creamos un objeto sin exponer la lógica al cliente y rifiriendonos al nuevo objeto usando una interfaz común.

### Implementación

Vamos a crear una interfaz Shape y clases concretas que la implementen. A continuación definiremos una clase ShapeFactory.

Nuestra clase FactoryPatternDemo usará ShapeFactory para obtener objetos de tipo Shape. Le pasará información(CIRCLE / RECTANGLE / SQUARE) a ShapeFactory para obtener los objetos que necesita. s Paso 1 ++++++ Creamos nuestra interfaz

public interface Shape {

void draw();

}

### Paso 2

Creamos las clases concretas implementando la interfaz

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.");

}

}

### Paso 3

Creamos nuestro objeto Fábrica(Factory) que genera nuestros objetos concretos basados en información proporcionada.

ShapeFactory.java

public class ShapeFactory {

//Usa el método getShape para obtener objetos de tipo 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;

}

}

### Paso 4

Usamos la Fábrica(Factory) para obtener los objetos de las clases concretas utilizando algún tipo de información como el tipo

FactoryPatternDemo.java

public class FactoryPatternDemo {

public static void main(String[] args) {

ShapeFactory shapeFactory = new ShapeFactory();

//obtenemos un objeto tipo Circle.

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

//llamamos al método draw de Circle

shape1.draw();

//obtenemos un objeto tipo Rectangle.

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

//llamamos al método draw de Rectangle

shape2.draw();

//obtenemos un objeto tipo Square.

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

//llamamos al método draw de circle

shape3.draw();

}

}

### Paso 5

Verificamos la salida

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

## Abstract Factory(Fábrica abstracta)

Los patrones Abstract Factory trabajan alrededor de una super-fábrica que crea otras fábricas. Esta fábrica también es llamada fábrica de fábricas. Este tipo de diseño cae en la categoría de **creational patter** ya que nos proporciona una de las mejores formas de crear un objeto

En el patrón Abstract Factory una interfaz es la responsable de crear una fábrica de objetos relacionados sin especificarle sus clases. Cada fábrica generada puede proporcionar los objetos de acuerdo con el patrón Factory.

### Implementación

Vamos a crear las interfaces Shape y Color y las clases concretas que las implementan. Creamos una clase de fábricas abstracta AbstractFactory en el siguiente paso. Definimos las clases Factory ShapeFactory y ColorFactory las cuales extienden Abstractactory. Y creamos una clase constructora/generadora FactoryProducer

AbstractFactoryPatternDemo usa FactoryProducer para obtener uno objeto de tipo AbstractFactory. Le pasará información (CIRCLE / RECTANGLE / SQUARE para Shape) a AbstractFactory para obtener el objeto que necesita. También le pasará información (RED / GREEN / BLUE para Color) a AbstractFactory para obtener el objeto que necesita.

### Paso 1

Creamos la interfaz Shape

Shape.java

public interface Shape {

void draw();

}

### Paso 2

Creamos las clases concretas implementando la interfaz.

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.");

}

}

### Paso 3

Creamos la interfaz Color

Color.java

public interface Color {

void fill();

}

### Paso 4

Creamos las clases concretas que implementan la interfaz.

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.");

}

}

### Paso 5

Creamos una clase abstracta para obtener las fábricas de objetos tipo Color y Shape.

AbstractFactory.java

public abstract class AbstractFactory {

abstract Color getColor(String color);

abstract Shape getShape(String shape) ;

}

### Paso 6

Creamos las clases Factory extendiendo de AbstractFactory para generar objetos de las clases concretas basados en información dada.

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;

}

}

### Paso 7

Creamos una clase Fábrica generadora para obtener fábricas pasandole información como Color o Shape

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;

}

}

### Paso 8

Usamos FactoryProducer para obtener AbstractFactory para obtener fábricas de clases concretas pasandoles información como el tipo

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();

}

}

### Paso 9

Verificamos la salida

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Inside Red::fill() method.

Inside Green::fill() method.

Inside Blue::fill() method.

## Composite

El patrón Composite es usado cuando necesitamos tratar un grupo de objetos en una manera similar a uno solo. El patrón compone objetos en términos de un árbol partes como la jerarquía completa. Este patrón de diseño cabe dentro de la categoria de **structural pattern** ya que crea una estructura de árbol a partir de un grupo de objetos.

Este patrón crea una clase que contiene grupos de sus mismos objetos. Esta clase proporciona maneras de modificar su grupo de mismos objetos.

Demostraremos el uso de este patrón con el siguiente ejemplo en el cual mostraremos la jerarquía de empleados en una organización.

### Implementación

Tenemos la clase Employee que tiene el rol de la clase actor en el patrón composite. CompositePatternDemo, usará la clase Employee para añadir jerarquía entre los departamentos e imprimir todos los empleados.

### Paso 1

Creamos la clase Employee con una lista de objetos tipo Employee.

Employee.java

import java.util.ArrayList;

import java.util.List;

public class Employee {

private String name;

private String dept;

private int salary;

private List<Employee> subordinates;

// constructor

public Employee(String name,String dept, int sal) {

this.name = name;

this.dept = dept;

this.salary = sal;

subordinates = new ArrayList<Employee>();

}

public void add(Employee e) {

subordinates.add(e);

}

public void remove(Employee e) {

subordinates.remove(e);

}

public List<Employee> getSubordinates(){

return subordinates;

}

public String toString(){

return ("Employee :[ Name : " + name + ", dept : " + dept + ", salary :" + salary+" ]");

}

}

### Paso 2

Usamos la clase Employee para crear e imprimir la jerarquía de empleados.

CompositePatternDemo

public class CompositePatternDemo {

public static void main(String[] args) {

Employee CEO = new Employee("John","CEO", 30000);

Employee headSales = new Employee("Robert","Head Sales", 20000);

Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

Employee clerk1 = new Employee("Laura","Marketing", 10000);

Employee clerk2 = new Employee("Bob","Marketing", 10000);

Employee salesExecutive1 = new Employee("Richard","Sales", 10000);

Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

CEO.add(headSales);

CEO.add(headMarketing);

headSales.add(salesExecutive1);

headSales.add(salesExecutive2);

headMarketing.add(clerk1);

headMarketing.add(clerk2);

//print all employees of the organization

System.out.println(CEO);

for (Employee headEmployee : CEO.getSubordinates()) {

System.out.println(headEmployee);

for (Employee employee : headEmployee.getSubordinates()) {

System.out.println(employee);

}

}

}

}

### Paso 3

Verificamos la salida

Employee :[ Name : John, dept : CEO, salary :30000 ]

Employee :[ Name : Robert, dept : Head Sales, salary :20000 ]

Employee :[ Name : Richard, dept : Sales, salary :10000 ]

Employee :[ Name : Rob, dept : Sales, salary :10000 ]

Employee :[ Name : Michel, dept : Head Marketing, salary :20000 ]

Employee :[ Name : Laura, dept : Marketing, salary :10000 ]

Employee :[ Name : Bob, dept : Marketing, salary :10000 ]

## State

En el patrón el comportamiento de una clase cambia dependiendo de su estado. Este tipo de patrón de diseño es de tipo behavior.

Creamos objetos que representan varios estados y un objeto de contexto cuyo comportamiento varia cuando su objeto de estado varia.

### Implementación

Vamos a crear la interfaz State definiendo una action y clase concretas implementando State. Context es una clase que contiene un State.

Usaremos Context y objetos de estado para demostrar el cambio de comportamiento en Context basado en el tipo de estado en el que se encuentra.

### Paso 1

Creamos una interfaz

State.java

public interface State {

public void doAction(Context context);

}

### Paso 2

Creamos la clases concretas implementando la misma interfaz.

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";

}

}

### Paso 3

Creamos la clase Context.

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;

}

}

### Paso 4

Usa el contexto para ver el cambio de comportamiente cuando State cambias

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());

}

}

### Paso 5

Verifica la salida

Player is in start state

Start State

Player is in stop state

Stop State

## MVC

## Fuentes

[tutorialspoint](http://www.tutorialspoint.com/design_pattern/index.htm)

[wikipediaState](http://www.wikiwand.com/en/State_pattern)