Trabajo de Taller de programación

**INYECCIÓN DE DEPENDENCIAS**

**Elaborado por:**

Kevin Astete Carpio



Universidad Científica Del Sur

Diciembre 2016

**TABLA DE CONTENIDO**

[**RESUMEN** 4](#_Toc468814984)

[**ANTECEDENTES** 4](#_Toc468814985)

[**FUNDAMENTOS TEÓRICOS** 5](#_Toc468814986)

[**CASO DEMOSTRATIVO** 9](#_Toc468814987)

[**DISCUSIÓN** 10](#_Toc468814988)

[**CONCLUSIÓN** 11](#_Toc468814989)

[**RECOMENDACIONES** 12](#_Toc468814990)

[**BIBIOLGRAFÍA** 13](#_Toc468814991)

# **RESUMEN**

La Inyección de Dependencias o DI 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. Este patrón nos ayudará a tener una arquitectura con bajo acoplamiento y conformado por componentes que tienen bien claros la responsabilidad que ocupa cada uno.

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 que se explicará más a detalle en los siguientes párrafos.

Uno puede aplicar el patrón de Inyección de Dependencias sin usar ningún Framework, simplemente utilizando clases con suficientes constructores, métodos, getters y setters. Un framework de inyección de dependencia simplifica la inicialización de las clases con los objetos correctos.

# **ANTECEDENTES**

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.

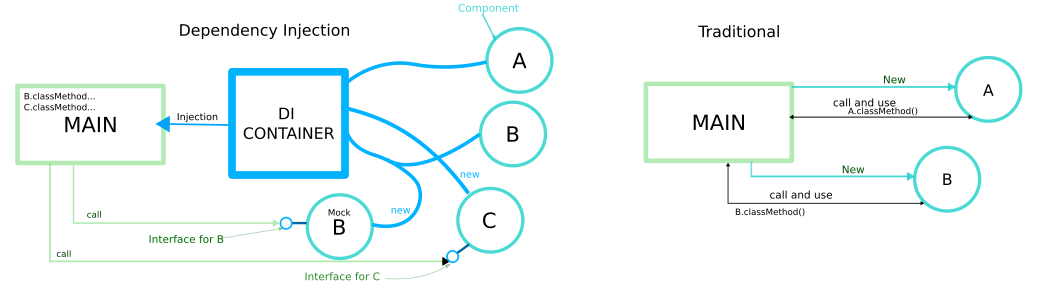
El problema de la dependencia se empieza a considerar lo suficientemente importante como para definir nuevos conceptos en el diseño:

* [Inversión de control](https://es.wikipedia.org/wiki/Inversi%C3%B3n_de_control) (IoC).
* Inyección de Dependencias (DI), que es una forma de inversión de control.

# **FUNDAMENTOS TEÓRICOS**

La inyección de dependencias es un patrón de diseño de software usado en la Programación Orientada a Objetos, que trata de solucionar las necesidades de creación de los objetos de una manera práctica, útil, escalable y con una alta versatilidad del código.

En la mayoría de los frameworks actuales se aplica la Inyección de dependencias como parte de las herramientas y modelos que facilitan al programador. Como cualquier patrón de diseño de software trata de solucionar de una manera elegante un problema habitual en el desarrollo de software, por lo que también es idóneo utilizar este patrón en el desarrollo de proyectos a pequeña escala.



Inyección de Dependencias vs Modo tradicional

**Inversión de Control (IoC)**

Los conceptos de Inversión de Control e Inyección de Dependencias no son nuevos, sino que se remontan hasta finales de la década de los 80. Sin embargo, estos conceptos han comenzado a popularizarse debido a la estrecha relación que mantienen con la aparición de Frameworks como *Spring* en Java o *Unity* en .NET.

Martin Fowler

El concepto de Inversión de Control fue acuñado originalmente por Martin Fowler, diseñador del patrón MVVM (Model View View-Model). Fowler definió el concepto de forma informal denominándolo como el Principio de Hollywood, en el que, tras una audición, se le decía al actor la famosa frase de No nos llames, nosotros te llamaremos.



El principio establece una diferenciación entre el concepto de biblioteca y framework, definiendo el primero como un simple conjunto de clases, métodos y funciones que son invocadas por el flujo del programa y que posteriormente devuelven el control a éste (control normal) y el segundo como un diseño más abstracto y elaborado que se encargará, en algún momento, de invocar el código que el programador se encargue de codificar (inversión de control).

El ejemplo expuesto por Fowler no puede ser más sencillo: un control normal sería un simple programa secuencial de consola en el que el programa va solicitando datos al usuario y realizando operaciones (cálculos, visualización por pantalla) al recibir las respuestas. Un programa que aplica una inversión de control, sin embargo, se podría representar como una ventana compuesta por cajas de texto, etiquetas y botones. El Framework, en este caso, expondría un bucle de espera que detectaría la emisión de eventos, como la pulsación de un botón, momento en el cual se ejecutaría el código de usuario. El Framework invocará nuestro código en lugar de realizar la operación contraria.

**El Principio de Inversión de Dependencias**

El segundo concepto se lo debemos a otro de los gurús de la ingeniería del Software, Robert C. Martin, creador del desarrollo ágil y de los principios básicos de la programación orientada a objetos, denominados SOLID.

Dos de estos principios, de hecho, sirvieron como base para otro de ellos: el concepto de Inversión de Dependencias:

* **Principio Abierto/Cerrado:** una entidad software debe ser abierta para su extensión, pero cerrada para su modificación.
* **Principio de Sustitución de Liskov:** un objeto siempre debe poder ser reemplazado por una instancia de una clase derivada sin alterar el funcionamiento del programa, es decir, todos los métodos de una clase padre deben estar presentes en una clase hija, y esta debe poder asumir el papel de su padre.

Partiendo de estos principios, Martin escribió un artículo en el que desarrollaría el concepto de Principio de Inversión de Dependencias. En este artículo establece, a grandes rasgos, que uno de los grandes problemas del software es el acoplamiento, es decir, la dependencia de clases entre sí.

Según Martin, las clases de las capas superiores no deberían depender de las clases de las capas inferiores, sino que deberían basarse en abstracciones. Del mismo modo, las clases inferiores deberían cumplir el mismo principio. Del mismo modo, las abstracciones no deberían depender de los detalles, sino que son los detalles los que deberían depender de las abstracciones.

El concepto de inversión proviene, por lo tanto, por el giro de 180 grados que se produce respecto al paradigma habitual en el que los módulos superiores tienden a construirse sobre los inferiores (por ejemplo, una clase de interfaz de usuario invocando un método de una clase de negocio) y en el que las abstracciones tienden a depender de los detalles (por ejemplo, una implementación de una interfaz).

A continuación se menciona algunas ventajas y desventajas del patrón Inyección de Dependencias:

**Ventajas:**

* Mejora y facilita las pruebas unitarias o Unit Tests y permite aplicar Mocking.
* Bajo acoplamiento entre los componentes, lo cual es una gran ayuda sobretodo en la mantenibilidad del software.
* El software se hace más mantenible a medida que va creciendo, ya que si se implementa una buena arquitectura con DI, la responsabilidad de cada uno de los componentes será muy clara y un cambio podrá ser más fácil de implementar.

**Desventajas:**

* La inyección de dependencia puede dificultar el rastreo del código (leer) porque separa el comportamiento de la construcción. Esto significa que los desarrolladores deben consultar más archivos para seguir cómo funciona un sistema.
* En los frameworks requiere configuración adicional para que pueda funcionar.
* Irónicamente la Inyección de dependencia puede generar dependencia del framework de Inyección de dependencia.

# **CASO DEMOSTRATIVO**

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;

}

}

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** void 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.

# **DISCUSIÓN**

**"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.

¿Qué 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 especialista y dedicado a ello [una responsabilidad], pero ahora tenemos otro diferenciador, el objeto responsable de ejecutar esa única tarea se 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 asignándole una instancia de objeto diferente.

# **CONCLUSIÓN**

Para implementar Inyección de Dependencias es necesario conocer bien de este patrón, sus ventajas y desventajas, saber cuándo usarlo frente a una solución de software. Este patrón está creado para soportar sistemas complejos, ya que implementarlo frente a una aplicación poco compleja o simple nos traería más problemas que beneficios. El uso de un framework haría más complejo el desarrollo de la aplicación sin embargo calzaría bien en sistemas con miras a expandirse, aplicaciones escalables o modulares, en donde sea común la reutilización de componentes o servicios

# **RECOMENDACIONES**

Una de las desventajas de este patrón es que al implementar un framework o librería de DI se tendrá que desarrollar más código y configuraciones extras para que esto logre funcionar. Se recomienda a las librerías o frameworks de este patrón ayudar más en este tema para que a la hora de desarrollar no sea muy tedioso implementarlo, como por ejemplo el mapeo manual de los tipos o por el archivo de configuración.

# **BIBIOLGRAFÍA**

García, D. (s.f.). *INVERSIÓN DE CONTROL E INYECCIÓN DE DEPENDENCIAS*. Obtenido de Let's code something up!: https://danielggarcia.wordpress.com/2014/01/15/inversion-de-control-e-inyeccion-de-dependencias/

Pacheco, J. C. (15 de 11 de 2016). *C# - Inyección de Dependencias*. Obtenido de Microsoft - Developer Network: https://msdn.microsoft.com/es-es/communitydocs/net-dev/csharp/inyeccion-de-dependencias

Wikipedia. (22 de 9 de 2016). *Wikipedia, la enciclopedia Libre*. Obtenido de Inyección de dependencias: https://es.wikipedia.org/wiki/Inyecci%C3%B3n\_de\_dependencias