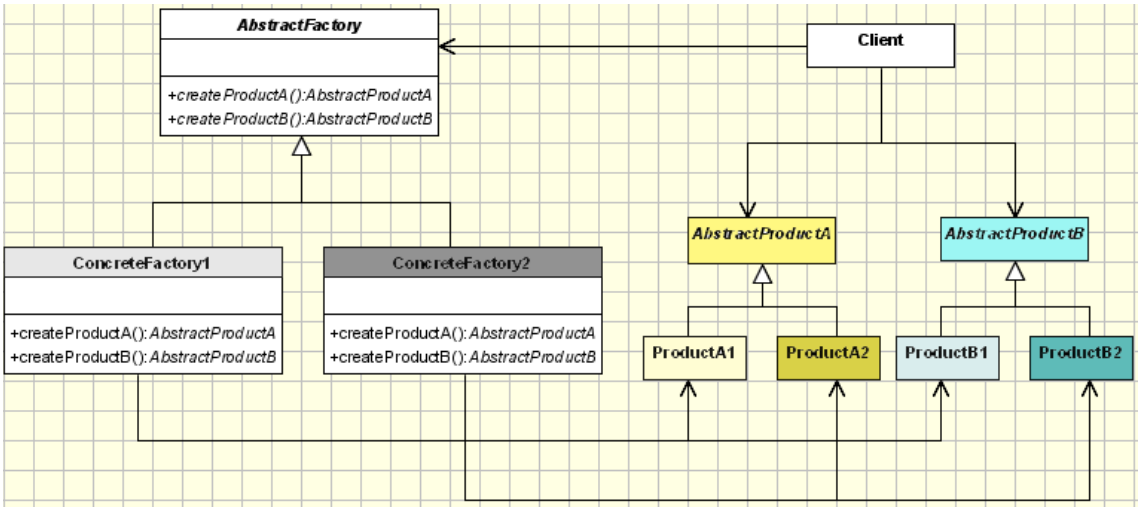
Abstract Factory

Diagrama de clases e interfaces



Intención

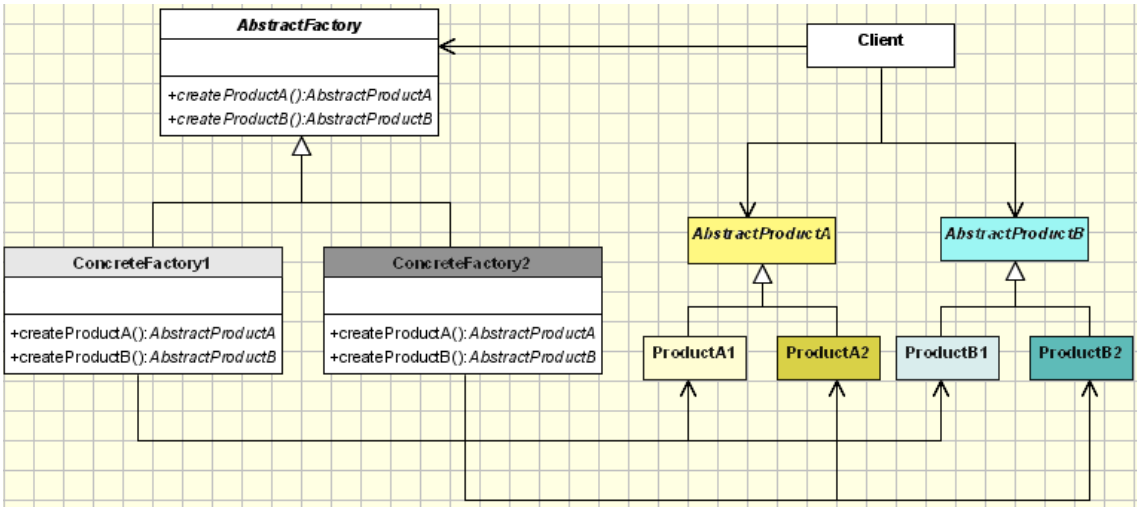
Abstract Factory permite seleccionar entre diferentes familias de objetos -que implementan las mismas interfaces- sin que sea necesario explicitar sus clases. Abstrae completamente a las clases cliente de los detalles de implementación de cada familia de productos, proporcionándoles un tipo interfaz o abstracto como referencia.

Motivación

Abstract Factory es un patrón de diseño muy potente que se puede utilizar para desacoplar a las clases cliente o a una capa de nuestra aplicación de *implementaciones específicas* de determinados servicios. Según esta definición podríamos pensar que esto es justamente lo que hace el patrón Factory o el Factory Method, pero no es así, puesto que Abstract Factory llega mucho más lejos: es una súper factoría que crea otras factorías; es una factoría de factorías. Estas *implementaciones específicas* que se han comentado son en realidad familias de clases que implementan las mismas interfaces y que nos interesa que sean intercambiables. Lo interesante de este patrón es que permite que las clases cliente funcionen igualmente con una u otra familia de clases sin tenerlas que modificar. La

posibilidad de diseñar una aplicación que pueda trabajar con diferentes familias de clases es de lo más sugerente.

Implementación



Clases que participan en el patrón Abstract Factory:

Client: Es la clase que llama a la factoría adecuada ya que necesita crear uno de los objetos que ésta proporciona, es decir, su interés es obtener una instancia de alguno de los productos (ProductAx, ProductBx). El cliente desconoce los tipos concretos que serán realmente instanciados, tan sólo podrá acceder a ellos mediante una referencia a la clase AbstractProductX.

AbstractFactory: Es el componente más importante del patrón, la factoría de factorías. Define la interfaz común a todas factorías. Debe de proporcionar un método para la obtención de cada objeto que pueda crear (createProductA(), createProductB()). AbstractFactory determina el tipo concreto de un objeto y lo crea, pero lo que devuelve es una interfaz al objeto concreto recién creado. Esto aísla al cliente sobre las implementaciones reales de los objetos que solicita.

ConcreteFactory: Estas son las diferentes familias de productos. Sólo pueden ser accedidas por la AbstractFactory, por lo que tendrán visibilidad de paquete. Crean y devuelven la instancia concreta que les compete, aunque mediante una referencia a un producto abstracto. De esta forma podemos tener una factoría para MySQL y otra para Oracle, por ejemplo. En la figura, la clase ConcreteFactory1 crea ProductA1 y ProductB1 que supongamos que fueran, respectivamente, implementaciones MySQL de las interfaces Connection y Statement.

AbstractProduct: Define la interfaz para un tipo de producto, por ejemplo la interfaz para Connetion. La implementación de Connection de cada fabricante tendrá que implementar esta interfaz. Así, la clase AbstractProductA del diagrama podría ser la interfaz Connection y AbstractProductB podría ser la interfaz Statement. El cliente trabajará directamente sobre estas interfaces.

Product: Los ProductX sólo pueden ser accedidos por las ConcreteFactoyX y por las AbstractProduct, por lo que tendrán visibilidad de paquete. Implementan la interfaz AbstractProduct y definen un producto concreto que será creado por la correspondiente ConcreteFactory. Por ejemplo, en el diagrama la clase ProductA1 podría ser ConnectionImplMySQL y la clase ProductA2 podría ser ConnectionImplOracle. Siguiendo con este supuesto, la clase ProductB1 podría ser StatementImplMySQL y la clase ProductB2 podría ser StatementImplOracle.

Código

A continuación se muestra el código de una implementación de Abstract Factory basada en una clase abstracta que implementa la interfaz IFactory. La ventaja de este enfoque es que soluciona el siguiente problema: ¿cómo sabe la aplicación que factoría concreta utilizar?¿ConcretaFactory1 o ConcretaFactory2? Veamos el código.

Creamos un fichero de propiedades en el mismo paquete que la factoría abstracta y sus subclases. En el fichero escribimos el nombre completamente cualificado de la factoría concreta que va a utilizar la aplicación. Este caso he escogido ConcretaFactory2:

familia.properties

```
familia=creacionales.abstractfactory.ConcreteFactory2
```

Ahora vamos con las clases.

Creamos una interfaz con los métodos que tendrán que implementar la factoría abstracta y, por consiguiente, las factorías concretas:

IFactoria.java

```
package creacionales.abstractfactory;

import creacionales.abstractfactory.products.AbstractProductA;
import creacionales.abstractfactory.products.AbstractProductB;

public interface IFactoria {

	public AbstractProductA createProductA();
	public AbstractProductB createProductB();

}
```

A continuación creamos la clase abstracta que implementa la interfaz anterior. Esta clase actúa de factoría y mediante un método getInstancia() del patrón Singleton las clases cliente pueden colaborar con ella y obtener una referencia a una de las instancias de sus subclases.

AbstractFactory.java

```
package creacionales.abstractfactory;

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

public abstract class AbstractFactory implements IFactoria {

	private static IFactoria instancia;

	public static synchronized IFactoria getInstancia() {
		if (instancia == null) {
			try {
				// La familia se obtiene leyendo del archivo
properties
				Class<?> clase =
Class.forName(getFamilia("familia"));
				// Crear una instancia
				instancia = (IFactoria) clase.newInstance();
				System.out.println("Familia utilizada:
"+instancia.getClass().getSimpleName());
			} catch (ClassNotFoundException e) { // No existe la
clase
				e.printStackTrace();
			} catch (Exception e) { // No se puede instanciar la
clase
				e.printStackTrace();
			}
		}

		return instancia;
```

```
	}

	// true indica que ya se han cargado la familia
	// desde el fichero de propiedades
	private static boolean propiedadesCargadas = false;

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

	public abstract AbstractProductA createProductA();

	public abstract AbstractProductB createProductB();

	// Lee un archivo properties donde se indican la familia que se
quiere utilizar
	private static String getFamilia(String nombrePropiedad) {
		try {
			// Carga de propiedades desde archivo -solo la
primera vez-
			if (!propiedadesCargadas) {
				InputStream is = IFactoria.class
						.getResourceAsStream("familia.prope
rties");

				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;
	}
}
```

Es el turno de las factorías concretas. Notad que tienen visibilidad de paquete, ya que tan sólo debe acceder a ellas la factoría abstracta y nunca una clase cliente.

ConcreteFactory1.java

```
package creacionales.abstractfactory;

/*
 * Tiene visibilidad de paquete, ya que tan sólo debe
 * acceder a ella la factoría abstracta.
 */
class ConcreteFactory1 extends AbstractFactory {
	public AbstractProductA createProductA(){
```

```
            return new ProductA1("ProductA1");
      }
      public AbstractProductB createProductB(){
            return new ProductB1("ProductB1");
      }
}
```

ConcreteFactory2.java

```
package creacionales.abstractfactory;

/*
 * Tiene visibilidad de paquete, ya que tan sólo debe
 * acceder a ella la factoría abstracta.
 */
class ConcreteFactory2 extends AbstractFactory{
      public AbstractProductA createProductA(){
            return new ProductA2("ProductA2");
      }
      public AbstractProductB createProductB(){
            return new ProductB2("ProductB2");
      }
}
```

Ahora vamos con las jerarquías de productos. Las clases clientes quieren obtener productos concretos, y de hecho los obtienen, pero sólo pueden referenciarlos mediante un tipo base, en nuestro caso: AbstractProductA o AbstractProductB. Hubiera sido perfectamente válido utilizar interfaces en lugar de clases abstractas.

Veamos primeros las clases abstractas:

AbstractProductA.java

```
package creacionales.abstractfactory;

public abstract class AbstractProductA {
      public abstract void metodoP();
      public abstract void metodoQ();
}
```

AbstractProductB.java

```
package creacionales.abstractfactory;

public abstract class AbstractProductB {
      public abstract void metodoR();
      public abstract void metodoS();
}
```

Ahora los productos concretos. Notad que tienen visibilidad de paquete, ya que tan sólo deben acceder a ellas la factoría concreta correspondiente.

ProductA1.java

```
package creacionales.abstractfactory;

/*
 * Tiene visibilidad de paquete, ya que tan sólo debe
 * acceder a ella la factoría concreta correspondiente
 */
class ProductA1 extends AbstractProductA {
	public ProductA1(String string) {
		System.out.println("Creado ProductA1");
	}

	public void metodoP() {
		System.out.println("ProductA1: Ejecutado metodoP()");
	}

	public void metodoQ() {
		System.out.println("ProductA1: Ejecutado metodoQ()");
	}
}
```

ProductA2.java

```
package creacionales.abstractfactory;

/*
 * Tiene visibilidad de paquete, ya que tan sólo debe
 * acceder a ella la factoría concreta correspondiente
 */
class ProductA2 extends AbstractProductA {
	public ProductA2(String string) {
		System.out.println("Creado ProductA2");
	}

	public void metodoP() {
		System.out.println("ProductA2: Ejecutado metodoP()");
	}

	public void metodoQ() {
		System.out.println("ProductA2: Ejecutado metodoQ()");
	}
}
```

ProductB1.java

```
package creacionales.abstractfactory;

/*
 * Tiene visibilidad de paquete, ya que tan sólo debe
 * acceder a ella la factoría concreta correspondiente
 */
class ProductB1 extends AbstractProductB {
	public ProductB1(String string) {
		System.out.println("Creado ProductB1");
	}
```

```
	public void metodoR() {
		System.out.println("ProductB1: Ejecutado metodoR()");
	}

	public void metodoS() {
		System.out.println("ProductB1: Ejecutado metodoS()");
	}
}
```

ProductB2.java

```
package creacionales.abstractfactory;

/*
 * Tiene visibilidad de paquete, ya que tan sólo debe
 * acceder a ella la factoría concreta correspondiente
 */
class ProductB2 extends AbstractProductB {
	public ProductB2(String string) {
		System.out.println("Creado ProductB2");
	}

	public void metodoR() {
		System.out.println("ProductB2: Ejecutado metodoR()");
	}

	public void metodoS() {
		System.out.println("ProductB2: Ejecutado metodoS()");
	}
}
```

Y para finalizar, creamos una clase cliente:

MainClient.java

```
package creacionales.abstractfactory.main;

import creacionales.abstractfactory.AbstractFactory;
import creacionales.abstractfactory.AbstractProductA;
import creacionales.abstractfactory.AbstractProductB;

/**
 * La familia con la que se trabaja se lee del fichero de
 * propiedades 'familia.properties'
 */
public class MainClient {

	public static void main(String[] args) {
		AbstractProductA productA;

		productA = AbstractFactory.getInstancia().createProductA();

		productA.metodoP();
		productA.metodoQ();
```

```
            AbstractProductB productB;

            productB = AbstractFactory.getInstancia().createProductB();
            productB.metodoR();
            productB.metodoS();
      }

}
```

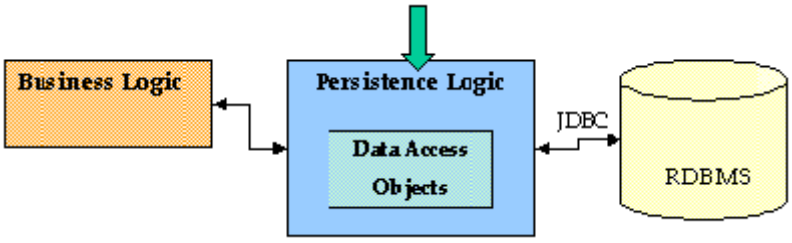
Salida:

```
Console
<terminated> MainClient (3) [Java Application] D:\java\jdk\jc
Familia utilizada: ConcreteFactory2
Creado ProductA2
ProductA2: Ejecutado metodoP()
ProductA2: Ejecutado metodoQ()
Creado ProductB2
ProductB2: Ejecutado metodoR()
ProductB2: Ejecutado metodoS()
```

Aplicabilidad y Ejemplos

Debemos usar el patrón Abstract Factory cuando:

-Un programa o módulo necesita funcionar independientemente de la manera en que se crean los productos con los que trabaja. Un ejemplo claro de esto lo tenemos con la capa de negocio y la capa de persistencia.



Si la capa de negocio interactúa con la capa de persistencia solamente a través de interfaces o clases abstractas, es decir, de manera independiente a la implementación de los DAO (Data Access Object), es posible implementar varias técnicas de almacenamiento (en base de datos, en ficheros planos, en memoria, etc.) y que la capa de negocio se mantenga totalmente operativa e ignorante del sistema de almacenamiento real.

-El sistema necesita configurarse para trabajar con múltiples familias de productos. Esto es muy propio de los frameworks y librerías y más raro en una típica aplicación de gestión.

Ejemplo 1 – API JDBC (No se proporciona código)

Una de las API's más populares de Java es el API JDBC. Esta potente colección de interfaces y clases hace posible que cambiemos de una implementación de base de datos a otra con tan sólo modificar una línea de código. Ha sido diseñada de esta manera para que las aplicaciones Java puedan trabajar con los drivers de bases de datos de cualquier fabricante. Por ejemplo, supongamos que hemos elaborado un programa que para el acceso a datos utiliza la familia de clases (driver o conector) del fabricante MySQL. Esto es perfecto, ya que la mayoría de clientes que compran nuestra aplicación utilizan este RDBMS. No obstante, sería deseable que si algún cliente utiliza otro RDBMS, por ejemplo Oracle, nuestra aplicación pudiera utilizar el driver de este fabricante sin que fuera necesario modificar nada de nuestro programa. En un fichero de propiedades podríamos tener una línea que indicara con qué RDBMS queremos trabajar. Este fichero se leería al cargarse la aplicación y serviría para que la propia aplicación utilizara el driver de uno o de otro. Por ejemplo:

rdbms.properties

```
mysql=com.mysql.jdbc.Driver
oracle=oracle.jdbc.OracleDriver
```

El API JDBC es un ejemplo perfecto del patrón Abstract Factory, puesto que hay un framework (los paquetes java.sql y javax.sql del JDK) que define una serie de interfaces y clases que son capaces de cargar una familia determinada de clases (un driver específico) que implementan los tipos definidos por el framework.

Para que el caso expuesto anteriormente sea posible tenemos que diseñar nuestra aplicación para que utilice las interfaces y clases de los paquetes java.sql y javax.sql, pero en ningún caso emplear las clases específicas incluidas en los drivers de los fabricantes. Otra condición indispensable es que las familias de clases deben implementar las mismas interfaces definidas por el framework.

De lo anterior se desprenden las siguientes consecuencias:

- El contenido de los paquetes java.sql y javax.sql define más que nada unas interdependencias entre tipos, pero a penas contiene implementación. Es lógico, pues no conoce a priori los objetos específicos que creará (MySQL?, Oracle?). Esto convierte a los paquetes del JDK en un API muy ligero.

- Nuestro programa no está acoplado en absoluto al driver de un fabricante de RDBMS y normalmente no costará esfuerzo alguno cambiar de una base de datos a otra (siempre y cuando no hayamos utilizado ninguna característica no estándar proporciona por el fabricante).

- El fabricante del driver puede realizar modificaciones en su código siempre que quiera, dado que ni los paquetes java.sql y javax.sql ni nuestra aplicación sostienen dependencias hacia sus tipos (clases).

Ejemplo 2 – Capa DAO

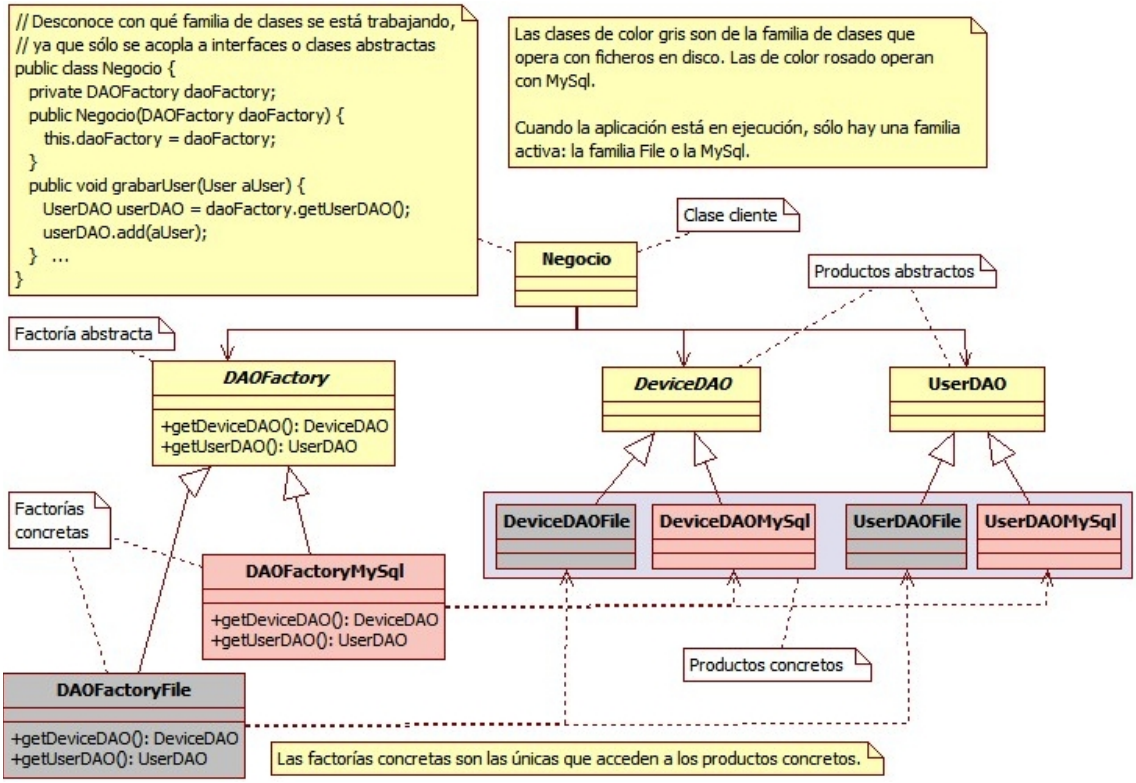
Imaginemos una aplicación web que permite realizar un mantenimiento de usuarios y de sus diferentes dispositivos (devices): ordenadores, impresoras, scanners, etc.

En esta aplicación, la parte web no utiliza ningún tipo de framework MVC, tan solo hace de uso del API Servlet. El diseño seguido consiste en el patrón Template Method (en su momento veremos en detalle este patrón), donde un servlet controlador abstracto define un algoritmo común (método service) y varios servlets concretos implementan la parte variable del algoritmo.

La capa de negocio es una simple clase de la cual se habría podido prescindir, ya que únicamente invoca a la capa de persistencia sin aportar nada. En una versión más sofisticada de la aplicación esta capa sería indispensable, pues implementaría las reglas de negocio, el control de las transacciones, etc.

La capa de acceso a datos es la más importante de todas desde el punto de vista del estudio del patrón AbstractFactory. Se ha implementado de manera que se puede utilizar como mecanismo de persistencia tanto MySql como ficheros en disco. Podemos indicar el mecanismo de persistencia a utilizar mediante un parámetro de inicio en el fichero web.xml.

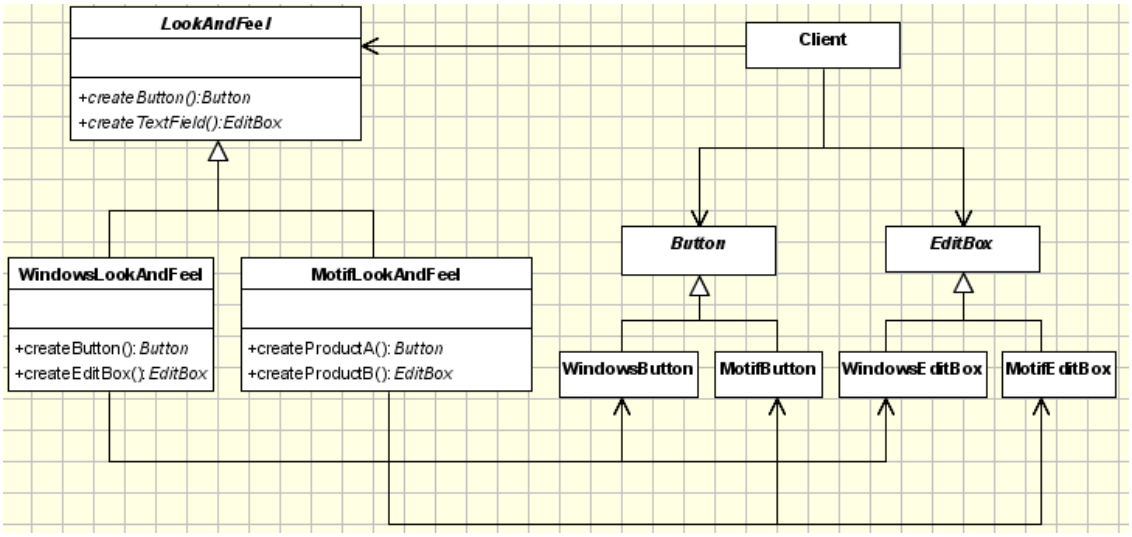
Veamos el diagrama de clases para la capa DAO:



Por su extensión, el código de esta aplicación se adjunta separadamente.

Ejemplo 3 – Look & Feel (No se proporciona código)

Otro ejemplo donde Abstract Factory es esencial es el del Look & Feel. Supongamos que un framework GUI tiene que dar soporte a varios temas gráficos como Motif y Window. Cada uno de estos estilos define su propio aspecto y comportamiento para los widgets gráficos (botones, barras de desplazamiento, cajas de texto, etc). Lógicamente, sería una locura que cada widget implementara en una estructura if-else "hardcoded" para contemplar cada look & feel soportado. Para evitar lo anterior se define una clase abstracta LookAndFeel, la cual instanciará -dependiendo de algún parámetro de configuración- una de las factorías concreta: WindowsLookAndFeel o MotifLookAndFeel. Cada petición que recibe LookAndFeel se delega a la factoría concreta activa, la cual retornará el control gráfico con el "sabor" adecuado a esa factoría.



Problemas específicos e implementación

El patrón Abstract Factory presenta beneficios e inconvenientes.

Beneficios

Las clases clientes no tienen que conocer el proceso de creación de objetos que necesitan. Se les ofrece únicamente la posibilidad de acceder a ellos a través de una interfaz, lo que hace más fácil su manipulación.

El intercambio o la adición de familias de productos es muy fácil, ya que la clase de una factoría concreta aparece en el código sólo cuando se crea una instancia (si utilizamos reflexión ni tan siquiera una vez).

Inconvenientes

La adición de nuevos productos a las factorías existentes es difícil, ya que la interfaz AbstractFactory utiliza un conjunto fijo de productos que se pueden crear. Es por eso que la adición de un nuevo producto implica ampliar la factoría abstracta y por ende, todas sus subclases.

Veamos esta sección las formas de implementar el patrón con el fin de evitar los problemas que puedan aparecer.

Factorías como Singletons

Una aplicación normalmente sólo necesita una instancia de la clase ConcreteFactory. Esto significa que lo mejor es implementar la factoría abstracta como un Singleton.

La creación de los productos

La clase AbstractFactory sólo declara la interfaz para la creación de los productos, ya que la implementación es responsabilidad de las clases ConcreteProductXX. Para cada familia, una buena opción es aplicar el patrón Factory Method (o Factory). De esta manera, una factoría concreta puede crear todos sus productos mediante la redefinición del método de factoría. Como recordatorio veamos de nuevo el código de la interfaz y de una de las factorías concretas. Notad que createProductA() y createProductB() son métodos de factoría:

```
public interface IFactoria {
	public AbstractProductA createProductA();
	public AbstractProductB createProductB();
}

class ConcreteFactory1 extends AbstractFactory {
	public AbstractProductA createProductA(){
		return new ProductA1("ProductA1");
	}
	public AbstractProductB createProductB(){
		return new ProductB1("ProductB1");
	}
}
```

Incluso ante una aplicación simple, esta técnica implica definir una factoría concreta por familia de productos, aunque la nueva subclase sea muy parecida a otras ya existentes.

Para simplificar el código y mejorar el rendimiento se puede utilizar el patrón Prototype (que aún no lo hemos visto) en lugar del Factory Method, sobre todo cuando hay muchas familias de productos. En este caso, la factoría concreta se inicia con un objeto prototipo de cada producto en la familia y cuando se solicite uno nuevo se devuelve uno que será copiado a partir del prototipo, en lugar de crearlo. Este enfoque elimina la necesidad de una nueva factoría concreta por cada nueva familia de productos.

Factoría que sabe qué familia utilizar en una ejecución puntual

En este mismo documento ver en página 3 la sección: *una implementación de Abstract Factory basada en una clase abstracta que implementa la interfaz IFactory*.

Patrones relacionados

Las clases de Abstract Factory normalmente se implementan como métodos de factoría (Factory Method), aunque también pueden implementarse usando el patrón Prototype.

Es habitual que una factoría concreta sea un Singleton. Debe serlo en un entorno multihilos.