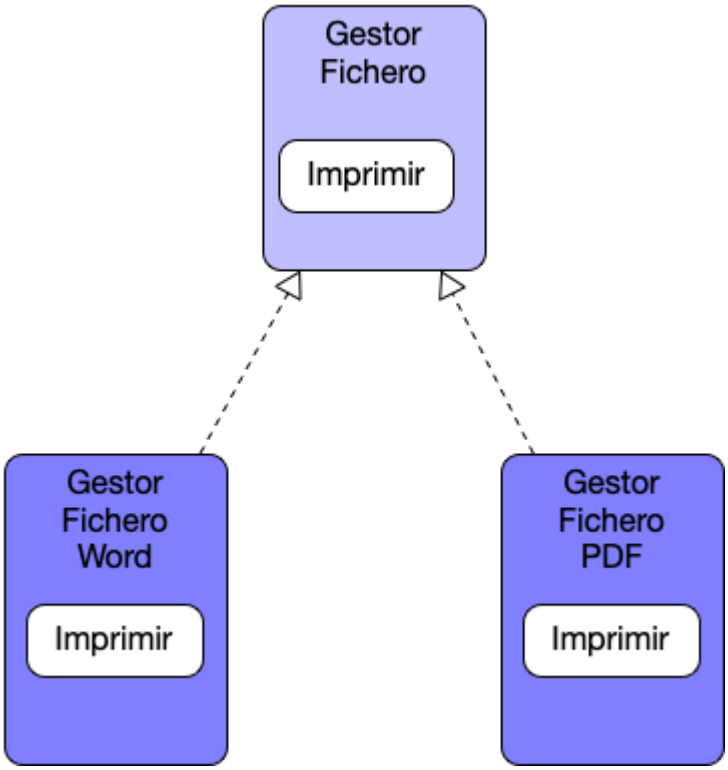
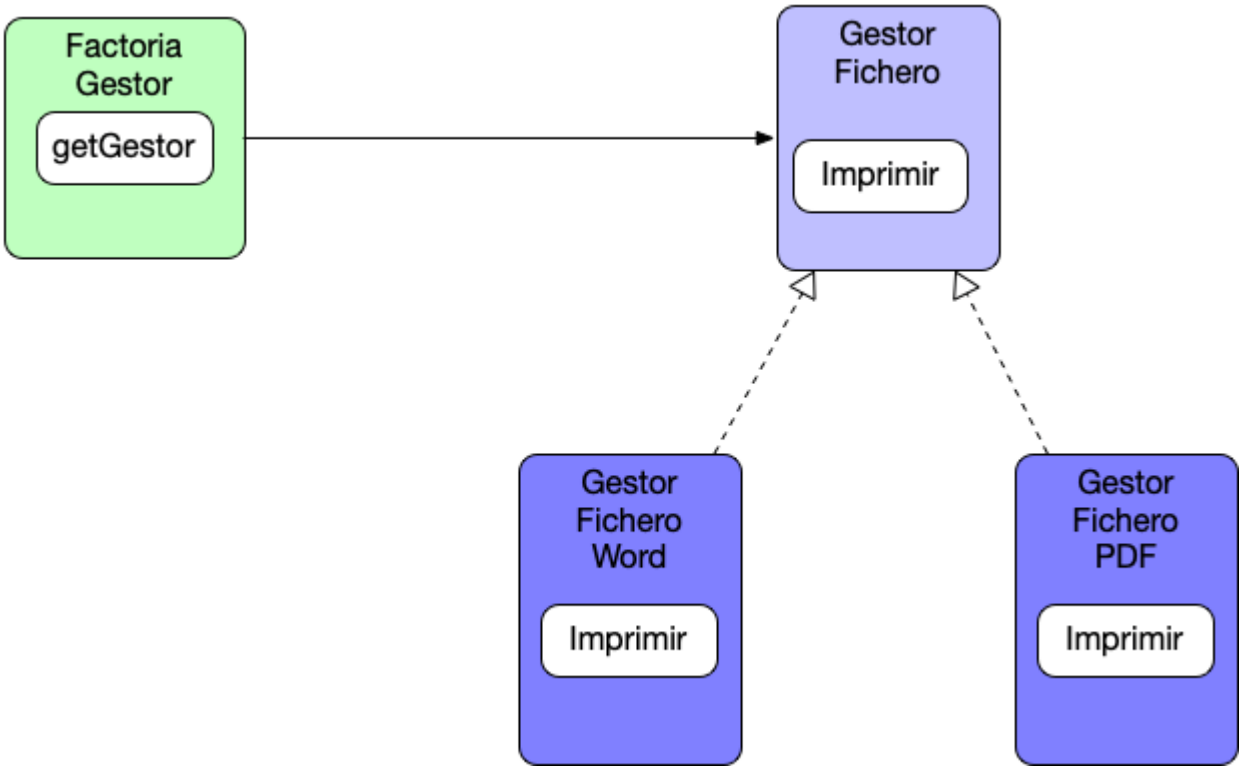
Patrones y Diseño . Muchas veces cuando hablo con desarrolladores y usan habitualmente patrones de diseño suelen considerar que el usar los patrones de diseño ya implica realizar un buen diseño . Hay situaciones sencillas que pueden ser así pero en muchas otras situaciones las cosas no son tan simples y se necesita que los desarrolladores ejerzan un mayor control sobre el software que están construyendo. Vamos a ver un ejemplo muy sencillo usando Factorias. Supongamos que queremos diseñar un sistema que gestione diferentes tipos de Ficheros podemos generar una Jerarquia de clases con GestorFichero GestorFicheroPDF y GestorFicheroWord.



Podemos diseñar una Factoria que se encargue de la generación de los diferentes tipos de GestoresFicheros:



Esto nos permite tener un mayor control sobre la construcción de los objetos . Sin embargo en varias ocasiones me han comentado que esta solución permite tanto instanciar los objetos con normalidad como a través de una factoria.

```
package com.arquitecturajava;

public class Principal {

	public static void main(String[] args) {
		GestorFichero gestor=
FactoriaGestorFichero.getGestor("f1", "pdf");
		gestor.imprimir();

		GestorFicheroPDF gestor2= new GestorFicheroPDF("f1");
		gestor2.imprimir();
	}
```

```
}
```

Esto puede parecer que no es un problema ya que nosotros para mejorar el nivel de abstracción vamos a recomendar siempre usar la factoria. Lamentablemente si es un problema porque la factoria se puede o no se puede usar es opcional. Hasta aquí lo que hemos usado es patrones de diseño “concepto de factoria” pero no hemos usado principios de diseño para diseñar mejor todo. Por ejemplo podriamos usar la encapsulación y obligar a que los gestores no fueran publicos y estuvieran con la factoria en otro package.

```
package com.arquitecturajava.gestores;

import com.arquitecturajava.GestorFichero;

public class FactoriaGestorFichero {

	public static GestorFichero getGestor(String ruta,String tipo)
{
		if (tipo.equals("pdf")) {
			return new GestorFicheroPDF(ruta);
		}else {
			return new GestorFicheroWord(ruta);
		}
	}
}
```

```
package com.arquitecturajava.gestores;

import com.arquitecturajava.GestorFichero;

class GestorFicheroPDF implements GestorFichero {

	private String fichero;
```

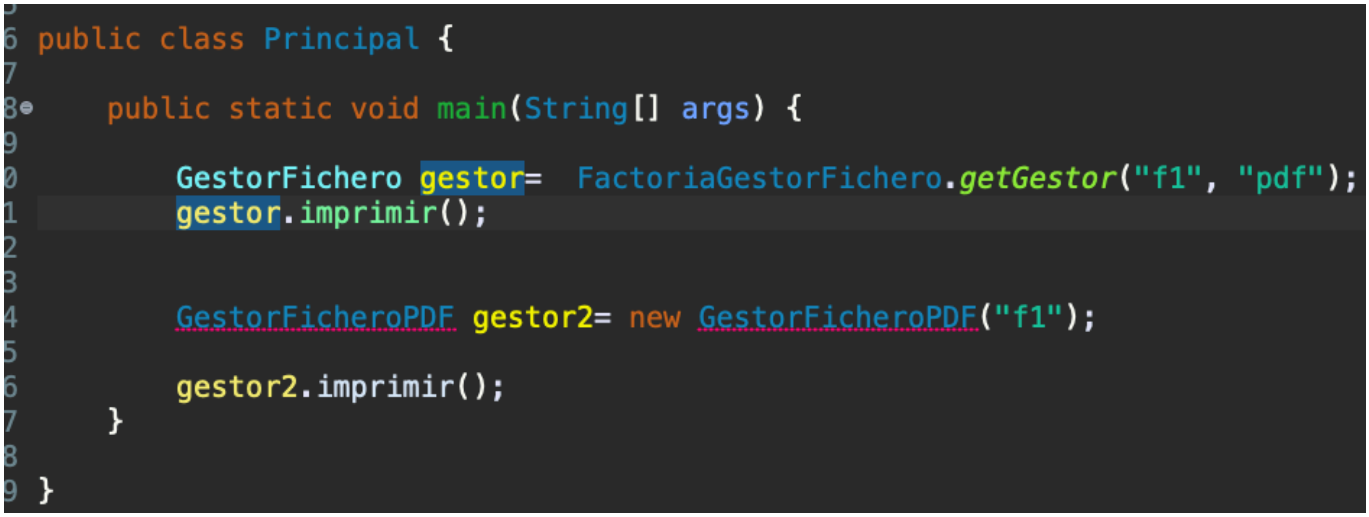
```
	public GestorFicheroPDF(String fichero) {
		super();
		this.fichero = fichero;
	}

	@Override
	public void imprimir() {
		System.out.println("imprimimos el fichero en pdf :" +
fichero);
	}

}
```

```
package com.arquitecturajava.gestores;

import com.arquitecturajava.GestorFichero;

 class GestorFicheroWord implements GestorFichero {

	private String fichero;
	public GestorFicheroWord(String fichero) {
		super();
		this.fichero = fichero;
	}


	@Override
	public void imprimir() {
		System.out.println("imprimimos el fichero en pdf :" +
fichero);
	}
```

}

Patrones y Diseño

De esta forma si intentamos instanciar una clase sin usar la Factoria las cosas simplemente no saldrán ya que hemos usado el principio de Encapsulación para aportar mayor control al programa .Ahora solo es posible instanciar algo usando la Factoria. Hemos combinado en una solución Patrones y Diseño



```
public class Principal {

    public static void main(String[] args) {

        GestorFichero gestor=  FactoriaGestorFichero.getGestor("f1", "pdf");
        gestor.imprimir();


        GestorFicheroPDF gestor2= new GestorFicheroPDF("f1");

        gestor2.imprimir();
    }

}
```

Otros artículos relacionados

- Utilizando Java Factories y Enums
- Java 8 Factory Pattern y su implementación
- Spring @ComponentScan y configuración
- Java Supplier Interface y Factories
- El patrón MVC , arquitectura cliente vs servidor