Introducción a la inyección de dependencias

por Franco Bulgarelli

Versión 1.0

Mayo 2013

1 Contexto

En el mercado del desarrollo de software, particularmente aquel construido en torno al lenguaje de programación Java, mucho se habla de la inyección de dependencias (DI, *Dependency Injection*, por sus siglas en inglés), y para aportar a la confusión, de la inversión de control.

Entre la comunidad de ingenieros de software encontraremos defensores y detractores de este principio de diseño, pero, sin embargo, rara vez encontraremos una definición aceptada generalmente del mismo, o dos Ingenieros de Software que usen este término de forma consistente.

Incluso nos toparemos muchas veces con ciertos frameworks, llamados contenedores (ej: Spring), diseñados para soportar la misma, pero pocas veces sus usuarios realmente entienden su utilidad. O, lo que es más grave, con arquitectos de software que no pueden justificar su uso, pero sin embargo, los promueven.

Para cerrar, un rápido vistazo a otros lenguajes y plataformas de programación nos arroja que la inyección de dependencias es algo (o al menos, parece ser) inexistente.

Entonces, surgen algunas preguntas: ¿Qué es la inyección de dependencias? ¿Cómo se relaciona con el principio de inversión de control? ¿Es realmente necesaria? ¿Qué es un contenedor?

En las próximas páginas intentaremos arrojar un poco de luz sobre estas cuestiones.

2 Un poco de terminología

Antes de avanzar con el patrón, nos preguntamos ¿qué es una dependencia?

Cuando tenemos componentes (desde algo tan pequeño como un objeto hasta algo tan grande como un nodo de red, pasando por módulos, bibliotecas, etc), normalmente este componente no será capaz de resolver todas sus responsabilidades por sí solo, y deberá recurrir a otros componentes (¿Se acuerdan sobre qué es diseñar?).

Pues bien, a estos componentes que internamente necesita para cumplir sus responsabilidades los llamamos dependencias.

En el caso de objetos, nuestras dependencias son justamente otros objetos, a los que accede de alguna forma, típicamente a través de una variable de instancia o de una variable global.

Hay bibliografías que identifican a las partes involucradas en la DI como *clientes* y *servicios*, donde:

- Los *clientes* son los objetos que necesitan de otros objetos para delegar alguna responsabilidad o comportamiento.
- Los *servicios* son los objetos que llamamos *dependencias* más arriba.

Por ende, un *cliente* necesita *servicios* para realizar alguna de sus tareas.

3 Inyección de dependencias como patrón de diseño

3.1 Una primera aproximación

Para empezar, tomaremos como ejemplo el problema de las listas de correo, en su versión Scala. Tenemos la siguiente implementación parcial de la lista de correo:

```
clase ListaDeCorreo

  constructor(usuarios){ ... }

  metodo enviarMail(mail)
    usuarios.foreach(usuario => mailSender.enviarMail(usuario, mail))
```

La pregunta es: suponiendo que un objeto mailSender existe, ¿cómo llegamos a conocerlo? ¿Qué opciones tenemos? Algunas son:

1. Tener una única instancia de su clase (llamémosla `MailSenderPosta`), accesible globalmente
2. Obtener ese objeto a través de otro que lo provea
3. Parametrizar la clase para que reciba al mailSender

A continuación analizaremos cada una de estas alternativas

3.2 Accediendo a un Singleton

La primera alternativa consiste en tener una (única) instancia de `MailSenderPosta`, que pueda ser accedida desde cualquier lugar de la aplicación, es decir, un objeto global a la aplicación.

Según la tecnología esto es más o menos complejo. En Scala, en particular, esto es una primitiva del lenguaje; basta con declarar un **object** MailSender:

```
object MailSenderPosta extends MailSender {
   ...implementacion...
}
```

Y luego, referenciarlo como una variable global:

```
clase ListaDeCorreo

  constructor(usuarios){ ... }

  metodo enviarMail(mail)
    usuarios.foreach(usuario => MailSenderPosta.enviarMail(usuario, mail))
```

Esto funciona y es ciertamente muy sencillo. Pero, ¿qué problemas trae? Ahora nuestra `ListaDeCorreo` está fuertemente acoplada a nuestro `MailSenderPosta`, y como consecuencia de esto, no es posible probar a la lista de forma unitaria, independientemente del comportamiento del `MailSenderPosta`.

Peor aún, el `MailSenderPosta` probablemente tenga efectos muy notorios: efectivamente mandará un mail cada vez que se le envíe mensaje enviarMail, lo que significa que cada vez que corra una prueba automatizada, llenaremos con spam casillas de correos de muchas personas.

Moraleja: no es testeable. Feo, feo, feo....

3.3 Accediendo mediante un Service Locator

Otra alternativa que tenemos es delegar la provisión de tal dependencia en otro objeto que tome la responsabilidad de conocer todas las dependencias de mi sistema. Por ejemplo:

```
clase ListaDeCorreo
```

```
  constructor(usuarios){ ... }

  metodo enviarMail(mail)
    usuarios.foreach(usuario =>
      ServiceLocator.mailSender.enviarMail(usuario, mail)
    )
```

Es decir, nuestro ServiceLocator tendrá métodos para obtener cada dependencia del sistema. A veces también podemos encontrarlo de la siguiente forma (perdiendo algo de chequeo de tipos estático):

```
clase ListaDeCorreo

  constructor(usuarios){ ... }

  metodo enviarMail(mail)
    usuarios.foreach(usuario =>
      ServiceLocator.get("mailSender").enviarMail(usuario, mail)
    )
```

En cualquier caso, lo que hemos introducido es una indirección, que desacopla a la `ListaDeCorreo` del `MailSenderPosta`.

Ahora, el singleton es nuestro `ServiceLocator` y no cada una de las dependencias, es decir, nuestras clases de dominio no quedan acopladas a implementaciones concretas de sus dependencias, lo cual nos da un único punto de acceso para cambiar la configuración de la aplicación.

Nuestro ServiceLocator podría exponer métodos para configurarlo y registrar componentes, por ejemplo:

```
ServiceLocator.set("mailSender", new MailSenderPosta)
```

Es decir, el `ServiceLocator` es stateful (tiene estado mutable): acapara toda la configuración de la aplicación. Sin embargo, no es la idea que los componentes del sistema muten esta configuración dinámicamente: dado que un singleton se comporta como una variable global, tiene sus mismos problemas respecto a su mutación. Por el contrario, esta configuración será establecida únicamente al inicio de la aplicación, o al preparar el contexto de los tests.

Ahora, si necesitamos probar alguna funcionalidad utilizando *impostores (mocks/stubs)* para ciertos componentes en nuestros tests, podremos hacer lo siguiente:

```
val mailSenderMock = mock(MailSenderPosta.class)
```

```
ServiceLocator.set("mailSender", mailSenderMock)
val listaDeCorreo = new ListaDeCorreo(...)
listaDeCorreo.enviarMail(...)
(mailSenderMock.enviarMail(...)).verify(...)
```

¿Es una buena solución? Sin duda, y es particularmente útil cuando tenemos dependencias que son propias del contexto de la aplicación. Un ejemplo el motor de persistencia: es bastante probable que en todo mi sistema siempre se recurra al mismo motor, y sólo voy a querer cambiarlo cuando ejecuto la aplicación en otro entorno (desarrollo, testing, producción, etc).

La contra es que ahora **muchos de los componentes están acoplados al contexto de la aplicación**: todos deberán conocer a un ServiceLocator, que se termina convirtiendo en una gran bolsa de gatos, y trazar el grafo de dependencias de un objeto puede ser menos evidente.

El Service Locator, entonces, no es la mejor solución si las dependencias son más que propias del contexto en el que se está usando al componente.

3.4 Inyección de dependencias

Finalmente, llegamos a la solución más simple de todas: parametrizar aquello de lo que se depende. Es decir, dejar que las dependencias sean inyectadas (por un tercero) en el componente que las necesita

```
clase ListaDeCorreo

  constructor(usuarios, mailSender){ ... }

  metodo enviarMail(mail)
    usuarios.foreach(usuario =>
      mailSender.enviarMail(usuario, mail)
    )
```

¿Verdad que no es nada del otro mundo? En lugar de que sea el componente que necesita dependencias quien las vaya a pedir, ya sea directamente contra el singleton o indirectamente contra un service locator, alguien las inyecta, y el componente se las guarda.

Pateamos la pelota, delegando el problema en algún otro componente. Por eso se dice a veces que la inyección de dependencias es un caso particular del principio de inversión de control, plasmado en el lema de Hollywood: “No nos llames, nosotros te llamamos”

Como señala Fowler en su artículo sobre Inyección de Dependencias, hay varias formas de inyectar dichas dependencias: por setter, por constructor. ¿En qué cambia?

- La inyección por setter es muchas veces la opción más simple, y muchas

tecnologías fuerzan a utilizar esta estrategia. Sin embargo, es incompatible con construir objetos inmutables. Y además, la construcción del objeto abarca más que solo la instanciación.

```
listaDeCorreo = new ListaDeCorreo()
listaDeCorreo.mailSender = ...
listaDeCorreo.usuarios = ...
```

Es decir, se pueden instanciar objetos inconsistentes. Pasa a ser responsabilidad del programador no usar estos métodos que configuran al objeto luego del tiempo de construcción.

- Por el contrario, la inyección por constructor asegura que se creen objetos completos, consistentes, y es compatible con crear objetos inmutables.

Estas ideas las retomaremos cuando veamos patrones creacionales.

Ahora, puedo escribir mi test empleando otro MailSender, por ejemplo, un impostor.

```
mailSenderMock = mock(MailSenderPosta.class)
listaDeCorreo = new ListaDeCorreo(.., mailSenderMock)
listaDeCorreo.enviarMail(...)
(mailSenderMock.enviarMail(...)).verify(...)
```

¿Que problema tiene DI? A veces es tedioso y antinatural tener como atributos ciertos objetos y luego tener que proveer una forma de inyectarlos. Al revés del ServiceLocator, esto ocurre típicamente cuando tengo objetos que son puramente dependientes del contexto global de la aplicación.

Pero por otro lado, es una solución más flexible y que no introduce un acoplamiento con el contexto de la aplicación cuando las dependencias son propias del contexto en el que se usa el componente.

Es decir, ServiceLocator e Inyección de Dependencias en algún punto se complementan.

3.4.1 Qué no es la inyección de dependencias?

Luego de haber visto qué es la inyección de dependencias, podemos hablar con un poco más de conocimiento sobre el tema. Al parecer, no es un concepto tan complejo como parece. Pero, entonces, ¿por qué antes dijimos que cuesta encontrar una definición generalmente aceptada del término?

Tomemos el ejemplo del artículo de Stackoverflow más valorado sobre DI que existe a día de hoy. La respuesta más votada dice que la DI es pasar de obtener la referencia de un componente dentro de un constructor, a pasarlo por parámetro a ese mismo constructor, y listo! Tenemos DI.

Ahora, con lo que vimos hasta acá, les parece que esta respuesta está completa? Muchos desarrolladores opinarían que sí (lo podemos ver en la aceptación de la misma), pero nosotros ya vimos que no es así:

- Primero y principal, existe la inyección de dependencias por setter.
- Hacer DI no se reduce a pasar una dependencia por parámetro, sino a entender por qué esa dependencia tiene que pasarse por parámetro.
- Existen Singletons, ServiceLocators, y demás, que utilizamos para poder comparar distintas formas de obtener la referencia a un componente. Sin estos en mente, no podemos valorar realmente por qué sirve inyectar una dependencia.
- Si fuera tan fácil implementar DI, ¿por qué existen frameworks específicos (y, a veces, bastante complejos) para utilizarla?

Muchos de estos puntos son los que abren la discusión sobre qué es la DI y qué no.

Tal vez, el punto más importante de discusión esté relacionado al último item de la lista. Hay muchos desarrolladores que entienden que la DI no es un patrón de diseño, sino un patrón arquitectural. Es un punto de vista válido, pero para lo que necesitamos en la materia no nos interesa indagar tanto en este aspecto, así que lo mencionaremos brevemente.

4 Inyección de dependencias como patrón arquitectural

A veces la inyección de dependencias se usa como un patrón arquitectural de vinculación de componentes para toda la aplicación. No nos interesa tanto en diseño (esto es un BONUS) así que lo comentaremos brevemente.

El elemento fundamental es el contenedor de dependencias: es un componente arquitectural que es responsable de guardar el estado de la aplicación (similar al service locator) y de construir todos los componentes del sistema (describiendo su forma de instanciación y dependencias de forma declarativa).

Este componente es, en tanto, provisto por un framework de inyección de dependencias. Existen distintos tipos e implementaciones de contenedores; en el momento de escribir este apunte, los más comunes para la JVM son:

- Spring
- Guice

Además de una especificación formal que los estandariza.

Algunos de estos emplean XMLs para definir la configuración del contenedor. Pero esto es una particularidad de dichas implementaciones. Dicho de otra forma: ¡DI no tiene nada que ver con XMLs!

5 Alternativas a la inyección de dependencias

La inyección de dependencias no es la única forma de desacoplar dos componentes. Por ejemplo, el observer es un patrón de diseño que nos permite también reducir el acoplamiento a través de la metáfora de producción y captura de eventos. Sin embargo, es una solución más compleja.

Por otro lado, como señala Gilad Bracha en su artículo Lethal Injection, según la tecnología nos toparemos con otras alternativas para vincular a dos componentes. Mencionaremos dos ejemplos:

5.1 Scala: Cake Pattern

Se basa en el uso de mixins para introducir las dependencias. Cada mixin es en tanto responsable de construir la dependencia que provee, de forma similar a un Abstract Factory.

Empezamos por definir la interfaz del proveedor de la dependencia:

```
trait MailSenderComponent {
   def mailSender : MailSender
}
```

Esta interfaz (implementada mediante un `trait` de Scala) define la firma del mensaje para poder obtener obtener un `MailSender`.

Una posible implementación de esta interfaz es aquella que provee un `MailSenderPosta`:

```
class MailSenderComponentPosta extends MailSenderComponent {
   override val mailSender = new MailSenderPosta
}
```

Finalmente, definimos nuestra `ListaDeCorreo`. Esta vez, en lugar de parametrizar la lista de correos, lo que hacemos es indicar que depende de un mixin `MailSenderComponent`.

```
class ListaDeCorreo(usuarios: List[Usuario]) {
  self : MailSenderComponent =>
  def enviarMail(mail: Mail) =
    usuarios.foreach { usuario =>
      mailSender.enviarMail(usuario, mail)
    }
}
```

Entonces, ahora, para incluir esa dependencia, en lugar de inyectarla por constructor o setter, lo que hacemos es crear una nueva instancia de ListaDeCorreo, incluyendo el míxin MailSenderComponentPosta:

```
new ListaDeCorreo() with MailSenderComponentPosta
```

Soluciones análogas son válidas para lenguajes con Mixins, como Groovy o Ruby

5.2 Ruby: Open classes

Se basa en "pisar" los métodos, ya sea de una clase, o de una instancia concreta.
Asociado a la idea de partial mocking.

El siguiente ejemplo muestra como en Javascript se puede pisar el método de una clase.

```
class Mailer () {
  send() {
     // envio mail
  }
}

class ListaCorreo() {
  constructor(usuarios) {
    this.usuarios = usuarios
  }

  enviarMail(mail) {
    const mailer = new Mailer();
    usuarios.foreach(usuario =>
      mailer.enviarMail(usuario, mail)
    )
  }

}

var sent = false;
Mailer.prototype.send = function () {
  sent = true
}

new ListaDeCorreo([new Persona()])
console.log(sent) // => true
```

6 Conclusiones

- La inyección de dependencias es una técnica de diseño más que sirve para desacoplar componentes, consistente en parametrizar estos componentes de los que depende y proveer una forma de inyectarlos.
- Es una técnica muy simple, de ahí su poder. Aunque no siempre es la mejor.
- En otras tecnologías más allá de Java hay otras técnicas que la complementan.

- Puede ser pensada tanto como patrón de diseño arquitectural (introduciendo el concepto de Contenedor). Creemos que su mayor utilidad es como patrón de diseño.
- No tiene nada que ver con archivos de configuración en XML

Anexo: Componentes vs sistemas externos: cuando la inyección de dependencias no aplica

¿Pero, si tengo una biblioteca de fechas, tengo que usar inyección de dependencias?

Ver discusión acá:
https://groups.google.com/forum/#!topic/dds-jv-cursada/iBL2dASE6lM