Contenido

[Inyección de dependencias 2](#_Toc359793403)

[Implementación del patrón en Java 2](#_Toc359793404)

[Ilustración de código usando Java 3](#_Toc359793405)

[Qué es Inyección de dependencias 4](#_Toc359793406)

[Inyección de dependencias. ¿Qué es y para qué sirve? 10](#_Toc359793407)

[Inversión de control (Wikipedia) 13](#_Toc359793408)

[Historia 13](#_Toc359793409)

[Definición 13](#_Toc359793410)

[Inversión de Control (IoC) 14](#_Toc359793411)

# Inyección de dependencias

A la hora de diseñar o desarrollar software, uno de los principios básicos que debemos seguir es: tratar de depender de abstracciones y no de objetos o clases concretas.

En [Informática](http://es.wikipedia.org/wiki/Inform%C3%A1tica), **Inyección de Dependencias** (en inglés *Dependency Injection*, DI) es un [patrón de diseño](http://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o) orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase quien cree el objeto. El término fue acuñado por primera vez por [Martin Fowler](http://es.wikipedia.org/w/index.php?title=Martin_Fowler&action=edit&redlink=1).

En los comienzos de la programación, los programas eran lineales y monolíticos. El flujo de ejecución era simple y predecible, ejecutándose línea tras línea.

Aparecieron dos conceptos para estructurar el código: la **modularidad** y la **reutilización** de los componentes: se crean bibliotecas de componentes reutilizables. El flujo se complica, saltando de componente a componente, y aparece un nuevo problema: la **dependencia** (acoplamiento) entre los componentes.

El problema de la dependencia se empieza a considerar lo suficientemente importante como para definir nuevos conceptos en el diseño:

* [Inversión de control](http://es.wikipedia.org/wiki/Inversi%C3%B3n_de_control) (IoC, por sus siglas en inglés)
* Inyección de Dependencias (Dependency Injection, DI) que es una forma de inversión de control.

Implementación del patrón en Java

* La forma habitual de implementar este patrón es mediante un "**Contenedor DI**" y objetos planos o simples por ejemplo los llamados [**POJO**](http://es.wikipedia.org/wiki/POJO) en java. El contenedor inyecta a cada objeto los objetos necesarios según las relaciones plasmadas en un fichero de configuración.
* Típicamente este contenedor es implementado por un framework externo a la aplicación (como [Spring](http://es.wikipedia.org/wiki/Spring_Framework) o uno propietario), por lo cual en la aplicación también se utilizará [inversión de control](http://es.wikipedia.org/wiki/Inversi%C3%B3n_de_control) al ser el contenedor (almacenado en una biblioteca) quien invoque el código de la aplicación. Ésta es la razón por la que los términos de *inversión de control* e *inyección de dependencias* se confunden habitualmente entre sí.

Ilustración de código usando Java

El siguiente ejemplo muestra una implementación sin inyección de dependencias.

**public** **class** Vehiculo {

**private** Motor motor = **new** Motor();

***/\*\* @retorna la velocidad del vehículo\*/***

**public** Double enAceleracionDePedal(**int** presionDePedal) {

motor.setPresionDePedal(presionDePedal);

**int** torque = motor.getTorque();

Double velocidad = ... *//realiza el cálculo*

**return** velocidad;

}

}

*//se omite la clase Motor ya que no es relevante para este ejemplo*

La implementación de arriba necesita crear una instancia de Motor para calcular su velocidad. El siguiente ejemplo sencillo muestra una implementación usando inyección de dependencias.

**public** **class** Vehiculo {

**private** Motor motor = **null**;

**public** setMotor(Motor motor){

**this**.motor = motor;

}

***/\*\* @retorna la velocidad del vehículo\*/***

**public** Double enAceleracionDePedal(**int** presionDePedal) {

Double velocidad = **null**;

**if** (**null** != motor){

motor.setPresionDePedal(presionDePedal);

**int** torque = motor.getTorque();

velocidad = ... *//realiza el cálculo*

}

**return** velocidad;

}

}

*//se omite la clase Motor ya que no es relevante para este ejemplo*

**public** **class** VehiculoFactory {

**public** Vehiculo construyeVehiculo() {

Vehiculo vehiculo = **new** Vehiculo();

Motor motor = **new** Motor();

vehiculo.setMotor(motor);

**return** vehiculo;

}

}

En este ejemplo VehiculoFactory representa al proveedor. Es una aplicación sencilla del patrón de diseño fábrica que hace posible que la clase Vehículo no requiera saber cómo obtener un motor por sí misma, sino que es la responsabilidad de VehiculoFactory.

# Qué es Inyección de dependencias

<http://msdn.microsoft.com/es-es/library/jj635998.aspx>

La inyección de dependencias o DI por sus siglas en inglés, es una herramienta comúnmente utilizada en varios patrones de diseño orientado a objetos, consiste en inyectar comportamientos a componentes.

Esto no es más que extraer responsabilidades a un componente para delegarlas en otro, estableciendo un mecanismo a través del cual el nuevo componente pueda ser cambiado en tiempo de ejecución. Es conveniente no confundir Inyección de dependencias (DI) con Inversión de Control (IoC), error muy comúnmente cometido que figura especialmente en la web. IoC es un tema para un próximo artículo.

**Ejemplo y Explicación detallada**

Imaginemos que estamos haciendo un videojuego en cual tenemos un personaje que es un robot, este robot puede realizar acciones de diferentes maneras, por lo cual su cabeza puede ser conectada a infinidad de cuerpos distintos.

Las soluciones tradicionales desde el punto de vista de POO son diversas, ejemplo:

* Una clase base Robot con un atributo cuerpo, crear varias clases asignando un atributo Cuerpo diferente en el constructor
* Una clase Robot con un atributo enumeración que le permita cambiar de cuerpo
* Muchas clases cuerpo que heredan de una clase Robot

Sin embargo al final la responsabilidad del cuerpo y de la cabeza sigue siendo confusa, otro de mis principios favoritos de Diseño Orientado a Objetos es de “Single Responsability” o mejor

“Un objeto una responsabilidad”

Claramente las soluciones clásicas al problema planteado no ofrecen esa opción.

¿Qué hacer?

Inyección de dependencias también puede llamarse inyección de comportamientos, en todo caso es un nombre muy profesional y estilizado, lo cual puede ser intimidante, pero es un nombre preciso. Que sucede cuando te inyectan algo, ¿una vacuna por ejemplo? Depositan nuevos componentes en tu cuerpo, que pueden modificar el funcionamiento, sin que esto implique que vuelvas a nacer.

De eso mismo se trata la Inyección de Dependencias, colocar dentro de un objeto otros que puedan cambiar su comportamiento, sin que esto implique volver a crear el objeto.

Esto nos permite tener un objeto que puede hacer un conjunto de tareas, cada una de esas tareas es una responsabilidad, que puede ser ejecutada por otro objeto únicamente especialista y dedicado a ello [una responsabilidad], pero ahora tenemos otro diferenciador, El objeto responsable de ejecutar esa única tarea

Puede establecerse en tiempo de ejecución

La implementación habitual en programación es crear un método capaz de establecer el comportamiento, es decir capaz de cambiar el valor de un atributo asignandole una instancia de objeto diferente.

Esta sería una posible implementación de Robot que no utiliza Inyección de Dependencias

**C#**

public class Robot

{

public Robot() { }

public void Caminar()

{

//Caminando Ando...

/\*Paso 1

\*Paso 2

\*Paso 3

\*Paso N\*/

}

/\*Hacer otras cosas cool que hacen los robots\*/

public void Disparar();

public void Volar();

public void BuscarAJhonConnor()

{

/\*Paso 1 buscar en internet

\*Paso 2 correr

\*Paso 3 caminar para ahorrar bateria

\*Paso N\*/

}

}

Ahora cambiemos la implementación por una que si haga Inyección de dependencias, primero creamos la clase Body la cual será la responsable de ejecutar las acciones del cuerpo:

C#

public class Body

{

public void Caminar()

{

//Caminando ando...

}

public void Volar()

{

//Volando ando...

}

public void Disparar()

{

//Volando ando...

}

}

Seguidamente creamos un Robot que camina por medio de un Body, teniendo atención en delegar las responsabilidades a dicho objeto:

C#

public class Robot

{

public Body \_body;

public Robot() { }

public void Caminar()

{

if(\_body != null)

\_body.Caminar();

}

public void Volar()

{

if (\_body != null)

\_body.Volar();

}

public void Disparar()

{

if (\_body != null)

\_body.Disparar();

}

/\*Hacer otras cosas cool que hacen los robots\*/

public void BuscarAJhonConnor()

{

/\*Paso 1 buscar en internet

\*Paso 2 \_body.Caminar();

\*Paso 3 \_body.Disparar();

\*Paso N\*/

}

}

Como nos podemos dar cuenta el Robot hace todo lo que hacía antes aunque no tiene asignado aún un body, así que establecemos un mecanismo para hacer esto:

C#

public class Robot

{

public Body \_body;

//Inyectar un cuerpo al Robot

public void SetBody(Body newBody)

{

\_body = newBody;

}

public Robot() { }

public void Caminar()

{

if(\_body != null)

\_body.Caminar();

}

public void Volar()

{

if (\_body != null)

\_body.Volar();

}

public void Disparar()

{

if (\_body != null)

\_body.Disparar();

}

/\*Hacer otras cosas cool que hacen los robots\*/

public void BuscarAJhonConnor()

{

/\*Paso 1 buscar en internet

\*Paso 2 \_body.Caminar();

\*Paso 3 \_body.Disparar();

\*Paso N\*/

}

}

Bueno eso fue muy ‘java’, pongámonos serios y volvámoslo más elegante, fácil de leer y de mantener, más C#. Hagamos uso de propiedades y deshagámonos del private storage

C#

public class Robot

{

//Inyectar un cuerpo al Robot

public Body Cuerpo { get; set; }

public Robot() { }

public void Caminar()

{

if (Cuerpo != null)

Cuerpo.Caminar();

}

public void Volar()

{

if (Cuerpo != null)

Cuerpo.Volar();

}

public void Disparar()

{

if (Cuerpo != null)

Cuerpo.Disparar();

}

/\*Hacer otras cosas cool que hacen los robots\*/

public void BuscarAJhonConnor()

{

/\*Paso 1 buscar en internet

\*Paso 2 Cuerpo.Caminar();

\*Paso 3 Cuerpo.Disparar();

\*Paso N\*/

}

}

Como se puede observar la clase Robot tiene la acción caminar, pero la responsabilidad es de la clase Body, también es claro que el valor de la propiedad Cuerpo lo podemos cambiar en tiempo de ejecución tan solo asignando una instancia diferente a la propiedad Cuerpo. Esta es una primera aproximación a la inyección de dependencias, siendo más formales incluso muchos comportamientos como Volar y Caminar incluso podrían llegar a estar en clases distintas.

# Inyección de dependencias. ¿Qué es y para qué sirve?

<http://codecriticon.com/inyeccion-de-dependencias/>

La inyección de dependencias (**DI**) es un patrón de diseño que deriva de un patrón más genérico llamado ***Inversión de Control***.**DI** hace uso de la modularidad y la reutilización, las cuales siempre deberíamos tener en cuenta si nuestra aplicación va a estar dotada de mayor funcionalidad. Más adelante, veremos en que consiste la Inversión de Control, pero hoy vamos a centrarnos en la **DI**, visto además de una manera simplista, con un ejemplo sencillo.

En el ejemplo, vamos a ver la representación de una puerta con cerradura de llave y una puerta con cerradura de código. Esta sería la representación sin el uso de **DI:**

**Start.java**

public class Start {

public static void main(String[] args) {

//Usamos una puerta con cerradura de llave

ObjetoPuertaLlave puertaLlave = new ObjetoPuertaLlave();

puertaLlave.abrir();

//Usamos una puerta con cerradura de codigo

ObjetoPuertaCodigo puertaCodigo = new ObjetoPuertaCodigo();

puertaCodigo.abrir();

}

}

Esta es la manera tradicional de hacerlo, se crea un nuevo objeto puerta que tenga una cerradura de tipo llave por un lado y otro que tenga una cerradura de tipo código por el otro.

**ObjetoPuertaLlave.java:**

public class ObjetoPuertaLlave {

private ObjetoCerraduraLlave cerradura;

public ObjetoPuertaLlave(){

cerradura = new ObjetoCerraduraLlave();

}

public void abrir(){

cerradura.accionar();

}

}

**CerraduraInterface.java:**

public interface CerraduraInterface {

void accionar();

}

**ObjetoCerraduraLlave.java:**

public class ObjetoCerraduraLlave implements CerraduraInterface{

@Override

public void accionar() {

System.out.println("Cerradura con Llave");

}

}

**ObjetoCerraduraCodigo.java:**

**public class ObjetoCerraduraCodigo implements CerraduraInterface{**

**@Override**

**public void accionar() {**

**System.out.println("Cerradura con Código");**

**}**

**}**

**Start.java:**

**public class Start {**

**public static void main(String[] args) {**

**ObjetoCerraduraLlave llave =new ObjetoCerraduraLlave();**

**ObjetoPuerta puerta = new ObjetoPuerta(llave);**

**puerta.usar();**

**ObjetoCerraduraCodigo codigo = new ObjetoCerraduraCodigo();**

**puerta = new ObjetoPuerta(codigo);**

**puerta.usar();**

**}**

**}**

La salida de la ejecución es la siguiente:

**Cerradura con Llave**

**Cerradura con Código**

**Como se puede ver, la salida es la misma en ambos casos, la diferencia es que si quisiéramos añadir una puerta con cerradura de tipo ”Reconocimiento\_dactilar“, en el primer caso, tendríamos que crear un nuevo tipo de puerta que tuviera como atributo un objeto de tipo CerraduraDactilar y ya tendríamos tres objetos distintos cuya única diferencia es el tipo de un atributo.**

**En el segundo caso, usando DI, bastaría con crear el objeto CerraduraDactilar y pasarla como parámetro al constructor de ObjetoPuerta. Seguiríamos teniendo un solo ObjetoPuerta y solamente añadiríamos la nueva implementación de la cerradura.**

**Uno de los problemas que plantea el uso de interfaces es que a la hora de depurar visualmente el código no vemos de que tipo es el objeto que pasamos, ya que solo vemos su representación como interface, nada que haciendo Debug en nuestro IDE no se solucione.**

**Es cierto que podríamos haber creado un ObjetoPuerta genérico del que heredasen todos los demás objetosPuerta con sus atributos cerradura concretos, pero no resolveríamos en nada el problema de tener objetos iguales cuyas únicas diferencias son un atributo.**

**Alguien puede preguntarse la utilidad real de este patrón teniendo en cuenta que pudiera ser que tuviéramos un ObjetoPuerta que tuviera dos o más atributos o un ObjetoCerradura que tuviera algún atributo y volvería a presentarse el problema inicial, creo dos objetos de tipo CerraduraConAtributo y CerraduraSinAtributo o vuelvo a usar DI.**

**Pero aún usando DI, a la fuerza tendremos que instanciar los dos tipos de cerradura y pasar como parámetro al ObjetoPuerta la que nos convenga en el momento, tal y como hemos hecho con ObjetoCerraduraCodigo y ObjetoCerraduraLlave.**

**Para solventar todo esto existe la inversión de control (IoC) donde veremos como mediante la inyección de dependencias y la reflexión se irán creando las instancias de clases que necesitemos según el momento y necesidad trasladando todo el control fuera de la clase principal.**

# Inversión de control (Wikipedia)

**Inversión de control** (Inversion of Control en inglés, IoC) es un método de programación en el que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales, en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones. Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones. En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

En cierto modo es una implementación del [**Principio de Hollywood**](http://es.wikipedia.org/w/index.php?title=Principio_de_Hollywood&action=edit&redlink=1), una metodología de diseño de software, cuyo nombre proviene de las típicas respuestas que se les dan a los actores amateurs en las audiciones que tienen lugar en la meca del cine [[1]](http://en.wikipedia.org/wiki/Hollywood_Principle): no nos llames; nosotros te llamaremos.

Es el principio subyacente a la técnica de [Inyección de Dependencias](http://es.wikipedia.org/wiki/Inyecci%C3%B3n_de_Dependencias), siendo términos frecuentemente confundidos.

Historia

En los comienzos de la programación, los programas eran lineales y monolíticos. El flujo de ejecución era simple y predecible, ejecutándose línea tras línea.

Aparecieron dos conceptos que revolucionaron la programación: la **modularidad** y la **reutilización** de los componentes: se crean [bibliotecas](http://es.wikipedia.org/wiki/Biblioteca_(inform%C3%A1tica)) de componentes reutilizables. El flujo se complica, saltando de componente a componente, y aparece un nuevo problema: la **dependencia** (acoplamiento) entre nuestros componentes.

El problema se empieza a considerar lo suficientemente importante como para definir nuevos conceptos en el diseño:

* Inversión de Control (IoC)
* [Inyección de Dependencias](http://es.wikipedia.org/wiki/Inyecci%C3%B3n_de_Dependencias) (Dependency Injection, DI)

Definición

El flujo habitual se da cuando es el código del usuario quien invoca a un procedimiento de una biblioteca.

La inversión de control sucede cuando es la biblioteca la que invoca el código del usuario.

Típicamente sucede cuando la biblioteca es la que implementa las estructuras de alto nivel y es el código del usuario el que implementa las tareas de bajo nivel.

La utilización de **interfaces** y la aparición de los [frameworks](http://es.wikipedia.org/wiki/Framework" \o "Framework) han popularizado este término. De hecho es el concepto central del [Framework de Spring](http://es.wikipedia.org/wiki/Spring_Framework), ya que implementa un "**Contenedor**" que se encarga de gestionar las instancias (así como sus creaciones y destrucciones) de los objetos del usuario. Por tanto las aplicaciones que utilicen el framework de Spring (no Spring propiamente dicho) utilizarán Inversión de Control.

# ****Inversión de Control (IoC)****

<http://elburgues.wordpress.com/2011/02/14/inversion-de-control-ioc/>

**Normalmente, para reutilizar código fuente de alguien, se debe llamar a una librería. Es lo que estamos haciendo cuando usamos .NET framework. Por ejemplo, cuando llamas a Console.WriteLine(“bla bla…”), estás usando código fuente que ha escrito alguien, tu haces la llamada, por lo tanto, tú tienes el control. Sin embargo, hay veces que es .Net framework el que reutiliza código fuente escrito por ti. Por ejemplo, cuando escribes un algoritmo de ordenación de números enteros usando la interfaz IComparable. En ese caso la dirección del control está invertida, alguien está llamando a tu código en vez de ser tu el que llama al código de alguien. Eso es la inversión de control (IoC). Veámoslo con el ejemplo que hemos descrito:**

**using System;**

**namespace InversionControl**

**{**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**// EnteroExtraño define un método de ordenación personalizado**

**// en donde los enteros negativos son mayores que los positivos.**

**EnteroExtraño[] ary = new EnteroExtraño[7];**

**ary[0] = new EnteroExtraño(-1);**

**ary[1] = new EnteroExtraño(1);**

**ary[2] = new EnteroExtraño(2);**

**ary[3] = new EnteroExtraño(-2);**

**ary[4] = new EnteroExtraño(-3);**

**ary[5] = new EnteroExtraño(3);**

**ary[6] = new EnteroExtraño(0);**

**// Array.Sort llama repetidamente al método CompareTo en EnteroExtraño**

**// Aquí es donde se produce la inversión de control.**

**Array.Sort(ary);**

**foreach (EnteroExtraño i in ary)**

**Console.WriteLine(i.ToString());**

**Console.ReadLine();**

**}**

**}**

**class EnteroExtraño : IComparable**

**{**

**private int valor;**

**internal EnteroExtraño(int valor)**

**{**

**this.valor = valor;**

**}**

**#region IComparable Members**

**/// <summary>**

**/// Ordena enteros de una forma extraña.**

**/// Cualquier entero negativo es mayor que un entero positivo.**

**/// De lo contrario, ordena con normalidad.**

**/// Esto significa que el -1 es el número más grande posible.**

**/// </summary>**

**public int CompareTo(object obj)**

**{**

**int otroValor = ((EnteroExtraño)obj).valor;**

**if (valor == otroValor)**

**return 0;**

**else if (valor < 0 && otroValor >= 0)**

**return 1;**

**else if (otroValor < 0 && valor >= 0)**

**return -1;**

**else**

**return valor.CompareTo(otroValor);**

**}**

**#endregion**

**public override string ToString()**

**{**

**return valor.ToString();**

**}**

**}**

**}**

**Si implementamos la interfaz IComparable en una clase, escribimos código en el método CompareTo(). Este método define cuando un objeto de la clase que lo contiene es mayor que otro. Entonces, cuando llamamos al método Array.Sort(…) sobre un array de objetos de esa clase, es el framework el que se encarga de llamar a nuestra rutina de ordenación de los objetos, acorde a nuestra definición. Nosotros no tenemos el control de cuando se realizan las llamadas, podemos pensar en todo esto como la inversión de control (IoC).**