D-IP: DEPENDENCY INVERSION PRINCIPLE

Básicamente lo que nos dice este principio es que:

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.

Inyección de Dependencias (en inglés *Dependency Injection*, DI) 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. Es una herramienta comúnmente utilizada en varios patrones de diseño orientado a objetos, consiste en inyectar comportamientos a componentes. El término fue acuñado por primera vez por Martin Fowler. 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.

La DI permite conseguir un desacoplamiento de las clases en el código, de tal manera que si una clase emplea otras clases, la inicialización de los objetos venga dada desde fuera. Además, si se puede sustituir las clases por interfaces, se puede extender a otro objeto que implemente la interfaz, sin variar el parámetro.

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

- **Inversión de control** (IoC, por sus siglas en inglés): También llamada Inverse of Control, se utiliza para evitar la excesiva creación de objetos de manera innecesaria que puede provocar la caída del servidor. Esto se suele utilizar con objetos DAO que crean una conexión a la BD ya que si cada usuario crea una conexión a la BD y el número de usuarios es alto pueden hacer que el servidor se caiga.
- La **inyección de dependencias** actúa generando los objetos cuando se arranca la aplicación y luego los inyecta en los demás objetos que los necesiten a través de métodos set o bien a través del constructor, pero estos objetos se instancian una vez, se guardan en una factoría y se comparten por todos los usuarios (al menos en el caso de Spring) y nos evitamos tener que andar extendiendo clases o tumbar el servidor de la BD

PRIMER EJEMPLO

Imagine que está haciendo un videojuego en cual tiene 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

Esta sería una implementación de Robot que NO utiliza Inyección de Dependencias

```
class RobotInicial {
        public RobotInicial() { }
        public void Caminar() {
                /*Paso 1 ,Paso 2, Paso N*/
        }
        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*/
        }
}
```

Sin embargo, al final, la responsabilidad del cuerpo y de la cabeza sigue siendo confusa, además, hay que intentar respetar el principio SRP "Single Responsability" que señala: "**Un objeto una responsabilidad**"

La aplicación de la inyección de dependencias permite diseñar un objeto que puede hacer un conjunto de tareas, (cada una de esas tareas es una responsabilidad), que pueden ser ejecutadas por otro objeto especialista y dedicado a ello [una responsabilidad], pero al ser un objeto… **¡¡puede establecerse su referencia en tiempo de ejecución!!** La implementación habitual de Inyección de dependencias es crear un método para establecer el comportamiento, es decir capaz de cambiar el valor de un atributo asignándole una instancia de objeto diferente.

SEGUNDO EJEMPLO

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.

Ejemplo:
Tenemos una entidad o clase que queremos persistir, ahora bien podemos usar por ejemplo una clase Libro:

```
public class Libro{
  private String ISBN;
  private String title;
  private int year;
  // más datos...
}
```

Usaremos una clase persistidora (bajo nivel) que nos permita guardar un objeto libro en una base de datos:

```
public class DBPersister{
   public void save(Libro libro){
    //Código para guardar..
  }
}

public class Libro{
  private String ISBN;
  private String title;
  private int year;
  private DBPersister p = new DBPersister();
  // más datos...

  public void guardar(){
    p.save(this);
  }
}
```

La clase Libro usará este persistidor internamente para realizar el trabajo de enviar la información de un objeto libro a la bd. Pero aquí nos llega el inconveniente de si luego quiero guardar la información del libro en un archivo xml por ejemplo tengo una dependencia basada en una clase concreta y no en una abstracción.

APLICANDO LA INVERSIÓN DE DEPENDENCIA

Rediseñando esta pequeña estructura se aplicará una abstracción ya sea con una Interface o clase abstracta. En este caso usaremos un Interface llamada **Persister** que nos especifique los métodos necesarios para la abstracción y después dos clases que la implementen.

```
public interface Persister{
  public void save(Libro libro)
}

public class DBPersister implements Persister{
  public void save(Libro libro){
    //Codigo para guardar en DB..
  }
}

public class XMLPersister implements Persister{
  public void save(Libro libro){
    //Codigo para guardar en XML..
  }
}
```

Ahora en vez de una clase concreta usaremos una interface dentro de la clase Libro y luego desde afuera le pasamos al Libro la implementación necesaria, sea la de DBPersister o la XMLPersister.

```
public class Libro{
  private String ISBN;
  private String title;
  private int year;
  private Persister persister;
  // mas datos...

  // Con este setter se pasa el Persister concreto
  public void setPersister(Persister persister){
    this.persister = persister;
  }
  public void guardar(){
    persister.save(this);
  }
}
```

```
public class Prueba{
  public static void main(String[] args){
   Libro l = new Libro(); //Datos del libro
   XMLPersister p = new XMLPersister();
   l.setPersister(p);
   l.guardar();
  }
}
```

Aquí Libro es la clase de nivel alto y los persistidores las clases de nivel bajo y basándonos en una abstracción llamada Persister se aplica este principio.

INVERSIÓN DE DEPENDENCIAS E INYECCIÓN DE DEPENDENCIAS.
La inversión de dependencias, como se ha dicho en el apartado anterior, lleva implícita la inyección de dependencias. Para explicar este concepto, se puede mostrar el siguiente ejemplo, partiendo de este código inicial.

```
class MiClase{
   public MiClase()   {
      ClaseA a = new ClaseA();
      ClaseB b = new ClaseB();
      a.UnMetodo();
      b.OtroMetodo();
   }
}
```

Este código tiene un problema de acoplamiento, ya que la clase depende de otras dos clases. Si cambiase el comportamiento de las clases **ClaseA o ClaseB**, habría que cambiar el código de esta clase, no digamos si los constructores tienen parámetros.

Una primera mejora consistiría en pasar los objetos por argumento, de tal manera que sean definidos fuera de la clase, eliminando la dependencia en esta fase, e i**nyectándola** mediante el constructor, por lo tanto, si el código de estos cambia, no será necesario editar esta clase, pero, nos aseguraremos que el código sigue siendo válido?:

```
class MiClase{
   public MiClase(ClaseA a, ClaseB b)   {
      a.UnMetodo();
      b.OtroMetodo();
   }
}
```

Una segunda mejora consistiría en sustituir los objetos por interfaces, de tal manera que lo que se pase al constructor de la función no sea una clase, sino un **contrato**, una implementación que tendrá que cumplir con la interfaz acordada, aunque el contenido de la clase cambie, o sea otra clase completamente diferente que implemente la interfaz:

```
class MiClase{
   public MiClase(IClaseA a, IClaseB b)   {
      a.UnMetodo();
      b.OtroMetodo();
   }
}
```

De esta manera si se crea una clase **OtherClass** que implemente **IClaseA**, se puede pasar por el constructor y el funcionamiento de **MiClase** no variará, con lo cual no será necesario editar el código. La consecuencia es que esta clase está más desacoplada, lo que permite, por ejemplo, poder pasar objetos de prueba para test unitarios, reducir los fallos y, en definitiva, tener un código más sólido.

INVERSION DE DEPENDENCIAS
El punto de inversión de dependencia es hacer software reutilizable. La idea es que en lugar de dos trozos de código dependen mutuamente, se basan en alguna interfaz abstracta. A continuación, puede volver a utilizar cualquier pieza sin el otro. La forma que más comúnmente esto se logra es a través de un contenedor de inversión de Control como resorte en Java. En este modelo, las propiedades de los objetos se configuran a través de una configuración de XML en lugar de los objetos salir y encontrar su depenency. Por ejemplo:

```
public class MyClass{
  public Service myService = ServiceLocator.service;
}
```

MyClass depende directamente de la clase de servicio y la clase de ServiceLocator. Tanto de los que necesita si desea utilizarla en otra aplicación. Ahora imagine esto...

```
public class MyClass {
  public IService myService;
}
```

Ahora, MyClass se basa en una sola interfaz, la interfaz IService. Permitiría que el contenedor de IoC realmente el valor de esa variable. Así que ahora, MyClass fácilmente pueden ser reutilizados en otros proyectos, sin que la dependencia de los otros dos clases junto con él. Mejor aún, no tienes que arrastrar las dependencias de MyService y las dependencias de esas dependencias.

Fuentes:
http://eljaviador.wordpress.com/2011/08/30/s-o-l-i-d-principio-de-inversion-de-dependencias/
http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=DependencyInjector
http://es.wikipedia.org/wiki/Inyecci%C3%B3n_de_Dependencias