 **UNIVERSIDAD CIENTIFICA DEL SUR**

**Facultad de Ingeniería**

**Escuela de Ingeniería de Sistemas de Información y Gestión**

**Programa CPE**

***Inyección de dependencias***

**Integrantes:**

Guillermo Alomía Monjaraz

Carlos Benites Luna

Profesor: Gustavo Coronel

**2017**

INDICE

[Resumen 3](#_Toc475557438)

[INTRODUCCIÓN 4](#_Toc475557439)

[Antecedentes 4](#_Toc475557440)

[Fundamento Teórico 5](#_Toc475557441)

[Inyección de Dependencias 5](#_Toc475557442)

[El Principio de Inyección de Dependencias 8](#_Toc475557443)

[Técnicas de Implementación 9](#_Toc475557444)

[Cuando utilizar DI 11](#_Toc475557445)

[Implementación del patrón en Java 11](#_Toc475557446)

[Discusión 14](#_Toc475557447)

[Caso Práctico 14](#_Toc475557448)

[Conclusión 15](#_Toc475557449)

[Recomendaciones 15](#_Toc475557450)

[Bibliografía 16](#_Toc475557451)

# Resumen

En el presente trabajo se realizó un estudio de diversas fuentes para comprender la “Inyección de Dependencias”, en este trabajo buscamos brindar a nuestros compañeros un mejor entendimiento sobre la DI y sus diferentes técnicas de implementación mediante el uso de constructor, setter o interfaz.

Para ello nos valemos de ejemplos ilustrativos llevados a la POO, esperamos que sea de mucho interés para el lector, dado que esperamos ayude a tomar una buena decisión para saber en qué momento se debe implementar.

En conclusión, se debe utilizar la inyección de dependencias porque es flexible, testeable y mantenible, La aplicación del patrón de Inversión de Control es una de las primeras decisiones que deben tomarse en el diseño de la arquitectura de un producto software.

# INTRODUCCIÓN

DI, corresponde a las siglas de Dependency Injection, es un patrón de diseño 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.

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

# 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 (IoC).

Inyección de Dependencias (DI), que es una forma de inversión de control.

# Fundamento Teórico

Una dependencia entre un componente y otro puede establecerse estáticamente o en tiempo de ejecución. En este último escenario es donde cabe el termino de inyección, y para que esto pueda ser posible, debemos referenciar interfaces y no implementaciones directas.

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 su 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.

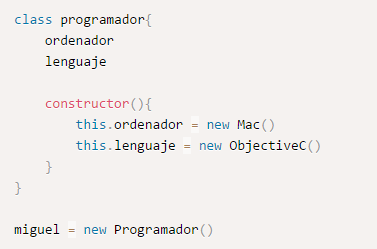
En la mayoría de los frameworks actuales se aplica la Inyección de dependencias como parte de las herramientas y modelos que facilitan al programador. Como cualquier patrón de diseño de software trata de solucionar de una manera elegante un problema habitual en el desarrollo de software, por lo que también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

# Inyección de Dependencias

Aparte de un patrón de diseño de software, vamos a explicar qué idea hay detrás de ese nombre. Este patrón, como muchos otros, nos ayuda a separar nuestro código por responsabilidades, siendo que en esta ocasión sólo se dedica a organizar el código que tiene que ver con la creación de los objetos. Uno de los principios básicos de la programación, y de las buenas prácticas, es la separación del código por responsabilidades. Pues la inyección de dependencias parte de ahí.

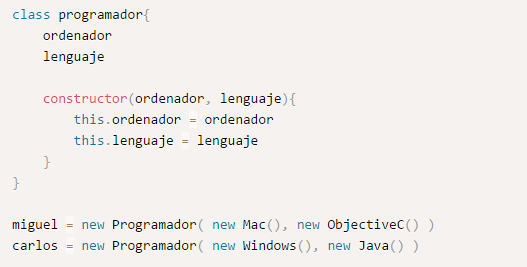
En el código de una aplicación con OOP (Programación Orientada a Objetos) tenemos una posible separación del código en dos partes, una en la que creamos los objetos y otra en la que los usamos. Existen patrones como las factorías que tratan esa parte, pero la inyección de dependencias va un poco más allá. Lo que dice es que los objetos nunca deben construir aquellos otros objetos que necesitan para funcionar. Esa parte de creación de los objetos se debe hacer en otro lugar diferente a la inicialización de un objeto

Por ejemplo, este código no sería el mejor:



El problema que nos encontramos es que la clase Programador está fuertemente acoplada con la el ordenador Mac o el lenguaje ObjectiveC. Si mañana queremos tener programadores de C que usan Windows, tal como está el código, tendríamos que crear una nueva clase Programador, porque esta no nos valdría.

Ahora veamos esta alternativa de código mucho más versátil.



Ahora nuestro programador es capaz de adaptarse a cualquier tipo de ordenador y cualquier tipo de lenguaje. De hecho observarás que hemos podido crear un segundo programador llamado "carlos" que es capaz de programar en Java bajo Windows.

El ejemplo es simple, pero sirve para clarificar el resto que viene detrás del concepto de inyección de dependencias. En realidad, es tan sencillo como apreciar que al constructor de los objetos se les están pasando aquellas dependencias que ellos tienen para poder realizar sus tareas.

El hecho en sí, de enviar por parámetros los objetos que son necesarios para que otro objeto funcione, es la inyección de dependencias.

# El Principio de Inyección de Dependencias

Robert C. Martin afirma en el Principio de Inyección de Dependencias:

    A. *Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.*   
    B. *Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.*

Imaginemos por un momento la solución inicial de la estación meteorológica (listado 1). La clase de alto nivel EstacioMeteorologica depende de la clase de bajo nivel Termometro (o Barometro, Anemometro, etc.). Toda la lógica de la solución se implementaría en la clase de alto nivel, y cualquier modificación en las clases de bajo nivel tendría repercusión no únicamente sobre la definición de la clase de alto nivel, sino sobre la propia lógica de la aplicación, llegando incluso a forzar cambios en la misma, cuando debería ser la clase de alto nivel la que debería forzar el cambio a las clases de bajo nivel sin comprometer la lógica de la aplicación; es decir, justamente lo contrario. Además, la clase de alto nivel sería difícilmente reusable debido a este acoplamiento. Sencillamente, y resumiendo, la clase Estacio Meteorologica no debe depender de la clase Termometro; en todo caso, al contrario. (Miguel Angel & José Miguel, 2015)

# Técnicas de Implementación

Existen tres formas de implementación de la Inyección de Dependencias:

* por constructor
* por setter
* por interfaz.

Para poder revisar las técnicas de implementación de DI los veremos a través de un código ejemplo:

tenemos la clase Schumacher que implementa la interface Piloto, que tiene el método conducir.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class Schumacher implements Piloto {       private Vehiculo;       public Persona(){        vehiculo = new Ferrari();     }       @Override       public void pilotar() {        vehiculo.conducir();     }  } |

En el ejemplo anterior, la clase Schumacher es dependiente de la interfaz Vehiculo que es implementada por la clase Ferrari en el constructor. Por lo anterior, se puede decir que Shumacher se encuentra fuertemente acoplado a Ferrari, pero qué pasa si queremos que Shumacher sea capaz de pilotar un Lamborghini o un BMW, para este caso la forma de la implementación se encuentra limitada y no es muy flexible. Entonces es cuando la inyección de dependencias llega a rescatarnos, o más bien a Schumacher…

**Inyección basada en constructor**

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

Para nuestro ejemplo quedaría así:

En su lugar el constructor de la clase recibe como argumento un objeto de la clase abstracta Vehículo lo que hace posible que Schumacher sea capaz de pilotar diferentes tipos de vehículos sin estar acoplado a alguno en particular.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class Schumacher implements Piloto {       private Vehiculo;       public Persona( Vehiculo v){        vehiculo = v;     }       @Override       public void pilotar() {        vehiculo.conducir();     }  } |

**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.

En el ejemplo sería:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Schumacher implements Piloto {       private Vehiculo;       public Persona( Vehiculo v) {        vehiculo = v;     }       public void setVehiculo(Vehiculo v) {        vehiculo = v;     }     @Override       public void pilotar() {        vehiculo.conducir();     }  } |

el uso de *setters*, permite mayor flexibilidad. Ya que incluso después de que un objeto de la clase haya sido creado, se podrá asignar un vehículo diferente. Es importante remarcar que en este caso estamos trabajando con una Interfaz, y no con una clase normal, y gracias al polimorfismo el uso de una interfaz nos da la ventaja de que si a mitad de la carrera (la ejecución), decidimos que Schumacher conduzca un nuevo vehículo, entonces será posible cambiar la implementación de la dependencia utilizando un setter y de una forma muy sencilla.

**Inyección basada en interfaces**

Aquí se utiliza una interfaz común que otras clases implementan para poderles luego inyectar dependencias

# Cuando utilizar DI

(Castillo, s.f.)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.

## 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 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 entre otros), por lo cual en la aplicación también se utilizará inversión 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.

El código que has visto anteriormente es muy sencillo, pero en estructuras más complejas observarás que hay muchos objetos que dependen de otros objetos. Esos otros objetos a su vez dependen de otros, que dependen de otros. Como has observado, dado el patrón de inyección de dependencias, necesito tener listos todos los objetos de los que depende el que voy a construir, porque se los tengo que enviar por parámetro al constructor.

Por ejemplo, el programador depende del ordenador y los lenguajes, pero el ordenador depende del sistema operativo y el teclado y ratón. A su vez el teclado depende de una conexión USB y de un conjunto de teclas, la conexión USB depende de las líneas de comunicación de datos y de la electricidad, la electricidad depende de que hayas pagado tu factura el mes anterior y de que así haya tensión en la red. Así podríamos continuar hasta donde nos lleve la imaginación.

Todo eso nos indica que, para conseguir instanciar un programador necesito haber instanciado antes la red eléctrica y las líneas de comunicación del USB, la conexión USB, cada una de las teclas del teclado, el teclado, el ratón, el sistema operativo, el ordenador, los lenguajes… y cuando tengo todo eso, por fin puedo invocar al constructor de la clase Programador para obtener el objeto que quería.

¿Complicado? No. Pero sí es laborioso. Quizás tengas una docena de líneas de código, o más, para poder hacer lo que tú querías, que era instanciar un programador al que necesitas inyectarle todas las dependencias, conforme nos dicta el patrón.

La solución a esta problemática nos la trae el contenedor de dependencias, también llamado inyector de dependencias, contenedor de servicios, etc.

Básicamente es como una caja a la que le pido construir las cosas. Esa caja sabe qué tiene que hacer para construir cada uno de los objetos de la aplicación. Si queremos un programador, simplemente le pedimos al contenedor de dependencias que nos cree un objeto de esa clase y él se dedica a crearlo. Finalmente el "dependency container" lo devuelve listo para ser usado.



El contenedor de dependencias nos permite que la instanciación de un objeto, por muchas dependencias que tenga, vuelva a ser tan simple como una llamada a un método. ¿Dónde está la magia? el contenedor de dependencias simplemente tiene todos los objetos que puedas necesitar para crear cualquier objeto complejo y si no cuenta en ese instante con las dependencias necesarias, sabe cómo conseguirlas en el acto.

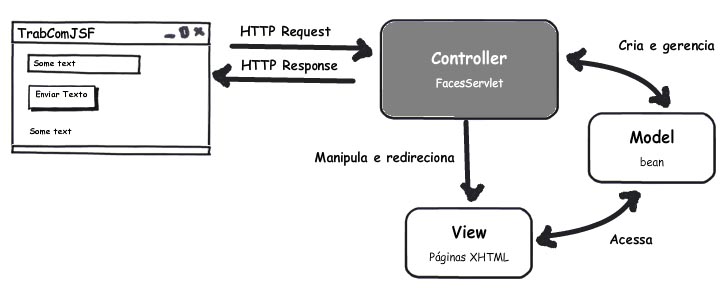
El uso de este contenedor de dependencias ya depende del lenguaje que estés usando y la librería o framework con la que trabajes. Además habitualmente hay que hacer algunas configuraciones básicas para que funcione. Si te fijas, en la línea anterior estoy diciendo que me cree un programador, pero en algún lugar le tendré que decir cómo puedo obtener ese objeto deseado y si tal como está ese programador será experto en Java, ObjectiveC o PHP

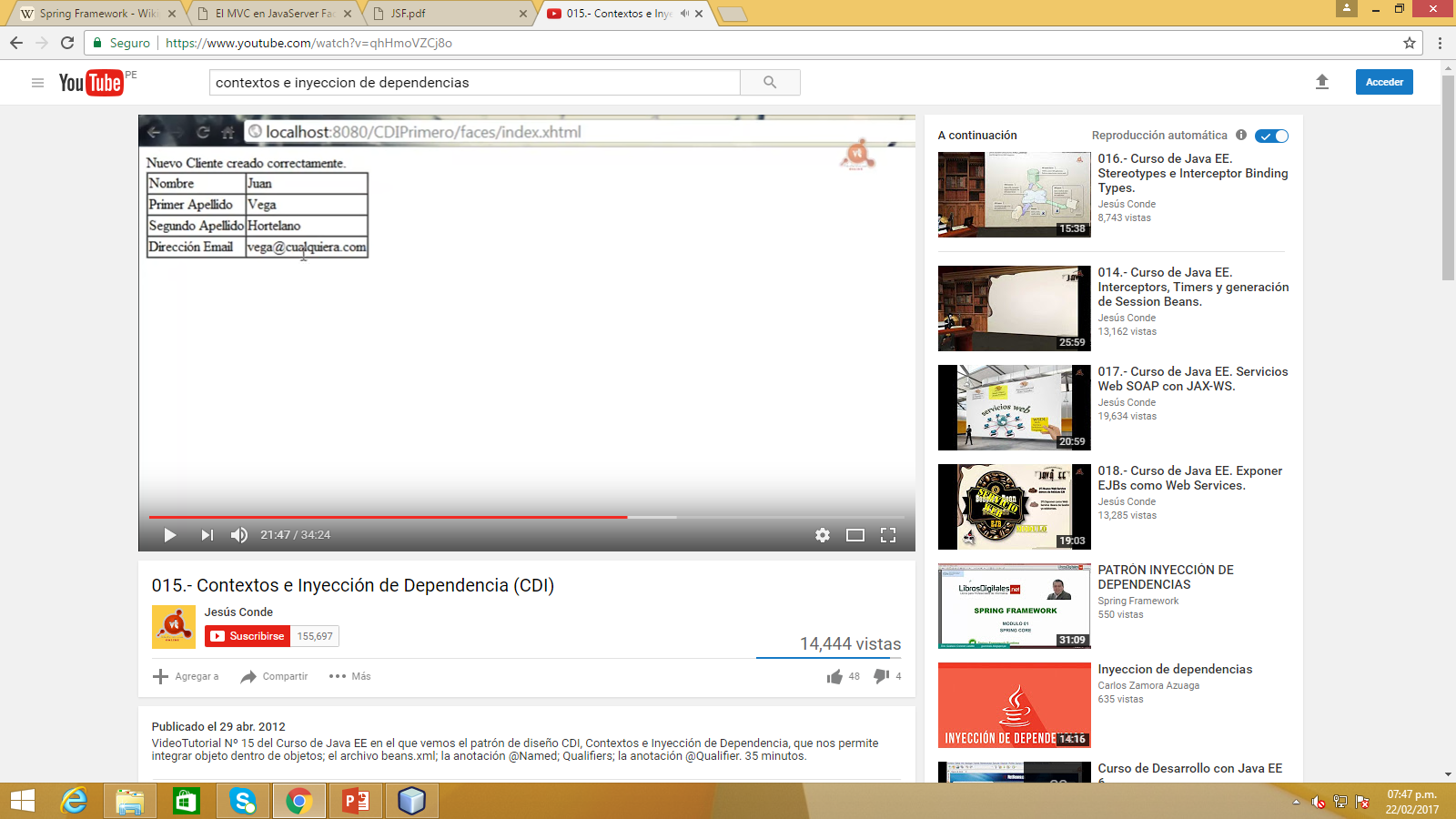
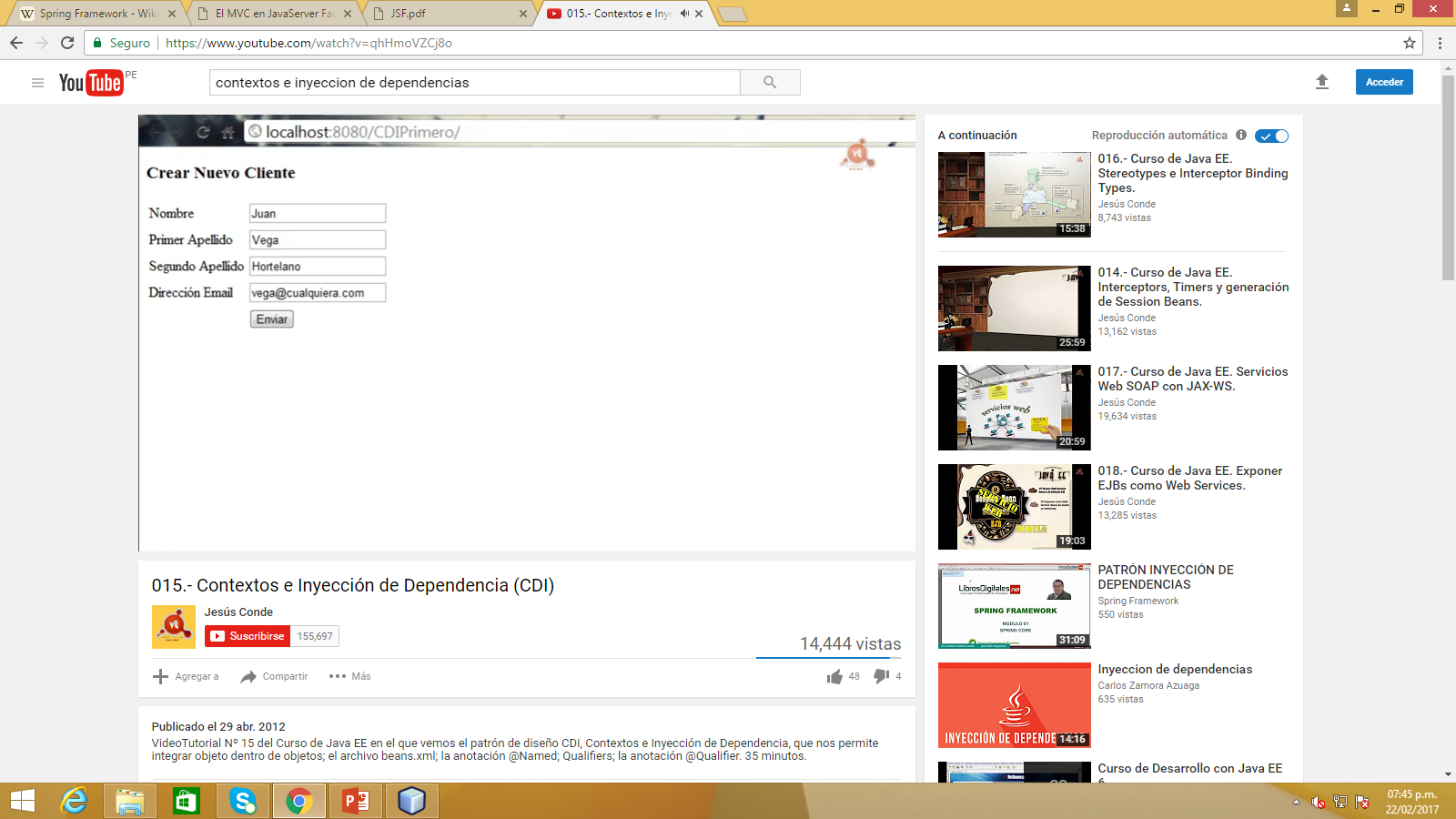
# Discusión

Si hablamos de Inyección de Independencias se nos viene a la mente Spring Framework ya que como habíamos mencionado es central su Contenedor de inversión de control (IoC), que proporciona una forma consistente de configuración y administración de objetos Java usando la Reflexión. En un principio íbamos a enfocarnos en desarrollar un caso práctico de este framework pero fuimos más allá. Quisimos desarrollar todo nuestros conocimientos adquiridos hasta este momento y es así como llegamos al framework oficial de Java Enterprise para el desarrollo de interfaces de usuario avanzadas en aplicaciones web. Tiene la característica de seguir el patrón MVC.

# Caso Práctico

La idea implementar un sencillo ejemplo con JSF, separando las distintas responsabilidades de la aplicación según el MVC y de disponer de un servidor de aplicaciones que tenga soporte para los contextos e Inyección de Dependencias (CDI) es la razón para desarrollar siguiente ejemplo vemos el patrón de diseño CDI, Contextos e Inyección de Dependencia, que nos permite integrar objeto dentro de objetos;   
el archivo beans.xml; la anotación @Named; Qualifiers; la anotación @Qualifier.





# Conclusión

Con este trabajo, hemos tratado de mostrar de una forma práctica la relación existente entre dependencias, detalles y abstracciones

Los patrones de diseño son conceptos o ideas. El código puede ser diferente según lo que necesites o el lenguaje que se esté usando (esto no es sólo para Java, si no para cualquier lenguaje que admita la programación orientada a objetos).

Es muy ventajoso si usas estructuras muy grandes (clases que requieran decenas de objetos para cobrar vida). Por otro lado, si tus objetos son pequeños, este tipo de patrón sólo haría de intermediario entre otras clases y sería molesto, ya que no merecería la pena mantenerlo en la aplicación.

Una de las principales ventajas de trabajar con la tecnología Java Server Faces es que está basado en el patrón MVC (Modelo Vista Controlador) ofreciendo una limpia separación entre el comportamiento y la presentación.

# Recomendaciones

Se debe utilizar la inyección de dependencias porque es flexible, testeable y mantenible.

# Bibliografía

Castillo, E. C. (s.f.). *https://github.com/gcoronelc/SISTUNI\_SPRING\_004*. Obtenido de https://github.com/gcoronelc/SISTUNI\_SPRING\_004/blob/master/Docs/SpringCap01\_InyeccionDependencias.pdf

M. A., & J. T. (08 de 05 de 2015). *desarrolloweb.com*. Obtenido de https://desarrolloweb.com/articulos/patron-diseno-contenedor-dependencias.html

Universidad de Alicante – España. (06 de 06 de 2014. http://www.jtech.ua.es/j2ee/publico/jsf-2012-13/sesion01-apuntes.html

Otras Referencias

<https://www.youtube.com/watch?v=qnufD0q9PIM>

<https://www.youtube.com/watch?v=Ds55iRZ1crU>

<https://www.youtube.com/watch?v=Gj_EbIN4P5w>

<https://www.youtube.com/watch?v=MWtrUxvdJlw&t=410s>