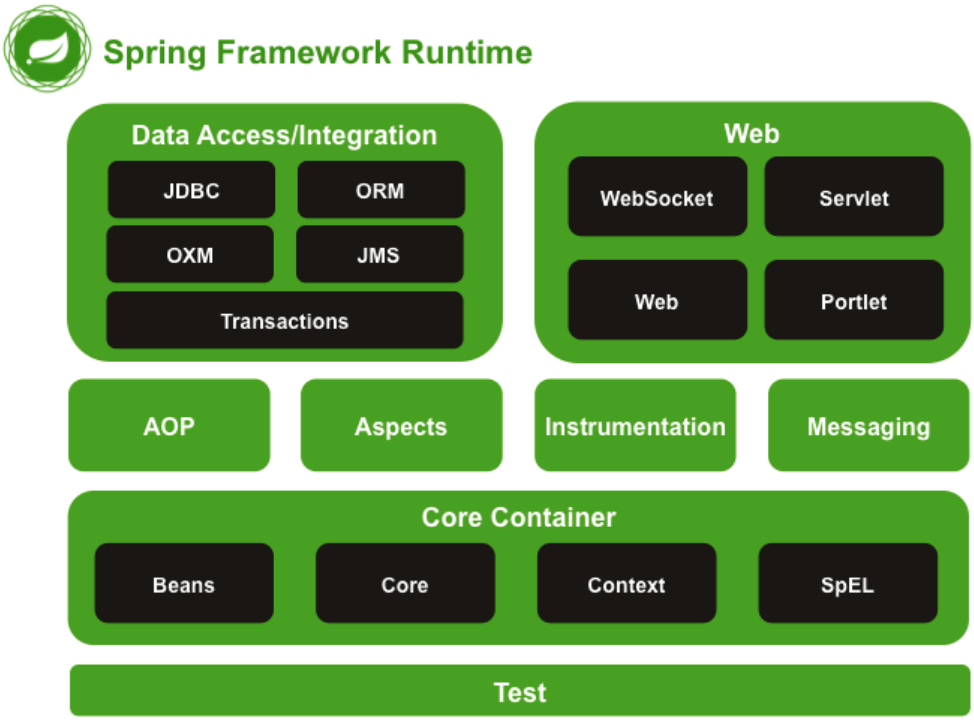
GUSTAVO CORONEL
DESARROLLA SOFTWARE

DESARROLLO WEB
CON SPRING BOOT

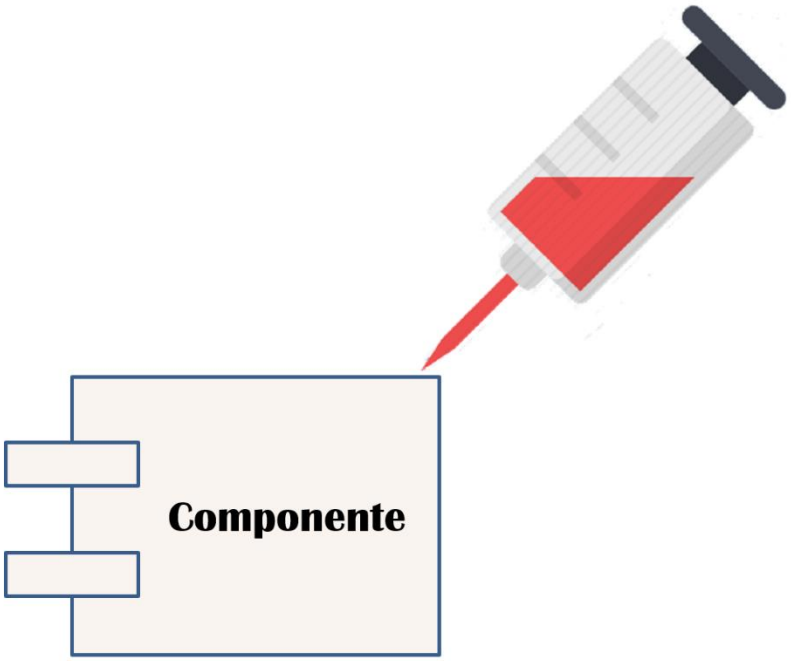


UNIDAD 01
PATRON INYECCION DE DEPENDENCIAS

Eric Gustavo Coronel Castillo
www.desarrollasoftware.com
gcoronelc@gmail.com
I N S T R U C T O R

CONTENIDO

INTRODUCCIÓN



DI, corresponde a las siglas de **Dependency Injection**, es un patrón de diseño pensado en permitir un menor acoplamiento entre componentes de una aplicación y fomentar así la reutilización de los mismos.

La principal ventaja de usar el patrón **DI** es la reducción del acoplamiento entre una clase y las clases de las cuales depende.

Una dependencia entre un componente y otro, puede establecerse estáticamente o en tiempo de compilación, o bien, dinámicamente o en tiempo de ejecución. Es en este último escenario es donde cabe el concepto de inyección, y para que esto pueda ser posible, debemos referenciar interfaces y no directamente las implementaciones.

En general, las dependencias son expresadas en términos de interfaces en lugar de clases concretas. Esto permite un rápido reemplazo de las implementaciones dependientes sin modificar el código fuente de la clase.

Lo que propone entonces la **Inyección de Dependencias**, es no instanciar las dependencias explícitamente en su clase, sino que declarativamente expresarlas en la definición de la clase. La esencia de la inyección de dependencias es contar con un componente capaz de obtener instancias validas de las dependencias del objeto y pasárselas durante la creación o inicialización del objeto.

EL PROBLEMA

Como todo patrón, comienza planteando un **problema** para el que plantea una solución. Muchas veces, un componente tiene dependencias de servicios o componentes cuyos tipos concretos son especificados en tiempo de diseño.

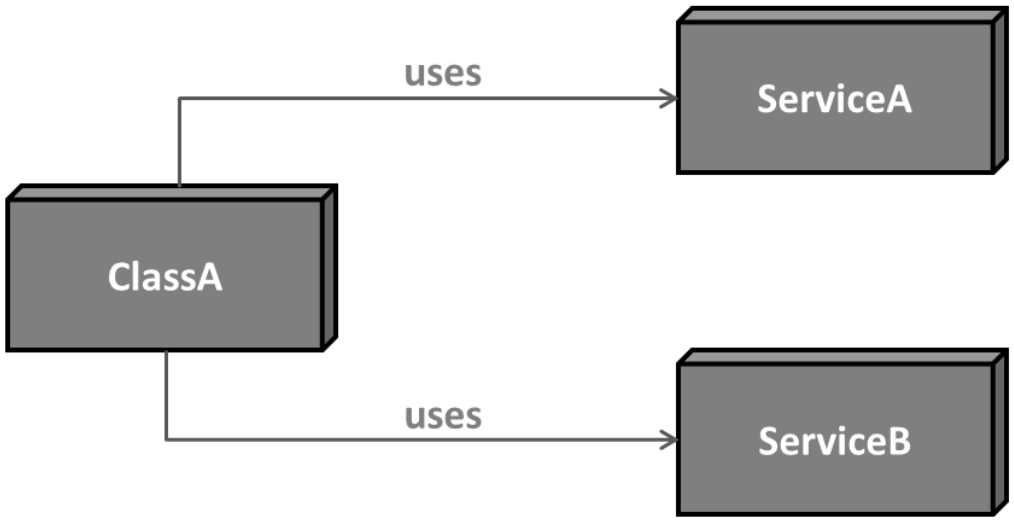


Figura 1

Para el caso de la Figura 1, **ClassA** depende de **ServiceA** y **ServiceB**.

Dentro de ClaseA normalmente tenemos el siguiente script:

```
ServiceA serviceA = new ServiceA();
```

Los problemas que esto plantea son:

- Al reemplazar o actualizar las dependencias, se necesita cambiar el código fuente de **ClassA**.
- Las implementaciones concretas de las dependencias tienen que estar disponibles en tiempo de compilación.
- Las clases son difíciles de testear aisladamente porque tienen directas definiciones a sus dependencias. Esto significa que las dependencias no pueden ser reemplazadas por componentes stubs o mocks.
- Las clases tienen código repetido para crear, localizar y gestionar sus dependencias.

SOLUCIÓN

La **solución** pasa por delegar la función de seleccionar una implementación concreta de las dependencias a un **componente externo**.

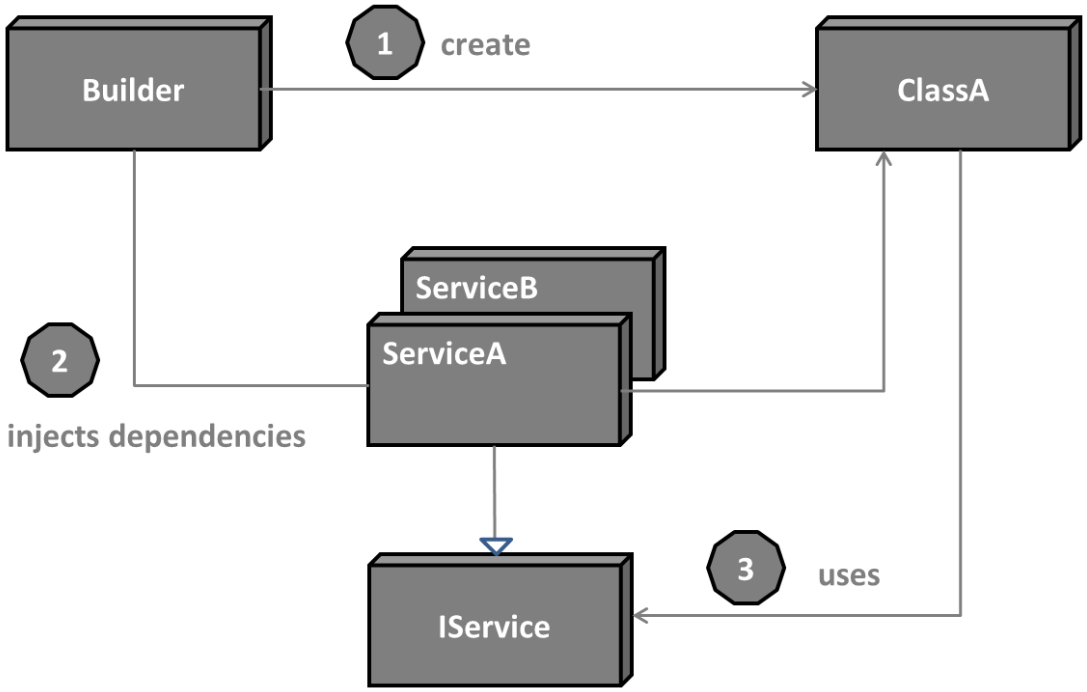
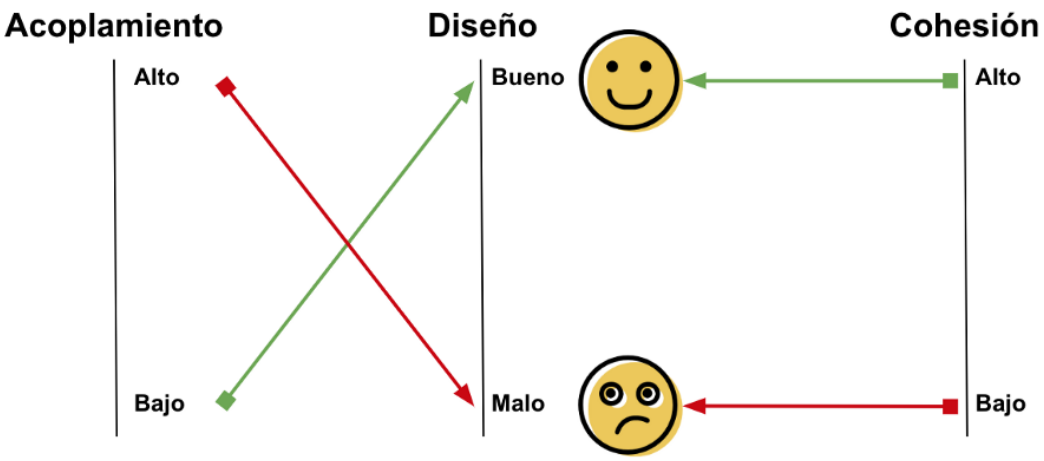


Figura 2

Con respecto a la Figura 2, el control de cómo un objeto A (De tipo ClassA) obtiene la referencia de un objeto B (De tipo ServiceB) es invertido. El objeto A no es responsable de obtener una referencia al objeto B, sino que es el **componente externo** (Builber) el responsable de esto. Esta es la base del patrón DI.

El patrón DI aplica un principio de diseño denominado **Principio de Hollywood** (*No nos llames, nosotros te llamaremos*).

USOS



El patrón DI se puede utilizar cuando:

- Se desee desacoplar las clases de sus dependencias de tal manera que las mismas puedan ser reemplazadas o actualizadas con muy pocos o casi ningún cambio en el código fuente de sus clases. Un caso ilustrativo es cuando en la empresa deciden cambiar de motor de base de datos, o quizás en la sucursales se utilizará un motor de base de datos diferente de menor costo de licenciamiento.
- Desea escribir clases que dependan de otras clases cuyas implementaciones no son conocidas en tiempo de compilación.
- Desea testear las clases aisladamente sin sus dependencias.
- Desea desacoplar las clases de ser responsables de localizar y gestionar el tiempo de vida de sus dependencias.

TÉCNICAS DE IMPLEMENTACIÓN

Existen tres maneras de implementar la inyección de dependencias:

1. Inyección basada en métodos setters
2. Inyección basada en constructor
3. Inyección basada en interfaces

Inyección basada en métodos setters

En este tipo de DI, se utiliza un método setters para inyectar una dependencia en el objeto que la requiere. Se invoca así al setter de cada dependencia y se le pasa como parámetro una referencia a la misma.

Considerando un diseño de dos capas donde tenemos una capa **service** donde se implementa los componentes de negocio, y una capa DAO donde se implementa los componentes de persistencia, se tiene una clase **VentaService** y una interface **VentaDaoApi**, **VentaService** depende de **VentaDaoApi** para operar correctamente. El diagrama lo tenemos en la Figura 3

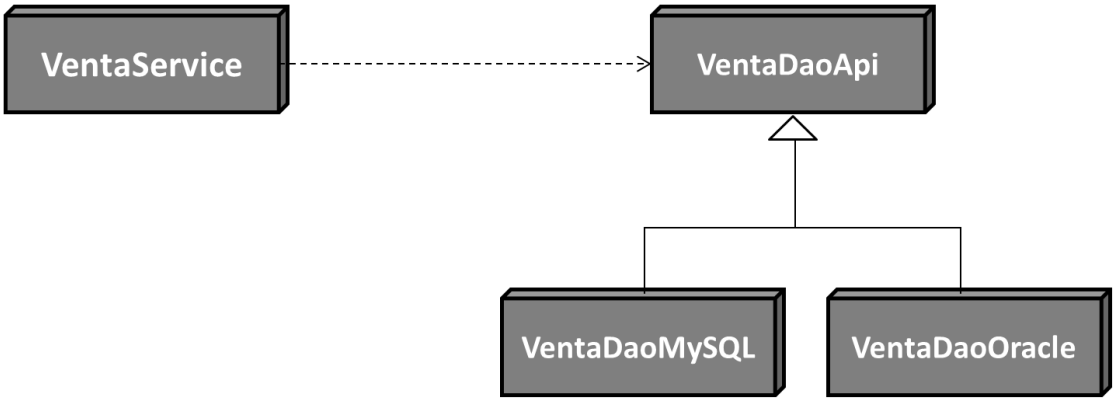


Figura 3

VentaDaoApi es una interface que tiene dos implementaciones, la clase **VentaService** depende de la interface.

El código de los componentes de persistencia sería así:

```
interface VentaDaoApi{

   // Some code
}


class VentaDaoMySQL implements VentaDaoApi{

   // Some code
}

class VentaDaoOracle implements VentaDaoApi{

   // Some code
}
```

La clase VentaService tiene implementado el método set para hacer la inyección del objeto respectivo:

```
public class VentaService{

   private VentaDaoApi ventaDao;

   public void setVentaDao(VentaDaoApi ventaDao){
      this.ventaDao = ventaDao;
   }

   // Some code
}
```

El objeto responsable de las dependencias realizará la inyección de la siguiente forma:

```
   VentaDaoApi ventaDao = new VentaDaoOracle();
   VentaService VentaService = new VentaService();
   ventaService.setVentaDao(ventaDao);
```

La ventaja aquí es que uno puede cambiar la dependencia entre VentaService y la implementación de VentaDaoApi luego de haber instanciado la clase VentaService.

La desventaja es que un objeto con setters no puede ser inmutable y suele ser complicado determinar cuáles son las dependencias que se necesitan y en qué momento se las

necesita. Se recomienda así utilizar DI basada en constructor a menos que necesite cambiar la dependencia y sepa claramente los momentos en los cuales realizar estos cambios.

Otra desventaja es que al utilizar setters (necesarios para la inyección), estamos exponiendo las propiedades de un objeto al mundo exterior cuando en realidad no era un requerimiento de negocio hacerlo.

Inyección basada en constructor

Las dependencias se inyectan utilizando un constructor con parámetros del objeto dependiente. Este constructor recibe las dependencias como parámetros y las establece en los atributos del objeto.

Considerando el caso de la Figura 3, en la inyección basada en un constructor, se creará una instancia de VentaService usando un constructor con parámetros al cual se le pasará una referencia a un objeto de una clase que implemente **VentaDaoApi** para poder inyectar la dependencia.

```
public class VentaService{

   private VentaDaoApi ventaDao;

   public VentaService(VentaDaoApi ventaDao){
      this.ventaDao = ventaDao;
   }

   // Some code
}
```

El objeto responsable de las dependencias realizará la inyección de la siguiente forma:

```
   VentaDaoApi ventaDao = new VentaDaoOracle();
   VentaService ventaService = new VentaService(ventaDao);
```

La principal desventaja de la DI basada en constructor es que una vez que la clase BusinessFacade es instanciada no podemos cambiar las dependencias inyectadas.

Inyección basada en interfaces

Aquí se utiliza una interfaz común que otras clases la implementen para poderles luego inyectar dependencias.

En el siguiente ejemplo, seguimos con el caso de la Figura 3, a toda clase que implemente **VentaServiceApi** se le podrá inyectar cualquier objeto que implemente **VentaDaoApi** mediante el método **injectVentaDao**.

```
interface VentaServiceApi{
   void injectVentaDao(VentaDaoApi ventaDao);
}

class VentaService implements VentaServiceApi{

   private VentaDaoApi ventaDao;

   public void injectVentaDao (VentaDaoApi ventaDao){
      this.ventaDao = ventaDao;
   }

   // Some code
}
```

El objeto responsable de las dependencias realizará la inyección de la siguiente forma:

```
   VentaDaoApi ventaDao = new VentaDaoOracle();
   VentaService ventaService = new VentaService();
   ventaService.injectVentaDao(ventaDao);
```

Comentarios

Las tres formas de inyección presentadas, pasan una referencia a una implementación de **VentaDaoApi**. Lo que busca finalmente este patrón es programar contra interfaces para tener un menor acoplamiento.

Cuando el acoplamiento se realiza a una interfaz (Interface), se puede utilizar cualquier implementación con un mínimo cambio.

APLICABILIDAD

Cuando utilizar DI

La inyección de dependencias no debería usarse siempre que una clase dependa de otra, sino más bien es efectiva en situaciones específicas como las siguientes:

- Inyectar datos de configuración en un componente.
- Inyectar la misma dependencia en varios componentes.
- Inyectar diferentes implementaciones de la misma dependencia.
- Inyectar la misma implementación en varias configuraciones
- Se necesitan alguno de los servicios provistos por un contenedor.

La DI no es necesaria si uno va a utilizar siempre la misma implementación de una dependencia o la misma configuración, o al menos, no reportará grandes beneficios en estos casos.

Contenedores

Para implementar el patrón DI en alguna de sus variantes, existen los denominados contenedores de inyección de dependencias (**DIC**), que son los componentes encargados de instanciar las dependencias y realizar las inyecciones necesarias entre todos los objetos y además **gestionar sus respectivos ciclos de vida**.

En cuanto a la instanciación e inyección, un **DIC** se configura para especificarle que dependencias debe instanciar y donde deberá luego inyectarlas. Además, para la instanciación se debe definir el modo de instanciación, es decir, si se creará una nueva instancia siempre que se lo requiera, o se reusará la instancia creada (**singleton**).

En cuanto a la gestión de los ciclos de vida de los objetos creados, implica que son capaces de administrar las diferentes etapas de la vida del componente (instanciación, configuración, eliminación).

El hecho de que el contenedor a veces mantenga una referencia a los componentes creados luego de la instanciación es lo que lo hace un contenedor.

No todos los objetos deben ser gestionados. El contenedor mantiene una referencia a aquellos objetos que son reusados para futuras inyecciones, como singletons. Cuando configuramos el contenedor para que siempre cree nuevas instancias, entonces éste se suele olvidar de dichos objetos creados, dejando la tarea al garbage collection para recolectarlos luego de que dejen de ser usados.

Existen varios DIC, según el lenguaje de programación que soportan, que modos de DI que soportan, etc. Para Java el más conocido es Spring Frameworks, pero existen otros como Butterfly Container, Pico Container, Google Guice, y otros.

CONCLUSIÓN

La natural mutabilidad de las partes o módulos de un sistema, hace que el desarrollar software con mínimo acoplamiento entre sus componentes sea una necesidad mas que un mero requerimiento.

La popularidad y efectividad del patrón DI en cualquiera de sus formas (Service Locator, Inyección de dependencias, etc.) se observa en la diversidad de frameworks y contenedores disponibles en diversos lenguajes.

En estos contextos, la aplicación del patrón DI es una de las primeras decisiones que deben tomarse en el diseño de la arquitectura de un producto software.