Dependency Injection

CAB402

Sara Alraddadi
N8779333

June 4, 2017

Contents

1 Introduction **2**

1.1 Motivations and Basic concepts 2

1.2 Maintainability 3

1.2.1 Testability 3

1.2.2 Flexibility and Extensibility 3

1.2.3 Loose Coupling 3

1.2.4 High Cohesion 3

2 Dependency Injection **4**

2.1 Definition 4

2.2 Types of Dependency Injection 6

2.3 Constructor Injection 6

2.4 Setter Injection 7

2.5 Interface Injection 8

3 Impact of DI in the non-functional requirements **9**

3.1 Maintainability 9

3.1.1 Testability 9

3.2 Flexibility and Extensibility 9

4 Dependency Injection Frameworks **10**

4.1 XML Configuration 10

4.2 Annotations 12

5 Complete Examples **14**

5.1 Running the example 14

6 Conclusions **15**

Listings

2.1 Naive implementation of a class dependency 4
2.2 Dependency Injection Principle. 5
2.3 Interface which must be implemented by Engine 5
2.4 Use of the Car implementation. 5
2.5 Use of the Car implementation using a different Engine Implementation. 6
2.6 Class Diagram of the Car/Engine example 6
2.7 Constructor 7
2.8 Setter 7
2.9 Dependency Injection Principle. 7
4.1 XML Dependencies Configuration Example. 10
4.2 Using the XML file (Beans.xml). 11
4.3 Dependencies Configuration using Annotations 12
4.4 Using the Configuration class showed before 12
5.1 Running the example code 14

Preface

In this report we are going to summarize our research about Dependency Injection. We are going to explore why this concept is important, when it is needed and how it can be applied to our daily enterprise systems. We shall see practical examples of the different types of Dependency Injection and also what tools can be helpful in our future implementations.

The idea behind Dependency Injection is just to remove the hard-coded use of dependencies implementations, and instead create more flexible way to create the instances that allow an extensible set of classes. This pattern will lead to a more organized code, which impact directly on the quality of the final application.

We are going to explain in the following chapters all the concepts briefly mentioned here.

1

Introduction

1.1 Motivations and Basic concepts

There are usually many requirements when a team (or even a single developer) is creating Enterprise Systems, however, all those requirements can be classified as *functional requirements and non-functional requirements.*[4]. We shall focus only in the non-functional requirements and we shall define some of the main concepts needed to understand the Dependency Injection principle.

Non-functional requirements are the requirements that judge the operation of a system, instead of specific behaviours. In many cases, it is also called "Quality Attributes" or "non-behavioral requirements" [4]. They are important because they are a key factor in the software maintenance, and it compromises around 70% of the software life-cycle[2]. Improving it will reduce time and cost of the projects.

Here we have a list of the non-functional requirements which are directly impacted by dependency injection:

1. Testability
2. Flexibility and Extensibility
3. Loose Coupling
4. High Cohesion

We shall define each of the concept in the this list and then we explain how they are impacted by dependency injection.

1.2 Maintainability

In this section we are going to explore a brief summary of each concept, focusing on the basic knowledge needed to understand its relationship with Dependency Injection.

1.2.1 Testability

It is basically the degree of a software to support testing in a testing context. The higher the degree means that it is easier to test.

It is important to see that the testing process must be incorporated in all the stages of the project, to reduce the amount of error, increase the efficiency in the debugging process.

1.2.2 Flexibility and Extensibility

The main idea with this concept is to keep in consideration any change that of the current behaviors of the software or the creation of new functionalities. A high flexibility/extensibility means that changing or adding new features to the application has very low impact in the internal flow and structure.

1.2.3 Loose Coupling

In summary, the idea is that every component knows as little as possible of other components. It gives the possibility of replacing components with alternative implementations but with the same services.

1.2.4 High Cohesion

Cohesion is a measure that describes the relationship between modules of an application. High cohesion is related with Loose coupling and it means that functions and methods inside a class are very similar. It is important because it reduce the complexity of the code, increases the system maintainability and reusability. The main idea is to organize the code in a way that functions do a very similar job.

2

Dependency Injection

2.1 Definition

Dependency Injection (DI) can be defined as a pattern that allows the injection of objects into another objects by using a container.[2] It is part of the design principle called Inversion of Control.

This can be explained with a very simple example. I am creating and developing this example throughout the report to illustrate my point of view in every section. It will be used to have an easier understanding of the situation. Let's imagine that we have a car, and it has an engine. It means that the class *car* depends on the class *engine*. Naive way to solve this would be as it is shown in the following code.

```
public class Car {

    private TurboEngine engine;

    public Car(){
        engine = new TurboEngine();
    }
}
```

Listing 2.1: Naive implementation of a class dependency

Let's omit the engine class, because it is not too relevant to the example. As you can see in this example, this brings the following limitations [3]:

1. if a dependency (in this case, the class *TurboEngine*) has to be replaced, the source code must be changed.
2. The implementation of the *engine* is only available in compile time. It means that in order to change something, the code must be recompiled.

3. Classes are difficult to test, because they have references to their dependencies. It means, that their dependencies cannot be replaced by mocks or stubs.

As a main requirement of DI is that the class must allow passing values (services in this case) from outside. Let's propose now a solution after applying the Dependency Inversion Principle.

```
public class Car {

    private IEngine engine;

    public Car(IEngine engineImplementation){
        engine = engineImplementation;
    }
}
```

Listing 2.2: Dependency Injection Principle.

We can see here that the Engine is being passed as parameter.

```
public interface IEngine {

    public void turnOn();
    public void turnOff();
    // More methods here.
}
```

Listing 2.3: Interface which must be implemented by Engine

```
Car myCar = new Car(new TurboEngine());
```

Listing 2.4: Use of the Car implementation.

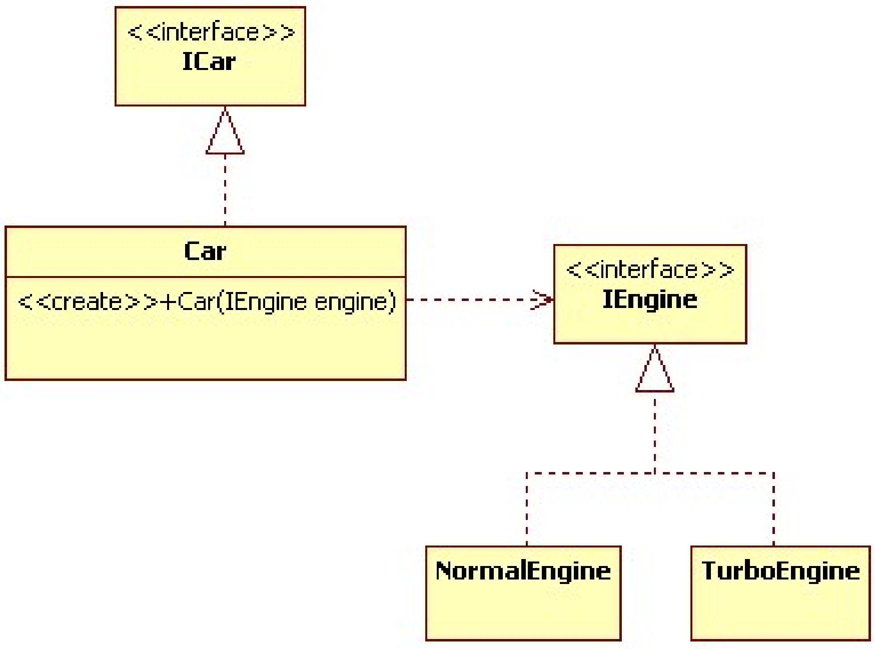
This allows the possibility of changing the implementation of the engine using the same class *Car*. It can be seen as the instance of the engine is *injected* as parameter of the constructor. In this case, we are using a *constructor injection*. We will explore more about this kind of dependency injection in the next subsection.

We can now also implement the *IEngine* class with different kind of engines, and use it with the same *Car* implementation. Let's imagine that we now have a Car with a Normal Engine, instead of a Turbo Engine. We can create the car instance easily, as illustrated below:

```
Car myCar = new Car(new NormalEngine());
```

Listing 2.5: Use of the Car implementation using a different Engine Implementation.

We can see the following class diagram:



Listing 2.6: Class Diagram of the Car/Engine example

2.2 Types of Dependency Injection

There are three types of dependency injection:

1. Constructor Injection.
2. Setter Injection.
3. Interface Injection.

2.3 Constructor Injection

This is simply a dependency injection that provide the service as parameter in a constructor. It will follow a pattern very similar to the following:

```
private IService service;

Client(Service service){
    this.service = service;
}
```

Listing 2.7: Constructor

It has a great advantage because it initializes all the dependencies at the beginning, removing the possibility of having dependencies not initialized which means invalid states. However, adding more dependencies will means to change the constructor and all the places where it was used.

2.4 Setter Injection

In contrast to the Constructor Injection, here, the services are passed to the client using the setters. It follows a pattern similar to the following:

```
private IService service;

setService(IService){
    this.service = service;
}
```

Listing 2.8: Setter

We can illustrate it with the same *Car* example given in the previous chapter. Let's have a look at the following code:

```
public class Car {

    private IEngine engine;


    public Car(){}

    public void setEngine(IEngine
       engineImplementation){
        engine = engineImplementation;
    }
}
```

Listing 2.9: Dependency Injection Principle.

It can be concluded that the flexibility of adding new dependencies can be done by only creating the setters needed, however, invalid states can be reached. The client has to know the state of the service before using it. It is common to create a state checker to confirm that it is on a valid state before using it.

2.5 Interface Injection

It is very similar to the Setter Injection, however, it adds the setter methods as client's dependencies. They are added to the interface, so all the implementations are forced to have those setters.

It has the advantage that dependencies can receive references of clients without knowing their implementations.

3

Impact of DI in the non-functional requirements

3.1 Maintainability

3.1.1 Testability

DI makes the testing process easier. Let's imagine that we have the naive implementation of a car example given in the previous chapters, and we need to do some testing of it. We are forced to test it with the *TurboEngine*.

However, after applying the Dependency Injection (take a look at the listing 2.6), However, after applying the dependency injection pattern, we can easily implement the interface *IEngine* as a Mock, testing the class *Car* in isolation.

3.2 Flexibility and Extensibility

When the traditional pattern is applied (called naive solution), each object is responsible for obtaining its own references, which generates a highly coupled, which also leads to harder tests.[2]

Dependency Injection pattern handle the collaboration between objects, which leads to a lower coupling. Which means that we can add or change the code easier.

It also improves the cohesion because when the Dependency Injection principle is applied, the developer has to think which class will be the client, which will be the service, what methods will have the interface, etc. It imply a major thinking process, which means to stronger cohesion between classes.

4

Dependency Injection Frameworks

Currently, due to the popularity of Dependency Injection, there are too many implementations of the it, and despite no mandatory framework, it offers some important advantages:

1. They control for you when the instances are created. It let you decide if you want to load them when the application is loading or instead taking a lazy-load approach.
2. You can configure the dependencies in external files, like XMLs. Based on that configuration file, some frameworks can create the instances and inject them where needed.
3. They can control how many instance are created in the application.

Dependency Injection can be implemented using XML configuration files or using annotations. Let's study both cases:

4.1 XML Configuration

Main characteristic of this implementation is that all the relationships are specified in a configuration file, usually a XML. The following XML file shows how the example of the *Car* can be implemented using Spring Framework (only the relevant part of the code was shown).

```
<?xml version = "1.0" encoding = "UTF-8"?>

<beans ...>

   <!-- Definition for Car bean -->
   <bean id = "car" class = "com.software.Car">
      <property name = "turboEngine" ref =
         "turboEngine"/>
   </bean>

   <!-- Definition for Engine bean -->
   <bean id = "turboEngine" class =
      "com.software.TurboEngine"></bean>

</beans>
```

Listing 4.1: XML Dependencies Configuration Example.

As it can be seen, we indicate to the framework that our Car class depends on the turboEngine class. We have to specify an id for each relationship, because they are going to be used when we use the instances of the Car. Now, let's see how the instances are created, again we are removing the irrelevant parts of the code:

```
package com.software;

// Here we must have all the imports
import ...


public class MainApp {
   public static void main(String[] args) {
      // Here we specify which config file we are
         going to use
      ApplicationContext context = new
         ClassPathXmlApplicationContext("Beans.xml");

      Car car = (Car) context.getBean("car");
      car.turnOnEngine();
   }
}
```

Listing 4.2: Using the XML file (Beans.xml).

Instances are created automatically by the *ApplicationContext*. In this case, a *car* instance is created.

4.2 Annotations

Some people might consider that creating a big XML configuration file is a tedious work, that is why there is also an option of specify the dependencies are annotations.

Let's examine the following example of Dependency Injection created using Spring Framework with Annotations. First than all, we need to create a configuration class which specify how the instance is created:

```
package com.software;

// Here we must have all the imports
import ...

@Configuration
public class CarConfig {
   @Bean
   public Car car(){
      return new Car( turboEngine() );
   }

   @Bean
   public TurboEngine turboEngine(){
      return new TurboEngine( );
   }
}
```

Listing 4.3: Dependencies Configuration using Annotations

It is exactly equivalent to the previous XML example. This class specify the relationship between the *Car* and *TurboEngine* class. Then we can create the instances as we see here:

```
package com.software;

// Here we must have all the imports
import ...


public class MainApp {
   public static void main(String[] args) {
      // Here we specify which config file we are
         going to use
      ApplicationContext context =
         new
            AnnotationConfigApplicationContext(
               CarConfig.class
           );

      Car car = (Car) context.getBean(Car.class);
      car.turnOnEngine();
   }
}
```

Listing 4.4: Using the Configuration class showed before

Car and TurboEngine will be exactly the same for both configurations. It will create the instance using the Constructor Injection type.

5

Complete Examples

Attached with this report you have available the complete code of the examples given. You need to following requirements to run it:

1. Java JDK 1.8 or above
2. maven 3.0.5 or above

Example was tested on Linux - Debian 8, however, it can run in any system that has the requirements mentioned.

5.1 Running the example

After unzipping the folder, it can be run with the following commands. This commands were run in bash, and the backslash means that the command continues in the same line. This behaviour might be different depending on the system were it runs

```
> cd dependencyinjection
> mvn clean compile package
> mvn org.apache.maven.plugins\
:maven-assembly-plugin:2.2-beta-5:single
> java -jar \
target/dependencyInjection-jar-with-dependencies.jar
```

Listing 5.1: Running the example code

6

Conclusions

Non-functional requirements are crucial in the Enterprise Systems, because they determinate how successful will be the application in medium and long terms. This criteria are the quality goals for the systems. One of the non-functional requirements is Maintainability, which represent most of the Software's Life cycle.

As the research proved, dependency injection has a great impact on maintainability, because it helps to reduce the coupling, increase the cohesion and testability.

According to the research done, we can conclude that it is not true that Dependency Injection will reduce the maintainability in all the types of projects or Frameworks. When projects become too big and we are handling the Dependency Injection using XML files, the XML files tends to also be too large, and it will mean more hours maintaining it. It is proved by the study published in the International Conference of Software Engineering and Applications. [2]

We can conclude that using Dependency Injection with a Framework in small and medium systems is a good practice, and despite having a bigger learning curve with a Framework as it can be clearly see in the example given, they help to keep a more organized code. Frameworks force to the developers to learn new libraries, however, in a long term they make the development process faster. Frameworks usually have annotations or automatic detection of dependencies which make the complete process simpler.

Bibliography

[1] Chen, Lianping; Ali Babar, Muhammad; Nuseibeh and Bashar. *Characterizing Architecturally Significant Requirements.* IEEE Software, 2013.

[2] Ekaterina Razina, David Janzen. *EFFECTS OF DEPENDENCY INJECTION ON MAINTAINABILITY.* International Conference SOFTWARE ENGINEERING AND APPLICATIONS, November 2007.

[3] *Solution Developer Fundamentals - Dependency Injection.* Online tutorial, April 2013.

[4] Stellman, Andrew; Greene, Jennifer *Applied Software Project Management.* 2005.