Dependency Injection frameworks: an improvement to testability?

Anders Lundberg
Abstract

Testing is a crucial part of any software project. Its importance can be seen in the increasing amount of developer striving towards producing code with higher testability, thus being able to verify and validate the functionality of their systems. The design and structure of the code is thereby very important, to incorporate testability at a satisfying level. Dependency-Injection (DI) is a way of reversing the dependency flow between objects in a system, to make them more isolated and easier to test. As DI will help the developer to build code in a good object-oriented way by promoting thought through structure, this will lead to higher testability in the produced code.

This report gives a overview of what testability is, if it can be measured and if DI frameworks can be beneficial from a testability point of view. By drawing conclusions on previous studies and work done in this field, these questions have been answered. Also a system for calculating dependency and coupling between objects is also presented, and shows that the conceptual use of DI gives more testability. The paper concludes that the use of DI frameworks will be beneficial for a systems testability, although DI itself is the major factor and not the frameworks. However as the frameworks make the use of DI that much easier there will be a significant increase in testability with the use of them.
Acknowledgements

I would like to thank my wife and two children for putting up with me during these four years of study, hopefully from now on there will be more Lego building with the sons. A special dedication to my father for his ongoing love and support and my mother who could not see this thesis completed.

I also want to thank my supervisor Dr. Andrew Wallace, for helping me straighten out many question marks along the path of this project. Furthermore my external-supervisor Karl Lindmark also deserves some credit for helping me out. Last but not least I would like to give credit to Christer Jakobsson, for helping me reason with all my stupid questions and giving me valuable feedback.
Contents

1 Introduction 1
   1.1 Purpose of this thesis 1
   1.2 Disposition 2

2 Method 3
   2.1 Q1: What is testability ? 3
   2.2 Q2: Can testability be measured ? 3
   2.3 Q3: Will the use of DI-frameworks improve testability ? 3

3 Dependency Injection: a detailed overview 5
   3.1 Dependency Inversion principle 5
   3.2 Inversion Of Control 6
      3.2.1 Interface Inversion 6
      3.2.2 Flow Inversion 7
      3.2.3 Creation Inversion 8
   3.3 Dependency Injection 8
      3.3.1 Constructor Injection 9
      3.3.2 Setter Injection 9
      3.3.3 Interface Injection 10

4 Dependency Injection frameworks 11
   4.1 Ninject 12
   4.2 Spring.NET 14

5 Testability of Code 17
   5.1 Measuring testability 19

6 Conclusion 25
   6.1 Q1: What is testability ? 25
1 Introduction

Testing software is crucial for the developers to ensure that the program does as intended and will function well together with already implemented systems. As modern systems tend to grow very big, the modularization of programs is essential as features are removed and added along the programs life-cycle. This will make the testability of produced code a key factor to ensure that what is added now is right and it works in conjunction with the already existing system. If the objects in the system are too tightly coupled it will make testing much harder as the object depends on other objects to display its functionality. This leads to pinpointing bugs and faults in the code base to be very hard, so it is desirable to isolate objects as much as possible.

Inversion of Control is a programming pattern for reversing execution flow in a system. Often this will be done in the reversal of execution flow from the code to the user/framework. Dependency Injection (DI) is one implementation of the Inversion of Control (IoC) pattern to utilize the reversal of control, and this to decouple and isolate classes/objects.

1.1 Purpose of this thesis

This thesis will in a theoretical way discuss and reason about what testability is, what defines it and wheter or not it can be measured in some way. Furthermore this thesis aims to discover if the use of DI frameworks is advantageous from a testability perspective. Although DI can be used manually without these frameworks, it will be a very tedious and repetitive work on large-scale systems. So Ninject and Spring.Net, two DI frameworks with different approaches to DI will be discussed and their usages exemplified.

**Summarization of the questions this thesis is trying to answer:**

Q1. What is testability ?

Q2. Can testability be measured ?

Q3. Will the use of DI frameworks improve code testability ?
1.2 Disposition

In Chapter 2, the method used in this thesis is described. Chapter 3 is a detailed overview of Dependency Injection and its concepts. A description of Dependency Injection frameworks will be presented in chapter 4 along with code examples\footnote{All code examples in this thesis are written in C#}, as well as how Ninject and Spring.Net utilizes Dependency Injection. Chapter 5 discusses what testability is and show an example of measuring structural change in a system. The conclusion of the previous chapters are found in chapter 6, that will provide answers for the 3 questions stated in chapter 1. Finally chapter 7 will provide a discussion of the thesis and future work.
2 Method

To answer the questions stated in the introduction, there was mainly a reasoning about what conclusions and statements that other writers of articles discovered. All facts and material for the conclusion of the report were gathered through articles and reports that have been published through IEEE, or in a few cases online sources with high credibility.

2.1 Q1: What is testability?

By gathering information and articles containing already made investigations in the different approaches of defining testability, the facts were put together and converging points of testability was considered as a common consensus of the different parts of testability.

2.2 Q2: Can testability be measured?

Also here the approach was to gather already made investigations and material of the subject. The different approaches of how to measure testability found were cross-referenced with facts found about what testability is, in the previous question. One concrete approach for how to measure testability were made as a conceptual example, because of the theoretical approach for this paper there was no real test case scenario to calculate. Instead a couple of typical structures where calculated using the method suggested by [1], this to make a point of what DI can bring to testability and how that correlates with other works conclusions.

2.3 Q3: Will the use of DI-frameworks improve testability?

This part was concluded from the findings in the previous two parts. With reasoning of how the different parts of testability are influenced by DI the conclusion was founded. It was also considered what general DI can accomplish against the use of frameworks that specializes in DI, primarily intending to establish what the frameworks contributes to the testability.
3 Dependency Injection: a detailed overview

As DI is one way to implement the IoC design pattern. The concept of IoC will be explained first and then a more in-depth explanation of DI will follow, describing its different ways to incorporate injection.

3.1 Dependency Inversion principle

Dependency Inversion Principle (DIP) [2] is a principle for inverting dependencies between objects in different layers in a program structure, and is a part of the general principles for agile design. The principle has two keystones:

- **High-level modules should not depend on low-level modules. Both should depend on abstractions.**
- **Abstractions should not depend upon details. Details should depend upon abstractions.**

This means that a high-level object like the Car class in Figure 1 should not be dependent on low-level objects. As seen in the example the Car has to take into consideration how the Engine and FuelInjection API works, in order to utilize them in a proper way. Worth mentioning is that every time a new low-level dependency is added to the Car, it has to be altered to accommodate the new dependencies.

With DIP in mind, the dependency would be inverted so that the Car defines how the dependencies should behave instead of the other way around. As Figure 2 displays, if the Car class defines the API with interfaces for the Engine and FuelInjection. The low-level implementations are dependent on the interfaces instead of the opposite (as seen in Figure 1), thus reversing the dependency flow for the Car class. With this behavior the Car will not be affected on changes in the low-level objects.

The statement that "Abstractions should not depend upon details. Details should depend upon abstractions", is fulfilled as the implementations of the interfaces defines the details, the code that make the implemented functions work. This means that the low-level object should decide how to implement the functions that the high-level object has declared.
3.2 Inversion Of Control

IoC is a design pattern based on DIP, that inverts dependency control to an exterior location or instance. IoC can be implemented in a number of ways, the most common and well known adaptations being the Factory, Service Locator and DI patterns [3].

According to Champatirays article on DI and IoC [4] and [5] there are three usual utilizations of IoC: Interface, Flow and Creation -inversion.

3.2.1 Interface Inversion

Even if the low-level objects depend on interfaces, or if there is an interface for each low-level object, there will still be a dependency to the actual abstraction(as each abstraction has to be treated separately). By gathering the existing interfaces into a single of fewer higher abstractions, Interface Inversion is able to help avoid the objects being over-coupled. For instance in Figure 3, both the NGK and KehinCarburator objects follows the high-level object Engines definition of abstraction. Even though they are used in the same fashion they are separate abstractions. Interface inversion gathers them under a common abstraction as in Figure 4 where the low-level objects are implementing the same abstraction IEngineComponent.
3.2.2 Flow Inversion

This will invert the flow of control, from the code/program to the user or an external framework. For instance, in a GUI that is code controlled, the user will be prompted by the program regarding what the input should be and the order of inputs to be made as in Figure 5. With flow inversion, control is inverted from the code and the user would be presented with all the different input fields and be able choose the order to fill them out as in Figure 6.

Figure 3: Engine dependent on two interfaces that performs similar tasks, thereby has to be treated as two separate abstractions

Figure 4: With Interface inversion, gathered as IEngineComponents instead

Figure 5: GUI with code controlled flow, user presented with input

Figure 6: Inverted flow of control, the user chooses fields and order
### 3.2.3 Creation Inversion

Creation Inversion aims to invert the creation of objects. Instead of a high-level instance creating an object, the object is created and received from another exterior instance. The Factory and Service Locator patterns define this type of inversion, as the creation of the objects are made in a dedicated environment. If the setup in the factory to create a Camaro is changed or replaced in the Code_ex 3.1 below, the Driver class can still remain unchanged as all the responsibility for the creation and initialization is directed to the external instance (in this example the CarFactory).

<table>
<thead>
<tr>
<th>Code_ex 3.1: Creation inversion performed with a factory</th>
</tr>
</thead>
</table>
| public class Driver {
  private Camaro car;
  public Driver() {
    car = CarFactory.GetCamaro();
  }
} |
| public static class CarFactory {
  public static Camaro GetCamaro() {
    return new Camaro(...);
  }
} |

### 3.3 Dependency Injection

The idea of DI is an implementation of IoC and will inject dependencies into objects from an exterior assembler, rather than creating or gathering them from a Factory or a Service Locator inside the object. For instance in code-Code_ex 3.2, the Engine class will create and hold the reference to the Carburator that is a part of the engine, which is not good dependency wise. Even if the design is altered so that the Carburator should be acquired from a Factory as in Code_ex 3.3, there will still be issues to deal with. If the Factory would be changed or replaced the Engine class would have to change and adapt to the new Factory. This is where DI comes into play, as the Engine class can be designed so that the constructor takes an argument of the Carburator instead as in example 3.4. Then the creation or gathering of the Carburator class is totally separated from the Engine class, as it is supplied without concern from where or who the creator is. This injection could also be made by a setter instead of passing it in as an argument to the constructor [2] [6].

<table>
<thead>
<tr>
<th>Code_ex 3.2: Creation of an dependency object</th>
</tr>
</thead>
</table>
| public class Engine {
  private Carburator carb;
  public Engine() {
    carb = new Carburator();
  }
} |
There are three types of injections that can be made when implementing DI from IoC, sometimes referred to as type 1 IoC (Interface injection), type 2 IoC (setter injection) and type 3 IoC (constructor injection) [5].

### 3.3.1 Constructor Injection

This is the most commonly used way to make use of DI, as can be seen in code ex: 3.4 the constructor will take the dependency as an argument. As discussed in [5], the best practice would probably be to use this type of injection when possible. This to achieve the creation of complete and valid objects at construction time.

### 3.3.2 Setter Injection

This is also a very common approach to use DI, as seen in code ex: 3.5 the dependency is injected through a setter in the receiving object. If there are many dependencies that are to be injected into an object, this might be the better choice as there will be a setter for each dependency, which in itself is more desirable than a constructor with a large number of arguments.
3.3.3 Interface Injection

Interface Injection works similar to the setter injection, but the setter functions are added from implementing interfaces instead. For instance, to use this approach on Code_ex 3.5, the setter function will be moved outside to an external interface that Engine has to implement to be able to gain the setter function as in Code_ex 3.6.

Code_ex 3.6: Interface Injection on the Engine class

```java
public class Engine : InjectCarburator {
    private ICarburator carb;

    public void injectCarburator(ICarburator carb) {
        this.carb = carb;
    }
}

public interface InjectCarburator {
    void injectCarburator(ICarburator carb);
}
```

Although some DI frameworks like Avalon use this approach to perform injection, the general consensus is that it is the least used method of the three mentioned in the comparison.
4 Dependency Injection frameworks

When the systems start to get bigger and the dependencies are numerous, coding manual injection can be very tedious and excessive. Here comes the real benefit from DI frameworks, as they provide an easy way of dealing with numerous and nested dependencies.

The basic principle of these frameworks is to construct modules of the dependencies that are to be used for the system. These modules are processed by an assembler that creates the necessary dependencies and injects them into the right place when creating the objects. To display how module injection functions see Figure 7. To create the dependencies for the Camaro module, the assembler is loaded with the module, the assembler creates those objects and injects them into the Car.

Figure 7: Assembly from modules

The system with modules is similar to the Factory pattern, but as discussed by Jenkov [7] the modules have an advantage over the factory. If the factory/module has to be changed or replaced, that will affect the base class using the factory. But with use of DI frameworks the dependency injection is outsourced to the assembler instead.

The modules will also be easier to maintain especially in the case where the modules are XML based, as they will be managed in a separate file without interaction with the code. Also multiple dependency nesting would be easier to use and maintain with modules, by the fact that modules can be composed of other modules and so on, for instance the engine in Figure 7 could have multiple camshafts and headers to complicate the structure. Then the engine module could consist of modules setting up the header, as well as a module with the camshaft to be used as seen in Figure 8. This nesting can be made very complex, but still very maintainable and understandable.
Figure 8: Nesting of modules for DI frameworks

4.1 Ninject

Ninject is a lightweight DI framework created by Nate Kohari [8], that has a fluent interface for producing the modules. The modules are produced directly in the code as the framework in itself does not directly support the use of XML. With some additional coding the modules can be stored in XML form and parsed/translated into ninject code, but the main focus is on the fluent code interface that is to be used.

As can be seen in Code ex 4.1, the CarBuilder creates a new Camaro which in turn will create a SuperCharger and a V8. This would be a poor choice of design as the dependencies will be too many. To solve this with use of the Ninject-framework the Camaro class has to be altered to adapt for use with DI, as can be seen in Code ex 4.2. The engine and fuelinjector are now implementations of Interfaces and the Camaro class has a constructor that takes a fuelinjection now, as well as a setter for the engine. Although the setter is unnecessary as the fuelinjection should be injected through the constructor as well, the setter is only to display the ease of which the injection is done.
**Code_ex 4.1: "No-DI: Building a Camaro"**

```java
public class Camaro {
    private SuperCharger fuelInjection;
    private V8 engine;
    public Camaro() {
        fuelInjection = new SuperCharger();
        engine = new V8();
    }

    public class CarBuilder {
        private Camaro car;
        public void BuildCamaro() {
            car = new Camaro();
        }
    }
}
```

**Code_ex 4.2: rebuild of the code to adapt to DI**

```java
public class Camaro {
    private IFuelInjector fuelinjection;
    private IEngine engine;
    public Camaro(IFuelInjector fuelinjection) {
        this.fuelinjection = fuelinjection;
    }

    [Inject]
    public void setEngine(IEngine engine) {
        this.engine = engine;
    }
}
```

With the Camaro class setup, the actual module for building a Camaro is created, see Code_ex 4.3. Here the command `Bind<Interface>().To<Implementation>()` decides what concrete implementation that is being used for the provided interface in the code. In Code_ex 4.4 the assembly is started, the kernel is initiated with the CamaroModule from Code_ex 4.3, and the assembled car is returned.

The binding of the engine and fuelinjection that is made in the ninject module in Code_ex 4.3 is matched against the different options that the Camaro class can offer. As the constructor can take a IFuelInjection as an argument the assembler injects that object there, the same goes for the engine that is injected through the setter. The assembler will automatically detect if the object can be injected into the module and where, first by the constructor and secondly into setters marked with Inpect in the code.

**Code_ex 4.3: "Ninject: CamaroModule.class"**

```csharp
using Ninject;

public class CamaroModule : NinjectModule {
    public override void Load() {
        Bind<IEngine>().To<V8>();
        Bind<IFuelInjector>().To<SuperCharger>();
    }
}
```
4.2 Spring.NET

Spring is based on the Java version of the Spring-framework, and applies DI by using a module system in XML form. There the modules will be defined and can be composed into larger modules, much like a Lego system where building blocks can be stacked into a larger component.

If the same example as with the Ninject framework is used (see Code_ex 4.4), it has to be prepared to use DI, as before seen in Code_ex 4.2. This with a slight tweak, the [Inject] annotation indicating that the setter accepts injection, is not needed when using Spring. The setters name will be stated in the XML, instead of the framework trying to find the marked setters for matching as Ninject does.

The XML in Code_ex 4.5 displays how the modules are setup. The Engine- and FuelInjector modules are associated with the implementations that are to be used for them, indicated by the (type="Class, Namespace") line. The Camaro-module defines which Engine- and FuelInjector module that is being used for that particular module. The separation of where the modules are injected into the Camaro.Class is stated in the XML, as the Fuelinjector is injected as a constructor argument and the Engine is injected through a setter instead.

As seen in Code_ex 4.6, the actual assembly of the car starts by first parsing the XML into the assembler, and subsequently defining which module present in the XML is to be used to create the car.
```csharp
using Spring;

public class CarBuilder {
    private ApplicationContext context = new XmlApplicationContext("CarSortiment.xml");
    public Car Car { get { return (Car)context.GetObject("Camaro"); } }
}
```

The approach with XML that Spring.Net utilizes is quite easy to manage, as the only section that is needed to be altered as new modules or setup of those emerge is in the actual XML file [9].
5 Testability of Code

Although there are many definitions and ideas for what testability is and how to define it, the general definition established from the IEEE glossary of software terminology states: *The degree of which a system or component facilitates the establishment of test criteria and performance of tests to verify whether those criteria have been met* [10]. In reality it means that if it can be verified that the code meets the test criteria set for the code, then the code has good testability. This presents a problem, because if the code written have met the criteria, then we would have made good and testable code. But how can we make decisions/design that will make the code have predetermined high testability before implementation.

Even if the notion of testability seems like a concept that would be fairly easy to grasp and achieve, there is a lot more to consider when designing for testability. As Binder [11] shows in his article, there are numerous aspects that will affect the level of testability as can be seen in figure 9. The six main branches that will affect the testability is in broad strokes:

1. **Representation**, is what the class/system is representing, the expected output, and what it is intended to perform. Without this information the testing will be very exploratory and more or less a guessing game of what to test.

2. **Test suite**, defines the actual testing and its suites, as how the test are designed will greatly affect the testing. Are the test reusable? Could they be automated? How efficient and precise are the oracles?

3. **Implementation**, defines the structure of the program, inheritance and dependency flows. Further topics for this branch is how deterministic the flow of execution is, as well as how exceptions are handled. All those topics will have a great affect on the whole testability.

4. **Test tools**, The lack of automation can cause a declination in the frequency of testing due to it being too much manual work, which in the end could manifest as an increased overall cost for the testing.

5. **Built-in test**, will deal with how the drivers function, the set/reset states of the tested objects to ensure a predetermined state to appear, as well as how the reporters can obtain the state of the objects under test.

6. **Process Capability**, is about how mature the overall process is, to incorporate testing and integration into the process. Designing, inspections, reviews and staff experience are things that need to be considered for the overall testability.

---

1 Mechanism for determining whether a program has passed or failed a test
2 Simulation of a higher-level module calling low-level objects, used with bottom-up testing
3 Code that can return an objects state for examination
Figure 9: The different criterias for achievement of testability, as defined by Robert V. Binder [11].
The testability structure that Binder [11] displays is the overall testability for a process from start to finish, showing all parts that could affect testability from the design of the system to the experience level of the testers. But in this thesis the focus will be on inheritance that is a part of the structure subbranch on the implementation branch seen in Figure 9, as the inheritance aspect is the most likely to be affected by the use of DI-frameworks. The complete figure with all the branches has been displayed to present the reader with the complexity of the definition of testability, as it incorporates a huge amount of different factors that will affect the definition of testability stated by IEEE [10].

There are a few more different approaches on how testability should be defined or achieved, and they could probably all be fitted somewhere into the testability tree above. Voas and Miller [12] defines that one part of testability would be the probability that a test will fail if the program has a fault. The code written should be designed so that faults don’t hide and go unnoticed, instead their approach is that programs should fail in a clear and noticeable way when a fault occur. In the article [13] Mouchawrab et.al point out different parts of testability attributes and what effect they have in different categories that will enhance the overall testability. They also state that coupling is a testability attribute that will affect both unit- and integration tests, because high coupling will make production of stubs more difficult. Joshi and Sardana [14] shows when testability improvement should take place, design- or code-time. Which is important, to decide what measures to consider in the design phase and what to leave to the implementation phase.

5.1 Measuring testability

A general measurement for testability is hard to obtain as the definition of testability is too large and abstract to pinpoint into a specific number. Although there are some measurements that can be made for different parts of testability, they are, mostly found on the structure and design side as the conceptual ideas are easier to generalize and measure.

Dietrich and Gottlieb [1] propose a way to measure the dependencies and inheritance as an indication for more refactoring. The metrics system they suggest is a measurement of how dependencies between classes are nested in a SUT (System Under Test). The equation for calculating (see Equation 5.1), will generate a value $T$ for each class where ($0 < T \leq 1$). If $T$=1 it would indicate that the class is very testable, and implies that it is very isolated and can be tested more easily. Binder [11] shares the same view, stating that good object-oriented design will promote testability.
The $T$ value is a score depicting what level of code coverage can be obtained with tests on the code. It will show that with lesser coupling and direct dependencies between objects there will be a much more test friendly environment in the SUT. Williams [15] also concludes that there is a correlation between code coverage and how many faults that are found in code.

\[
T(C) = \frac{1 + 5C_f + 4C_j}{1 + 5C_n \cdot \log_2(C_n + 1) \cdot (C_nN)!} \cdot \prod_{D \in C} T(D) \tag{5.1}
\]

$C_f$ Number of injected interface dependencies  
$C_j$ Number of injected non-interface dependencies  
$C_s$ Number of static-state dependencies for C  
$C_n$ Total number of dependencies  
$N$ Total number of classes in the SUT

Jungmayr [16] also displays a metric that can be calculated to find critical dependencies that are undesirable. The method will help to find out if the most test-critical dependency has been removed from a SUT and how many stubs that is needed to break the dependency cycle found. The approach is similar to what [1] shows, but is more focused on finding the dependencies that cause the problem, instead of measuring after each re-factorization to find if there was a positive improvement.

Similar to Jungmayr, Baudry [17] determines the testability of a SUT by analyzing the complexity of the UML over the SUT. This also is based on a dependency graph of the SUT, to calculate the complexity of dependence paths and how the inheritance structure is built. More examples of measuring structural changes can be seen in the articles from Chidamber [18], Bruntink [19] and Briand [20]. All of those also shows with small differences how to calculate the structure and inheritance, which seems to be most easy and common way to measure testability.

With the use of the metrics suggested by [1], a conceptual structure is made example of, to display one way of measuring subparts of testability. The SUT in Figure [10] shows a naive structure for a car that contains an engine and a fuelinjection.

---

4Substitute for an object during testing
Figure 10: Structure of a car, with no use of DI. Dependency flow indicates that the car is dependant on the components.

In Figure 11, the breakdown for each class can be seen as well as the calculated values using Equation 5.1. The number of dependencies for the classes are quite low in this really naive example. Although the problems with the dependencies in Figure 10 are displayed, as the low score for both Car.class and EngineV8.class will tell. As both classes depend on low-level classes created by them, testing Car for instance would mean that EngineV8 and SuperCharger also has been tested with the Car as a single module.

<table>
<thead>
<tr>
<th>Class</th>
<th>T(C)</th>
<th>Cf</th>
<th>Cj</th>
<th>Cs</th>
<th>Cn</th>
<th>T(D)</th>
<th>Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0.17</td>
<td>EngineV8, SuperCharger</td>
</tr>
<tr>
<td>EngineV8</td>
<td>0.17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>Header1</td>
</tr>
<tr>
<td>SuperCharger</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>-</td>
</tr>
<tr>
<td>Header1</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 11: Calculation of the T-value for the SUT in Figure 10

After altering the previous SUT and adjusting it to comply with use of DI, see Figure 12. Both IEngine and IfuelInjection interfaces has been added to comply with the rules from DIP, see Section 3.1. High-level modules should not depend on low-level modules, both should depend on abstractions. Interfaces have been introduced to keep the dependencies at the same level, both in the case for Car and V8. This causes the concrete implementations to move down a level, and also depend on the interfaces in question.
Calculating the DI fitted SUT reveals that the testability level has risen significantly, see Figure 13. The poor value from Table 11 at 0.01 for the Car has changed into a much better 0.65, indicating that the code coverage for that SUT would probably be much higher. Although there could be done more to the SUT, if the interface inversion from IoC would be implemented (see Section 3.2.1).

<table>
<thead>
<tr>
<th>Class</th>
<th>T(C)</th>
<th>Cf</th>
<th>Cj</th>
<th>Cs</th>
<th>Cn</th>
<th>T(D)</th>
<th>Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td>0.65</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1.00</td>
<td>IEngine, IFuelinjection</td>
</tr>
<tr>
<td>V8</td>
<td>1.00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>IHeader</td>
</tr>
<tr>
<td>SuperCharger</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>-</td>
</tr>
<tr>
<td>Header1</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 13: Calculation of the T-value for the SUT in Figure 12

To achieve interface inversion, the engine and fuelinjection interfaces are replaced with a IEngineComponent interface. Now the SuperCharger and the V8 will be concrete implementations of that interface, see Figure 14.
Recalculating the SUT with interface inversion applied, results in that the score for the class is even higher this time. Due to the fact that Car only depends on one interface now, the score will be at 1.00 for all classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>T(C)</th>
<th>Cf</th>
<th>Cj</th>
<th>Cs</th>
<th>Cn</th>
<th>T(D)</th>
<th>Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td>1.00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>ICarComponent</td>
</tr>
<tr>
<td>V8</td>
<td>1.00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1.00</td>
<td>IHeader</td>
</tr>
<tr>
<td>SuperCharger</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>-</td>
</tr>
<tr>
<td>Header1</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
<td>-</td>
</tr>
</tbody>
</table>

**Figure 15:** Calculation of the T-value for the SUT in Figure 14
6 Conclusion

In this chapter the conclusion to the three main questions for this thesis will be presented. First the question of what testability is will be resolved and concluded. Secondly there will be a conclusion if it is possible to measure testability. Last but not least, the question if the use of DI-frameworks will be beneficial for the testability will be answered.

6.1 Q1: What is testability?

As the official definition [10] states, testability in general is a really large and vast area. However as Binder [11] shows in his article, there are several fields in the process that are all a part of the testability definition (see Figure 9). The spectrum goes from project-start with the specifications and requirements all the way down to the type of assertions that have been chosen for a specific test. As all of these small pieces together will form a stronger and stronger testability for the code created, they are all important pieces in the huge puzzle that a larger project is.

Binder [11] also states that high coupling will make it more difficult to control a class under test. This correlates well with Jungmayr [16] who concludes that introducing a component with low coupling to a component with high coupling would be a test-critical dependency. Those type of dependencies are preferable to remove, as they complicate and generate a more hostile test environment. Both these articles discusses the troubles that dependencies can create for testing. Mouchawrab [13] follows Binder’s thoughts and also concludes that coupling is a testability issue. They define coupling of objects to be a testability attribute that both will affect unit- and regression testing. Joshi [14] also shows that in design for testability at code-time, DI will be a beneficial method to use.

The conclusion is that, with support from the studies mentioned above and in the context of DI, testability is low coupling between objects and simple dependency structure.

6.2 Q2: Can testability be measured?

Calculating testability in general would probably be impossible as there are too many and abstract parameters for the big and general testability definition. But in small subparts it would be quite possible to calculate values for how good chance there is that the testability will be enhanced for a specific aspect or method. As [1], [18], [19], [20] all give examples on how to measure small subparts of testability, all of those focus on measuring the structure and design of code.

To measure the concluded definition of testability in the context of DI, there all of the above mentioned metrics would be sufficient. An example of how to calculate a testability score
based on the coupling of objects in a SUT (see Section 5.1). The metrics suggested by [1] is applied and calculated there for a conceptual SUT, displaying one way to calculate the structure of a system. Jungmayr [16] also suggest an approach to measure coupling in a SUT, looking for dependency cycles and objects with multiple dependents. The method will calculate if the most test-critical dependency has been removed from the SUT, while removing dependencies.

As the articles above suggest there are multiple ways to measure various parts that will affect the overall testability defined by the IEEE [10]. The conclusion will then be that, for the general definition of testability, it will not be measurable, but potentially possible for subparts of the testability definition, like suggested by Binder [11] in Figure 9.

So to conclude the question, yes there are multiple ways of measuring subparts of testability.

6.3 Q3: Will the use of DI-frameworks improve testability?

To answer this question the question if DI in general will improve testability would have to be answered first. The conclusion drawn from the first question of how to define testability gives an indication of the answer to this question. As the concluded definition of testability in aspect of DI is low coupling, and DI by definition is all about minimizing coupling between objects.

As Fowler [5] display in his examples of DI, if used correctly the coupling of objects will be reduced. According to Chinnaya [4] the backbone in the IoC principles are the DIP [2] shown in Section 3.1. DIP states that high-level object should not depend on low-level objects, both should depend on abstractions. Given that the objects should depend on abstractions there will be a decoupling of the objects, which is the intent with the IoC principles.

The result in Section 6.2 using the calculation metrics from [1]. With DI implemented there will be an improvement in testability, as the metric is correlated to what code coverage that is expected from the SUT. Williams [15] shows that there is a correlation between the number of faults found in code and the percent of code-coverage. Finding and eliminating faults in the code-base is one of the primary goals of testability, so that would imply that DI is beneficial to testability.

So with all that fact combined with the concluded definition of testability in the context of DI, the conclusion will be that DI will improve testability.

Although it is not really the frameworks that will be the key factor for improving testability, the general DI is the key here. As the frameworks help implementing DI structured and more easily in large systems, there will be a major advantage using them compared to manual DI.

So the conclusion will be that DI-frameworks will indeed be beneficial to the testability of a system.
7 Discussion

Finding relevant material for this paper was quite easy as the subjects involved are not especially new topics. But as the testability area is so large and vast, there had to be some limitation of how to define it, thereby the conclusion defining testability in this paper is in the context of DI. To define testability in detail would probably be a quite large work for a couple of persons.

7.1 Future work

As this paper focus on the theoretical side of DI and testability, the next step would be to apply some of the mentioned metrics to code. A comparison between the different metrics like [19], [16], [1], on a larger code base would be interesting to find out if there are any method that stands out in locating how testable the code will be. Also investigate how these methods could be applied during a real coding project, in order to constantly review if the code produced follows the requirements. Although maybe it would be too time-consuming to evaluate the code by these metrics, it would be interesting to see if a gain in code quality could be obtained from it.

It would also be interesting to investigate how to combine multiple different testability measurements to find a metric that could be more compliant with the IEEE definition of testability [10]. Although it would probably be a lot of work due to the high amount of parameters to consider, it would be an interesting topic to read more about.

7.2 The benefits of DI-frameworks

As the general testability gain are obtained from the use of DI principles, and not the actual frameworks. Questions can be raised whether its beneficial to use these, as the perceived overhead could be intimidating. Although there is some overhead by using them, as setting up modules, arranging XML structures and inclusion of libraries. The small effort to start using the frameworks would be greatly compensated by the benefits, not merely by the fact that complicated structures will be more manageable. But because the use of these frameworks will urge the developers to maintain good object orientation for the code, as otherwise the module system cant be utilized.

Maybe these frameworks could be applied for educational purposes also, to stimulate students to make use of the DI principles, in order to comply with the frameworks. On the downside it could be easy to neglect what the framework do in the background, so vital information that needs to be understood gets lost. But probably the introduction of this kind of frameworks would be introduced after the general concepts are learned.
References


