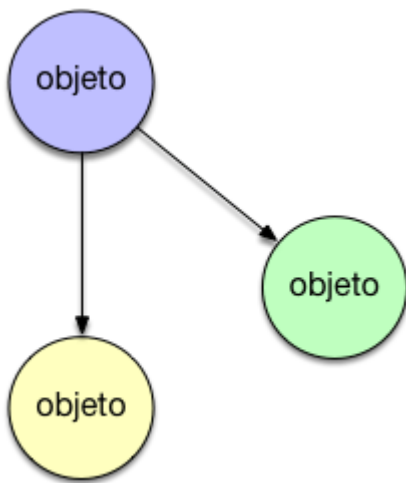


El patrón de inyección de dependencia siempre ha sido uno de los conceptos que cuesta entender en el mundo del desarrollo de software sobre todo a la gente que esta empezando. ¿Para qué sirve este patrón de diseño y cual es su utilizad? Normalmente cuando nosotros programamos en el día a día con la programación orientada a objeto nos encontramos construyendo objetos y relacionando objetos utilizando dependencias.



Por ejemplo podemos tener un programa principal que use un sencillo servicio de impresión para imprimir un documento.

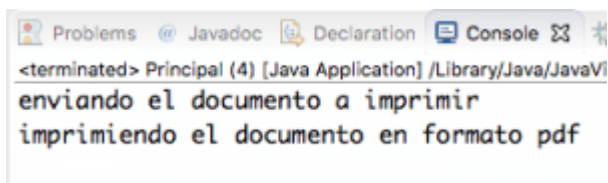
```
package com.arquitecturajava;  
  
public class ServicioImpresion {  
  
    public void imprimir() {  
        System.out.println("enviando el documento a  
imprimir");  
        System.out.println("imprimiendo el documento en  
formato pdf");  
    }  
}
```

```
    }  
}
```

Utilizamos un programa main e imprimimos:

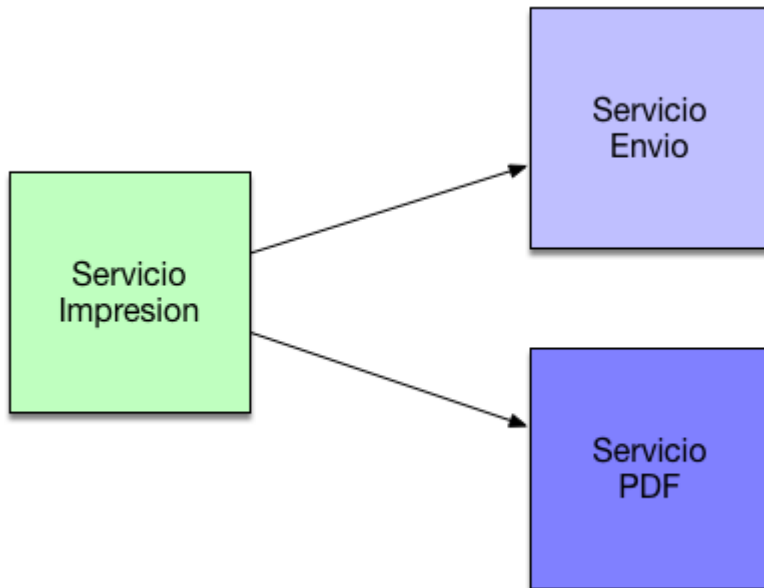
```
package com.arquitecturajava;  
  
public class Principal {  
  
    public static void main(String[] args) {  
        ServicioImpresion miServicio= new ServicioImpresion();  
        miServicio.imprimir();  
    }  
}
```

Hasta ahí no tiene nada de especial y veremos impreso el resultado en la consola:



## Inyeccion de Dependencia

Sin embargo lo lógico es que este programa divida un poco más sus responsabilidades y este compuesto de varios servicios algo como lo siguiente:



Hasta aquí todo es bastante razonable y lo que conseguimos es que nuestro servicio de impresión dependa de otros servicios y las responsabilidades queden mas claras. Acabamos de generar dependencias al Servicio de impresión . El código sería algo como lo siguiente.

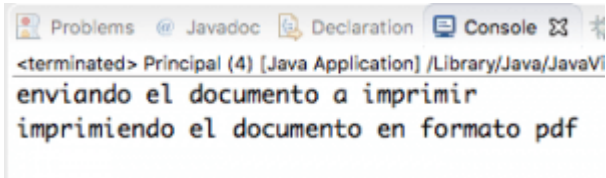
```
public class ServicioImpresion {  
  
    ServicioEnvio servicioA;  
    ServicioPDF servicioB;  
    public ServicioImpresion() {  
        this.servicioA= new ServicioEnvio();  
        this.servicioB= new ServicioPDF();  
    }  
    public void imprimir() {  
        servicioA.enviar();  
        servicioB.pdf();  
    }  
}
```

```
    }  
}
```

```
public class ServicioEnvio {  
  
    public void enviar() {  
        System.out.println("enviando el documento a  
imprimir");  
    }  
}
```

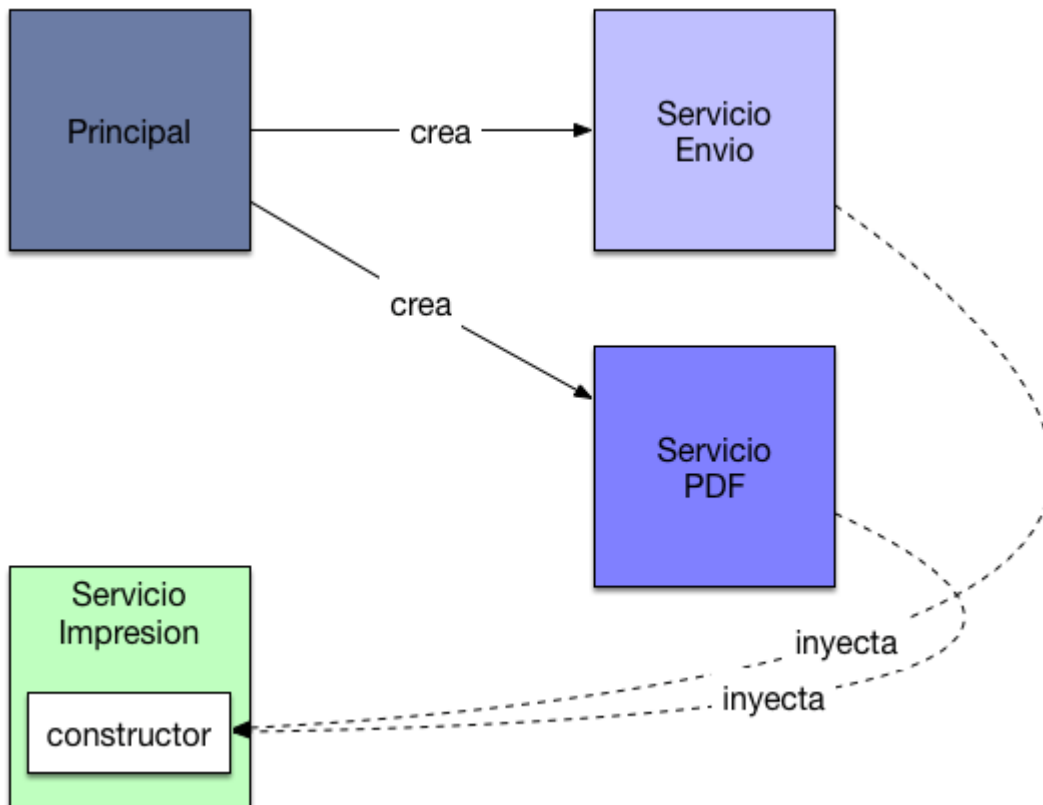
```
public class ServicioPDF {  
  
    public void pdf() {  
        System.out.println("imprimiendo el documento en  
formato pdf");  
    }  
}
```

El resultado en la consola será el mismo solo que hemos dividido mejor.



```
<terminated> Principal (4) [Java Application] /Library/Java/JavaVi
enviando el documento a imprimir
imprimiendo el documento en formato pdf
```

Se puede realizar la misma operación inyectando las dependencias al ServicioImpresión y que no sea él el que tenga que definir las en el constructor.



Este parece en principio un cambio sin importancia, el código quedaría:

```
package com.arquitecturajava.parte3;

public class ServicioImpresion {

    ServicioEnvio servicioA;
    ServicioPDF servicioB;
    public ServicioImpresion(ServicioEnvio servicioA,ServicioPDF
servicioB) {
        this.servicioA= servicioA;
        this.servicioB= servicioB;
    }
    public void imprimir() {
        servicioA.enviar();
        servicioB.pdf();
    }
}

public class Principal {

    public static void main(String[] args) {
        ServicioImpresion miServicio=
        new ServicioImpresion(new ServicioEnvio(),new ServicioPDF());
        miServicio.imprimir();

    }

}
```

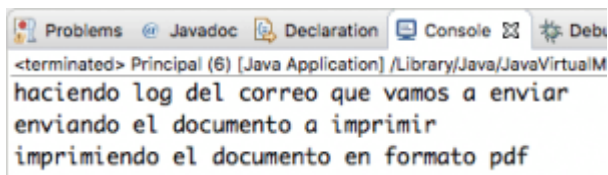
El resultado sigue siendo el mismo. Acabamos de inyectar las dependencias en el servicio desde nuestro programa main . ¿Qué ventajas aporta esto? . La realidad es que en principio parece que ninguna . Pero hay algo que ha cambiado ya no es el propio servicio el responsable de definir sus dependencias sino que lo es el programa principal. Esto abre las puertas a la extensibilidad. Es decir ¿tenemos nosotros siempre que inyectar las mismas dependencias al servicio?. La respuesta parece obvia... claro que sí están ya definidas. Sin embargo la respuesta es NO , nosotros podemos cambiar el tipo de dependencia que inyectamos , simplemente extendiendo una de nuestras clases y cambiando el comportamiento, vamos a verlo.

```
public class ServicioEnvioAspecto extends ServicioEnvio {  
  
    @Override  
    public void enviar() {  
        System.out.println("haciendo log del correo que vamos a  
enviar");  
        super.enviar();  
    }  
}
```

Acabamos de crear una clase que extiende ServicioEnvio y añade una funcionalidad adicional de log que hace un “log” del correo que enviamos . Ahora es tan sencillo como decirle al programa principal que cuando inyecte la dependencia no inyecte el ServicioEnvio sino el ServicioEnvioAspecto de esta forma habremos cambiado el comportamiento de forma considerable.

```
public class Principal {  
  
    public static void main(String[] args) {  
ServicioImpresion miServicio=  
new ServicioImpresion(new ServicioEnvioAspecto(),new ServicioPDF());  
        miServicio.imprimir();  
  
    }  
  
}
```

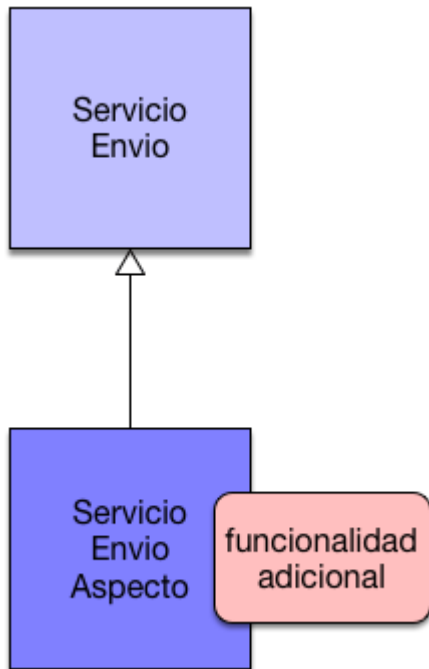
El resultado en la consola será:



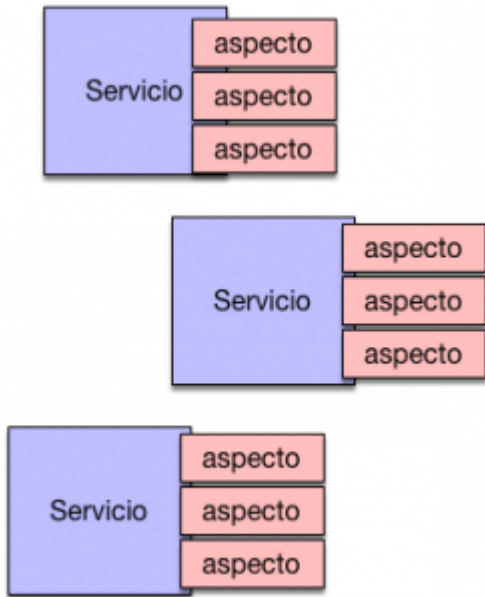
The screenshot shows a Java IDE's console window. The title bar includes tabs for Problems, Javadoc, Declaration, Console, and Debug Console. The console output shows the execution of the Principal class, which has terminated. The output text is: "haciendo log del correo que vamos a enviar", "enviando el documento a imprimir", and "imprimiendo el documento en formato pdf".

Acabamos de modificar el comportamiento de nuestro programa de forma significativa gracias al uso del concepto de inyección de dependencia.





La inyección de dependencia nos permite inyectar otras clases y añadir funcionalidad transversal a medida. Este patrón de diseño es el que abre la puerta a frameworks como Spring utilizando el concepto de inyección de dependencia de una forma más avanzada. En estos framework los aspectos que se añaden a nuestras clases son múltiples y la complejidad alta.



La importancia del patrón de inyección de dependencia es hoy clave en la mayoría de los frameworks.

## Otros artículos relacionados

1. [Spring @Qualifier utilizando @Autowired](#)
2. [Spring @import , organizando Spring framework](#)
3. [Java 8 Factory Pattern y su implementación](#)

## Artículos externos

1. [Dependency Injection](#)