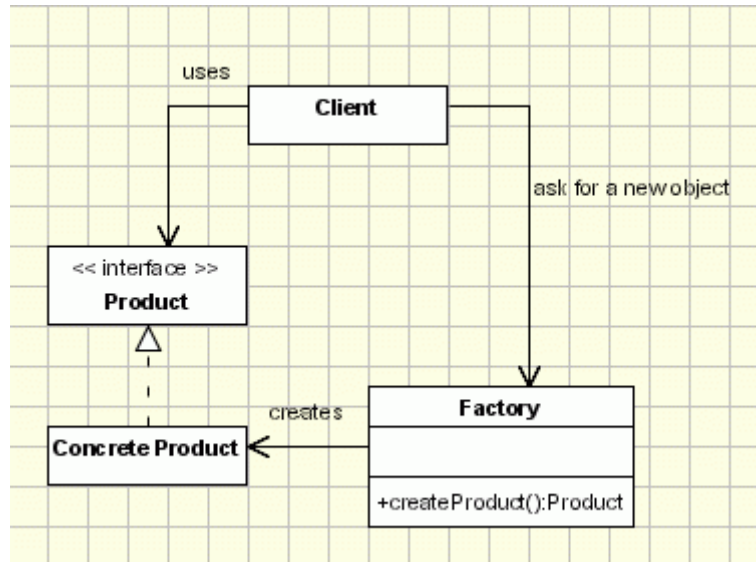


# Factory

## Diagrama de clases e interfaces



## Intención

- Crear objetos sin que quede expuesta al cliente la lógica de creación de las instancias.
- Que se tenga que hacer referencia al nuevo objeto a través de una interfaz conocida.

## Motivación

Factory -fábrica o factoría- es probablemente el patrón de diseño más utilizado en los lenguajes de programación modernos como Java y C#. Se presenta en diferentes variantes e implementaciones. No obstante, puede no ser considerado por algunos como un patrón de diseño, sino más bien un “programming idiom”.

Este patrón no lo encontraremos en la serie de patrones GoF. En GoF tenemos los patrones Factory Method y Abstract Factory, que están estrechamente relacionados con Factory. El origen del patrón Factory es otro muy similar: el patrón Factory Method, que veremos más adelante.

## Implementación

La implementación es muy sencilla. Una clase cliente necesita un objeto, pero en lugar de crearlo directamente con el operador *new*, solicita a la clase factoría que le proporcione el nuevo objeto. Para ello, es necesario que la clase cliente proporcione la información sobre el tipo de objeto que necesita. La fábrica (factoría) crea una instancia de un tipo concreto y la devuelve a la clase cliente, pero sólo le permite obtener una referencia al objeto a través de un tipo interfaz o una clase abstracta, lo que mantiene al cliente totalmente ajeno al tipo real del objeto instanciado.

## Aplicabilidad y Ejemplos

Por ejemplo, supongamos que tenemos que desarrollar una aplicación gráfica basada en figuras geométricas. En nuestra implementación, las clases clientes representan el framework de dibujo y las figuras geométricas son los objetos (también llamados productos en el contexto de los patrones) que se desean crear. Todas las figuras se derivan de una clase abstracta (podría ser también una interfaz) llamada Shape, que define las operaciones de dibujo y de movimiento que deben ser implementadas por las figuras concretas (Circle, Square, etc).

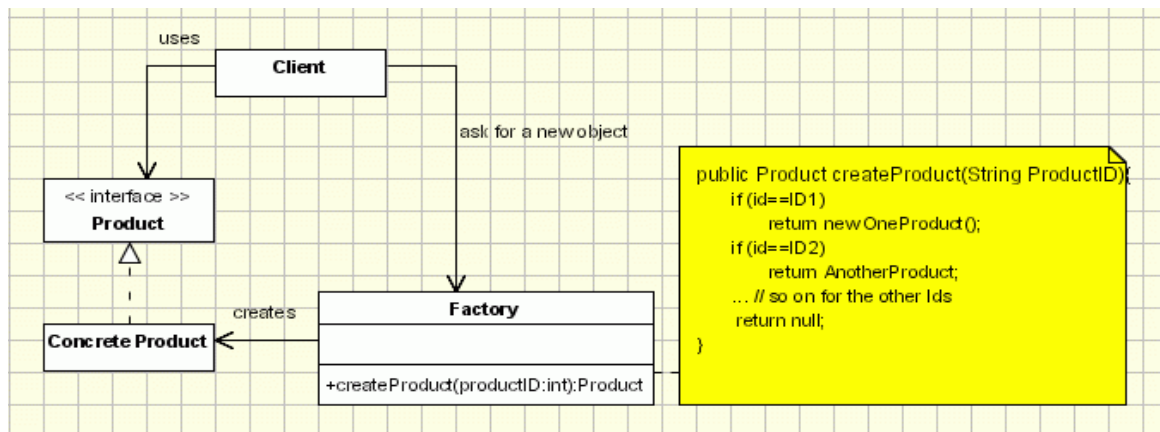
Supongamos que mediante un menú se selecciona la opción para crear un nuevo círculo. El framework de dibujo recibe a través de la opción de menú una cadena de texto (un String) que indica el tipo de la figura a construir, a continuación solicita a la factoría que cree una nueva figura a partir de esa cadena de texto. La factoría crea un nuevo círculo y lo devuelve al framework de dibujo, pero lo que devuelve es una referencia del tipo de la clase abstracta Shape y no de la clase concreta Circulo. A partir de aquí, el framework de dibujo utiliza el objeto recibido sin ser consciente del tipo concreto del objeto.

Esto desacopla totalmente a las clases cliente de las implementaciones utilizadas por los diseñadores de librerías, lo que permite que éstos puedan hacer cambios en las librerías sin afectar a los programas que las utilizan. Por ejemplo, es posible añadir nuevas figuras sin cambiar una sola línea de código en el framework de dibujo (el código cliente que utiliza las figuras creadas por la factoría). Como se verá más adelante, hay ciertas implementaciones de la factoría que permiten añadir nuevos tipos de objetos (productos) sin que sea necesario modificar siquiera la clase factoría

(normalmente la técnica consiste en leer de un fichero de propiedades o XML el nombre completamente cualificado de la clase de implementación de la que se tienen que crear objetos).

## Problemas específicos e implementación

### Solución procedural - instanciación de novato basada en switch/case



A esta implementación también se la conoce por factoría parametrizada. El método que crea el producto (la instancia) está escrito de manera que puede generar uno de entre varios subtipos de **Producto**. Para ello utiliza una condición que puede haber llegado a la factoría como un parámetro de método o haberse obtenido de algún parámetro de configuración global. A continuación se muestra una implementación para el primer caso:

Comenzamos por la jerarquía de Productos.

La interfaz:

```
package creacionales.factory.product;

public interface Product {

}
```

El producto A:

```
package creacionales.factory.product;

public class ProductA implements Product {
```

```
}
```

El producto B:

```
package creacionales.factory.product;

public class ProductB implements Product {

}
```

Seguimos con la clase factoría. Notad que se implementa como un Singleton:

```
package creacionales.factory;

import creacionales.factory.product.Product;
import creacionales.factory.product.ProductA;
import creacionales.factory.product.ProductB;

// Factoria implementada como Singleton
public class ProductFactoryNoob {
    private static ProductFactoryNoob factory =
        new ProductFactoryNoob();

    private ProductFactoryNoob() {

    }

    public static ProductFactoryNoob getInstance() {
        return factory;
    }

    public Product createProduct(String ProductID) {
        if (ProductID.equals("A"))
            return new ProductA();
        if (ProductID.equals("B"))
            return new ProductB();
        // lo mismo para otros Ids
        //...
        // si el id no tiene ninguno de los valores esperados
        return null;
    }
    //...
}
```

Terminamos con la clase cliente:

```
package creacionales.factory.main;

import creacionales.factory.ProductFactoryNoob;
import creacionales.factory.product.Product;

public class MainClientNoob {
```

```

    public static void main(String[] args) {

        // Crear un ProductA
        System.out.println("Creando un objeto ProductA");
        Product product = (Product) ProductFactoryNoob
            .getInstance().createProduct("A");
        System.out.println("Tipo creado: "+product.getClass());

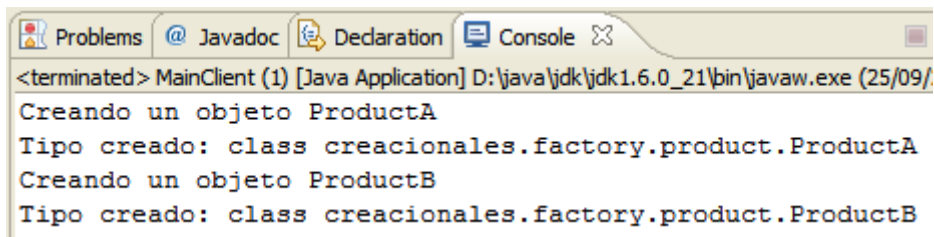
        // Crear un ProductB
        System.out.println("Creando un objeto ProductB");
        product = (Product) ProductFactoryNoob
            .getInstance().createProduct("B");
        System.out.println("Tipo creado: "+product.getClass());

    }

}

```

Salida:



```

<terminated> MainClient (1) [Java Application] D:\java\jdk\jdk1.6.0_21\bin\javaw.exe (25/09/
Creando un objeto ProductA
Tipo creado: class creacionales.factory.product.ProductA
Creando un objeto ProductB
Tipo creado: class creacionales.factory.product.ProductB

```

Esta implementación es la más sencilla e intuitiva de todas las que veremos, vamos a llamarla la implementación de novato. El problema aquí es que cada vez que se añade un producto concreto nuevo tenemos que modificar la factoría. No es muy flexible y se viola el principio de abierto/cerrado.

Aunque es posible subclasificar la factoría y crear una jerarquía de factorías, esto no es muy común, ya que generalmente una factoría se diseña como un Singleton para evitar problemas en entornos multihilo.

Además, puesto que la técnica de la factoría se basa en llamadas a métodos estáticos -luego no actúa el polimorfismo- si se lleva a cabo la subclasificación de la factoría, hay que pensar que en las clases cliente, en la parte de código que corresponda, habrá que sustituir las llamadas a la factoría base por las nuevas llamadas a la factoría subclase. Por ejemplo:

```

MiFactoria.getInstance().createProduct("A");
MiFactoriaHija.getInstance().createProduct("A");

```

### Uso de ficheros de propiedades para almacenar las clases de producto instanciables

Si tenemos en un fichero de propiedades la correspondencia entre un identificador y el nombre completamente cualificado de una subclase de Product, la factoría puede cargar este fichero y almacenarlo en memoria mediante una clase Properties (como una HashMap pero sólo válido para claves y valores de tipo String). Entonces, cuando una clase cliente invoque a createProduct() y pase como argumento un identificador de producto, la factoría buscará en el objeto Properties la clase que debe instanciar.

Veamos el código necesario:

#### La clase de factoría:

```
package creacionales.factory;

import java.io.IOException;
import java.io.InputStream;

import creacionales.factory.product.Product;

// Factoria implementada como Singleton
public class ProductFactoryPropFile {

    private static ProductFactoryPropFile factory = new
ProductFactoryPropFile();

    // true indica que ya se han cargado los nombres de las clases
    // desde el fichero de propiedades
    private static boolean propiedadesCargadas = false;

    // Propiedades
    private static java.util.Properties prop = new
java.util.Properties();

    public static ProductFactoryPropFile getInstance() {
        return factory;
    }

    public Product createProduct(String productID) {
        try {
            // La clase se obtiene leyendo del archivo properties
            Class<?> clase = Class.forName(getClase(productID));
            // Crear una instancia
            return (Product) clase.newInstance();
        } catch (ClassNotFoundException e) { // No existe la clase
            e.printStackTrace();
            return null;
        } catch (Exception e) { // No se puede instanciar la clase
            e.printStackTrace();
            return null;
        }
    }
}
```

```

        // Lee un archivo properties donde se indican las clases
        // instanciables
        private static String getClase(String nombrePropiedad) {
            try {
                // Carga de propiedades desde archivo -solo la
                // primera vez-
                if (!propiedadesCargadas) {
                    InputStream is = ProductFactoryPropFile.class
                        .getResourceAsStream("mapaProducts.
                        properties");

                    prop.load(is);
                    propiedadesCargadas = true;
                }

                // Lectura de propiedad
                String nombreClase =
                prop.getProperty(nombrePropiedad, "");
                if (nombreClase.length() > 0)
                    return nombreClase;

                } catch (IOException e) {
                    e.printStackTrace();
                }
                return null;
            }
        }
    }
}

```

Ahora el fichero de propiedades, que tenemos que crearlo en el mismo paquete que la factoría:

#### mapaProperties.properties

```

A=creacionales.factory.product.ProductA
B=creacionales.factory.product.ProductB

```

Finalmente, la clase cliente:

#### MainClientPropFile.java

```

package creacionales.factory.main;

import creacionales.factory.ProductFactoryPropFile;
import creacionales.factory.product.Product;

public class MainClientPropFile {

    public static void main(String args[]) {
        // Crear un ProductA
        System.out.println("Creando un objeto ProductA");
        Product product = (Product) ProductFactoryPropFile
            .getInstance().createProduct("A");
        System.out.println("Tipo creado: "+product.getClass());
        System.out.println(product.getDescripcion());
    }
}

```

```

        // Crear un ProductB
        System.out.println("Creando un objeto ProductB");
        product = (Product) ProductFactoryPropFile
            .getInstance().createProduct("B");
        System.out.println("Tipo creado: "+product.getClass());
        System.out.println(product.getDescripcion());
    }
}

```

Notad que es una solución muy simple y elegante, pues nos permite crear nuevos productos concretos sin tener que recompilar el código.

### Registro de clase - Uso de la reflexión

Mediante la reflexión se pueden registrar nuevos tipos de productos en la factoría sin tener que modificar el propio código de la factoría. La idea es que una clase cliente invoca al método `createProduct()` proporcionándole un `ProductID` que sirve como clave de búsqueda dentro de un `HashMap` que registra la correspondencia entre el `ProductID` y la subclase de `Product` que se debe crear. Este mapa se implementa en la factoría y el código que lo invoca se puede situar en cualquier parte de la aplicación, aunque un lugar conveniente suele ser en las propias subclases de `Producto`, como veremos dentro de poco.

Veamos a continuación la implementación de la factoría. Se han considerado las siguientes características:

- Contiene un `HashMap` para almacenar la correspondencia entre cada `ProductID` y cada subclase de `Product` que puede instanciar
- Contiene un método llamado `registerProduct()` que es invocado por cada subclase de `Product` para añadir al `HashMap` comentado anteriormente una nueva correspondencia `ProductID-clase`.
- Utiliza el API `Reflection` para instanciar subclases de `Producto` mediante introspección. Concretamente, contiene dos métodos para construir las subclases:
  - Uno para crear instancias utilizando el constructor sin parámetros de la subclase de `Producto` cuyo `ProductID` recibe por parámetro.
  - Otro que sobrecarga al anterior y que permite invocar a cualquier constructor de la subclase de `Producto` cuyo `ProductID` recibe como



primera parámetro. El segundo parámetro es del tipo varargs, por lo que puede contener un número variable de argumentos que sirven para instanciar a uno de los constructores.

El código para la factoría queda como sigue:

```
package creacionales.factory;

import java.lang.reflect.Constructor;
import java.util.HashMap;

import creacionales.factory.product.Product;

// Factoria implementada como Singleton
public class ProductFactoryReflex {
    private static ProductFactoryReflex factory =
        new ProductFactoryReflex();

    // Mapa para correspondencia ProductID-Clase
    private HashMap<String, Class<?>>
        m_RegisteredProducts = new HashMap<String, Class<?>>();

    public void
        registerProduct (String productID, Class<?> productClass) {
        m_RegisteredProducts.put(productID, productClass);
    }

    public static ProductFactoryReflex getInstance() {
        return factory;
    }

    // Método para cuando se quiere ejecutar el
    // constructor sin parámetros del Producto
    public Product createProduct(String productID) {
        Class<?> productClass =
            (Class<?>) m_RegisteredProducts.get(productID);

        try {
            return (Product) productClass.newInstance();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    // Metodo capaz de ejecutar cualquier constructor del Producto
    // Notad que el segundo argumento es un varargs, lo cual
    // nos permite buscar entre la lista de constructores de Product
    public Product
        createProduct(String productID, Object... argsConstructor)
    {
        Class<?> productClass =
            (Class<?>) m_RegisteredProducts.get(productID);
        Constructor<?> productConstructor = null;

        try {
```

```

        boolean continuar = false;

        // Obtener un array con todos los constructores
        // de la clase
        Constructor<?>[] ctors =
            productClass.getDeclaredConstructors();
        Constructor<?> ctor = null;

        // Si algun constructor coincide con el numero de
        // parámetros de parametrosConstructor entonces, a
        // priori, hay coincidencia y se podra instanciar el
        // constructor
        for (int i = 0; i < ctors.length; i++) {
            ctor = ctors[i];
            if (argsConstructor.length ==
                ctor.getGenericParameterTypes().length) {
                continuar = true;
                // hay coincidencia en el num. de params
                break;
            }
        }

        if (continuar) {
            // Recorrer los argumentos enviados al
            // constructor y comprobar que son del mismo
            // tipo que el constructor candidato
            for (int i = 0;
                i < argsConstructor.length; i++) {
                String tipoArgumentoCons =
                    argsConstructor[i].getClass().getName();
                String tipoParamCons =
                    ctor.getParameterTypes()[i].getName();
                if (!tipoArgumentoCons
                    .equals(tipoParamCons)) {
                    throw new RuntimeException("No hay
ningun constructor adecuado para esos argumentos");
                }
            }

            // Ha habido coincidencia completa entre los
            // argumentos enviados y uno de constructor de
            // la clase, por tanto se puede invocar

            productConstructor = productClass
                .getDeclaredConstructor(ctor
                    .getParameterTypes());

            return (Product) productConstructor
                .newInstance(argsConstructor);
        }

        throw new RuntimeException("No hay ningun constructor
adecuado para esos argumentos");
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

```

```
}  
}
```

Para poner a prueba la nueva versión de la factoría vamos a modificar la jerarquía de Product. Haremos que la interfaz Product defina un método llamado `getDescripcion()`, por lo que las subclases tendrán que implementarlo:

#### Product.java

```
package creacionales.factory.product;  
  
public interface Product {  
    public String getDescripcion();  
}
```

Las subclases tendrán ahora las siguientes características:

- Un atributo de String denominado 'descripcion'.
- Un inicializador estático para registrarse como subclase de Product en el HashMap de la factoría.
- Dos constructores:
  - Uno sin parámetros. Asignará el texto 'sin descripcion' al atributo 'descripcion'.
  - Uno con un parámetro de tipo String que se asignará al atributo 'descripcion'.
- Implementación del método `getDescripcion()` definido en la interfaz Product. La implementación consistirá simplemente en retornar el atributo 'descripcion'.

Veamos las subclases:

#### ProductA.java

```
package creacionales.factory.product;  
  
import creacionales.factory.ProductFactoryReflex;  
  
public class ProductA implements Product {  
    private String descripcion;  

```

```

    static {
        ProductFactoryReflex.getInstance()
            .registerProduct("A",
                creacionales.factory.product.ProductA.class);
    }

    public ProductA () {
        this.descripcion = "sin descripcion";
    }

    public ProductA (String descripcion) {
        this.descripcion = descripcion;
    }

    @Override
    public String getDescripcion() {
        return descripcion;
    }
}

```

### ProductB.java

```

package creacionales.factory.product;

import creacionales.factory.ProductFactoryReflex;

public class ProductB implements Product {
    private String descripcion;

    static {
        ProductFactoryReflex.getInstance()
            .registerProduct("B",
                creacionales.factory.product.ProductB.class);
    }

    public ProductB () {
        this.descripcion = "sin descripcion";
    }

    public ProductB (String descripcion) {
        this.descripcion = descripcion;
    }

    @Override
    public String getDescripcion() {
        return descripcion;
    }
}

```

Nos resta ver cómo queda la clase que contiene el método main(). Pero antes veamos la siguiente consideración:

Problema: Tenemos que asegurarnos de que las clases de productos concretos ya estén cargadas cuando la factoría lleve a cabo el proceso de registro de las subclases.

En caso contrario no quedarán registradas en la factoría y el método createProduct() devolverá un valor nulo.

Solución: Para garantizar esto, vamos a utilizar el método Class.forName() en la sección del inicializador estático de la clase donde tenemos definido el método main() (un inicializador estático se ejecuta siempre inmediatamente después de la carga de la clase). Recordemos que Class.forName() tiene que devolver una instancia de la clase indicada. Si el cargador de clases (ClassLoader) aún no hubiera cargado alguna de las subclases de Producto, ésta será cargada al invocarse Class.forName(). Por tanto, al cargarse cada subclase de Product se ejecutará su correspondiente bloque estático.

El código para la clase con el main() podría quedar como sigue:

```
package creacionales.factory.main;

import creacionales.factory.ProductFactoryReflex;
import creacionales.factory.product.Product;

public class MainClientReflex {
    static {
        try {

            Class.forName("creacionales.factory.product.ProductA");

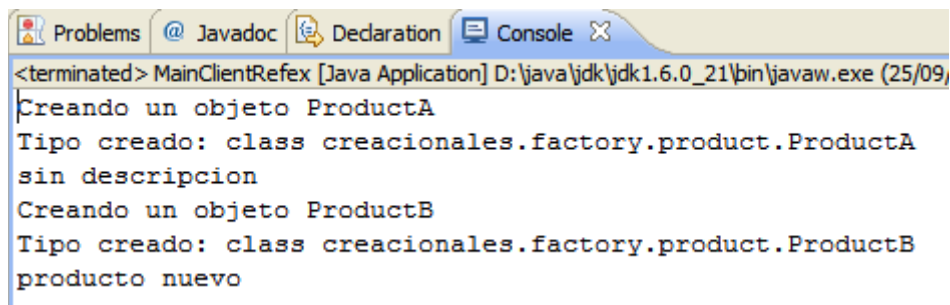
            Class.forName("creacionales.factory.product.ProductB");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) {

        // Crear un ProductA
        System.out.println("Creando un objeto ProductA");
        Product product = (Product) ProductFactoryReflex
            .getInstance().createProduct("A");
        System.out.println("Tipo creado: "+product.getClass());
        System.out.println(product.getDescripcion());

        // Crear un ProductB
        System.out.println("Creando un objeto ProductB");
        product = (Product) ProductFactoryReflex
            .getInstance().createProduct("B", "producto nuevo");
        System.out.println("Tipo creado: "+product.getClass());
        System.out.println(product.getDescripcion());
    }
}
```

Salida:



```
<terminated> MainClientRefex [Java Application] D:\java\jdk\jdk1.6.0_21\bin\javaw.exe (25/09/)  
Creando un objeto ProductA  
Tipo creado: class creacionales.factory.product.ProductA  
sin descripcion  
Creando un objeto ProductB  
Tipo creado: class creacionales.factory.product.ProductB  
producto nuevo
```

Notad que hemos creado el producto A de manera diferente al producto B.

Cuando creamos el primero no proporcionamos una descripción, por lo que la factoría invoca al constructor sin parámetros de ProductA, cuyo comportamiento es asignar el texto 'sin descripcion' al atributo 'descripcion'.

Sin embargo, cuando creamos el segundo producto pasamos una descripción a la factoría, quien utiliza este argumento para localizar dentro de ProductB un constructor que tenga una lista de un único parámetro y que además sea de tipo String. Cuando lo localiza, lo instancia y su ejecución implica que se asigne el texto "producto nuevo" al atributo 'descripcion'.

Hay que destacar la flexibilidad que nos proporciona la factoría. Por ejemplo, sería muy sencillo añadir un nuevo constructor que aceptase además de la descripción la fecha de creación. Tan sólo tendríamos que modificar las subclases de Product y la clase que contiene el main(). Veamos como queda para el producto B:

#### ProductB.java

```
package creacionales.factory.product;  
  
import java.util.Date;  
  
import creacionales.factory.ProductFactoryReflex;  
  
public class ProductB implements Product {  
    private String descripcion;  
    private Date fechaCreacion;  
  
    static {  
        ProductFactoryReflex.getInstance()  
            .registerProduct("B",  
creacionales.factory.product.ProductB.class);  
    }  
  
    public ProductB () {
```

```

        this.descripcion = "sin descripcion";
    }

    public ProductB (String descripcion) {
        this.descripcion = descripcion;
    }

    public ProductB (String descripcion, Date fechaCreacion) {
        this.descripcion = descripcion;
        this.fechaCreacion = fechaCreacion;
    }

    @Override
    public String getDescripcion() {
        return descripcion;
    }

    @Override
    public String toString() {
        return "ProductB [descripcion=" + descripcion + ",
fechaCreacion=" + fechaCreacion + "]";
    }
}

```

Y en la clase con el main(), si quisiéramos crear un producto B adicional, pero proporcionando además la fecha de creación, añadiríamos lo siguiente al final del código:

```

...
// Crear otro ProductB
System.out.println("Creando otro objeto ProductB");

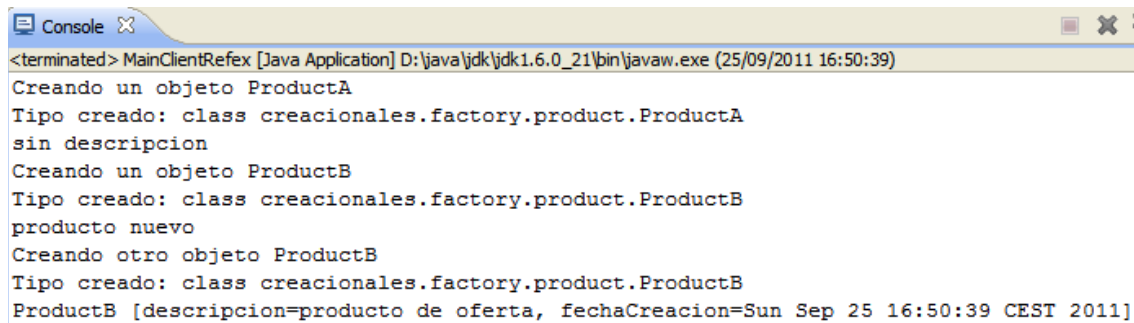
Calendar cal = Calendar.getInstance();
cal.set(Calendar.DAY_OF_MONTH, 25);
cal.set(Calendar.MONTH, 8); // 0=enero - 8=septiembre
cal.set(Calendar.YEAR, 2011);

product = (Product) ProductFactoryReflex
    .getInstance().createProduct("B", "producto de
oferta", cal.getTime());
System.out.println("Tipo creado: "+product.getClass());
System.out.println(product);
...

```

La factoría, por supuesto, no tendríamos que tocarla para nada.

Salida:



```
<terminated> MainClientReflex [Java Application] D:\java\jdk\jdk1.6.0_21\bin\javaw.exe (25/09/2011 16:50:39)
Creando un objeto ProductA
Tipo creado: class creacionales.factory.product.ProductA
sin descripcion
Creando un objeto ProductB
Tipo creado: class creacionales.factory.product.ProductB
producto nuevo
Creando otro objeto ProductB
Tipo creado: class creacionales.factory.product.ProductB
ProductB [descripcion=producto de oferta, fechaCreacion=Sun Sep 25 16:50:39 CEST 2011]
```

El único inconveniente que se le puede encontrar a la implementación de la factoría vista en esta sección es una mínima pérdida de rendimiento. Lo cierto es que las versiones modernas del Java no acusan este problema como las versiones antiguas. Por tanto, esta implementación es una buena decisión a no ser que tuviéramos alguno requerimiento donde esa mínima pérdida de rendimiento fuese un factor decisivo.

### Registro de clases - evitando la reflexión

Como vimos anteriormente, la factoría utiliza internamente un HashMap para mantener la correlación entre los parámetros (en nuestro caso Strings) y las subclases de Product. El registro en el HashMap se realiza desde fuera de la factoría (desde cada subclase de Product) y la factoría usa el API Reflection para crear cada objeto, por lo que no conoce hasta tiempo de ejecución el tipo concreto de la subclase que debe crear.

Podemos emplear otra implementación de la factoría si por cualquier motivo no queremos utilizar el mecanismo de reflexión pero al mismo tiempo queremos tener un acoplamiento reducido entre la fábrica y la subclase de Product.

Para ello, tenemos que extraer el proceso de construcción de objetos de la clase factoría y llevarlo a algún objeto que sí esté claramente acoplado con cada clase concreta de Product. La solución obvia pasa por hacer que sea cada subclase de Product la que se responsabilice de crear un objeto de su propio tipo.

Por tanto, añadimos un nuevo método llamado createProduct() en la interfaz Product y lo implementamos en cada una de sus subclases. Este método creará un objeto de la



propia subclase y lo retornará (Desde Java 5 el overriding es válido aunque el método de la subclase devuelva un subtipo del método de la superclase).

Al no utilizar en la factoría la técnica de la reflexión, también tenemos que cambiar el método de registro de manera que lo que se registre sea un objeto del tipo concreto de la subclase y no un objeto Class (como hacíamos antes).

Veamos como luce el nuevo código:

La factoría. Notad que ahora el código es mucho más reducido que en la implementación con reflexión.

#### ProductFactoryNoReflex.java

```
package creacionales.factory;

import java.util.HashMap;

import creacionales.factory.product.Product;

// Factoria implementada como Singleton
public class ProductFactoryNoReflex {
    private static ProductFactoryNoReflex factory =
        new ProductFactoryNoReflex();

    // Mapa para correspondencia ProductID-Subclase
    private HashMap<String, Product>
        m_RegisteredProducts = new HashMap<String, Product>();

    public void
        registerProduct (String productID, Product product) {
        m_RegisteredProducts.put (productID, product);
    }

    public static ProductFactoryNoReflex getInstance() {
        return factory;
    }

    public Product createProduct (String productID) {
        Product product =
            (Product) m_RegisteredProducts.get (productID);
        return product.createProduct ();
    }
}
```

Ahora la interfaz.

#### Product.java

```
package creacionales.factory.product;
```

```
public interface Product {
    public Product createProduct();
    public String getDescripcion();
}
```

Las subclases que implementan la interfaz:

#### ProductA.java

```
package creacionales.factory.product;

import creacionales.factory.ProductFactoryNoReflex;

public class ProductA implements Product {
    private String descripcion;

    static {
        ProductFactoryNoReflex.getInstance()
            .registerProduct("A", new ProductA());
    }

    public ProductA () {
        this.descripcion = "sin descripcion";
    }

    public ProductA (String descripcion) {
        this.descripcion = descripcion;
    }

    @Override
    public String getDescripcion() {
        return descripcion;
    }

    @Override
    public ProductA createProduct() {
        return new ProductA();
    }
}
```

#### ProductB.java

En este caso se han eliminado las características que se habían añadido en esta clase en la versión que usaba la factoría que utilizaba el API Reflection (varios constructores, el atributo fecha). La razón es que en esta nueva versión todo es más limitado al no poder hacer introspección de clases desde la factoría.

```
public class ProductB implements Product {
    private String descripcion;

    static {
        ProductFactoryNoReflex.getInstance()
            .registerProduct("B", new ProductB());
    }
}
```

```

    public ProductB () {
        this.descripcion = "sin descripcion";
    }

    @Override
    public String getDescripcion() {
        return descripcion;
    }

    @Override
    public ProductB createProduct () {
        return new ProductB ();
    }
}

```

Sólo falta la clase que tiene el método main().

```

package creacionales.factory.main;

import creacionales.factory.ProductFactoryNoReflex;
import creacionales.factory.product.Product;

public class MainClientNoReflex {
    static {
        try {

            Class.forName("creacionales.factory.product.ProductA");

            Class.forName("creacionales.factory.product.ProductB");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) {
        // Crear un ProductA
        System.out.println("Creando un objeto ProductA");
        Product product = (Product) ProductFactoryNoReflex
            .getInstance().createProduct("A");
        System.out.println("Tipo creado: "+product.getClass());
        System.out.println(product.getDescripcion());

        // Crear un ProductB
        System.out.println("Creando un objeto ProductB");
        product = (Product) ProductFactoryNoReflex
            .getInstance().createProduct("B");
        System.out.println("Tipo creado: "+product.getClass());
        System.out.println(product.getDescripcion());
    }
}

```

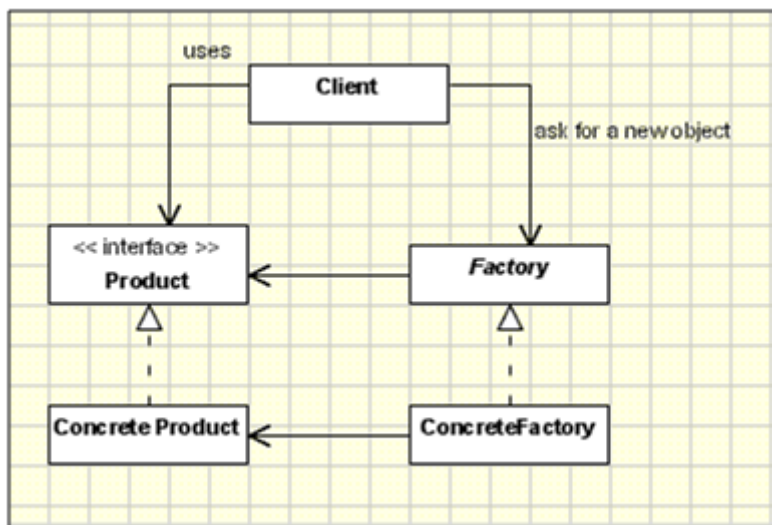
### Una solución más avanzada - El patrón Factory con abstracciones (Factory Method)

Esta implementación representa una alternativa para las implementaciones que usan un HashMap para el registro de las subclases.

Supongamos que tenemos que añadir un nuevo subtipo de Product a la aplicación. Mediante la implementación switch-case (la primera que hemos visto) tendremos que modificar la factoría para contemplar este nuevo tipo de producto, mientras que en las implementaciones con HashMap todo lo que necesitamos es registrar el nuevo subtipo en la factoría, pero sin que sea 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. Como podemos intuir, la solución para evitar la modificación de la factoría es simple: extenderla.

Esto es lo que hace el (clásico) patrón Factory Method. A continuación podemos ver el diagrama de clases de este patrón:



Es importante tener en mente que Factory Method tiene varios inconvenientes y no tiene demasiadas ventajas respecto a la implementación basada en el HashMap:

Ventajas sobre las implementaciones HashMap:

- El método de factoría derivado se puede redefinir para realizar operaciones adicionales cuando se crean los objetos (por ejemplo, algunas tareas de inicialización en función de parámetros globales).

Inconvenientes respecto a las implementaciones HashMap:

- La factoría no puede utilizarse como un Singleton.
- Cada factoría debe ser inicializada antes de poder usarse.
- Es más difícil de implementar.
- Se debe crear una nueva factoría cada vez que se crea un nuevo subtipo de Product.

De todos modos, es necesario conocer la implementación del Factory Method, dado que nos ayudará a comprender el funcionamiento de otro patrón muy interesante: el patrón Abstract Factory.

Más adelante veremos ejemplos de código tanto de Factory Method como de Abstract Factory.

## Conclusión

Cuando se diseña una aplicación hay que pensar detenidamente si realmente necesitamos una factoría para crear objetos. Podría ser que no la necesitáramos y en cambio estuviéramos añadiendo una complejidad innecesaria a la aplicación. Si tenemos muchos objetos cuyas clases implementan las mismas interfaces (o tienen la misma superclase) y manipularlos implica enmarañar el código con líneas donde abundan las operaciones de cast, posiblemente necesitemos una factoría. Por ejemplo, si tenemos en varias partes del código sentencias como la siguiente, tendríamos que considerar la implementación de una factoría:

```
(if (ConcreteProduct)genericProduct typeof )  
    ((ConcreteProduct)genericProduct).doSomeConcreteOperation()
```

Si decidimos implementar la factoría, lo recomendable es usar una de las implementaciones basadas en HashMap (registro de subclases), con o sin reflexión, y evitar el patrón Factory Method (patrón Factory con abstracciones). Finalmente, hemos de tener en cuenta que la implementación switch-case (novato) es la más simple de todas, pero viola el principio de abierto/cerrado y se ha visto sólo para contrastarla con

los demás casos. Su uso, sólo queda justificado para clases temporales hasta que sea reemplazada por una factoría bien diseñada.