

Paradigmas de Programación

# Patrones de Diseño

Patrones de diseño

**GRUPO RHO**

JAVIER KOROLL

HERNAN ZARZYCKI

## Tabla de contenido

INTRODUCCIÓN .....	2
PATRON ELEGIDO: FACTORY METHOD .....	2
IMPLEMENTACIÓN EN JAVA .....	3
IMPLEMENTACIÓN EN PYTHON .....	6
COMPARATIVA .....	8
CONCLUSIÓN.....	10
REFERENCIAS.....	10

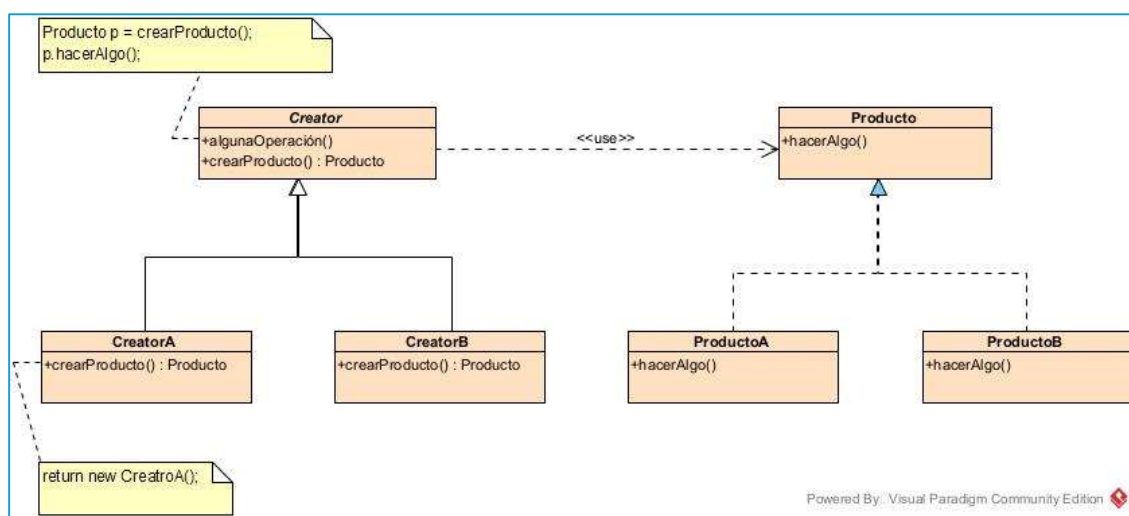
## INTRODUCCIÓN

El presente documento busca explorar y comparar cómo se trabajan los conceptos vistos en Java con el Paradigma Orientado a Objetos, en el lenguaje de programación Python tomando como objeto de estudio un patrón de diseño.

## PATRON ELEGIDO: FACTORY METHOD

Hemos seleccionado para trabajar el patrón creacional Factory Method. Este patrón nos proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

El diagrama del patrón es el siguiente:



La elección del patrón se debe principalmente a que este posee una serie de elementos interesantes en su implementación:

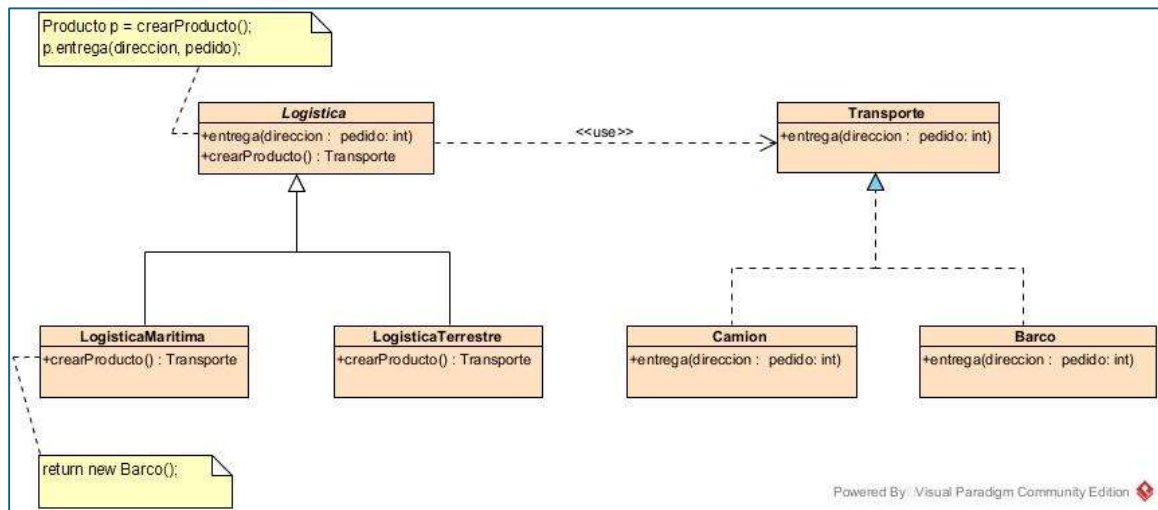
- Interfaz
- Clase abstracta
- Métodos abstractos
- Herencia

Estos elementos son recurrentes en la implementación de los demás patrones, por lo que es un buen punto de partida para explorar y comparar cómo trabajan ambos lenguajes con estos elementos. Y así, nos estaremos allanando el camino para muchos otros patrones.

## IMPLEMENTACIÓN EN JAVA

Aplicaremos el patrón a la siguiente problemática: se tiene una empresa de logística que gestiona transporte en camiones o en barcos.

El diagrama a implementar sería el siguiente:



Defino la interfaz común que implementarán todos mis productos:

```

gestiona package creacional.factorymethod;

public interface Transporte {

    void entrega(String direccion, int pedido);

}

```

Defino las clases que implementarán dicha interfaz:

```

package creacional.factorymethod;

public class Camion implements Transporte {

    @Override
    public void entrega(String direccion, int pedido) {

        System.out.println("Se ha despachado el pedido " + pedido + " en
el camion hacia " + direccion);

    }

}

```

```
package creacional.factorymethod;

public class Barco implements Transporte {

    @Override
    public void entrega(String direccion, int pedido) {

        System.out.println("Se ha despachado el pedido " + pedido + " en
el barco hacia " + direccion);
    }
}
```

Defino una clase abstracta Logística, que declara el método de fábrica que devuelve nuevos objetos de Transporte.

```
package creacional.factorymethod;

public abstract class Logistica {

    public abstract Transporte crearLogistica();

    public void entrega(String direccion, int pedido) {

        Transporte transporte = this.crearLogistica();
        transporte.entrega(direccion, pedido);
    }
}
```

Extiendo la clase abstracta e implemento el método abstracto de creación crearLogistica().

```
package creacional.factorymethod;

public class LogisticaMaritima extends Logistica {

    @Override
    public Transporte crearLogistica() {

        return new Barco();
    }
}
```

```
package creacional.factorymethod;

public class LogisticaTerrestre extends Logistica {

    @Override
    public Transporte crearLogistica() {

        return new Camion();
    }
}
```

Finalmente, hago uso de la clase de creación en la clase principal:

```
package creacional.factorymethod;

public class Main {

    private static TipoLogistica TIPO_DE_LOGISTICA =
    TipoLogistica.LOGISTICA_MARITIMA;

    public static void main(String[] args) {

        Logistica;
        if (TIPO_DE_LOGISTICA == TipoLogistica.LOGISTICA_MARITIMA) {
            logistica = new LogisticaMaritima();
        } else {
            logistica = new LogisticaTerrestre();
        }
        logistica.entrega("Avenida Siempreviva 742", 4678);
    }

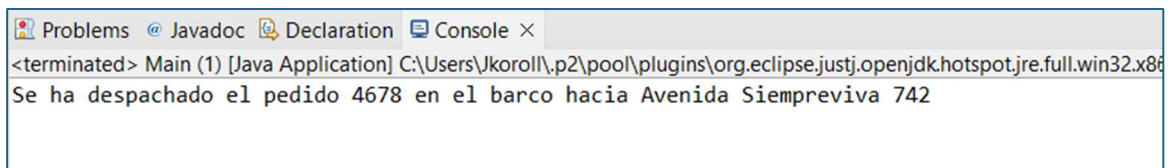
}
```

```
package creacional.factorymethod;

public enum TipoLogistica {

    LOGISTICA_TERRESTRE,
    LOGISTICA_MARITIMA
}
```

La salida del programa es:



The screenshot shows the Eclipse IDE's console window. The title bar includes 'Problems', 'Javadoc', 'Declaration', and 'Console'. The console output shows the program has terminated and the message: 'Se ha despachado el pedido 4678 en el barco hacia Avenida Siempreviva 742'.

```
<terminated> Main (1) [Java Application] C:\Users\Jkoroll\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86
Se ha despachado el pedido 4678 en el barco hacia Avenida Siempreviva 742
```

## IMPLEMENTACIÓN EN PYTHON

Veamos ahora el mismo ejemplo, pero en Python.

Defino la interfaz común que implementarán todos mis productos.

Para eso, haremos uso de las **interfaces formales**, las cuales definen una forma de crear interfaces (a través de metaclasses).

```
from abc import abstractmethod
from abc import ABCMeta

class Transporte(metaclass=ABCMeta):

    @abstractmethod
    def entrega(self, direccion, pedido) -> str:
        pass
```

Podemos observar que se usa el decorador @abstractmethod. Un método definido con este decorador forzará a las clases que implementen dicha interfaz a codificarlo.

Defino las clases que implementarán la interfaz Transporte utilizando el mecanismo de herencia en Python, donde se recibe el nombre de la clase padre dentro de los paréntesis inmediatos al nombre de la clase. Y para implementar los métodos abstractos de la interfaz sólo basta con desarrollarlos con la sintaxis normal del lenguaje.

```
from Transporte import Transporte

class Barco(Transporte):

    def entrega(self, direccion, pedido) -> str:
        print(f"Se ha despachado el pedido {pedido} en el barco hacia {direccion}.");
```

```
from Transporte import Transporte

class Camion(Transporte):

    def entrega(self, direccion, pedido) -> str:
        print(f"Se ha despachado el pedido {pedido} en el camion hacia {direccion}.");
```

Ahora defino la clase abstracta Logistica:

```
from abc import abstractmethod
from abc import ABCMeta

from Transporte import Transporte

class Logistica(metaclass=ABCMeta):

    def entrega(self, direccion, pedido) -> str:
        transporte = self.crearLogistica()
        return transporte.entrega(direccion, pedido)

    @abstractmethod
    def crearLogistica(self) -> Transporte:
        pass
```

Al igual que sucedió con la interfaz se utilizan metaclasses y el decorador @abstractmethod para definir interfaces.

Ahora creamos las clases derivadas que implementan el método abstracto de creación:

```
from Barco import Barco
from Logistica import Logistica
from Transporte import Transporte

class LogisticaMaritima(Logistica):

    def crearLogistica(self) -> Transporte:
        return Barco()
```

```
from Camion import Camion
from Logistica import Logistica
from Transporte import Transporte

class LogisticaTerrestre(Logistica):

    def crearLogistica(self) -> Transporte:
        return Camion()
```



Un enumerado que usaremos en la clase principal:

```
from enum import Enum

class Logistica(Enum):
    LOGISTICA_TERRESTRE = 1
    LOGISTICA_MARITIMA = 2
```

Por último, creamos la clase principal:

```
from LogisticaMaritima import LogisticaMaritima
from LogisticaTerrestre import LogisticaTerrestre
from TipoLogistica import Logistica

def main():
    tipoLogistica = Logistica.LOGISTICA_MARITIMA

    if(tipoLogistica == Logistica.LOGISTICA_MARITIMA):
        logistica = LogisticaMaritima()
        logistica.entrega("Avenida Siempreviva 742", 4678)
    else:
        logistica = LogisticaTerrestre()
        logistica.entrega("Avenida Siempreviva 742", 4678)

if __name__ == '__main__':
    main()
```

## COMPARATIVA

Ambas implementaciones del patrón terminan utilizando los siguientes elementos:

- Interfaz
- Clase abstracta
- Herencia

Comparando la manera de implementar una **interfaz**:

JAVA	PYTHON
<pre>package creacional.factorymethod;  public interface Transporte {      void entrega(String direccion, int pedido); }</pre>	<pre>from abc import abstractmethod from abc import ABCMeta  class Transporte(metaclass=ABCMeta):      @abstractmethod     def entrega(self, direccion, pedido) -&gt; str:         pass</pre>

Vemos que Python no hace uso de una palabra reservada como en Java. Además, para que tenga el mismo comportamiento que en Java es necesario recurrir a metaclasses y anotaciones.

Comparando las clases **abstractas**:

JAVA	PYTHON
<pre>package creacional.factorymethod;  public abstract class Logistica {      public abstract Transporte crearLogistica();      public void entrega(String direccion, int pedido) {          Transporte transporte = this.crearLogistica();         transporte.entrega(direccion, pedido);      }  }</pre>	<pre>from abc import abstractmethod from abc import ABCMeta  from Transporte import Transporte  class Logistica(metaclass=ABCMeta):      def entrega(self, direccion, pedido) -&gt; str:         transporte = self.crearLogistica()         return transporte.entrega(direccion, pedido)      @abstractmethod     def crearLogistica(self) -&gt; Transporte:         pass</pre>

Al igual que con las interfaces, Python no usa palabras reservadas para definir una clase abstracta. En su lugar utiliza metaclasses y anotaciones. De hecho, la manera de crear una interfaz o una clase abstracta es la misma para ambas. Solamente que la interfaz únicamente tendrá métodos con la anotación `@abstractmethod`.

Comparando la **herencia**:

JAVA	PYTHON
<pre>package creacional.factorymethod;  public class LogisticaMaritima extends Logistica {      @Override     public Transporte crearLogistica() {          return new Barco();      }  }</pre>	<pre>from Barco import Barco from Logistica import Logistica from Transporte import Transporte  class LogisticaMaritima(Logistica):      def crearLogistica(self) -&gt; Transporte:         return Barco()</pre>

Como se puede ver, Java utiliza la palabra reservada **extends** para extender la clase base Logistica. Y sobrescribe los métodos de la clase padre usando la anotación `@Override`. En cambio, en Python se recibe como parámetro en la definición de la clase derivada el nombre de la clase padre. Y se implementan los métodos sin requerir anotaciones para la sobreescritura.

## CONCLUSIÓN

Python es un lenguaje orientado a objetos que, si bien no utiliza palabras reservadas como **interface** y **abstract**, para diferenciar claramente si se trata de una interfaz o clase abstracta, o **extends** e **implements**. para diferenciar si se está extendiendo de una clase o implementando una interfaz, sí utiliza estos conceptos y por lo tanto se pueden implementar sin problemas patrones como **Factory Method** al igual que muchos patrones más. que hacen uso de los mismos elementos de la **POO**.

## REFERENCIAS

<https://refactoring.guru/es/design-patterns>

<https://ellibrodepython.com/>

<https://www.gyata.ai/es/python/python-main-function>