PATRÓN FACTORÍA ABSTRACTA

Objetivo

El objetivo de la guía es el de afianzar el conocimiento acerca del patrón Factoría Abstracta, para resolver problemas reales con el uso del lenguaje de programación Java. Además le presentará las diferentes variantes de dicho patrón y los pros y contras de cada una.

Competencias

- Entender el concepto de patrón Factoría Abstracta
- Utilizar el lenguaje Java para implementar el patrón
- Resolver problemas reales mediante el uso del patrón

Definición

Definición 1

El patrón Abstract Factory dice que sólo se define una interfaz o clase abstracta para crear familias de objetos relacionados (o dependientes), pero sin especificar sus subclases concretas. Esto significa que Abstract Factory permite que una clase devuelva una fábrica de clases. Por lo tanto, esta es la razón por la que patrón factoría abstracta es de más más alto nivel que el patrón de fábrica.

Un patrón factoría abstracta también se conoce como Kit.

Definición 2

Los patrones factoría abstracta trabajan alrededor de una super factoría que crea otras factorías. Este tipo de patrón de diseño viene es un patrón de creación ya que proporciona una de las mejores maneras 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 objetos según el patrón de la fábrica.

Ventaja del patrón factoría abstracta

- El patrón aísla el código del cliente de las clases concretas (de la ejecución).
- Facilita el intercambio de familias de objetos.

- Promueve la consistencia entre los objetos.

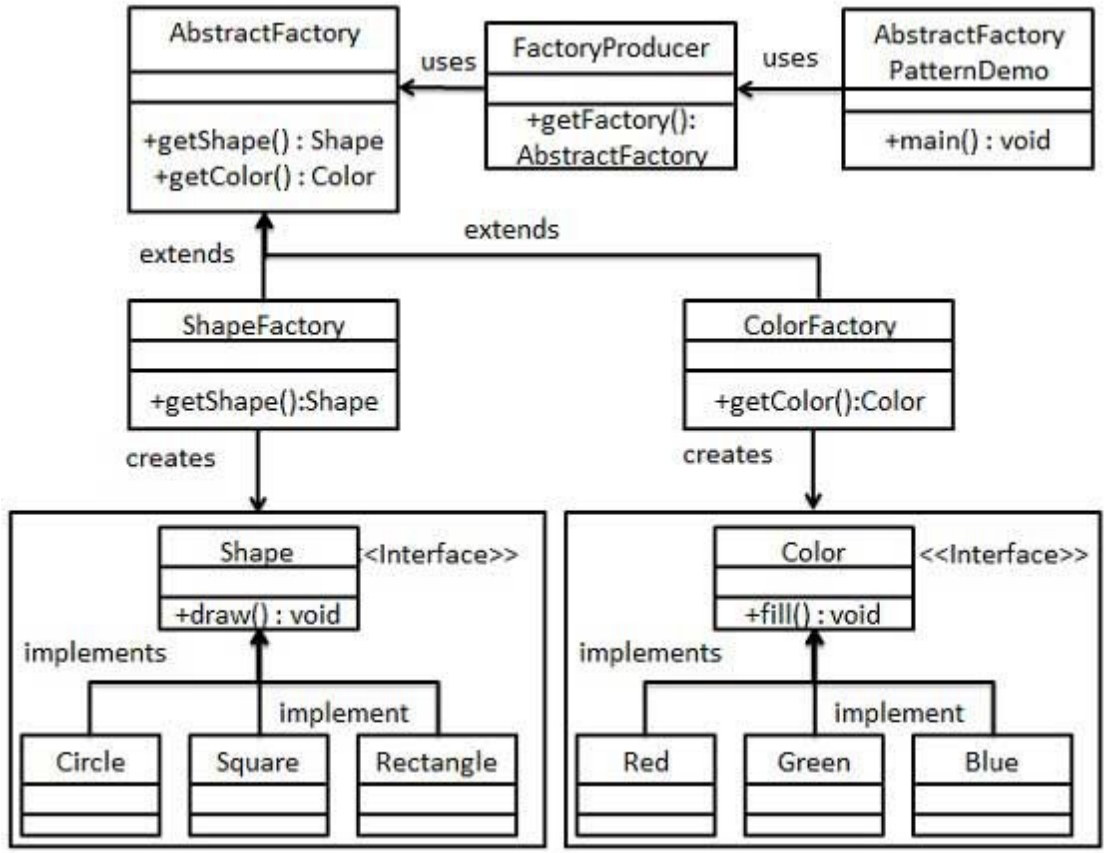
Uso de patrón factoría abstracta

- Cuando el sistema necesita ser independiente de cómo su objeto es creado, compuesto y representado.
- Cuando la familia de objetos relacionados tienen que usarse juntos, entonces esta restricción necesita ser reforzada.
  Cuando desea proporcionar una biblioteca de objetos que no muestra implementaciones y sólo revela interfaces.
- Cuando el sistema necesita ser configurado con uno de una familia múltiple de objetos

Implementación 1

Vamos a crear interfaces Shape and Color y clases concretas implementando estas interfaces. Crearemos luego una clase abstracta `AbstractFactory`. Las clases factoría `ShapeFactory` y `ColorFactory` extienden `AbstractFactory`. Se crea una clase creadora/generadora llamada `FactoryProducer`.

AbstractFactoryPatternDemo, será nuestra clase de demostración y utiliza `FactoryProducer` para obtener un objeto `AbstractFactory`. Se le pasará información (CIRCLE / RECTANGLE / SQUARE para Shape) a `AbstractFactory` para obtener el tipo de objeto que necesita. También se le pasará información (RED / GREEN / BLUE para Color) a `AbstractFactory` para obtener el tipo de objeto que necesita.



Paso 1

Crear la interfaz para Shapes.

Shape.java

```
public interface Shape {
   void draw();
}
```

Paso 2

Creamos las clases concretas que implementan la misma 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 una interfaz para Colors.

Color.java

```
public interface Color {
   void fill();
}
```

Paso 4

Creamos las clases concretas que implementan la misma 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 factorías para Color y Shape.

AbstractFactory.java

```
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Shape getShape(String shape) ;
}
```

Paso 6

Creamos una factoría que extienda AbstractFactory para generar objetos de la clase concreta basados en la 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 generadora/productora de factorías pasando información tal como Shape o 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;
   }
}
```

Paso 8

Ese la clase FactoryProducer para obtener AbstractFactory con el fin de obtener fábricas de clases concretas pasandoles información de 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

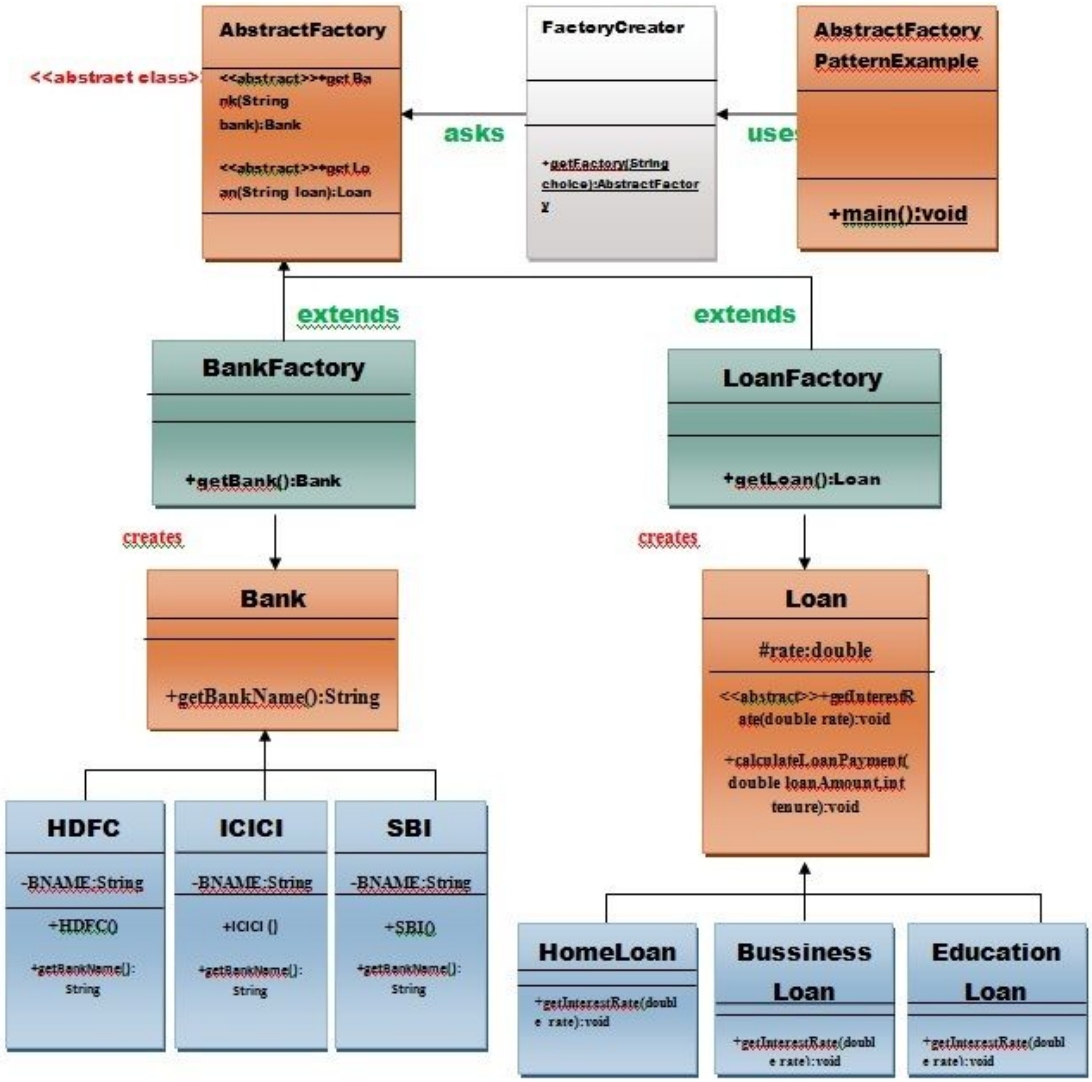
Verifique la salida:

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside Red::fill() method.
Inside Green::fill() method.
Inside Blue::fill() method.
```

Implementación de un ejemplo real

UML para patrón de fábrica abstracta

- Vamos a crear una interfaz de `Bank` (Banco) y una clase abstracta `Loan` (Préstamo), así como sus subclases.
- A continuación, crearemos la clase `AbstractFactory`.
- Luego, crearemos las clases concretas, `BankFactory` y `LoanFactory` que extenderán la clase `AbstractFactory`
- Después de eso, la clase `AbstractFactoryPatternExample` utiliza el `FactoryCreator` para obtener un objeto de la clase `AbstractFactory`.



Mediante este ejemplo estamos calculando el pago del préstamo para diferentes bancos como HDFC, ICICI, SBI, etc.

Step 1

Crearemos una interfaz Banco

```
import java.io.*;
interface Bank{
        String getBankName();
}
```

Paso 2

Crearemos las clases concretas que implementan la interfaz Bank.

```
public class HDFC implements Bank{
         private final String BNAME;
         public HDFC(){
                BNAME="HDFC BANK";
        }
```

```
        public String getBankName() {
                return BNAME;
        }
}

public class ICICI implements Bank{
       private final String BNAME;
       ICICI(){
                BNAME="ICICI BANK";
        }
        public String getBankName() {
                return BNAME;
       }
}

class SBI implements Bank{
      private final String BNAME;
      public SBI(){
                BNAME="SBI BANK";
        }
       public String getBankName(){
                return BNAME;
       }
}
```

Paso 3

Creamos la clase abstracta Loan.

```
public abstract class Loan{
   protected double rate;
   abstract void getInterestRate(double rate);
   public void calculateLoanPayment(double loanamount, int years)
   {
        /*
              to calculate the monthly loan payment i.e. EMI

              rate=annual interest rate/12*100;
              n=number of monthly installments;
              1year=12 months.
              so, n=years*12;

           */

         double EMI;
         int n;

         n=years*12;
         rate=rate/1200;
         EMI=((rate*Math.pow((1+rate),n))/((Math.pow((1+rate),n))-1))*loanamount;
```

```
System.out.println("your monthly EMI is "+ EMI +" for the amount"+loanamount+" you
have borrowed");
 }
}// end of the Loan abstract class.
```

Paso 4

Creamos las clases concretas que extienden Loan.

```
class HomeLoan extends Loan{
     public void getInterestRate(double r){
         rate=r;
    }
}//End of the HomeLoan class


class BussinessLoan extends Loan{
    public void getInterestRate(double r){
          rate=r;
     }

}//End of the BusssinessLoan class.

class EducationLoan extends Loan{
     public void getInterestRate(double r){
       rate=r;
 }
}//End of the EducationLoan class.
```

Paso 5

Creamos la clase abstracta (i.e AbstractFactory) para obtener las fábricas para objetos Bank y Loan.

```
1. abstract class AbstractFactory{
2.    public abstract Bank getBank(String bank);
3.    public abstract Loan getLoan(String loan);
4. }
```

Paso 6

Creamos las clases factoría que heredan AbstractFactory para generar los objetos de las clases concretas dado cierto parámetro.

```
public class BankFactory extends AbstractFactory{
   public Bank getBank(String bank){
      if(bank == null){
         return null;
      }
      if(bank.equalsIgnoreCase("HDFC")){
         return new HDFC();
      } else if(bank.equalsIgnoreCase("ICICI")){
```

```
         return new ICICI();
      } else if(bank.equalsIgnoreCase("SBI")){
         return new SBI();
      }
      return null;
   }
   public Loan getLoan(String loan) {
      return null;
   }
}//End of the BankFactory class.


public class LoanFactory extends AbstractFactory{
           public Bank getBank(String bank){
                return null;
           }

     public Loan getLoan(String loan){
      if(loan == null){
         return null;
      }
      if(loan.equalsIgnoreCase("Home")){
         return new HomeLoan();
      } else if(loan.equalsIgnoreCase("Business")){
         return new BussinessLoan();
      } else if(loan.equalsIgnoreCase("Education")){
         return new EducationLoan();
      }
      return null;
   }

}
```

Paso 7

Creamos una clase FactoryCreator para obtener las factorías pasando información tal como Bank o Loan.

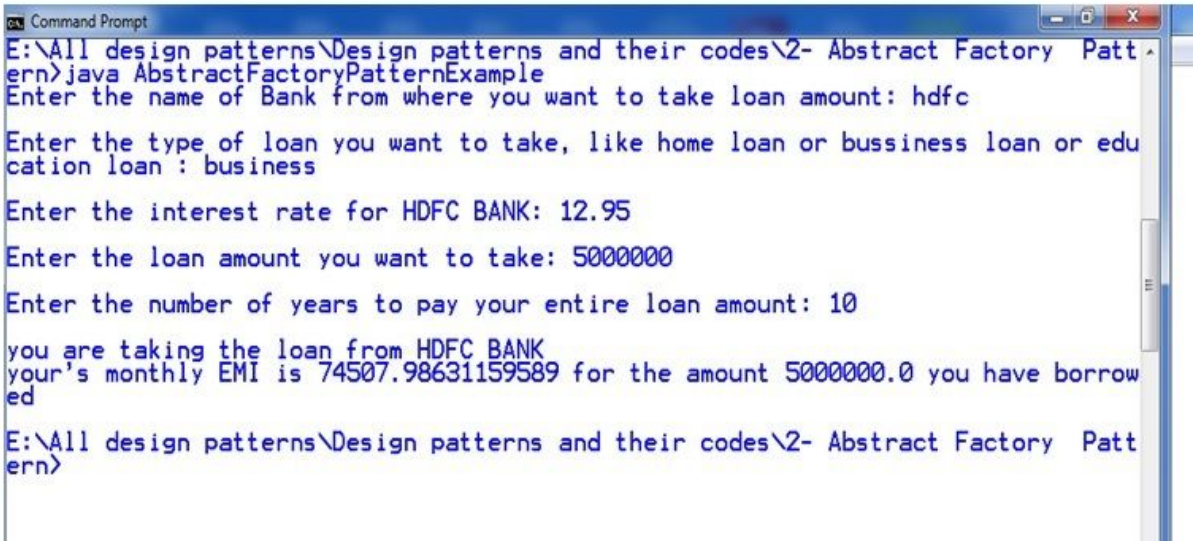
```
class FactoryCreator {
     public static AbstractFactory getFactory(String choice){
      if(choice.equalsIgnoreCase("Bank")){
         return new BankFactory();
      } else if(choice.equalsIgnoreCase("Loan")){
         return new LoanFactory();
      }
      return null;
   }
}//End of the FactoryCreator.
```

Paso 8

Usamos el FactoryCreator para obtener AbstractFactory para tomar factorías de clases conretas pasando información de tipo.

```
import java.io.*;
class AbstractFactoryPatternExample {

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

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

       System.out.print(
              "Enter the name of Bank from where you want to take loan amount: ");
       String bankName=br.readLine();

       System.out.print("\n");
       System.out.print(
              "Enter the type of loan e.g. home loan or business loan or education
       loan : ");

       String loanName=br.readLine();
       AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");
       Bank b=bankFactory.getBank(bankName);

       System.out.print("\n");
       System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");

       double rate=Double.parseDouble(br.readLine());
       System.out.print("\n");
       System.out.print("Enter the loan amount you want to take: ");

       double loanAmount=Double.parseDouble(br.readLine());
       System.out.print("\n");
       System.out.print("Enter the number of years to pay your entire loan amount:
");
       int years=Integer.parseInt(br.readLine());

       System.out.print("\n");
       System.out.println("you are taking the loan from "+ b.getBankName());

       AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");
            Loan l=loanFactory.getLoan(loanName);
            l.getInterestRate(rate);
            l.calculateLoanPayment(loanAmount,years);
     }
}//End of the  AbstractFactoryPatternExample
```

Salida



Antipatrones

Aunque esta guía y la anterior tratan sobre patrones factoría, usar patrones sólo par usarlos es peor que nunca usarlos. Este comportamiento es un anti-patrón. De hecho, la mayoría de los patrones hacen que el código sea más difícil de entender. La mayoría de las veces, no se usan factorías. Por ejemplo:

- Cuando se codifica sólo en casa / trabajo, se evita usarlos.
- Para proyectos pequeños que no cambian mucho, también se evitan las factorías.
- Para proyectos medianos o grandes involucrando a varios desarrolladores que utilizan el mismo código, allí son útiles.

Podría decirse que las fáctorías son como una negociación entre sus ventajas y la legibilidad y comprensión del código.

El objetivo principal de las fábricas es instanciar objetos. Pero, ¿por qué no crear directamente objetos con llamadas de constructor?

Para casos de uso simples, no hay necesidad de usar una fábrica. Echemos un vistazo a este código.

```
public class SimpleClass {
   private final Integer arg1;
   private final Integer arg2;

   SimpleClass(Integer arg1, Integer arg2) {
      this.arg1 = arg1;
```

```
      this.arg2 = arg2;
   }

   public Integer getArg1(){
      return arg1;
   }

   public Integer getArg2(){
      return args;
   }
}
...
public class BusinessClassXYZ {
   public static void someFunction(){
      SimpleClass mySimpleClass = new SimpleClass(1,2);
       // some stuff
   }
}
```

En este código, SimpleClass es una clase muy simple con un estado, sin dependencias, sin polimorfismo y sin lógica de negocio. Podría utilizar una fábrica para crear este objeto, pero duplicaría la cantidad de código. Por lo tanto, haría el código más difícil de entender. Si puede evitar el uso de fábricas hágalo, usted terminará con un código más simple!

Sin embargo, a menudo encontrará casos más complejos al escribir aplicaciones de gran tamaño que requieren muchos desarrolladores y muchos cambios de código. Para estos casos complejos, las ventajas de las fábricas superan sus inconvenientes.

La necesidad de las factorías

Ahora que ya esta advertido sobre el uso de las factorías, veamos por qué son tan poderosas y, por lo tanto, utilizadas en la mayoría de los proyectos.

Control sobre la instancia

Un caso de uso común con aplicaciones empresariales es limitar el número de instancias de una clase. ¿Cómo lograría tener sólo una (o 2, o 10) instancias de una clase porque consume un recurso como un socket, una conexión de base de datos o un descriptor de sistema de archivos o lo que sea?

Con el enfoque constructor, sería difícil para diferentes funciones (de diferentes clases) saber si ya existe una instancia de una clase. Y, incluso si hubo una instancia, ¿cómo podría una función obtener esta instancia? Podrías hacerlo usando variables compartidas que cada función verificaría pero:

- Enlazaría el comportamiento de todas las funciones que necesitan instanciar la misma clase ya que están utilizando y modificando las mismas variables compartidas,
- Varias partes del código tendrían la misma lógica para comprobar si la clase ya ha sido instanciada lo que conduciría a la duplicación de código.

Usando un singleton, usted podría hacer fácilmente eso.

Pérdida de acoplamiento

Otra ventaja de las fábricas es la pérdida de acoplamiento.

Supongamos que escribes un programa que calcula cosas y necesita escribir registros. Puesto que es un proyecto grande, uno de los códigos los escribe tu compañero, aquella clase que escribe los registros en un sistema de archivos (la clase FileSystemLogger) mientras que tú estás codificando las clases de negocio. Sin fábricas, es necesario instanciar FileSystemLogger con un constructor antes de usarlo:

```
public class FileSystemLogger {
   ...
   public void writeLog(String s) {
   //Implemation
   }
}
...
public void someFunctionInClassXXX(some parameters){
   FileSystemLogger logger= new FileSystemLogger(some paramters);
   logger.writeLog("This is a log");
}
```

¿Pero qué sucede si hay un cambio repentino y ahora usted necesita escribir registros en una base de datos con la implementación `DatabaseLogger`? Sin fábricas, tendrás que modificar todas las funciones usando la clase `FileSystemLogger`. Dado que este logger se utiliza en todas partes, necesitará modificar cientos de funciones / clases mientras que con una fábrica se puede cambiar fácilmente de una implementación a otra modificando sólo la fábrica:

```
//this is an abstraction of a Logger
public interface ILogger {
   public void writeLog(String s);
}

public class FileSystemLogger implements ILogger {
   ...
   public void writeLog(String s) {
      //Implemation
   }
}
```

```
public class DatabaseLogger implements ILogger {
   ...
   public void writeLog(String s) {
      //Implemation
   }
}

public class FactoryLogger {
   public static ILogger createLogger() {
      //you can choose the logger you want
      // as long as it's an ILogger
      return new FileSystemLogger();
   }
}
//////////////////////some code using the factory
public class SomeClass {
   public void someFunction() {
      //if the logger implementation changes
      //you have nothing to change in this code
      ILogger logger = FactoryLogger.createLogger();
      logger.writeLog("This is a log");
   }
}
```

Si miras este código, puedes cambiar fácilmente la implementación del logger de `FileSystemLogger` a `DatabaseLogger`. Sólo tienes que modificar la función `createLogger` () (que es una fáctoría). Este cambio es invisible para el código de cliente (negocio) ya que el código de cliente utiliza una interfaz de logger (`ILogger`) y la elección de la implementación del logger la realiza por la factoría. Al hacerlo, está creando un desacoplamiento entre la implementación del logger y las partes de los códigos que utilizan el logger.

Encapsulación

A veces, usar una factoría mejora la legibilidad de su código y reduce su complejidad por encapsulación.

Supongamos que necesita usar una clase de negocio CarComparator que compara 2 autos. Esta clase necesita una DatabaseConnection para obtener las características de millones de coches y una FileSystemConnection para obtener un archivo de configuración que parametriza el algoritmo de comparación (por ejemplo: agregar más peso al consumo de combustible que la velocidad máxima).

Sin una fábrica podría codificar algo como:

```
public class DatabaseConnection {
   DatabaseConnection(some parameters) {
      // some stuff
```

```
   }
   ...
}

public class FileSystemConnection {
   FileSystemConnection(some parameters) {
      // some stuff
   }
   ...
}

public class CarComparator {
   CarComparator(DatabaseConnection dbConn, FileSystemConnection fsConn) {
      // some stuff
   }

   public int compare(String car1, String car2) {
      // some stuff with objets dbConn and fsConn
   }
}
...
public class CarBusinessXY {
   public void someFunctionInTheCodeThatNeedsToCompareCars() {
      DatabaseConnection db = new DatabaseConnection(some parameters);
      FileSystemConnection fs = new FileSystemConnection(some parameters);
      CarComparator carComparator = new CarComparator(db, fs);
      carComparator.compare("Ford Mustang","Ferrari F40");
   }
...
}

public class CarBusinessZY {
   public void someOtherFunctionInTheCodeThatNeedsToCompareCars() {
      DatabaseConnection db = new DatabaseConnection(some parameters);
      FileSystemConnection fs = new FileSystemConnection(some parameters);
      CarComparator carComparator = new CarComparator(db, fs);
      carComparator.compare("chevrolet camaro 2015","lamborghini diablo");
   }
...
}
```

Este código funciona, pero se puede ver que para utilizar el método de comparación, es necesario instanciar:

- Un DatabaseConnection,
- Un FileSystemConnection,
- Luego un CarComparator.

Si necesita usar la comparación en múltiples funciones, tendrá que duplicar su código, lo que significa que si la construcción de la clase CarComparator cambia, tendrá que modificar

todas las partes duplicadas. El uso de una fábrica podría factorizar el código y ocultar la complejidad de la construcción de la clase CarComparator.

```
...
public class Factory {
   public static CarComparator getCarComparator() {
      DatabaseConnection db = new DatabaseConnection(some parameters);
      FileSystemConnection fs = new FileSystemConnection(some parameters);
      CarComparator carComparator = new CarComparator(db, fs);
   }
}
//////////////////////////////some code using the factory
public class CarBusinessXY {
   public void someFunctionInTheCodeThatNeedsToCompareCars() {
      CarComparator carComparator = Factory.getCarComparator();
      carComparator.compare("Ford Mustang","Ferrari F40");
   }
...
}
...
public class CarBusinessZY {
   public void someOtherFunctionInTheCodeThatNeedsToCompareCars() {
      CarComparator carComparator = Factory.getCarComparator();
      carComparator.compare("chevrolet camaro 2015","lamborghini diablo");
   }
...
}
```

Si comparas ambos códigos, puedes ver que usando una fábrica:

- Se reduce el número de línea de código.
- Se evita la duplicación de código.
- Se organiza el código: la fábrica tiene la responsabilidad de construir un CarComparator y la clase de negocios sólo lo utiliza.

El último punto es importante. Porque es una cuestión de separación de preocupaciones. Una clase de negocio no debe tener que saber cómo construir un objeto complejo que necesita para usar: la clase de negocio necesita centrarse sólo en las preocupaciones de negocio. Además, también aumenta la división del trabajo entre los desarrolladores del mismo proyecto:

- Uno trabaja en el CarComparator y la forma en que se crea.
- Otros trabajan en objetos de negocio que utilizan el CarComparator.

Desambiguación

Supongamos que tiene una clase con varios constructores (con comportamientos muy diferentes). ¿Cómo puede estar seguro de que no usará el constructor equivocado por error?
Echemos un vistazo al siguiente código:

```
class Example{
     //constructor one
     public Example(double a, float b) {
         //...
     }

     //constructor two
     public Example(double a) {
         //...
     }

     //constructor three
     public Example(float a, double b) {
         //...
     }
}
```

Aunque el constructor uno y dos no tienen el mismo número de argumentos, puede fallar rápidamente para elegir el correcto, especialmente al final de un día ocupado usando el autocompletado de su IDE favorito. Es aún más difícil ver la diferencia entre el constructor uno y el constructor tres. Este ejemplo parece falso, pero lo vi en código legado.
La pregunta es, ¿cómo podrías implementar diferentes constructores con el mismo tipo de parámetros (evitando un camino sucio como los constructores uno y tres)?

Aquí está una solución limpia usando una fábrica:

```
class Complex {
     public static Complex fromCartesian(double real, double imag) {
         return new Complex(real, imag);
     }

     public static Complex fromPolar(double rho, double theta) {
         return new Complex(rho * Math.cos(theta), rho * Math.sin(theta));
     }

     private Complex(double a, double b) {
         //...
     }
 }
```

En este ejemplo, el uso de una fábrica añade una descripción de lo que es la creación con el nombre del método factoría: puede crear un número Complejo a partir de coordenadas cartesianas o de coordenadas polares. En ambos casos, usted sabe exactamente de qué se trata la creación

Bibliografía

- Patrón factoría abstracta https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm
- Patrón Factoría http://www.javatpoint.com/factory-method-design-pattern
- Patrón de diseño: patrones de fábrica: http://coding-geek.com/design-pattern-factory-patterns/