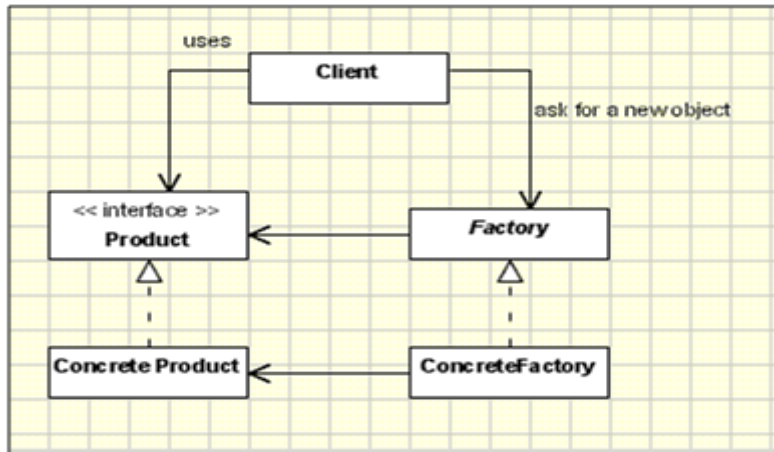


Factory Method

Diagrama de clases e interfaces



Intención

Un objeto ha de crear objetos de otra clase.

- El objeto que ha de crear los otros objetos no puede anticipar su clase.
- Queremos delegar la responsabilidad de especificar la clase de los objetos que se han de crear en nuestras subclases.

Motivación

También conocido como Constructor Virtual, el patrón Factory Method representa una alternativa para las implementaciones del patrón Factory que usan un HashMap o ficheros de propiedades para el registro de las subclases.

Supongamos que tenemos que añadir un nuevo subtipo de **Product** a la aplicación y la implementación de nuestra factoría es del tipo switch-case, esto es, con los nombres de las subclases de **Product** especificados literalmente (hardcoded) en cada sentencia 'case'. En este caso tendremos que modificar la factoría para contemplar este nuevo tipo de producto. En cambio, con una implementación del patrón Factory basada en HashMap o en ficheros de propiedades todo lo que necesitaríamos sería registrar

(dinámicamente) el nuevo subtipo en la factoría, pero sin que fuese necesario alterarla en absoluto, lo que supone una gran ventaja.

La implementación procedural -la basada en estamentos switch/case- es el clásico mal ejemplo que incumple el principio de diseño abierto-cerrado. Si por alguna razón no podemos o no queremos utilizar el patrón Factory basado en HashMap o Properties, ¿existiría igualmente alguna manera de utilizar la aproximación switch/case? Sí, como podemos intuir, la solución para evitar la modificación de la factoría es simple: extenderla, de manera que cada subclase sea una encapsulación de cada 'case' de la estructura condicional. Esto es lo que hace el (clásico) patrón Factory Method.

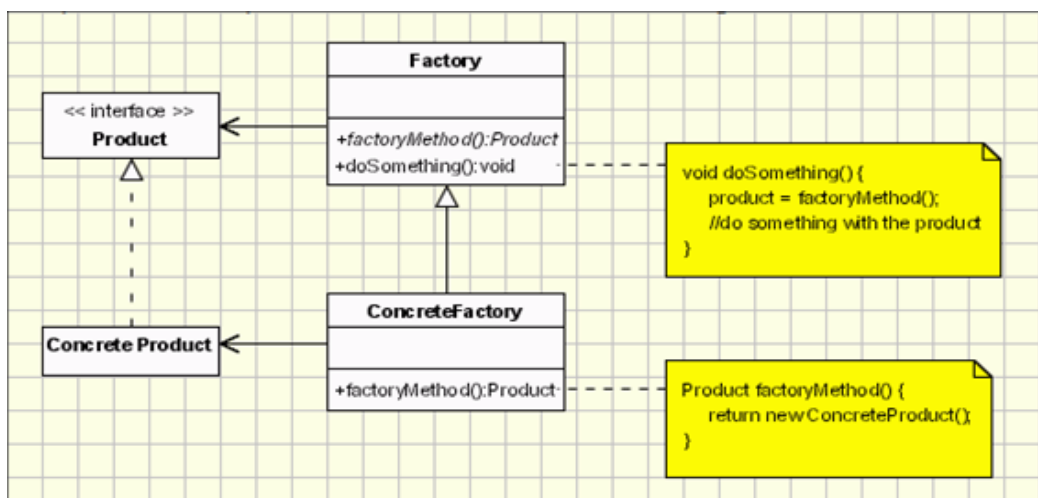
Contexto: Creación de un objeto del que se desconoce la subclase a la que pertenece.

Problema: Instanciar un objeto y tener que determinar su subclase en tiempo de ejecución. Esta situación se presenta muy a menudo en la creación de frameworks en los que hay clases abstractas que reciben peticiones para crear objetos dentro del propio framework.

Solución: Factory (Creator) define una clase abstracta para crear un objeto, pero deja que sean sus propias subclases quienes decidan, en tiempo de ejecución, el tipo concreto de objeto a crear. Esto se consigue definiendo en la subclase un método abstracto -el método de factoría- que las subclases tendrán que implementar.

Implementación

El funcionamiento del patrón es el mostrado en el diagrama de clases siguiente:



Las clases participantes en el patrón son las siguientes:

- **Product:** Define la interfaz que todo objeto susceptible de ser creado por la factoría debe implementar. La factoría sólo se acopla con esta interfaz, nunca con las subclases que la implementan.
- **ConcreteProduct:** Cada una de las subclases que implementan la interfaz Product y cuyos objetos interesa crear.
- **Factory/Creator:** Clase que contiene el método de factoría y que devuelve un objeto concreto de Product, aunque a través de una referencia Product, nunca mediante una referencia a alguna subclase de Product.
- **ConcreteFactory/ConcreteCreator:** Subclase de Factory/Creator que reemplaza el método de factoría para crear una instancia concreta de alguna subclase de Product.

Product es la clase abstracta de los objetos que se crean y Creator es la clase abstracta de los objetos creadores de productos. Un creador de una subclase concreta (ProductCreator) instancia un producto de una subclase concreta ConcreteProduct mediante la operación factoryMethod(). La operación doSomethig() obtiene una instancia nueva del producto llamando a factoryMethod().

Dado que todos los productos concretos son subclases de Product, todos tienen la misma implementación básica, en cierta medida. La clase Factory/Creador especifica el comportamiento estándar y genérico de los productos y cuando se necesita un nuevo producto envía los detalles de la creación (que previamente han sido suministrados por una clase cliente) a (una de las clases) ConcreteFactory/ConcreteCreator.

A continuación se muestra el código de la implementación básica del patrón:

Participante: Product

```
package creacionales.factorymethod.simple.product;

// Definimos la interfaz del producto
public interface Product {
    // Tendrá un operacion
    public void operacion();
}
```

Participante: ConcreteProduct

```
package creacionales.factorymethod.simple.product;

// Definimos un producto concreto
public class ConcreteProductA implements Product {
    public void operacion() {
        System.out.println("operacion del producto A");
    }
}
```

Participante: ConcreteProduct (la gracia del patrón es que haya varios)

```
package creacionales.factorymethod.simple.product;

//Definimos otro producto concreto
public class ConcreteProductB implements Product {
    public void operacion() {
        System.out.println("operacion del producto B");
    }
}
```

Participante: Factory/Creator

```
package creacionales.factorymethod.simple;

import creacionales.factorymethod.simple.product.Product;

// Definimos la clase abstracta constructora
public abstract class Creator {
    // Operación común a todas las subclases de Creator
    // Podríamos guardar cada producto en una lista, por ejemplo.
    public Product aMethod() {
        Product product = factoryMethod();
        return product;
    }

    // Definimos el método abstracto - método factoria
    protected abstract Product factoryMethod();
}
```

Participante: ConcreteFactory/ConcreteCreator

```
package creacionales.factorymethod.simple;

import creacionales.factorymethod.simple.product.ConcreteProductA;
import creacionales.factorymethod.simple.product.Product;

// Definimos un creador concreto
public class ConcreteCreatorA extends Creator {
    @Override
    protected Product factoryMethod() {
        return new ConcreteProductA();
    }
}
```

Participante: ConcreteFactory/ConcreteCreator (la gracia del patrón es que haya varios)

```
package creacionales.factorymethod.simple;

import creacionales.factorymethod.simple.product.ConcreteProductB;
import creacionales.factorymethod.simple.product.Product;

// Definimos un creador concreto
public class ConcreteCreatorB extends Creator {
    @Override
    protected Product factoryMethod() {
        return new ConcreteProductB();
    }
}
```

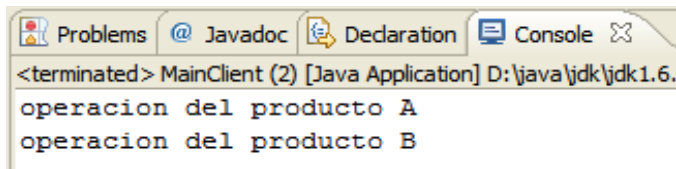
Participante: Clase cliente

```
package creacionales.factorymethod.simple.main;

import creacionales.factorymethod.simple.ConcreteCreatorA;
import creacionales.factorymethod.simple.ConcreteCreatorB;
import creacionales.factorymethod.simple.Creator;
import creacionales.factorymethod.simple.product.Product;

public class MainClient {
    public static void main( String arg[] ) {
        Creator creator = new ConcreteCreatorA();
        Product product = creator.aMethod();
        product.operacion();
        creator = new ConcreteCreatorB();
        product = creator.aMethod();
        product.operacion();
    }
}
```

Salida:



The screenshot shows the 'Console' tab of a Java IDE. The title bar indicates the application is 'MainClient (2) [Java Application]' running on 'D:\java\jdk\jdk1.6.'. The output text is as follows:

```
<terminated> MainClient (2) [Java Application] D:\java\jdk\jdk1.6.
operacion del producto A
operacion del producto B
```

Aplicabilidad y Ejemplos

Ejemplo 1 - Creación de triángulos

Imaginemos un programa en el que el usuario puede crear diferentes tipos de triángulos proporcionando valores para sus tres lados. El usuario solicitará la creación de un triángulo y a partir de los valores suministrados, la aplicación determinará de qué triángulo se trata.



Los triángulos se pueden clasificar por las medidas de sus lados o por sus ángulos. Veamos cómo.

Por sus lados:

- Equilátero: tres lados iguales
- Isósceles: dos lados iguales y uno diferente
- Escaleno: los tres lados diferentes

Por sus ángulos:

- Triángulo rectángulo: tiene un ángulo de 90°
- Triángulo acutángulo: los tres ángulos son menores de 90°
- Triángulo obtusángulo: tiene un ángulo mayor de 90°

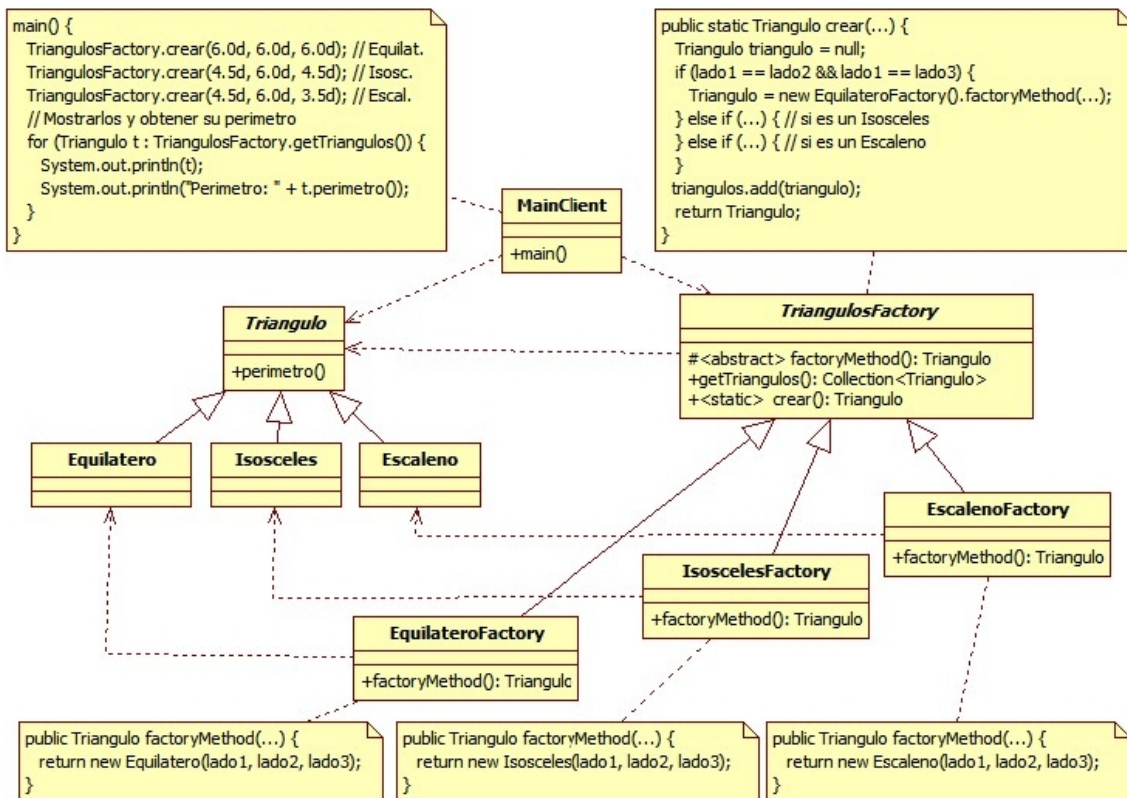
A nosotros no interesa en esta aplicación clasificarlos por sus lados.

El patrón Factory Method nos puede venir muy bien en este caso (Ver un poco más abajo el diagrama de clases). Una clase abstracta con funcionalidad común (Creator) y tres clases concretas (ConcreteCreator), donde cada una creará un tipo de triángulo específico (ConcreteProduct).

La clase abstracta implementará, entre otros, un método público (en el código que se adjunta un poco más adelante se llama crear()) que recibirá los tres lados del nuevo triángulo en sus parámetros y los evaluará para determinar a qué ConcreteCreator invocar para obtener el tipo de triángulo adecuado. A continuación, guardará el nuevo triángulo en una lista y lo devolverá al usuario. El usuario siempre podrá recuperar la lista con todos los triángulos creados.

Pero, ¿Cómo se produce la llamada desde Creator a los ConcreteCreator? Bien, la clase abstracta definirá un método abstracto y protegido, factoryMethod(), que servirá para que cada subclase instancie al ConcreteProduct. Este método protegido es invocado directamente por Creator en su método crear(), una vez que ya ha instanciado la ConcreteCreator adecuada (efectivamente, Creator define el método factoryMethod como abstracto y él mismo lo invoca sobre la ConcreteCreator correspondiente).

Las clases cliente no deben instanciar a ningún ConcreteProduct, pues queremos generalidad en nuestras clases clientes y deseamos que no haya acoplamiento innecesario). Después de una ejecución exitosa del patrón, la única referencia que consiguen las clases cliente es del tipo Product, aunque obviamente el objeto apuntado por la referencia será de alguno de los tres tipos de triángulo.



Comencemos por ver la jerarquía de Product:

Triangulo.java

Se trata de la clase base de la jerarquía, es abstracta e implementa los métodos comunes para las clases hijas: `perimetro()` y `toString()`.

```
package creacionales.factorymethod.triangulos.products;
```

```
public abstract class Triangulo {
    private double lado1, lado2, lado3;

    Triangulo(double lado1, double lado2, double lado3) {
        this.lado1 = lado1;
        this.lado2 = lado2;
        this.lado3 = lado3;
    }

    public double perimetro() {
        return (lado1+lado2+lado3);
    }

    public double getLado1() { return lado1; }
    public double getLado2() { return lado2; }
    public double getLado3() { return lado3; }

    public void setLado1(double lado1) {
        this.lado1 = lado1;
    }
}
  
```



```

    public void setLado2(double lado2) {
        this.lado2 = lado2;
    }

    public void setLado3(double lado3) {
        this.lado3 = lado3;
    }

    @Override
    public String toString() {
        return "Triangulo " + this.getClass().getSimpleName() +
            "[lado1=" + lado1 + ", lado2=" + lado2 +
            ", lado3=" + lado3 + "]";
    }
}

```

Veamos las subclases de Triangulo. Observad que la implementación de las tres es de lo más trivial.

Equilatero.java

```

package creacionales.factorymethod.triangulos.products;

public class Equilatero extends Triangulo {

    public Equilatero(double lado1, double lado2, double lado3) {
        super(lado1, lado2, lado3);
    }

}

```

Isosceles.java

```

package creacionales.factorymethod.triangulos.products;

public class Isosceles extends Triangulo {

    public Isosceles(double lado1, double lado2, double lado3) {
        super(lado1, lado2, lado3);
    }

}

```

Escaleno.java

```

package creacionales.factorymethod.triangulos.products;

public class Escaleno extends Triangulo {

    public Escaleno(double lado1, double lado2, double lado3) {
        super(lado1, lado2, lado3);
    }

}

```

Ahora veamos las clases de estructura del patrón:

TriangulosFactory.java

Es el Creator (Factory). En esta ocasión implementa toda la lógica necesaria para instanciar un ConcreteFactory y que se ejecute su correspondiente método factoryMethod(). Para más detalle leer los comentarios específicos de esta clase en el enunciado del ejemplo.

```
package creacionales.factorymethod.triangulos;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;

import creacionales.factorymethod.triangulos.products.Triangulo;

public abstract class TriangulosFactory {

    private static Collection<Triangulo> triangulos = new
    ArrayList<Triangulo>();

    protected abstract Triangulo factoryMethod(double lado1, double
    lado2, double lado3);

    public static Triangulo crear(double lado1, double lado2, double
    lado3) {

        Triangulo triangulo = null;

        if (lado1 == lado2 && lado1 == lado3) {
            triangulo = new
            EquilateroFactory().factoryMethod(lado1, lado2, lado3);
        } else if ((lado1 == lado2) && (lado2 != lado3) || (lado1
        == lado3)
            && (lado1 != lado2)) {
            triangulo = new
            IsoscelesFactory().factoryMethod(lado1, lado2, lado3);
        } else if (lado1 != lado2 && lado2 != lado3 && lado3 !=
        lado1) {
            triangulo = new
            EscalenoFactory().factoryMethod(lado1, lado2, lado3);
        }

        triangulos.add(triangulo);

        return triangulo;
    }

    public static Collection<Triangulo> getTriangulos() {
        if (triangulos.isEmpty()) {
            return Collections.emptyList();
        }
        return triangulos;
    }
}
```

```
}
```

Ahora las subclases. Notad que tienen visibilidad de paquete, por lo que sólo son accesibles desde TriangulosFactory. Esto nos permite no acoplar clases concretas del patrón con las clases cliente. También apreciad que la implementación de las tres es de lo más trivial.

EquilateroFactory.java

```
package creacionales.factorymethod.triangulos;

import creacionales.factorymethod.triangulos.products.Equilatero;
import creacionales.factorymethod.triangulos.products.Triangulo;

// Visibilidad de paquete
class EquilateroFactory extends TriangulosFactory {

    @Override
    public Triangulo factoryMethod(double lado1, double lado2,
double lado3) {
        return new Equilatero(lado1, lado2, lado3);
    }

}
```

IsoscelesFactory.java

```
package creacionales.factorymethod.triangulos;

import creacionales.factorymethod.triangulos.products.Isosceles;
import creacionales.factorymethod.triangulos.products.Triangulo;

// Visibilidad de paquete
public class IsoscelesFactory extends TriangulosFactory {

    @Override
    public Triangulo factoryMethod(double lado1, double lado2,
double lado3) {
        return new Isosceles(lado1, lado2, lado3);
    }

}
```

EscalenoFactory.java

```
package creacionales.factorymethod.triangulos;

import creacionales.factorymethod.triangulos.products.Escaleno;
import creacionales.factorymethod.triangulos.products.Triangulo;

//Visibilidad de paquete
public class EscalenoFactory extends TriangulosFactory {
```

```

        @Override
        public Triangulo factoryMethod(double lado1, double lado2,
double lado3) {
            return new Escaleno(lado1, lado2, lado3);
        }
    }
}

```

Finalmente, la clase cliente. Notad que la clase sólo se acopla con TriangulosFactory y con Triangulo, esto es, con clases abstractas.

MainClient.java

```

package creacionales.factorymethod.triangulos.main;

import creacionales.factorymethod.triangulos.TriangulosFactory;
import creacionales.factorymethod.triangulos.products.Triangulo;

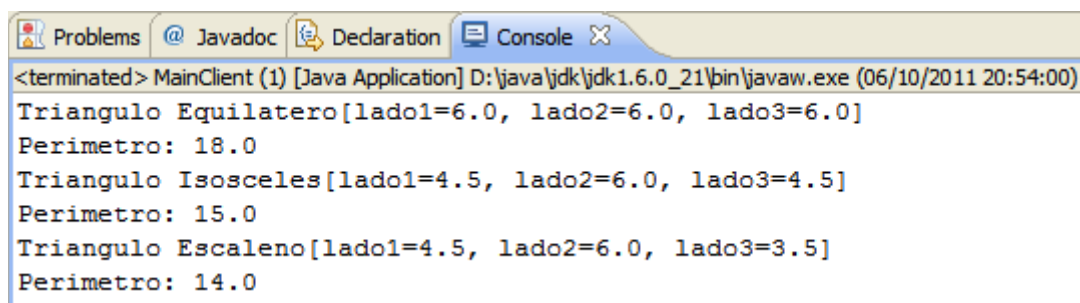
public class MainClient {
    public static void main(String arg[]) {

        TriangulosFactory.crear(6.0d, 6.0d, 6.0d);
        TriangulosFactory.crear(4.5d, 6.0d, 4.5d);
        TriangulosFactory.crear(4.5d, 6.0d, 3.5d);

        // La factoria ha almacenado los triangulos creados
        // en una coleccion que ahora podemos recuperar
        if (!TriangulosFactory.getTriangulos().isEmpty()) {
            for (Triangulo t : TriangulosFactory.getTriangulos()) {
                System.out.println(t);
                System.out.println("Perimetro: " +
t.perimetro());
            }
        } else {
            System.out.println("La lista esta vacia");
        }
    }
}

```

Salida:



```

<terminated> MainClient (1) [Java Application] D:\java\jdk\jdk1.6.0_21\bin\javaw.exe (06/10/2011 20:54:00)
Triangulo Equilatero[lado1=6.0, lado2=6.0, lado3=6.0]
Perimetro: 18.0
Triangulo Isosceles[lado1=4.5, lado2=6.0, lado3=4.5]
Perimetro: 15.0
Triangulo Escaleno[lado1=4.5, lado2=6.0, lado3=3.5]
Perimetro: 14.0

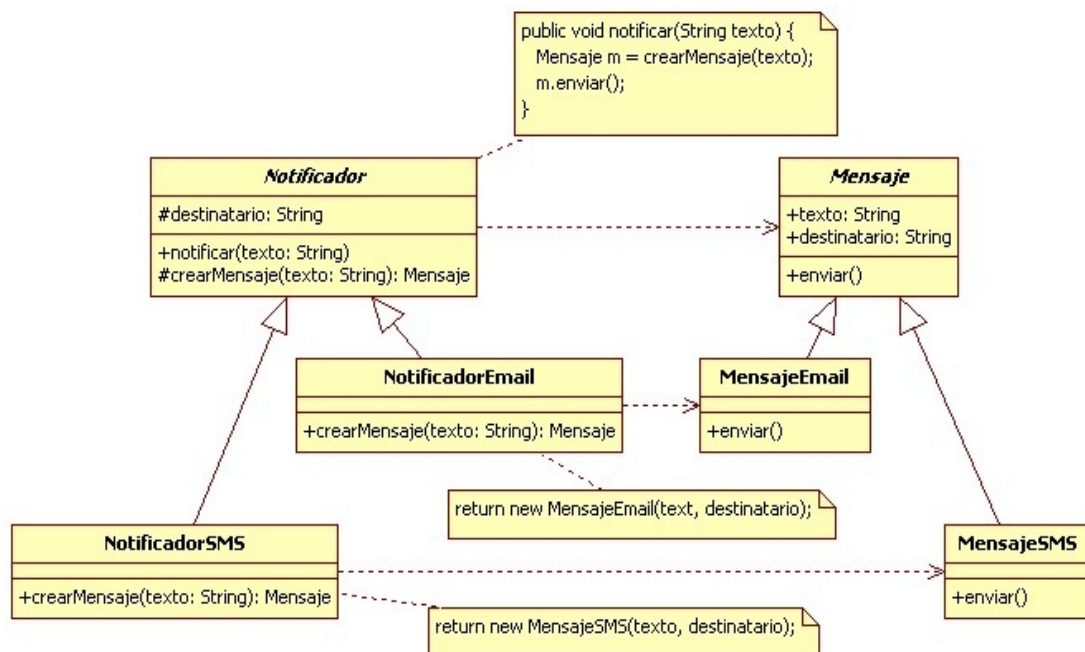
```

Ejemplo 2 – Notificador (no se incluye código fuente)

Supongamos que estamos desarrollando un sistema que ha de notificar ciertos eventos al usuario. La clase encargada de gestionar las notificaciones ha de crear un mensaje y enviarlo, pero queremos delegar en las subclases la selección del tipo de mensaje a enviar: un mensaje de correo electrónico o un mensaje SMS de móvil.

Aplicando el patrón Factory Method, podemos definir en la clase abstracta *Creator* (la que crea los objetos), un método llamado *crear()* responsable de la creación (es el método factoría). Por otro lado, proporcionaremos a cada subclase de *Creator* una implementación del método factoría que cree un tipo concreto de objeto.

En nuestro ejemplo, aplicaremos el método factoría haciendo que la decisión del tipo de mensaje a crear sea de la subclase. De esta manera, los usuarios que trabajen con notificaciones de tipo *NotificadorEmail* recibirán los mensajes por correo electrónico y los que utilicen un *NotificadorSMS* los recibirán en su teléfono móvil. Además, si queremos añadir a nuestro sistema nuevos mecanismos de notificación, tan sólo tendremos que añadir las nuevas subclases de *Notificador* y de *Mensaje*, respetando así el principio de abierto-cerrado.



Problemas específicos e implementación

Al implementar el patrón Factory Method podemos encontrarnos con alguno de los siguientes temas:

Definición de la clase Creator

Si aplicamos el patrón a código ya existente podrían aparecer problemas con la manera en que tenemos definida la clase Creator. Se dan tres casos:

- La clase Creator es abstracta y el método de factoría que define no tiene implementación alguna. En este caso, las clases ConcreteCreator tienen que definir su propio método de factoría. Esta situación suele darse en los casos en que la clase Creator no puede prever lo que la clase ConcreteProduct instanciará.
- La clase Creator es una clase concreta y el método de factoría presenta una implementación por defecto que podría perfectamente crear instancias y devolverlas. Si esto sucede, las clases ConcreteCreator utilizarán el método de factoría más por flexibilidad que por necesidad, pues siempre es interesante que el programador de una subclase tenga la opción de modificar el tipo de objeto que la super clase instancia.
- Existe una variante en que la clase Creator es una clase abstracta y el método de factoría presenta una implementación por defecto, pero es menos frecuente.

Métodos de factoría parametrizados

Hay una variante del patrón en que el método de factoría es capaz de crear múltiples clases de productos. El método de factoría recibe un parámetro que le permite identificar la clase de objeto a crear, todos implementando la interfaz Product.

```
public class Creator {  
  
    public Product factoryMethod(String tipo){  
        if (type.isEqual("A"))  
            return new ProductA();  
        if (type.isEqual("B"))  
            return new ProductB();  
        if (type.isEqual("C"))  
            return new ProductC();  
    }  
}
```

No obstante, esta variante tal y como está infringe el principio abierto-cerrando, por lo que en caso de necesitar nuevos productos tendríamos que redefinir el método factoría en la nueva ConcreteProduct.

```
public class MyCreator {  
  
    public Product factoryMethod(String tipo){  
        if (type.isEqual("A"))  
            return new ProductA();  
        if (type.isEqual("D"))  
            return new ProductD();  
  
        return super.factoryMethod(tipo);  
    }  
}
```

Notad que la clase MyCreator sólo se responsabiliza crear de productos A y productos D, por lo que si el tipo recibido por parámetro no fuera ninguno de estos identificadores hay que invocar a la super clase (esto es lo que hace la última línea).

Inconvenientes y beneficios

Veamos por último las ventajas e inconvenientes de Factory Method.

Ventajas:

- La razón principal por la que se utiliza este patrón es que introduce una separación clara entre la aplicación y una familia de clases (produce un acoplamiento débil al ocultar las clases concretas a la aplicación. Proporciona una forma sencilla de extender la familia de productos con tan sólo pequeños cambios en el código.
- Proporciona puntos de "enganche" personalizados. Cuando los objetos se crean directamente dentro de una clase es difícil, cuando surja la necesidad, sustituirlos por otros objetos que extiendan su funcionalidad. Si en lugar de esto, se utiliza una factoría para crear una familia de objetos, entonces es posible los objetos personalizados pueden reemplazar fácilmente a los objetos originales.

Inconvenientes:

- La factoría sólo puede utilizarse una familia de objetos. Si las clases no extienden de una misma clase base o implementar una interfaz entonces no pueden utilizarse según el patrón Factory Method.
- Para ver más inconvenientes consultar el patrón Factory.

Patrones relacionados

Es habitual implementar las clases de Abstract Factory como métodos de factoría.

Los métodos de factoría se utilizan a menudo junto con métodos plantilla (Template Method, patrón que veremos en su momento cuando abordemos los patrones de comportamiento).