**University Of Amsterdam**
Faculty of Science

Master Thesis Software Engineering

---

# Testability of Dependency injection

An attempt to find out how the testability of source code is affected when the dependency injection principle is applied to it.

---

**Ricardo Lindooren**

UNIVERSITEIT VAN AMSTERDAM

venspro

| | |
|---|---|
| Student number: | 5636078 |
| Host Organization: | Venspro |
| Thesis Supervisor: | Dr. Jurgen J. Vinju |
| Internship Supervisor: | Drs. Johan van Vulpen |
| Availability: | Public domain |
| Date: | 10 September 2007 |

# Preface

Dear reader, the document you are reading right now is my final thesis with which I conclude the one year during Software Engineering master program that is offered to students by the University of Amsterdam.

The reason to sign myself up for this master program was the fact that after two years working fulltime as a developer I came to the conclusion that there was more to creating a good software product than just writing the code for it. This conclusion made me realize that in order to increase my skills and professionalism I had to increase my awareness of the aspects that together form the complete software engineering process.
While I'm writing this as the last part of my final thesis document I truly believe that my awareness of the software engineering process has increased. I hope that my final thesis document proofs this awareness to you, the reader.

For me there is no doubt this could have been a more complete document, especially when it comes to the actual research I've performed to base my conclusion on. The reason for this, is that during the available period of time that students have to work on their final thesis research I decided to change my research subject because I believed the research subject I started out with did not have a concrete link with the software engineering process. I was convinced that when I continued my original research it would result in writing a final thesis document that I could not ever be satisfied with.
This meant I had to find a new research subject as well as a company giving me the change to perform a new research within the limited available amount of time that was still left.
Because of this limited available amount of time, it took a lot of hard work to complete my final thesis document before the original deadline. And although it may not be as complete as it might have been if I had started with my second research subject right away, I'm still reasonably satisfied with the content that I was able to produce.

Working on my thesis research was a true learning experience for me. It has given me a better understanding of what scientific research actually is. It especially has given me a lot of respect for people that dedicate themselves to collecting facts about any research subject and sharing these with other people in the world making it possible to learn from.

## *Acknowledgement*

At the beginning of the program all students were warned that it was going to require a lot of work from them to successfully complete the Software Engineering master program within one year. Looking back on this year I can confirm it really did require a lot of work, dedication and motivation. But not only from the students! The driving force behind the Software Engineering master program are people that are truly dedicated to offer it in the form of a one year program and also in the best way possible.
Therefore I would like to thank the following persons for investing their time in the Software Engineering master program; **prof.dr. Paul Klint** – course: Software Evolution, **prof. dr. Hans van Vliet** – course: Software Architecture, **dr. Patricia Lago** – course: Software Architecture, **prof. dr. Jan van Eijck** – course: Software Testing and **Peter van Lith** – courses: Software Construction and Software Process. In particularly I would like to thank **drs. Hans Dekkers, Msc**. – course: Requirements engineering and **dr. Jurgen J. Vinju** – course: Software Evolution. I would like to thank Hans for his support to students during many of the practical lab sessions. I would like to thank Jurgen for his support as my thesis supervisor.

The person that made it possible to finish my final thesis research, as well as writing this document, before the original final deadline is **drs. Johan van Vulpen**. I would like to thank Johan for giving me the change to do my research at Venspro after making only one phone call to him.

Last, but certainly not least, I would like to thank **my parents**. Without their support I probably would not have started with the Software Engineering master program in the first place.

## Summary

Before starting with this research my hypothesis was that *test automation will become more complex when using a dependency injection (DI) solution*. Suspected reasons for this increase of complexity where that (1) *the DI solution should be configured for each test* and (2) *it may even be impossible to use a DI solution in a test environment since DI solutions act as a managed environment on their own*. Both of these reasons, in my opinion, could be seen as how a DI solution can intrude a software product and especially its test environment.

Both reasons (1) and (2) proved not to be true during this research. The cause for both reasons not being true is that the components implemented with DI during this research where done so with either the constructor or setter method DI strategy. This means that dependencies can also be injected without having to make use of a DI solution. From within a JUnit testcase it proved to be no problem injecting, for example a mock object, as a constructor or setter argument to the component under test without having to make use of a DI solution. No proof could be found that the DI solutions used during this research did intrude, or dictate, the test environment.

---

In order to do a research focusing on how DI intrudes the software testing environment first of all an attempt has been made to find out what the effects of using DI are on software testability in general. Therefore, based on a literature study, a description has been given of what software testing is in general and what makes software testable (appendix A). As well as what dependency injection is and how it affects a software product on component/source code level (appendix B).
The two outcomes of this literature study (what makes software testable & how DI affects a software product) have been brought in to relation with each other (chapter 2). Bringing them both in to relation with each other resulted in the ability to make assumptions of how DI affects the testability of software product (paragraph 2.1). Based on these assumptions a set of metrics has been selected that could be used as a factual representation of how DI affects software testability and especially the level of intrusion a selected DI solution forms for the whitebox/JUnit test environment of a software product (paragraph 2.2).

The limited scope of this research (2 components implemented with 2 DI solutions), as well as that the extracted metrics did not really give an indication of how testability is affected on component level, mean that the results of this research cannot be seen as a factual representation of how the testability of software is affected when making use of dependency injection.
Under ‘Future work’ I therefore describe a possible second research iteration that assumingly can improve this research when it comes to collecting facts on how DI affects software testability.

# Index

Preface ..... i
Acknowledgement ..... ii

Summary ..... iii

Index ..... 1

1 Context and background ..... 3
1.1 Motivation ..... 3
1.2 Research method ..... 3
2 The effects of Dependency Injection on testability ..... 5
2.1 Assumptions of why and how DI effects testability ..... 5
2.1.1 How DI effects a complete software product ..... 5
2.1.2 How DI effects software testability ..... 6
2.2 How can we make this measurable? ..... 9
2.2.1 Scope definition ..... 9
2.2.2 Measurement definition ..... 10
3 Research results ..... 13
3.1 The paper manager example ..... 13
3.2 The Greetz! customer component ..... 14
3.3 Overview ..... 16
3.3.1 Paper manager example ..... 16
3.3.2 Greetz! customer component ..... 16
3.4 Validation ..... 17
3.4.1 The TLOC metric ..... 17
3.4.2 The McCabe Cyclomatic complexity metric ..... 17
3.4.3 Needed DI configuration in testcases ..... 18
3.4.4 Coverage ..... 18
3.4.5 Test cases ..... 18
4 Conclusion ..... 20
Future work ..... 21

Appendix A - Introduction to Software Testing ..... 23
Why is software tested? ..... 23
Why does software not behave? ..... 23
When is software tested? ..... 24
How is software tested? ..... 25
Planning phase ..... 26
Design phase ..... 26
Coding phase ..... 26
Testing phase ..... 27
What makes software testable? ..... 27
Multiple input possibilities ..... 28
Source code complexity ..... 28
Dependencies ..... 29
Controllability & observability ..... 30
Traceability of requirements ..... 30
Test automation ..... 30

Appendix B - Introduction to Dependency Injection ..... 32
What are dependencies? ..... 32
What are the effects of dependencies? ..... 32
Rigid ..... 32
Fragile ..... 32
Immobile ..... 32
Uncontrollable ..... 33
An example ..... 33
What is dependency injection? ..... 34
From static factory pattern… ..... 35
…To dynamic wiring ..... 35
Current solutions ..... 37
Types of dependency injection ..... 37
Containers / Managed environments ..... 39

Appendix C – The paper manager example ..... 40
Technical design ..... 40
The paper manager interface ..... 41
The paper provider interface ..... 42
The unknown paper exception ..... 42
The abstract paper order ..... 43
Implemented without dependency injection ..... 44
PaperManagerNonDi.java ..... 45
PaperManagerNonDiTest.java ..... 47
Note about the Metrics tool for Eclipse ..... 50
Implemented with PicoContainer dependency injection ..... 52
PaperManagerPicoDi.java ..... 53
PaperManagerPicoDiTest.java ..... 55
PaperCompanyMock.java ..... 58
Implemented with Java EE5 dependency injection ..... 60
PaperManagerBean.java ..... 60
PaparManagerBeanTest.java ..... 63
Note about the Metrics tool for Eclipse ..... 66
What about the other dependencies? ..... 67

References ..... 70

# 1 Context and background

Venspro is a company that creates concepts for the gift and greet branch. Their biggest concept at the moment is Greetz!. Greetz! is an online service allowing customers to design real greeting cards which are delivered -to the recipient(s) of the card- by normal mail. (www.greetz.nl).

The goal of Venspro is to create a worldwide Greetz! greeting card network. Meaning; it should be possible to print Greetz! cards in as many countries as possible. E.g.: a card with Australian recipients created in the Netherlands by a Dutch customer, will be printed in the nearest location of the Australian recipient. This ultimately makes next-day-delivery possible all over the world.

To make this world-wide next-day-delivery approach feasible, a strong and well thought trough software system is needed. Currently Venspro develops and uses Java code which runs in Servlet-container servers to facilitate its Greetz! service in two countries; The Netherlands and Belgium. Next to these two countries Venspro will expands its Greetz! network to England, France and Australia very soon as well.

## *1.1 Motivation*

One of the valuable lessons learned by Venspro over the last years is that it is vital to have a solid software development environment and process. E.g. the Venspro development team has invested in professionalizing their development environment by introducing (Unit) testing combined with a Continues integration strategy.

In theory it is possible that Venspro, in the future, will have to use an Enterprise application development approach. Meaning; instead of developing Servlet-container based Java code, code that runs in Java application servers will have to be developed.

My research springs from the recent introduction of Java Enterprise Edition version 5. Sun has drastically changed their model for Enterprise development, which until version 5 was based on complex code and XML configuration files. Sun has changed this by simplifying the way Enterprise Java Beans are coded. For example by making use of dependency injection based on annotations. JEE5 Applications servers are responsible for this dependency injection behavior when the code is being executed.

The goal of my research is to find out how the testability of software is affected when implementing it with dependency injection (DI). My hypothesis is that *test automation will become more complex when using a dependency injection solution*. Suspected reasons for this increase of complexity are that (1) the dependency injection solution should be configured for each test and (2) it may even be impossible to use a dependency injection solution in a test environment since DI solutions act as a managed environment on their own. Both of these reasons can be seen as how the DI solution intrudes a software product and its test environment.

## *1.2 Research method*

To determine how software testability is affected by dependency injection it is important to first define what software testing is and determine what makes software testable. Secondly it has to be defined what dependency injection exactly is and what effects it has on the source code of a software product.

The outcome of these two research sub-questions have to be brought into relation with each other, so assumptions can be made of how testability is affected by dependency injection.
Based on these assumptions the actual research can be done; relevant dependency injection and testability metrics can be retrieved from different software products, implemented with and without dependency injection. The goal is to use these metrics as facts to form a valid conclusion on how dependency injection affects software testability.

# 2 The effects of Dependency Injection on testability

In order to determine how software testability is affected by dependency injection a literature study has been performed. The goal of this literature study was to find out what software testing is and what makes software testable as well as to find out what dependency injection is.

In this chapter the outcomes of this literature study; Appendix A - 'Introduction to Software Testing' and Appendix B - 'Introduction to Dependency Injection' are brought into relation with each other with the goal to determine how software testability is possibly affected when applying the dependency injection principle to it.

## *2.1 Assumptions of why and how DI effects testability*

In the chapter 'What makes software testable?' of appendix A, six factors that have influence on the testability of software have been defined. These are;

- Multiple input possibilities
- Source code complexity
- Controllability & observability
- Dependencies
- Traceability of requirements
- Test automation

To determine how DI possibly can affect software testability we must try to imagine what the effects of DI are on a software product. A loosely coupled design for example can be seen as the goal of DI. But to realize this with DI means that it will affect a software product in a certain way. The source code for example will most likely be different than when another approach is used to create a loosely coupled design (or than when choosing not to create loosely coupled components at all).

### 2.1.1 How DI effects a complete software product

In appendix B a description is given of what dependencies in software are. It focuses on the negative effects of interdependency between source code components that together form a software product. The following definition is used to define this interdependency between source code components; Component A depends on component B if "correct execution of B may be necessary for A to complete the task described in its definition" [Jackson03].

The chapter 'What is Dependency Injection' describes how components can be changed to break their dependencies by removing logic from a component that defines its dependencies. With DI, this 'which, where and how dependency logic' [Nene05] is not needed in a component that depends on one or more other components. But while this logic can be removed from a component it cannot be taken away from the software product completely. The interdependency of components has to be defined at another place; the DI solution that is used has to be configured.

So when we look at this from the number of lines of code (that are needed to implement the logic of a software product) viewpoint; the lines of code in a depending component will become less, but the lines needed to configure the DI solution will increase.

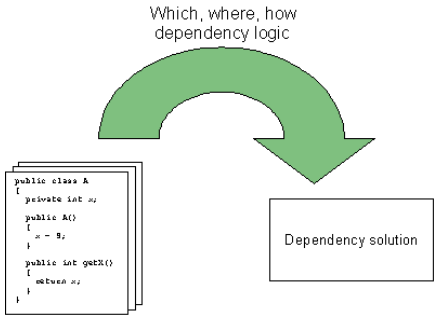



Figure 1: Dependency logic moved from component to DI solution configuration

DI helps creating a loosely coupled design because a component only depends on an abstraction. The DI solution will provide (or better: inject) the correct implementation of this abstraction to the depending component. On component level this creates a more loosely coupled design. But on software product level a new dependency is introduced; the software product now depends on the DI solution.

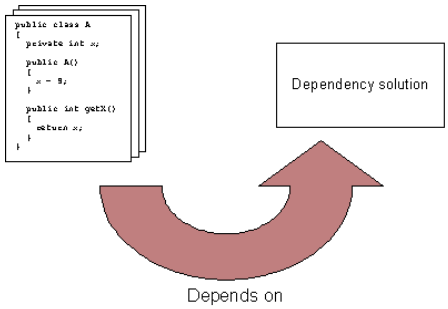



Figure 2: Complete software product depends on dependency solution

### 2.1.2 How DI effects software testability

The question is how the previously described effects on software product level can be related to the factors that influence the testability of a software product? Below the effects of using DI in a software product are related to software testability per testability factor.

**Source code complexity**

If we talk about source code in general, the effect that DI has on the source code of a software product is that depending components don't have to contain code used for obtaining their dependencies anymore. The source code that forms the logic of the software product will therefore decrease.

This decrease of lines of code (LOC) cannot be seen as compressing or squeezing the code because a part of the code is removed instead of rewritten to reduce the number of LOC. Squeezing the code is seen as something that increases the complexity [Kaner99] because it becomes harder to read and understand what a piece of code exactly does. A higher number of LOC is sometimes also seen as something that increases complexity. Especially at method level it becomes more difficult to understand what a method exactly does if it consist out a large number of LOC [McConnell04].

So based on the decreased number of lines of code (without squeezing the code) it seems that DI helps reducing the complexity that exists in the source code that makes up the logic of a software product.
On the other hand, the dependency logic itself is now defined at another place, assumable outside the code that makes up the logic of the software product. This definition, or configuration, (depending on the used DI solution) will most likely introduce some sort of complexity.

**Dependencies**
The goal of DI is to loosen up dependencies; high level components will not be depending on implementation specific lower level components. Instead they will depend on abstractions only describing functionality. The DI solution will provide the high level component with the correct implementation of the abstractions it depends on.

Tests mostly focus on a specific piece of code. If this piece of code depends on another component to complete its task then this component is required to be available during the test as well. This is not always desired. Take for example a component that, through another component, retrieves data from a database because this data is needed to complete its task. This database connectivity may not be available during tests. If the component, for its database connectivity, depends on an abstraction instead of a specific implementation, then it becomes possible to create a component that fakes this database connectivity and provide it to the depending component during tests. A so called mock object makes it possible to have full control over the behavior of the component that the component under test depends on.

For example; with full control over the data that otherwise would be retrieved from a database it is for example more easy to test the component on what would happen when wrong data would be returned by the database. This use of mock objects and the control they provide during testing is often mentioned as the most important reason why DI improves software testability. For example in [Weiskotten06].

But when dependencies are managed by a DI solution, then the software product depends on this DI solution for its own correct behavior. It is possible that this dependency on software product level affects how software is tested. For example; can the DI solution be used during tests? If so, can it easily be configured? Preferably in the setup of a test case, so that in a test it can be defined which mock objects should be injected in the component under test. Or should the test code contain logic to create a work around for correct and controllable DI during tests?

**Controllability & observability**
The ability to inject mock objects into a component under test increases both controllability and observability. A mock object can implement an abstraction with code developed for a specific test. The previous given database example improves controllability. The mock object and the custom code it consist of make it much easier to control its output (which forms the input for the component under test during its execution).
A mock object can also help with improving observability. The mock object can also consist out of code that, for example, logs how it is called by the component under test that depends on it.

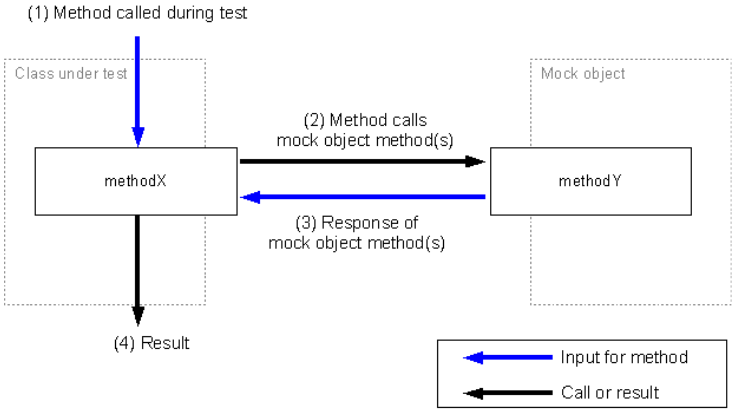



Figure 3: Mock object providing input to the method under test

**Traceability of requirements**

When using DI, components do not depend on specific components but rather on abstractions. These abstractions describe the required functionality that an implementing component should provide. So this abstraction can be seen as a contract describing what the depending component can use and what the implementing component should provide. These abstractions are normally defined during the design phase of the development process and are based on the requirements for the software product. So it can be assumed that when a component implements an abstraction based on requirements it should take less effort to match code that is used to implement an abstraction to the original requirements for this abstraction. Than it is to first having to find out what the function of certain pieces of code is that do not implement a contractual abstraction.

**Test automation**

The ability to automate testing when using DI has actually already been described for the testability factor 'dependencies'. The ability to automate tests will be based on the level of intrusion of the DI solution. The DI solution will most likely introduce some sort of configuration for dependency management. It is possible that this configuration dictates how to use the components which dependencies are managed by this DI solution. During tests it is mostly (if not always) desired to have control over which components are injected into the component that is under test.

When it is not possible to control dependencies through the DI solution during tests it might limit the way these components can be tested since a work around will have to be developed. Then tests will have to contain logic that provide the correct test dependencies themselves.

## *2.2 How can we make this measurable?*

Dependency injection is a principle; it can be seen as a design pattern that can be used to create loosely coupled components. It is a principle because it can be implemented in more than one way. There are different types of dependency injection strategies (constructor, setter, etc.) and there are different solutions to manage the injected dependencies with.

It is possible that the different DI implementation strategies and DI solutions will have a different effect on testability. Next to that; source code of different software products is also never the same and may be affected differently when applying the DI principle to it.

So it is difficult to generalize DI when there are multiple variations; applying DI on different sets of source code with different types of DI strategies and solutions can have different effects. This means that it is difficult to speak about the general effects that dependency injection has on testability.

### 2.2.1 Scope definition

Due to the limited amount of time available for this thesis research, an effective scope has to be defined. A decision has to be made about how this research can be limited but still provide correct information.

Because there are multiple dependency injection possibilities (different sets of source code and different DI solutions) it seems that at least two components from two different sets of source code have to be implemented using two different dependency injection solutions.

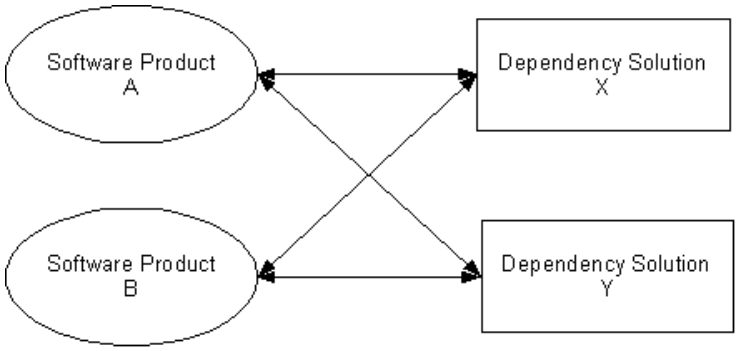



Figure 4: Two different sets of source code implemented with two different DI solutions

For this research we will use two DI solutions that are somewhat situated at both ends of the DI solution spectrum. These are PicoContainer[1], a small and lightweight DI solution, and on the other end; Java Enterprise Edition 5[2] which is a complete Enterprise framework that supports DI.
The components that will be implemented using DI will be a component from a controlled environment and a component from the more extensive Greetz! source code base.

[1] http://www.picocontainer.org
[2] http://java.sun.com/javaee/

In total we also have recognized/defined five testability factors. Below an overview of the assumed influences of DI on these factors has been given;

| Testability factor | Influence of dependency injection |
|---|---|
| **Source code complexity** | Less code needed in depending component |
| | Increase of DI configuration code |
| **Dependencies** | Loosely coupled design based on abstractions gives the possibility to use mock objects more easily |
| | Software product depends on DI solution |
| **Controllability & observability** | Mock objects make it possible to control the output the component under test receives from it. |
| **Traceability (of requirements)** | When code implements an abstraction it is easier to link this code to the requirements which the abstraction is based on |
| **Test automation** | Injection of mock objects gives more control over component under test |
| | (Configuration of) DI solution might interfere with relative ease of testing a component |

| |
|---|
| *Assumed to have a **positive effect** on testability.* |
| *Assumed to have a **negative effect** on testability.* |

Table 1: Assumed effects of dependency injection on software testability

Supposedly the two testability factors ‘source code complexity’ and ‘test automation’ are affected most by the use of dependency injection. Therefore, and this seems most logical, the focus will be on these two testability factors that seem most affected by DI.

### 2.2.2 Measurement definition

To form a valid conclusion on how DI affects testability assumptions are not enough. Instead of drawing a conclusion based on personal interpretation a factual representation of how DI affects testability is needed. Testability has to be measured in some way so that the resulting metrics can be used as facts.

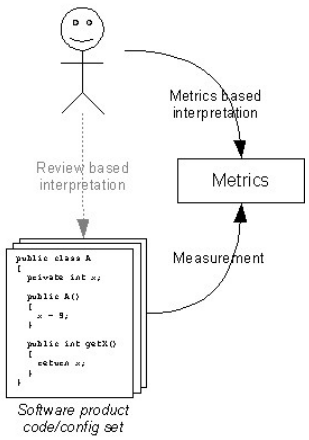



Figure 5: Facts based on metrics extracted from source code instead of personal interpretation

When using metrics as facts to base a conclusion on it is important to use a measuring approach that is valid for this research. The measurements have to provide metrics that actually give a correct insight in how DI affects testability.

**Source code complexity**
Source code complexity can be measured statically. Meaning; it is possible to determine complexity without having to execute the code. Two kinds of metrics that give an indication of the complexity of source code seem most appropriate for this research. These are the McCabe Cyclomatic Complexity and more general the total number of lines of code (TLOC) of a component

The tool used for retrieving these metrics is the Metrics project for Eclipse[3].

**Test automation**
How DI affects test automation on the other hand seems something that can be measured partly static, but can also be experienced in practice.

During this research we will try to give an indication of the intrusion that the selected DI solution forms for automated testing. Based on agile development methods and the testing approach used by Venspro, this intrusion of DI will be tested by integrating (regression) tests with the help of the JUnit unit-test-framework[4].

Per component we will create one testcase and one test-method for every method that exists in the component that is tested. The goal is to create a test that executes all lines of code and branches in the component under test. The coverage metrics will be extracted from a test report generated by Cobertura[5] after each test execution.

As for experiencing how test automation is affected in practice when using DI. The test case should also inject a mock object into the component under test. The amount of needed configuration (counted as lines of code, LOC) and the location of this configuration will then be used to determine the needed effort for test automation. The goal for this dependency configuration is to make it part of the JUnit setUp() method that exists in a testcase class.

**Compare metrics**
By retrieving metrics for a component of a software product that is implemented with and without the help of dependency injection we are able to compare these metrics with each other. The difference should give insight in the effects of implementing the dependency injection principle.

[3] http://metrics.sourceforge.net/
[4] http://www.junit.org/
[5] http://cobertura.sourceforge.net/

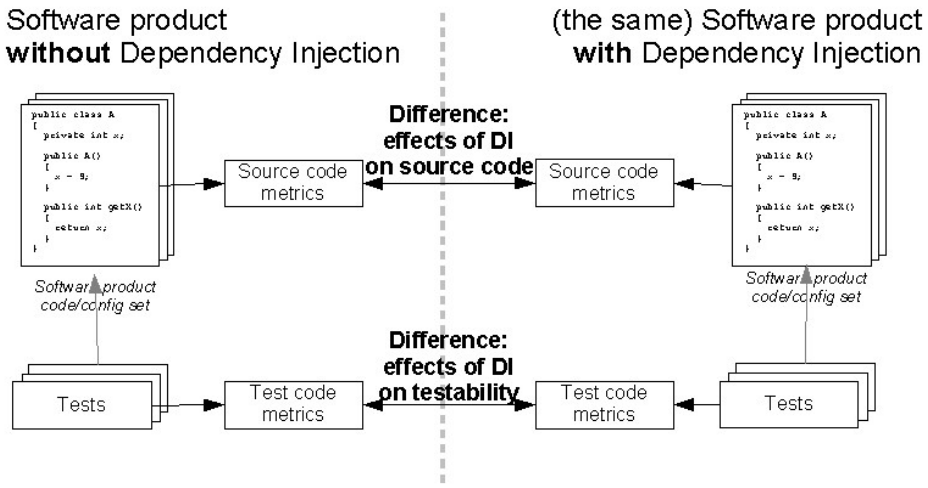



Figure 6: Difference between metrics gives an indication of how DI affects testability

# 3 Research results

## *3.1 The paper manager example*

The paper manager example is a collection of components that could be part of a real software product. The idea behind these components is based on a dependency that could exist in the real world. In this case this is a company that depends on another company to supply paper that is needed during its production process.
A more complete description of this example is available in appendix C.

The paper manager example components function as a controlled environment used to calibrate the research method. By developing code that contains pre-defined dependencies it becomes possible to make predictions of why and how the research method metrics, that are extracted from both DI and non-DI implementations of the paper manager components, will differ from each other. Comparing earlier made predictions with the actual extracted metrics can give insight in possible shortcomings of the research method.

**Without dependency injection**
The `PaperManagerNonDi` Java class is the Non-DI implementation of the `PaperManagerInterface`. This concrete class will form the base class; metrics extracted from the DI implementing classes will be compared with the metrics that are extracted from this class. The difference between them should give insight in the effects of DI on component/source code level.

Up front there is little to say about expectations for this non-DI implementation other than it should contain a reference to another concrete class. And this is true for the `PaperManagerNonDi` class because it contains a reference to the concrete class `PaperCompanyA`.

**With dependency injection using PicoContainer**
The `PaperManagerPicoDi` Java class implements the `PaperManagerInterface` with the intention to use PicoContainer as the dependency injection solution. We will make use of the logic already defined in the previous described `PaperManagerNonDi` class. This means that the reference to `PaperCompanyA` will be replaced with the class containing only a variable of the type `PaperProviderInterface` (instead of `PaperCompanyA`).

Since PicoContainer is based on the constructor injection strategy, the constructor signature of the `PaperManagerPicoDi` class will have to be extended with an argument so it will accept an instantiation of a class that implements `PaperProviderInterface`.

It was expected and found true that these changes will not cause differences for the TLOC and McCabe complexity metrics compared with the `PaperManagerNonDi` class. Because the signature of the constructor will be changed and a variable type will be changed from the specific implementing class `PaperCompanyA` to the abstraction `PaperManagerInterface`.

**With dependency injection using Java EE5**
It was expected that the Java EE5 implementation was going to have the least amount of TLOC and that the McCabe complexity metric stayed the same compared to the `PaperManagerNonDi` class. The reason for this expectation is that with the `@EJB` annotation the field dependency injection strategy is supported. This means that only a variable of the type `PaperManagerInterface` with the needed `@EJB` annotation above it is needed.

But two things became obvious when implementing this component (based on the code in `PaperManagerNonDi` class) in the form of a EJB (Enterprise Java Bean) class.

The first thing was that Metrics tool didn't show the expected difference in TLOC, because annotations are counted as LOC as well. This was unexpected and raised the question; should these be counted as TLOC since it is actually DI configuration code? Since the TLOC metric is used to give insight in the source code complexity it was decided that it should be part of the TLOC metric since we approach this metric as: more lines means more complex source code.

The second thing that became obvious is that this field dependency injection wasn't testable outside an application server because the application server is responsible for assigning the correct dependencies to variables. When the application server is not available the dependency must be assigned to the variable from within a test. Being private the `paperProvider` variable wasn't accessible so a mock object could not be injected. One option was to make the variable accessible by changing its modifier from `private` to `public`. Buts since Java EE5 also allows the `@EJB` injection annotation to be used for setter methods, the setter dependency injection strategy was chosen over the field dependency injection strategy.

## *3.2 The Greetz! customer component*

Next to the paper manager example a real life software product has been selected as research subject with the goal to get an indication of how the testability of an already existing software product is affected when it is re-implemented with DI. In this case the Greetz! code base was selected and narrowed down to one component that contains dependencies, that are considered relevant, to other components.
This is the `Customer` component.

It is hard to define the specific task that the `Customer` component has since it functions as a data entity but also contains a lot of business logic. An example of a part of this logic is that it contains code that is used to send email whenever the state of a customer is changed. For sending these emails the `Customer` component depends on the `GreetzMailProvider` component. This is also the specific dependency that is focused on during this research. Di will be used to change the Customer component not depending on the implementation specific `GreetzMailProvider` but rather on a `MailProviderInterface` abstraction.

**Without dependency injection**
The original, already existing, implementation of the `Customer` component also serves as the base implementation with which the metrics extracted from the other two dependency implementation variants will be compared. Because a JUnit testcase did not exist for the `Customer` class the plan was to create one aiming at the highest possible coverage rate.

Unfortunately the `Customer` class has a lot more dependencies; some of these dependencies involve settings that are retrieved from a database. Since the database wasn't available in the environment in which the test was going to be developed and executed the Customer class proved to be un-testable. And because the limited available amount of time still left for this research it wasn't possible to develop a complete JUnit testcase for this class.

**With dependency injection using PicoContainer**
While implementing the PicoContainer constructor dependency injection strategy in the `Customer` component, it proved that constructor injection wasn't the right DI implementation strategy to be used in the `Customer` component (and much of the other components in the Greetz! code base for that matter).

The reason for this is the fact that instances of the `Customer` class -as well as many other Greetz! components/classes- are provided by Hibernate[6]. The `Customer` component represents customer data that is stored in a database. Hibernate retrieves this data and creates a new instance of the `Customer` class with the retrieved database values and does this by ignoring constructors with arguments.

Therefore the setter dependency injection strategy seems a better choice. But this means that the dependency injection should find place after instantiation.
Does this mean that components that use instances of the `Customer` class are responsible for injecting the right dependency? This is certainly not desirable and it would also cause a rippling effect of changes throughout all components that make use of the `Customer` component.

Fortunately all other components that use the `Customer` component retrieve new instances from the `CustomerFacade` component. The role of the `CustomerFacade` is being the spokesperson for all other components that want to retrieve or persist an instance of the `Customer` component. So the logic for injecting the correct dependency through a setter method could become part of the `CustomerFacade` without other components knowing about it.

Although constructor dependency injection is preferred, PicoContainer does support the setter dependency injection. Unfortunately and due to the short available amount of time it wasn't possible to implement this strategy in a test. PicoContainer kept throwing an `UnsatisfiableDependenciesException` and there was no quick way of finding a solution for this problem (it seemed that PicoContainer wants to manage and/or inject something into all setter methods).

[6] An object relational persistence service used to persist to and retrieve data from a database. http://www.hibernate.org/

But using the setter dependency injection strategy it was still possible to inject the `MailProviderInterface` dependency by making a custom call to this setter method from within the test.

**With dependency injection using Java EE5**
After implementing the `Customer` component with the setter dependency injection strategy the component was easily configurable for Java EE5 dependency injection by only needing to add the needed `@EJB` annotation above the `setMailProvider(MailProviderInterface gmp)` method (as well as the `@Stateful` annotation above the class itself).

In a testcase the component implementing the `MailProviderInterface` instance can be injected with a call to this setter method.

## *3.3 Overview*

### 3.3.1 Paper manager example

| | Static reference | DI Pico container | DI JEE5 (EJB 3.0) | |
|---|---|---|---|---|
| **Component** | PaperManagerNonDi.java | PaperManagerPicoDi.java | PaperManagerBean.java | Software product business logic |
| **DI type/solution** | None (Static reference) | Constructor | Setter (EJB Annotation) | |
| **TLOC** | 106 | 106 | 113 | |
| **McCabe Class** | 1,25 | 1,25 | 1,23 | |
| **Testcase** | PaperManagerNonDiTest.java | PaperManagerPicoDiTest.java | PaperManagerBeanTest.java | Test |
| **Class under test** | PaperManagerNonDi.java | PaperManagerConstructorDi.java | PaperManagerBean.java | |
| **DI type/solution** | None/Static reference | Constructor/Pico container | Setter/Custom | |
| **DI configuration location** | None/Static reference | In testcase code (calls to Pico container) | In testcase code (Custom injection in setter) | |
| **DI configuration LOC** | 0 | 5 | 3 | |
| **TLOC** | 108 | 126 | 122 | |
| **Junit asserts** | 16 | 22 | 22 | |
| **Line coverage** | 93,0% | 100,0% | 100,0% | |
| **Branch coverage** | 83,0% | 100,0% | 100,0% | |
| **Automation framework** | Ant | Ant | Ant | |
| **Test framework** | Junit | Junit | Junit | |

Table 2: Research metrics extracted from the paper manager example

### 3.3.2 Greetz! customer component

| | Static reference | DI Pico container | DI JEE5 (EJB 3.0) | |
|---|---|---|---|---|
| **Component** | Customer.java | CustomerPicoDi.java | CustomerBean.java | Software product business logic |
| **DI type/solution** | None (Static reference) | Setter | Setter (EJB Annotation) | |
| **TLOC** | 937 | 948 | 952 | |
| **McCabe Class** | 1,675 | 1,667 | 1,667 | |
| **Testcase** | CustomerTest.java | CustomerPicoDiTest.java | CustomerBeanTest.java | Test |
| **Class under test** | Customer.java | CustomerPicoDi.java | CustomerBean.java | |
| **DI type/solution** | None/Static reference | Setter/Custom | Setter/Custom | |
| **DI configuration location** | None/Static reference | In testcase code (Custom injection in setter) | In testcase code (Custom injection in setter) | |
| **DI configuration LOC** | 0 | 3 | 3 | |
| **TLOC** | N/A | N/A | N/A | |
| **Junit asserts** | N/A | N/A | N/A | |
| **Line coverage** | N/A | N/A | N/A | |
| **Branch coverage** | N/A | N/A | N/A | |
| **Automation framework** | Ant | Ant | Ant | |
| **Test framework** | Junit | Junit | Junit | |

Table 3: Research metrics extracted from Greetz! code source base

## *3.4 Validation*

It is important to validate that the chosen metrics actually give a correct indication of the source code complexity and the needed effort for creating tests (as well as automating them).

To achieve this we fist created a controlled environment in which the research was performed. The controlled environment in this case is the paper manager example; the small software product developed specifically for this research (see also appendix C). By developing such an example application it is possible to make a precise prediction of how it will change when the dependency injection principle is applied to it. If these predictions are confirmed by the retrieved metrics we can be more certain of the research method validity.

### 3.4.1 The TLOC metric

During the implementation of the paper manager example predictions about the TLOC metric proved not to be correct. This lead to the conclusion of how the Metrics tool actually calculated the TLOC metric and also how important it was to keep code formatting the same throughout the research because formatting can affect this metric.

When it comes to the meaning of the TLOC metric for this research; more lines of code make the code harder to understand and therefore it becomes more difficult to write a test that tests this code. It became very doubtful during this research that a significant change in level of testability was something that could be discovered based on differences between the TLOC metric of the non-DI and the DI implementations. The differences are very small as you can see in the results overview.

This is because logic inside the components, that were focused on during this research, was hardly altered when (re-) implementing them with a DI implementation strategy. The reason for the small differences is the fact that most of the changes are found in the import block of a class (and sometimes a small setter method is added when the setter DI implementation strategy is used).

If implementing DI affects the testability of a component, then the TLOC metric seems not very usable as a factual representation of this change in testability at all. The components where not altered significantly during this research when it comes to the total lines of code their made out of.

### 3.4.2 The McCabe Cyclomatic complexity metric

When it comes to the McCabe Cyclometic complexity metric it also seems that this metric, as it was used during this research, is not usable as a fact indicating that testability is affected by DI. The difference for this metric between the non-DI and DI implementations is very small. The reason that the differences are so small is because for this research the class's average McCabe Cyclometic Complexity is used. This is the average of the McCabe Cyclomatic complexity of all methods in a class.

Because the components where hardly altered after (re-)implementing them with DI meant that the control flow (the possible paths) in the component was not changed. And when measuring the average McCabe Cyclomatic complexity for a component it

also means that when adding a simple setter method (which scores 1 for its Cyclomatic complexity) this average decreases while actually the number of TLOC increases! This is conflicting with the idea behind the TLOC metric as how it is used during this research.

If implementing DI affects the testability of a component than the McCabe Cyclomatic complexity metric also seems not very usable as a factual representation of how DI affects testability. This is because the control flow in the components used during this research was not altered significantly.

### 3.4.3 Needed DI configuration in testcases

The amount of needed DI configuration that has to be done to manage dependencies in tests and the location of the configuration is used to give an indication of the needed effort for test automation and the intrusion of the dependency injection solution in the test environment. The idea behind this is that more lines of code increases complexity; each line can be seen as a step for solving the dependency management problem. This is also based on [Wikipedia-Complexiteitsgraad] which is also referred to in 'Source code complexity' in appendix A. When the configuration has to be done outside the actual code of a testcase it means that this also increases complexity and therefore the needed test effort.

The only significant difference in needed lines of configuration code can be found in the `PaperManagerPicoDiTest` testcase. This is because it uses and configures the PicoContainer to set up all needed dependencies. But still it is not a fact showing that more effort is needed. Because it was also possible to directly create a mock object and inject it into the component under test from within the testcase. In fact all components implemented with DI during this research are testable from within testcases without having to make use of a dependency solution. This can be interpreted as the fact that the used dependency injection solutions did not intrude and dictated the test environment during this research.

### 3.4.4 Coverage

The test coverage in the component under test was a metric that really did show a significant difference for the paper manager example after implementing it with DI. Increasing coverage was actually only possible when injecting a mock object into the component under test (the original component was also designed with this intention). The coverage metric shows without a doubt that DI can have a positive effect on the testability of a component. This seems to prove the often made point in different DI related literature that DI improves testability (mentioned in [Weiskotten06] for example).

### 3.4.5 Test cases

The testcases used during this research were developed to make it possible to measure both the 'needed DI configuration' and 'coverage' metrics. When it comes to the coverage metric, the goal of a testcase was to get the highest value for this metric as possible. In order to do this the code in a testcase was written so that as much lines of code and branches within the code (the control flow) of the class under test are executed.

The testcases for the paper manager example were able to completely cover all lines but only when the component under test was re-implemented with the help of DI. This was to be expected because the non-DI version of the component was developed to be not fully testable because of a failing component it depends on. Therefore making use of dependency injection in the testcases used for the paper manager example can be seen more as a goal instead of only a meaning to improve the way the component under test is tested. This certainly may have influenced the integrity of the testcase, something that should not have been the case for the testcases with which the DI implementations of the Greetz! Customer component are tested. Unfortunately no complete testcase could be developed for the Customer component. Meaning that more independent testcases (as compared to those from the paper manager example ) have not been used during this research. It is certainly possible that the validity of the research and upcoming conclusions are affected by this situation.

# 4 Conclusion

My hypothesis was that *test automation will become more complex when using a dependency injection solution*. Suspected reasons for this increase of complexity where that (1) *the dependency injection solution should be configured for each test* and (2) *it may even be impossible to use a dependency injection solution in a test environment since DI solutions act as managed environments on their own*. Both of these reasons in my opinion could be seen as how a DI solution can intrude (and/or dictate) a software product and especially its test environment.

But both reasons (1) and (2) proved not to be true during this research. The cause for both reasons not being true is that the components implemented with DI during this research where done so with either the constructor or setter method DI strategy. This made it possible that dependencies could also be injected without having to make use of a DI solution. From within a JUnit testcase it proved to be no problem providing for example a mock object as a constructor or setter argument to the component under test, without having to make use of the DI solution that the software product in its complete form depends on for its normal behavior.

This could have been different when the choice had been made to settle for the field dependency injection strategy when developing components during this research that made use of the JEE5 dependency solution (based on `@EJB` annotations).
While developing the first component that made use of the JEE5 field dependency injection strategy, it became obvious that these dependencies could only be injected with the help of a Java application server. This is what I already expected before starting with this research and my hypothesis is for a big part based on this assumption (see also 1.1 Motivation). But during my research I found out that the setter injection strategy is supported as well when making use of JEE 5 `@EJB` annotations. Changing from field to the setter method DI implementation strategy made it possible to call this setter from within a testcase.

For me this is an indication that the intrusion of a DI solution, in the test environment of a software product, is partly formed by the DI implementation strategies that are supported by the chosen DI solution for that product. If the DI solution makes use of either the constructor or setter method DI strategy then it is also possible to provide the needed dependencies to the components without having to make use of the DI solution. When glassbox testing the component with a JUnit testcase, all needed dependencies can be injected with code in the testcase itself.

I specifically mention both glassbox testing and that the level of intrusion is ‘partly’ formed by the DI implementation strategies that are supported by the chosen DI solution. The reason for doing so is that during this research only glassbox testing has been used as an attempt to collect facts about how DI intrudes these kinds of tests. Based on the metrics for the amount of configuration, as well as the location of this configuration, that is needed to inject mock objects into the component under test. Glassbox testing is obviously not the only way to test a component, meaning that this research gives no indication of how DI affects or forms an intrusion for other test strategies that are used throughout the complete test process.

Another outcome of this research was that choosing a DI implementation strategy for already existing components can be dictated by how they are implemented and used by other components. For the Greetz! Customer component it showed that constructor dependency injection could not be implemented. The reason for this is that instances of the Customer component are provided by the Hibernate persistence service layer that is used throughout the Greetz! codebase. Needed dependencies therefore had to be injected with the help of setter methods.

When it comes to the actual dependency of the Customer component that was focused on during this research (the `GreetzMailProvider` component) it can be questioned if the Customer component should have this dependency at all. The main obvious reason for the Customer component containing this dependency is because it functions as a data entity (storing information about a specific customer) but it also contains a lot of general customer business logic (like sending emails when the status of a customer changes). DI in this case can help with the Customer component not depending on specific implementing components but not with making a clearer separation between the concerns that exist in the Customer component. Therefore dependency injection is something that can improve the way components are coupled but will not fix other problems that may exist in a design or implementation of a software product.

## *Future work*

The limited scope of this research (2 components implemented with 2 DI solutions) as well as the fact that the extracted metrics do not really give an indication of how testability is affected, mean that the results of this research cannot be seen as a factual representation of how testability of software is affected when making use of dependency injection.

The metrics used during this research, the Total Lines Of Code (TLOC) and the McCabe Cyclomatic Complexity, do not show significant differences between components implemented with and without dependency injection. The reason for this is that the testability of components is not affected by DI in such a way that it could be measured with these metrics. Therefore the biggest question I had at the end of my research was; what metrics could be used to measure the effects of DI on component level?

In search for an answer on my question I came across [Bruntink03] which is the final thesis document of Magiel Bruntink. In his thesis Magiel focuses on testability of object-oriented Systems with a metrics-based approach. Magiel also uses metrics for test-critical dependencies. The dependency related metrics used by him are the Fan Out (FOUT) and the Response For Class (RFC). In his conclusion he discusses the metrics used during his research. An excerpt from his conclusion concerning the FOUT metric is given below:

> *"We showed that FOUT is a significantly better predictor of the dLOCC metric than of the dNOTC metric (at the 95% confidence level for DocGen, 99% for Ant). Thus, the association between the fan out of a class and the size of its test suite is significantly stronger than the association between the fan out and the number of test cases. The fan out of a class measures the number of other classes that the class depends on. In the actual program, these classes*

*will have been initialized before they are used. In other words, the fields of the classes will have been set to the appropriate values before they are used. When a class needs to be (unit) tested, however, the tester will need to take care of the initialization of the (objects of) other classes and the class-under-test itself. The amount of initialization required before testing can be done will thus influence the testing effort, and by assumption, the dLOCC metric."*

I interpret this excerpt as that the FOUT metric can be a predictor of the needed test effort, in the form of needed dependency configuration in a test case. A higher Fan Out assumingly results in the testcase containing more lines of code.
I believe that this increase of needed test effort based on needed dependency configuration in a testcase, is closely related to my assumption that; using a DI solution will increase the amount of needed configuration code in (or outside) a testcase. But my assumption proved to be wrong during this research; it was not needed to configure a DI solution when using a JUnit testcase because mock objects could be injected into the component under test directly from code. The thing I did not focus on during this research where the mock objects themselves. Effort is of course also needed to write the code for these mock objects.

Why I think it is relevant to mention the FOUT metric here is that the FOUT metric doesn't say anything about the nature of dependencies in terms of how components are coupled. When using DI, components can become less tightly coupled; depending on abstractions instead of implementations. Allowing mock objects to be injected into components under test, which eliminates the need to configure other components on which the component depends. These other components are not needed since they can be replaced by mock objects. But as said; creating mock objects also requires effort. This effort may even be more than configuring the needed component in a testcase but improves test controllability and observability.

During a second iteration I would like to have made an attempt to introduce a more specific metric than, but still based on, the FOUT metric. Not only giving an indication of which other components are called by a component, but also if the called components can be substituted with a mock object if the DI principle is applied to it. In my opinion the nature of a dependency determines if DI can be used to loosen up this dependency. When for example a component inside one of its method creates an instance of a component it depends on every time the method is executed, then it is not possible to change this dependency to an abstraction and inject an implementing component. A dependency that for example is initialized once during construction of the depending component is a perfect candidate for becoming less tightly coupled by implementing a DI strategy. Based on this I would then also like to have found a solution to analyze the specific depending code, when the proposed metric indicates that DI can be applied to it, in order to get insight in the behavior of the mock object that is needed to test this code. This needed behavior (or input from the mock object during test, *see also Figure 3*) can possibly be based on the McCabe Cyclomatic Complexity. The needed effort for creating a mock object with this behavior could be measured based on making use of a mock object tool like EasyMock[7]. After this the ultimate goal, in my opinion, would be to completely automate the creation of mock objects as well as the needed transformation of objects allowing them to be injected.

[7] http://www.easymock.org

# Appendix A - Introduction to Software Testing

## *Why is software tested?*

According to [SWEBOK04] testing is an activity performed to evaluate the quality of a product. Based on the outcome of this activity: the identified defects and problems, it is possible to improve the product. When testing a software product, the behavior of the software product under test is compared with the expected behavior for this software product.

More simply put, software testing can be seen as *checking if a software product behaves as it is supposed to do*. So the most obvious explanation for the reason why software is tested is *because it can happen that software doesn't behave as intended*.

From a commercial point of view it is for any company important to develop products that are considered by customers as good quality. A software program that is not functioning like the customer requires it to do so will most likely not be accepted by the customer as good quality. The goal therefore is to develop a program that behaves like the customer requires it to behave. With the help of software testing a development copy can be used to check if there are problems/errors that negatively affect the intended behavior and therefore need to be fixed.

## *Why does software not behave?*

There can be several reasons for software not behaving like intended. In my opinion the most obvious reasons are probably mistakes made by programmers during development.
Just like normal human beings developers can make mistakes. These mistakes are mostly pieces of code that, unintentionally, have a negative effect on the behavior of a program.
Often these mistakes go unnoticed during development; when executed to see if the program runs, all seems ok. But when the actual program will be used in a production environment these mistakes have the potential to make the program not behave like intended. A reason for this is clearly described by Andreas Zeller;

In [Zeller05] a program execution is described as a succession of states. Initially the program is in a sane state (hopefully) and during execution it goes trough different states. Somewhere between two states a piece of code may be executed that causes the program to fail (and with failure we mean: not doing what its supposed to do). But this failure may not propagate immediately. When a malfunctioning peace of code is executed the next state becomes infected. During execution this infection has influence on the next states and eventually may also cause a failure in one of the next states.

A very simple real life example of this, is when two methods use the same global variable X. Executed independently from each other during development both methods show the expected behavior. But when in production it can happen that the first method may change the value of X in such a way that the second method that uses this variable as well (in one of the next states of execution) can not behave like intended. *This truly must be seen as only a very simple example supporting the above*

*description from [ZELLER05]. It is not intended to argue if this is bad design or a bad programming habit.*

Other than unintentionally made programming mistakes in the code of a software product it is possible that code is written based on wrong and/or incomplete requirements. Also, if not specific enough; requirements may also be wrongly interpreted by developers. In the case of problems with the requirements the behavior of program will also not be equal to the actually intended behavior.

Also after implementation (when done with the initial development) the behavior of a software product can still be influenced negatively. Unexpected behavior of external elements (like failures in hardware and other software products for example) on which the software product depends for its own correct behavior.

## *When is software tested?*

Probably the best way to answer the question "*how is software tested?"* is by first describing *when software is tested*.

Software testing is a Software Engineering knowledge area that has really matured from just being seen as *an activity* to being seen as *a process closely interwoven with the complete Software Engineering process*. In [SWEBOK04] this is described as;

> *"Testing is no longer seen as an activity which starts only after the coding phase is complete, with the limited purpose of detecting failures. Software testing is now seen as an activity which should encompass the whole development and maintenance process and is itself an important part of the actual product construction"*.

This description tells us that previously software testing was mostly done at the end of the development process (after all the code was written) and that this approach is limited.

[Gelperin88] explains the growth of software testing over the years by describing how the purpose of software testing has changed. Until the beginning of 1980, test models where classified as 'Phase models'. The word *phase* describes that these models are executed/processed once (and not re-occurring) during the development of a software product. There are two test models that make up this period of phase testing; the demonstration model and the destruction model.

The primary goal of the *demonstration* model is to *make sure that the software satisfies it specification*. In [Gelprin88] it is mentioned that the words 'make sure' where often translated as; *showing it works*. But due to the increase of amount, complexity and costs of applications as well as the fact that computer systems contained a large number of deficiencies it became clear software products needed to be tested better before they were delivered to the customer.

After the demonstration oriented model the *destruction* model was introduced. The reason for doing so was because the goal of testing shifted from demonstrating the software behaves as intended to finding problems before releasing the software. This model tries to overcome the fact that the demonstration model is prone to not being

effective in detecting errors. Because it is possible that for demonstrations test data is used that has a low probability of causing the software not to behave like intended ("You see, it works!").

Around 1983 the first life cycle method was introduced; the evaluation model.
The goal of the *evaluation* model was to detect faults during the complete development process. Each phase in the development process has an associated set of products and activities. The evaluation model aimed at increasing the quality of the tests and with that increasing the quality of the end product. Not only should the end product be tested at the end of the development process but also the requirements and design that lead to the actual coding of the end product.

The next step from the evaluation model is the prevention model. This test model can be seen as a more professional version of the evaluation model. The goal of the *prevention model* is not only to detect problems during the complete development process but to also prevent problems from occurring in the first place. This is for instance possible trough timely test planning and test design. Designing and planning tests early on in the development process have a positive effect on the quality of requirements/specifications and code as well. The effect of focusing on what should be tested before starting with actual coding is that flaws in the requirements (like ambiguity, incorrectness, inconsistency, etc.) are detected early on.

A reason (and it is probably the most obvious reason) for the increasing professionalism of software testing can be found in [Kaner99]; *problems in software can have a big financial impact*. In [Kaner99] we can read that the effects of software errors are that they become more expensive to fix during the development process. Correcting faulty requirements in the beginning of the development process is far less expensive than fixing errors after the product already has been released.
Based on this reason it seems that software is best tested from early on in the development process and also during the complete development process.

| | | |
|---|---|---|
| **Phase models** | 1957 – 1978 | Demonstation model |
| | 1979 – 1982 | Destruction model |
| **Life cycle models** | 1983 – 1987 | Evaluation model |
| | 1988 – now | Prevention model |

Table 4: Overview of different software test models over the years

## *How is software tested?*

Software testing isn't cheap. To test software, a software development company has to free up resources needed for testing. In [Christensen03] is stated that the development of a software product is mostly driven by four parameters: resources, time, scope and quality. In many cases the parameters resources, time and scope have a fixed value, meaning that, to finish (or survive) a project, the parameter quality is adjusted negatively; *the quality level of the end product is lowered*.

Lowering/decreasing the quality goes against the increased professionalism that software testing has gone through the last couple of decades (see: When is software tested?). The goal of software testing is to assure the quality of the end product. None

the less, in the real world resources are almost always limited. This means that to develop a high quality end-product the available resources should be used optimal. All development activities should be adequate, therefore a development company should decide during its test planning which testing activities should be performed to assure the quality (the expected behavior) of the end product.

There are many types of test activities. Each type of test has its own place in the development process. In [Kaner99] a complete overview is given of (well known) test types and their place in the development process. This development process is divided in the following phases; planning, design, coding and documentation, testing/fixing and maintenance. Although testing/fixing is mentioned as a separate phase, it clearly focuses on how to integrate testing in all phases of development. Below a summarization of this available information is given focusing on all phases except for maintenance.
*Being the last phase after end-product delivery to the customer the maintenance phase is considered out of scope for this research. Someone might argue if both the planning and design phases should to be considered out of scope as well for this research. But considering the fact of how important testing during these phases is (which is also described in the summarization below) a description in this document is vital to understand the importance of testing during the complete development process.*

## Planning phase

At the beginning of the development process there's no code yet to test. At this point it is critical to lay a solid foundation for further development. This is possible by reviewing the contents of the requirements and functional documentation on which actual coding is based. During these reviews the requirements are tested on the following issues: Are these the right requirements? Are they complete? Are they achievable/reasonable? And very important for testing during the further development process: *Are the requirements testable?*

## Design phase

Based on the requirements documentation the to-be-developed software product can be designed. Gross there are two types of designs; external design and internal design. The external design basically describes the (user) interfaces of the end product. The internal design describes the structural design for example. The structural design can be seen as the architecture of the application. But internal design can also describe how data is used (data design). Designs are mostly tested on the following issues: Is the design good? *Does the design meet the requirements?* Is the design complete? And is the design possible? A common practice used to test a (part of the) design, is to make a prototype. For example the user interface can be simulated with a paper mockup prototype. Parts of the data design could be tested with a coded prototype. In this way it is possible to test if a design is for example even possible in the first place.

## Coding phase

The phase, during which the actual code is written based on the earlier made (and hopefully tested) requirements and designs. Testing during development is often revered to as glassbox or whitebox testing. The reason for this term is the fact that developers test their code knowing the internal working of the code. Testing during the coding phase is meant to test the structure of the code (control flow and data

integrity for example). By creating test-cases during coding it is possible to re-run these tests during the entire coding/development phase. Re-running tests is referred to as regression testing. Regression testing (and especially automation of regression testing) benefits integration of components that together form the complete software product. When adding a new component, executing earlier made test-cases for the already existing components can determine if the newly added component has a negative impact on the existing code. This is a more effective approach than adding all components together at the end of development, because this makes it hard to find out in which component(s) the problem(s) exist.

*A strong trend in the software development process is to set the earlier mentioned quality parameter of the end product to a fixed value and adjust the scope parameter when the project is endangered from not being completed in time. The development process models that use this approach are categorized under the name 'Agile development'. Agile development models (eXtreme Programming (XP) for example) focusses on glassbox testing during development in the form of unit testing. Unit testing stands for testing small parts of code; normally tests are written per method (but not necessarily limited to only one test per method).*

### Testing phase

Although testing during the complete development process is important to ensure the quality of the product there is also a phase when coding is finished. At this point it is important to test the complete product. During this phase the behavior of the program is tested against the expected behavior documented in the requirements documentation. These tests normally focus on giving input and checking the generated output without knowing about the inner workings op the software product. This is the opposite of glassbox testing and called blackbox testing. Common tests are stability and performance/load testing. It is common practice that these tests are executed by persons that don't have a developer role.

## *What makes software testable?*

Before describing what makes software testable it is important to define testable software or *Software Testability*. In [Binder94] is stated *that testability is the relative ease and expense of revealing software faults*. So how harder it becomes to find existing faults the less testable software becomes.

There are several reasons why faults/problems might still occur when a software product has been taken into production, even though the software has been tested. In [Whittaker00] acceptable reasons are given for this phenomenon;

In the production environment…

- …code was executed that hasn't been tested.
- …the execution order of code statements differs from the order when tested.
- …untested input has been supplied.
- …the production environment is different from the test environment.

This short list makes it clear that these failures may go unnoticed during testing if tests do not resemble situations that might occur in a production environment. The difficulty with this statement is that there are often many possible situations that may occur in a production environment.

## Multiple input possibilities

In [Dijkstra69] an example of a multiplication mechanism is given. This example describes the difficulty of testing if this multiplier mechanism is 100% correct. When blackbox testing this mechanism only the output for a given input can be checked. This means that, to be 100% sure that this mechanism behaves correctly, all possible input has to be tested. In the described example this would take more than 10,000 years, meaning that in practice it is impossible to completely test the mechanism and proof it is entirely correct. Dijkstra tells us that in order to be more accurate in proofing the correctness of software, reasonable test-cases have to be defined (*is it really necessary to test all possible input?*) and that we should take the structure of the mechanism into account. Meaning; the focus of testing should not only be on the output, but more on the individual parts of code that together provide the functionality.

The difficulty of many input possibilities is also described in [Whittaker00]. It is stated that testers have the task to simulate the interaction between software and its environment. To do this, testers have to identify the interfaces of the software product and the input possibilities per interface. An interface can be for example the User Input interface or a more 'underwater' interface like the file system interface. Because it is impractical (and mostly even impossible) to test all possible input, testers must carefully select the input that's used during testing.

## Source code complexity

To select a finite set of input for a test it is necessary to analyze the source code that will be tested. By analyzing the source code the person who creates a test for existing code can get insight in the inner workings of the code. With knowledge of the inner workings it is for example possible to determine the control-flow of the piece of code.

In software code many decision points exist. Each decision point has influence on which code is executed next, or; the next state the software will be in. A very simple example is an if-statement. If the evaluation for the if-statement is true, the code within the scope of the if-statement is executed. So this simple if -statement creates two possible paths. This means at least two tests are needed to test this code. One test supplying true as input, and one with false as input.

```
public void deleteCustomer(Long customerId, boolean removeHistory)
{
      // Code to remove customer object from database goes here

      If (removeHistory)
      {
            // Code to remove customer history from database goes
here
      }
}
```

Code example 1: Code executed based on decision

This if-statement is very simple to understand, and for the given example it is also pretty easy to figure out that at least two tests are necessary. It becomes much harder to understand which test input is required to correctly test the structure of code when the complexity becomes greater.

During this research we use the following definition of complexity based on the "Complexiteitsgraad" [Wikipedia-Complexiteitsgraad]; *the amount of steps needed to solve a problem*. More steps needed to solve a problem increased the complexity.

Complexity in source code is closely related to understandability of that source code. The control flow of source code becomes much harder to understand when a lot of decision points exist in the code (*which decisions are taken at which point?*). Nested decisions (like if-statements) for example are harder to understand than the previous given example of a single if-statement. So the person creating the test(s) should use a mental-aid like a truth table to keep track of all possible input cases; more steps are needed to solve the problem of determining the needed tests.

T. McCabe developed a complexity value indication named 'Cyclomatic complexity'. Simply put, this indicator stands for the amount of closed loops in source code (which translates to the amount of decision points) + 1. It is an indication for the level of complexity of the source code structure when it comes to the amount of needed test-cases. A complexity level over 20 is considered complex and means that a program is hard to test.

| | |
|---|---|
| 1-10 | A simple program |
| 11-20 | A more complex program |
| 21-50 | A complex program |
| > 50 | An untestable program |

Table 5: Indication of software testability based on the cyclomatic complexity of that software

Structural complexity in the form of Cyclometic Complexity can be measured with static analysis. Meaning; tools can count the number of decision points without having to execute the code.

## Dependencies

When testing a specific piece of code a test normally focuses on that piece of code alone. But in software products it is very common for components to rely on other components. So when testing a component it is possible that other components are executed as well during the test. This can interfere with the actual test, for example when another component is failing. This dependency between components also affects testing such a depending component in another way; if you want to test a component that relies on other components you need to have these components available as well.

A tactic to overcome these influences on testing is developing loosely coupled components. An example of loosely coupled components are components that don't have a direct relation with other components. This can be realized by using interfaces/abstractions and information hiding. An interface describes the functionality of a component that can be used by other components. The component implementing this interface hides the actual implementation code, and by referencing to an interface the depending component is not depending on a specific piece of code anymore.

When testing a component depending on an interface implementation it becomes easier to substitute such a component with another component. For example a component depending on another component that is used to read values from a database. This dependency can during testing be replaced with a component that fakes this database interaction. Such components created for independent testing purpose are called Mock Objects.

## Controllability & observability

According to [Binder94] (software) testability has two key facets; controllability and observability. This is actual pretty logical, *because when you cannot control the input of your test how can you tell if the output is correct*? And the other way around it is pretty much the same story; *how can you tell if something works correctly if you cannot check the output*?

Normally when performing a blackbox test, input is given and the output is checked to see if what happened between these two steps works/behaves correctly. Controllability is the amount control over the input that can be given during the test. Observability is the relative ease of checking the output of a piece of code.

When talking about a method, giving a certain input and checking its return value seems pretty straight forward. And when blackbox testing this is normally the case. But often the correct execution of a piece of code depends on other components. When testing a piece of code that depends on other components a correct output is not something that is always related to a certain given input (like for example a method argument value). The interaction with other components (their behavior) can play a big role as well. So input doesn't necessarily consist only out of the argument value(s) given to a method. Input can also be given by components a piece of code depends on during its execution.

When testing a piece of code it can be tested more thoroughly if its interaction with other components can be controlled. Something that can be very difficult because it may be necessary to alter the behavior of such components to force a specific needed test output (which forms the input for the depending piece of code). When altering the behavior of such components during tests, it would be for example possible that the piece of code under test deliberately receives incorrect input from the altered component it depends on. This makes it possible to test the behavior of the piece of code in a not desirable but still possible situation that might occur after it is delivered to the customer and taken into production.

## Traceability of requirements

When testing it is important to know what the expected test outcome should be. It is meaningless to test software without knowing what to test for. When developing a test it should be possible to trace the requirements for that specific piece of code describing the expected behavior. Traceability therefore describes the possibility to determine if requirements are met by the software under test.

## Test automation

Testing requires an investment of resources like time and personnel which ultimately can be translated to money, or better said: financial resources.

First of all tests have to be developed. And after a test has been developed it has to be executed and the test results have to be analyzed.

Often when tests will be executed multiple times during a development process (regression testing) a development company will make the effort to automate such tests. Using a test framework the test code can be executed and test results can be analyzed automatically. But in some cases it is very difficult to automate testing. Good examples of hard to automate tests are User Interface tests. It is relatively easy to test changes to a data-set but much more difficult to test results that are only visible on screen.

# Appendix B - Introduction to Dependency Injection

## *What are dependencies?*

Earlier, in 'What makes software testable?' dependencies and their effects on testing have been discussed briefly. In [Jackson03] a precise definition is given of when software components depend on each other; Component A depends on component B if "correct execution of B may be necessary for A to complete the task described in its definition"

## *What are the effects of dependencies?*

When software components depend on other components to complete their task it is normal that in the source code of these components a reference is made to other components. In [Nene05] this is described as; components need to know with *which* other components to communicate, *where* to locate these components and *how* to communicate with them.

Adding this '*which, where, how dependency logic*' to a component can have a negative effect on the source code of a software product when changes are needed to be made. In [Martin96] an example of three negative effects of dependencies between components, on the architecture/design of a software product, are given. These effects; rigidity, fragility and immobility, as well as fourth one: uncontrollability are described below:

### Rigid

Component interdependency makes it harder to make changes to components. When making changes to a component that another component is depending on it may be hard to tell what the effects of these changes are on the depending components.
A change may have a rippling effect of needed related changes throughout depending modules making it hard to predict the impact of the change.

### Fragile

Often when a software product is build from code with poor quality, single changes to a component may introduce new problems in depending components. Maintenance becomes a real problem because fixing these problems often also causes new problems in other parts of the software product. So maintenance can be compared with a dog chasing its own tail.

### Immobile

When components depend on specific other components to complete their own task it becomes hard to re-use these components in another software product without having to include all logic that is not needed in the new product but still is required by the components that the re-used component is depending on. The costs of component separation are often higher than redevelopment of the desired logic that exists in such a single component.

## Uncontrollable

In most software designs dependency of software components is a top-down situation. Meaning; high level components contain the business logic of a software product and depend on lower level modules to make this logic happen.
The problem with high level components strongly depending on lower level components is that they often become dictated by the implementation details of the lower level components. Changes to lower level components can force the higher level components to change. As stated in [Martin96]; *High level modules should be forcing the low level modules to change, not the other way around*.

## An example

Probably the two biggest problems of dependencies in high level components are that these high level components become immobile and are prone for changes dictated by lower level components they are depending on. An approach to overcome these problems is described in [Martin96], in which is stated that *high level modules should not depend on details of lower level modules*. Instead they should depend on abstractions, meaning that specific *implementation logic of lower level components should be hidden from high level components*.

In practice, when focusing on the Java programming language, this abstraction (or hiding of implementation detail) can be realized by using interfaces. An interface describes the provided functionality (methods including their arguments/signatures and return value). A software component (a Java class in this case) implements this functionality without the higher level components (other depending classes) knowing anything about the implementation details.

The interface can be seen as a contract between two components. On one hand the high level component knows which functionality is provided by the component implementing this interface. On the other hand the interface describes which functionality it should provide.

The big advantage of this abstraction is that a lower level component, on which a higher level component is depending, can be replaced by another component that also implements the interface. This is a design approach that creates *loosely coupled components.*

At one point in the code of a higher level component that depends on a lower level component a reference has to be made to this lower level component. And with reference we mean; code that initializes an instantiation of this lower level component. Below a very simple Java example is given of such a lower level component instantiation;

```
public class PaperManager
{
      private PaperProvider currentPaperProvider;

      public PaperManager ()
      {
            currentPaperProvider = new PaperCompanyA();
      }
```

```
      public void checkPaperInStock()
      {
            if (getAmountOfPaperUnitsInStock() < 100)
            {
                  currentPaperProvider.orderPaper(200);
            }
      }
}
```

Code example 2: Instantiation of required component within a component

In the Java example above, the class `PaperManager` is meant to contain all high level logic for managing the paper stock inventory of a company. This company orders paper, needed for its production process, from paper company A. In the example it is clear that although `PaperManager` depends on the abstract type `PaperProvider` it still also depends on a specific implementation of `PaperProvider` (which is `PaperCompanyA`).

Although it is a very simple example it illustrates the dependency of a company that can exist in the real world. Such a real world dependency can form a risk for a business. If the company needs paper for its production process, the production process depends on Company A. Therefore it is important for the company to be able to switch to another paper provider when needed.

## *What is dependency injection?*

In the previous given example it is fairly easy to change from an `PaperCompanyA` to a `PaperCompanyB`, all that has to be changed is the instantiation line in the constructor. But still, `PaperManager` keeps depending on a specific implementing lower level component.

One way to overcome this is to use the Factory design pattern. In the Factory pattern a component is held responsible for the correct instantiation of an implementing component. This causes that the high level component will no longer be depending on a specific implementation but solely on the abstract interface description. Below a very simple Java example is given of how a Factory can be used.

```
public class PaperManager
{
      private PaperProvider currentPaperProvider;

      public StockInventoryManager()
      {
            currentPaperProvider =
PaperProviderFactory.getCurrentPaperProvider();
      }

      public void checkPaperInStock()
      {
            if (amountOfPaperUnitsInStock() < 100)
            {
                  currentPaperProvider.orderPaper(200);
            }
      }
```

```
}

public class PaperProviderFactory
{
      public static PaperProvider getCurrentPaperProvider()
      {
            return new PaperCompanyB();
      }
}
```

Code example 3: Obtaining a required component from a Factory

The `PaperProviderFactory` becomes responsible for initializing the correct paper company. The High level PaperManager only depends on a the PaperProvider interface and not on specific companies any more.

## From static factory pattern…

So with introducing a factory pattern the `PaperManager` becomes more loosely coupled from the specific paper company implementations. But this factory pattern has its own drawbacks. Now that the `PaperManager` is decoupled from a specific `PaperProvider` implementation it now relies on a `PaperProviderFactory` component to get the currently active `PaperProvider` implementation. When it comes to the *which, where, how dependency logic* in a component, the Factory Pattern takes away the which and how logic from the component. Because a factory component initializes (*how*) the correct (*which*) implementation. The component containing the dependency keeps responsible for calling the factory; it still contains the logic for finding the dependency (*where*).

Two other shortcomings (or maybe better: points for improvement) are that when more high level components need to be decoupled from other components, the amount of Factory objects often increase as well. In most cases these factory objects aren't much different from each other. Meaning that a loosely coupled design can cause a lot of boilerplate coding. The other shortcoming is that changing to another implementing object means that the code has to be modified. The factory component has to be altered so that it will return the correct implementation. When the company in the previous example changes to a new/different paper company, the source code has to be modified and the software product needs to be redeployed.

## …To dynamic wiring

Looking further than the Factory design pattern it is possible to create an even more loosely coupled and less static design. This is where dependency injection enters the ring.

Going back to the example used throughout this chapter, the `PaperManager` component, after applying the Factory pattern, is still responsible for knowing where to get the needed dependency; it has to call the `PaperProviderFactory` component.
Another approach is to provide the `PaperManager` component with the needed dependency so that it doesn't have to retrieve it itself. For example the needed dependency can be provided as constructor argument as the example below shows.

```
public class PaperManager
{
	private PaperProvider currentPaperProvider;

	public StockInventoryManager(PaperProvider pp)
	{
		currentPaperProvider = pp;
	}
}
```

Code example 4: Obtaining a required component as a constructor argument

Next to the providing the needed dependency through the constructor a setter-method can be used as well to provide the needed `PaperProvider` dependency (even after initialization of the `PaperManager` component).

```
public class PaperManager
{
	private PaperProvider currentPaperProvider;

	public StockInventoryManager(PaperProvider pp)
	{
		currentPaperProvider = pp;
	}

	public void setCurrentPaperProvider(PaperProvider pp)
	{
		currentPaperProvider = pp;
	}
}
```

Code example 5: Obtaining a required component as a setter method argument

Providing a component with the needed dependencies it needs to complete its tasks, with this component only knowing what these dependencies should do (thanks to the abstract description) and without knowing how they do it (the specific implementation) is often referred to as '*dependency injection*'.

*Another term often used to describe the same principle is 'Inversion of Control' (IoC). Although in my opinion it more describes the power of loosely coupled design when it comes to depending components no longer be in charge of the dependencies they use. And lower level components not forcing higher level components depending on them to change (better described in 'What are the effects of dependencies?'). Therefore we use the term 'dependency injection' (DI) throughout the rest of this document.*

So dependency injection helps with removing the *which, where, how dependency logic* from a component. But the true power of DI lies in the ability of automating it with configuration based solutions.

Recalling the business in our example that relies on a specific paper company (that provides paper needed for its production process); what if the company would switch to another paper company? The `PaperManager` component should now use the `PaperCompanyB` implementation of the `PaperProvider` interface instead of `PaperCompanyA`. When using DI the `PaperManager` can be injected with the correct `PaperProvider` implementation.

## *Current solutions*

Needless to say, little helper gnomes (instead of bugs) are very rare in the software engineering business. Or in other words; this dependency injection thing doesn't happen out of itself. When providing dependencies to components this logic has to be developed or an existing DI solution can be used.

## Types of dependency injection

As the examples earlier on showed, dependency injection is (for example) possible through supplying the dependency as an argument for a constructor or a setter-method. The different types of DI (or DI implementation strategies) are often referred to as 'type x', where x is the number corresponding to a certain type of DI implementation strategy.
Throughout different literature these DI types are often given a name that describes the DI implementation strategy, instead of using only a number. Using names instead of numbers is obvious much more clearer. But the problem is that sometimes these DI types are not given the same name. For example in [Fowler04] type 1 DI is given the name 'Interface Injection'. And in [CodehausPico] the same type has been given the name 'Contextualized Dependency Lookup'. In [CodehausPico] it is also mentioned that the 'type x' definitions can be seen as obsolete all together.
Throughout the rest of this document the following names for the different DI implementations (based on [CodehausPico]) are used:

**Contextualized Dependency Lookup**
Also known as *Type 1*. The component contains logic to call another component that provides the needed dependency. This DI implementation strategy on component level causes the component still having a dependency to a component (or the context) that provides the needed dependency. A possible solution is to provide this dependency provider as an interface implementation and injecting it through a constructor or a setter method.

```
public class PaperManager
{
	private PaperProvider currentPaperProvider;

	public StockInventoryManager(PaperCompanyContext pcc)
	{
		currentPaperProvider =
pcc.retrieveCurrentPaperProvider();
	}

	public StockInventoryManager(PaperProvider pp)
	{
		currentPaperProvider = pp;
	}

	public void setCurrentPaperProvider(PaperProvider pp)
	{
		currentPaperProvider = pp;
	}
}
```

Code example 6: Obtaining a required component from a provided context

**Setter Dependency Injection**

Also know as *Type 2*. The dependency is provided to the component as a setter-method argument. The problem with this DI implementation strategy is that when developing a custom DI solution (the logic that makes the injection happen) it is possible to forget to call such a setter method. When initializing a component it is not mandatory to call setter methods (only a call to a constructor is mandatory), meaning that after initialization the needed dependency might never be provided. This of course will result in the component not behaving like intended.

```
public class PaperManager
{
	private PaperProvider currentPaperProvider;

	public StockInventoryManager(PaperCompanyContext pcc)
	{
		currentPaperProvider =
pcc.retrieveCurrentPaperProvider();
	}

	public StockInventoryManager(PaperProvider pp)
	{
		currentPaperProvider = pp;
	}

	public void setCurrentPaperProvider(PaperProvider pp)
	{
		currentPaperProvider = pp;
	}
}
```

Code example 7: Obtaining a required component through a setter method

**Constructor Dependency injection**

Also known as *Type 3*. The dependency is provided to the component as a constructor argument. The constructor will always have to be called when initializing a component. This means that, if all constructors require the needed dependencies as arguments and null-values are not allowed, the component is always provided with the needed dependency. The problem that might exist when multiple dependencies are required is that the signature of a constructor can become too large and beyond the point they are still easy to read/understand.

```
public class PaperManager
{
	private PaperProvider currentPaperProvider;

	public StockInventoryManager(PaperProvider pp)
	{
		currentPaperProvider = pp;
	}

	public void setCurrentPaperProvider(PaperProvider pp)
	{
		currentPaperProvider = pp;
	}
}
```

Code example 8: Obtaining a required component through a constructor

**Field Dependency injection**
Also known as *type 4* but is less common than type other three types. The dependency is assigned to a field (in Java called a class member variable). No constructor argument or specific setter method is needed to assign an instance of dependency to a variable. The logic for this form of dependency is very complex to develop. A managed environment that is responsible for controlled execution of code seems to be the best option to implement this form of dependency.

## Containers / Managed environments

Multiple third party Dependency Injection solutions exist. Actually they're not always called Dependency Injection solutions, some are called Inversion of Control frameworks but focus on the dependency injection principle. Others provide more functionality than just DI. On [Wikipedia-DI] a list of frameworks that support DI is given. From this list three well known Java frameworks that support DI are described below;

**PicoContainer**
PicoContainer is (and this is also one of its goals) a lightweight DI framework. The developers of PicoContainer believe that constructor injection is the best DI implementation approach (but setter-injection is supposedly also supported).

**Spring**
Spring is more than a DI solution. Spring is a complete (and very popular) J2EE framework offering a lot of other possibilities, like Aspect Oriented Programming (AOP) for example.

**Java EE5**
Java Enterprise Edition 5 is the API for Java Enterprise Applications. The previous version of the Java Enterprise Edition, version 1.4, was considered to be very cumbersome to implement. Especially its core components; Enterprise Java Beans (EJB) proved to be very hard to develop and configure. As response to upcoming frameworks like Spring, Sun tries with JEE5 to simplify EJB development for example by supporting dependency injection.

# Appendix C – The paper manager example

The idea behind the paper manager example is a company that uses paper during its production process. For some products they use white paper, for others they use brown colored paper. The company wants a solution that automatically orders paper when the available amount in stock drops below 100 (measured in meters).

One component will be used to manage the paper, this is the `PaperManager` component. When paper (white or brown) is used during the production process the `PaperManager` component is notified. Its the `PaperManager`'s responsibility to order 200 meters of paper from a paper company when the amount in stock becomes less than 100. To keep the amount of paper in stock at a sufficient level the `PaperManager` depends on a `PaperProvider`.

Because its production process depends on paper, the company doesn't want to depend on only one paper company. The solution should make it possible to switch to another paper company if this may be necessary in the future. Therefore the solution will incorporate an abstraction of the paper ordering functionality that a paper company provides; the `PaperProviderInterface` abstraction. This means the `PaperManager` component will make use of a component that implements this `PaperProvider` abstraction.

The question is what the `PaperProvider` abstraction should look like. Which functionality should it provide or better; how can the `PaperManager` component order the needed paper? The `PaperManager` component must be able to order brown and white paper. It must also be possible to notify the paper company that an order has been received.

## *Technical design*

A generic solution is preferred; the company foresees the possibility that it will be using other different kind of papers in the future. Therefore the decision has been made that the `PaperManager` and the `PaperProvider` implementation will communicate in the form of generic paper orders. These will be derived from an abstract `PaperOrder` component class.

Because we focus on the `PaperManager` component (implementing it with and without the dependency injection principle) we will also create an abstraction (interface) for this component called `PaperManagerInterface`. This way we create a contract so all implementations (concrete classes) implement the same desired logic.

## The paper manager interface

This abstraction/interface describes the functionality that a concrete implementation class should provide, despite using dependency injection or not.

```
package model.papermanager;
import model.paperorder.AbstractPaperOrder;

/**
 * PaperManager is the highlevel component-interface used to manage the information
about paperflow in the company
 * A paper Manager is responsible to order paper when the amount in stock drops below
100
 * @author Ricardo Lindooren
 */
public interface PaperManagerInterface
{
	/**
	 * Called by other components to inform the PaperManager how much brown paper
is used during the production process
	 * @see #whitePaperUsedInProductionProcess(int)
	 * @param amountOfMeters the amount of paper used
	 */
	public void brownPaperUsedInProductionProcess(int amountOfMeters);

	/**
	 * @see #brownPaperUsedInProductionProcess(int)
	 * @param amountOfMeters the amount of paper used
	 */
	public void whitePaperUsedInProductionProcess(int amountOfMeters);

	/**
	 * Called by other components to inform the PaperManager ordered Paper has been
received
	 * @param apo the paper order that has been received by the company
	 */
	public void paperOrderReceivedFromPaperProvider(AbstractPaperOrder apo) throws
UnknownPaperOrderException;

	/**
	 * Returns the number of meters of brown paper that's in stock
	 * @see #getAmountOfWhitePaperInStock()
	 * @return the current amount of brown paper in stock
	 */
	public int getAmountOfBrownPaperInStock();

	/**
	 * @see #getAmountOfBrownPaperInStock()
	 * @return the current amount of white paper in stock
	 */
	public int getAmountOfWhitePaperInStock();

	/**
	 * Sets the amount of brown paper that's in stock
	 * The purpose of this setter is to give a begin value of the amount in stock
	 * @see #setAmountOfWhitePaperInStock(int)
	 * @param amountOfMeters
	 */
	public void setAmountOfBrownPaperInStock(int amountOfMeters);

	/**
	 * @see #setAmountOfBrownPaperInStock(int)
	 * @param amountOfMeters
	 */
	public void setAmountOfWhitePaperInStock(int amountOfMeters);

	/**
	 * Which orders have been dispatched to the paper company
	 * @return a set of paper orders
	 */
	public Set<AbstractPaperOrder> getPaperCurrentlyInOrder();
}
```

## The paper provider interface

This is the abstraction the paper manager component depends on for ordering paper.

```
package model.paperprovider;
import java.util.Set;
import model.paperorder.AbstractPaperOrder;

/**
 * Interface describing the functionality of a paper provider
 * @author Ricardo Lindooren
 */
public interface PaperProviderInterface
{
	/**
	 * Orders paper from the paper provider
	 * @param apo the paper order
	 */
	public void orderPaper(AbstractPaperOrder apo);

	/**
	 * Checks which paper orders are being processed by the paper provider
	 * @return the orders that are being processed by the paper provider
	 */
	public Set<AbstractPaperOrder> checkCurrentlyProcessedPaperOrders();

	/**
	 * Used to let the paper provider know which order has been received by the
paper company
	 * @param apo
	 */
	public void confirmReceivedPaperOrder(AbstractPaperOrder apo) throws
UnknownPaperOrderException;
}
```

## The unknown paper exception

Should be thrown by the PaperManager and PaperProvider implementations when a not earlier identified paper order is supplied as argument.

```
package model.paperorder;

public class UnknownPaperOrderException extends Exception
{
	/**
	 * Generated by Eclipse
	 */
	private static final long serialVersionUID = 1081125365526999630L;

	public UnknownPaperOrderException()
	{
		super();
	}

	public UnknownPaperOrderException(String message)
	{
		super(message);
	}

	public UnknownPaperOrderException(Throwable cause)
	{
		super(cause);
	}

	public UnknownPaperOrderException(String message, Throwable cause)
	{
		super(message, cause);
	}
}
```

## The abstract paper order

This is the information that defines a paper order communicated between the paper manager and a paper company.

```
package model.paperorder;
import java.io.Serializable;
import java.util.Date;

/**
 * @author Ricardo Lindooren
 */
public abstract class AbstractPaperOrder implements Serializable
{
	private Long id;

	private Date orderDate;

	private int amount;

	public Long getId()
	{
		return id;
	}

	public void setId(Long id)
	{
		this.id = id;
	}

	public Date getOrderDate()
	{
		return orderDate;
	}

	public void setOrderDate(Date orderDate)
	{
		this.orderDate = orderDate;
	}

	public int getAmount()
	{
		return amount;
	}

	public void setAmount(int amount)
	{
		this.amount = amount;
	}
}
```

```
package model.paperorder;

public class WhitePaperOrder extends AbstractPaperOrder
{
	/**
	 * Generated by Eclipse
	 */
	private static final long serialVersionUID = -3966465969285239587L;
}
```

```
package model.paperorder;

public class BrownPaperOrder extends AbstractPaperOrder
{
	/**
	 * Generated by Eclipse
	 */
	private static final long serialVersionUID = 1582952228765022555L;
}
```

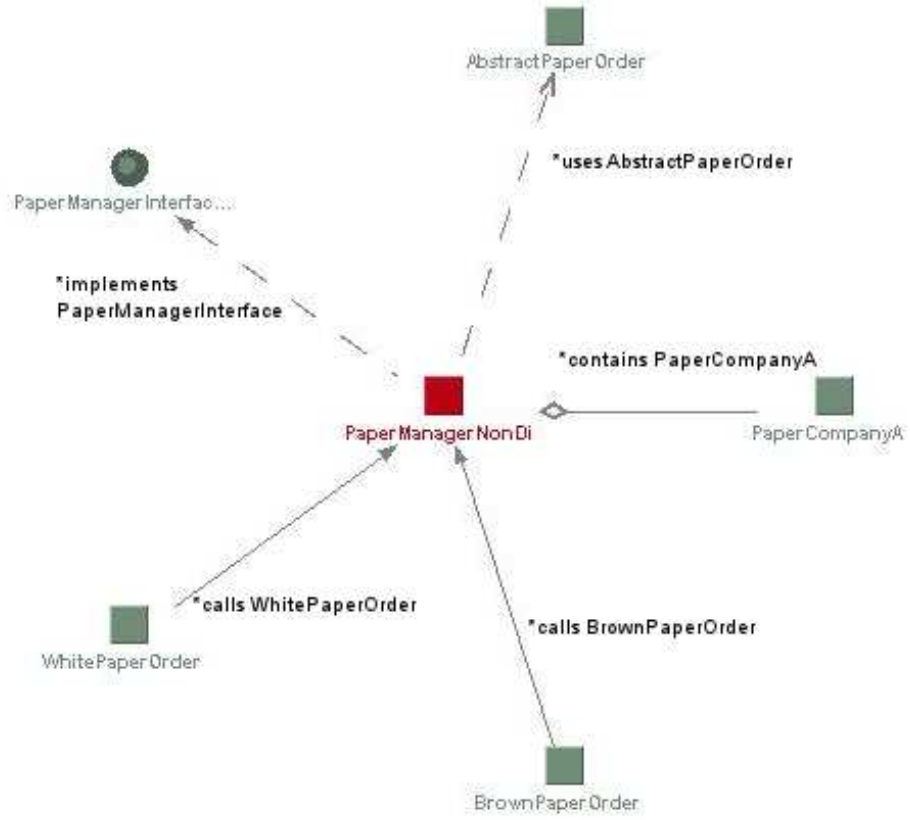
### *Implemented without dependency injection*

Without dependency injection the `PaperManagerNonDi` implementation of the `PaperManager` interface contains a reference to a `PaperProviderInterface` implementation. In this case this is the concrete class `PaperCompanyA`. `PaperCompanyA` class is meant to fail under all circumstances. This may sound unusual but actually this is done to prove how hard it becomes to test a component (`PaperManagerNonDi`) when a component it depends on (`PaperCompanyA`) is not available during test.

The import block in a Java source file is a good indication of the dependencies of the classes inside that source file. The `PaperManagerNonDi.java` class depends on 6 other classes (ignoring the classes that are part of Java language framework, like `java.util.Date` etc.):

```
import model.papermanager.PaperManagerInterface;
import model.paperorder.AbstractPaperOrder;
import model.paperorder.BrownPaperOrder;
import model.paperorder.UnknownPaperOrderException;
import model.paperorder.WhitePaperOrder;
import model.paperprovider.impl.PaperCompanyA;
```

With the 'IBM Structural Analysis for Java' toolkit[8] this can be visualized in an UML like notation:




[8] http://www.alphaworks.ibm.com/tech/sa4j

To make this dependency visualization more clear, the meaning of the lines between the components have been added (beginning with *). Also, this toolkit ignores Exception classes by default, that's why the `UnknownPaperOrderException` component is not displayed.

Below the complete code of the `PaperManagerNonDi` class is given.

### PaperManagerNonDi.java

```
package model.papermanager.impl;

import java.util.Collections;
import java.util.Date;
import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;

import model.papermanager.PaperManagerInterface;
import model.paperorder.AbstractPaperOrder;
import model.paperorder.BrownPaperOrder;
import model.paperorder.UnknownPaperOrderException;
import model.paperorder.WhitePaperOrder;
import model.paperprovider.impl.PaperCompanyA;

/**
 * An implementation of the PaperManagerInterface not making use of dependency
injection
 * @author Ricardo Lindooren
 */
public class PaperManagerNonDi implements PaperManagerInterface
{
	private PaperCompanyA paperCompanyA;
	private int amountOfBrownPaperInStock;
	private int amountOfWhitePaperInStock;
	private SortedSet<AbstractPaperOrder> paperOrdered;
	private int ORDER_WHITEPAPER_BELOW = 100;
	private int ORDER_BROWNPAPER_BELOW = 100;
	private long lastOrderId;

	/**
	 * Constructor initializing the reference to a paper company
	 */
	public PaperManagerNonDi()
	{
		paperCompanyA = new PaperCompanyA();
		paperOrdered = Collections.synchronizedSortedSet(new
TreeSet<AbstractPaperOrder>());
		lastOrderId = 0;
	}

	@Override
	public void brownPaperUsedInProductionProcess(int amountOfMeters)
	{
		amountOfBrownPaperInStock -= amountOfMeters;
		checkBrownPaperInStock();
	}

	@Override
	public void whitePaperUsedInProductionProcess(int amountOfMeters)
	{
		amountOfWhitePaperInStock -= amountOfMeters;
		checkWhitePaperInStock();
	}

	@Override
	public int getAmountOfBrownPaperInStock()
	{
		return amountOfBrownPaperInStock;
	}

	@Override
	public int getAmountOfWhitePaperInStock()
	{
```

```
                return amountOfWhitePaperInStock;
        }

        @Override
        public void paperOrderReceivedFromPaperProvider(AbstractPaperOrder apo) throws
UnknownPaperOrderException
        {
                if (paperOrdered.contains(apo))
                {
                        // Dispatch paper company
                        paperCompanyA.confirmReceivedPaperOrder(apo);
                        // Delete from local history
                        paperOrdered.remove(apo);
                }
                else
                {
                        throw new UnknownPaperOrderException("Order did not exist in
paper orders");
                }
        }

        @Override
        public void setAmountOfBrownPaperInStock(int amountOfMeters)
        {
                amountOfBrownPaperInStock = amountOfMeters;
        }

        @Override
        public void setAmountOfWhitePaperInStock(int amountOfMeters)
        {
                amountOfWhitePaperInStock = amountOfMeters;
        }

        @Override
        public Set<AbstractPaperOrder> getPaperCurrentlyInOrder()
        {
                return paperOrdered;
        }

        /**
         * Checks and orders white paper when needed
         */
        private void checkWhitePaperInStock()
        {
                if (getAmountOfWhitePaperInStock() < ORDER_WHITEPAPER_BELOW)
                {
                        WhitePaperOrder wpo = new WhitePaperOrder();
                        wpo.setId(getNewOrderId());
                        wpo.setAmount(200);
                        wpo.setOrderDate(new Date(System.currentTimeMillis()));
                        paperCompanyA.orderPaper(wpo);
                }
        }

        /**
         * Checks and orders brown paper when needed
         */
        private void checkBrownPaperInStock()
        {
                if (getAmountOfBrownPaperInStock() < ORDER_BROWNPAPER_BELOW)
                {
                        BrownPaperOrder bpo = new BrownPaperOrder();
                        bpo.setId(getNewOrderId());
                        bpo.setAmount(200);
                        bpo.setOrderDate(new Date(System.currentTimeMillis()));
                        // Keep in local history
                        paperOrdered.add(bpo);
                        // Dispatch to paper company
                        paperCompanyA.orderPaper(bpo);
                }
        }

        /**
         * Creates a new order Id
         * @return last order id + 1
         */
```

```
        private synchronized Long getNewOrderId()
        {
                return new Long(lastOrderId + 1);
        }
}
```

It consists of 106 total lines of code (TLOC). And the average McCabe cyclomatic complexity is 1,25.

The JUnit testcase for this component is given below.

### PaperManagerNonDiTest.java

```
package model.papermanager.impl;

import java.util.Iterator;

import junit.framework.TestCase;
import model.paperorder.AbstractPaperOrder;
import model.paperorder.BrownPaperOrder;
import model.paperorder.UnknownPaperOrderException;
import model.paperorder.WhitePaperOrder;

/**
 * JUnite testcase for class PaperManagerNonDi
 *
 * @see PaperManagerNonDi
 * @author Ricardo Lindooren
 */
public class PaperManagerNonDiTest extends TestCase
{
        private PaperManagerNonDi pmndiUnderTest;

        private int brownPaperInStockToStartWith = 500;
        private int whitePaperInStockToStartWith = 500;
        private int orderBrownPaperBelow = 100;
        private int orderWhitePaperBelow = 100;
        private int brownPaperAmountThatShouldBeOrdered = 200;
        private int whitePaperAmountThatShouldBeOrdered = 200;

        @Override
        protected void setUp() throws Exception
        {
                super.setUp();
                pmndiUnderTest = new PaperManagerNonDi();

        pmndiUnderTest.setAmountOfBrownPaperInStock(brownPaperInStockToStartWith);

        pmndiUnderTest.setAmountOfWhitePaperInStock(whitePaperInStockToStartWith);
        }

        public void testBrownPaperUsedInProductionProcess()
        {
                /* Test if amount in stock decreases correctly */
                int usedAmountOfMeters = 5;
                pmndiUnderTest.brownPaperUsedInProductionProcess(usedAmountOfMeters);
                assertEquals(brownPaperInStockToStartWith - usedAmountOfMeters,
pmndiUnderTest.getAmountOfBrownPaperInStock());

                /* Test if paper manager orders paper from paper company */
                // Clear all pending orders
                pmndiUnderTest.getPaperCurrentlyInOrder().clear();

                // Use paper so that a new order has to be placed
                pmndiUnderTest.setAmountOfBrownPaperInStock(orderBrownPaperBelow);
                pmndiUnderTest.brownPaperUsedInProductionProcess(1);
                assertEquals("There should be an order", false,
pmndiUnderTest.getPaperCurrentlyInOrder().isEmpty());

                // Check order validity
                Iterator<AbstractPaperOrder> paperOrderIterator =
pmndiUnderTest.getPaperCurrentlyInOrder().iterator();
```

```
                AbstractPaperOrder apo = paperOrderIterator.next();
                assertTrue(apo instanceof BrownPaperOrder);
                assertEquals(brownPaperAmountThatShouldBeOrdered, apo.getAmount());

                /* Impossible to test if PaperCompanyA has been called! */
        }

        public void testWhitePaperUsedInProductionProcess()
        {
                /* Test if amount in stock decreases correctly */
                int usedAmountOfMeters = 5;
                pmndiUnderTest.whitePaperUsedInProductionProcess(usedAmountOfMeters);
                assertEquals(whitePaperInStockToStartWith - usedAmountOfMeters,
pmndiUnderTest.getAmountOfWhitePaperInStock());

                /* Test if paper manager orders paper from paper company */
                // Clear all pending orders
                pmndiUnderTest.getPaperCurrentlyInOrder().clear();

                // Use paper so that a new order has to be placed
                pmndiUnderTest.setAmountOfWhitePaperInStock(orderWhitePaperBelow);
                pmndiUnderTest.whitePaperUsedInProductionProcess(1);
                assertEquals("There should be an order", false,
pmndiUnderTest.getPaperCurrentlyInOrder().isEmpty());

                // Check order validity
                Iterator<AbstractPaperOrder> paperOrderIterator =
pmndiUnderTest.getPaperCurrentlyInOrder().iterator();
                AbstractPaperOrder apo = paperOrderIterator.next();
                assertTrue(apo instanceof WhitePaperOrder);
                assertEquals(whitePaperAmountThatShouldBeOrdered, apo.getAmount());

                /* Impossible to test if PaperCompanyA has been called! */
        }

        public void testGetAmountOfBrownPaperInStock()
        {
                assertEquals(brownPaperInStockToStartWith,
pmndiUnderTest.getAmountOfBrownPaperInStock());
        }

        public void testGetAmountOfWhitePaperInStock()
        {
                assertEquals(whitePaperInStockToStartWith,
pmndiUnderTest.getAmountOfWhitePaperInStock());
        }

        public void testPaperOrderReceivedFromPaperProvider()
        {
                WhitePaperOrder wpo = new WhitePaperOrder();
                // fake Id
                wpo.setId(new Long(324));
                wpo.setAmount(1);

                UnknownPaperOrderException upoex = null;
                try
                {
                        pmndiUnderTest.paperOrderReceivedFromPaperProvider(wpo);
                }
                catch(UnknownPaperOrderException ex)
                {
                        upoex = ex;
                }
                assertNotNull("Unknown order should throw an exception", upoex);

                // Use paper so that a new order has to be placed
                pmndiUnderTest.setAmountOfBrownPaperInStock(orderBrownPaperBelow);
                pmndiUnderTest.brownPaperUsedInProductionProcess(1);
                Iterator<AbstractPaperOrder> paperOrderIterator =
pmndiUnderTest.getPaperCurrentlyInOrder().iterator();
                AbstractPaperOrder apo = paperOrderIterator.next();

                UnknownPaperOrderException upoex2 = null;
                try
                {
                        pmndiUnderTest.paperOrderReceivedFromPaperProvider(apo);
```

```
                }
                catch(UnknownPaperOrderException ex)
                {
                        upoex2 = ex;
                }

                assertNull(upoex2);

                /* Impossible to test if PaperCompanyA has been called! */
        }

        /**
         * Simple getter/setter test
         */
        public void testSetAmountOfBrownPaperInStock()
        {
                int testValue = 10;
                pmndiUnderTest.setAmountOfBrownPaperInStock(testValue);
                assertEquals(testValue, pmndiUnderTest.getAmountOfBrownPaperInStock());
        }

        /**
         * Simple getter/setter test
         */
        public void testSetAmountOfWhitePaperInStock()
        {
                int testValue = 20;
                pmndiUnderTest.setAmountOfWhitePaperInStock(testValue);
                assertEquals(testValue, pmndiUnderTest.getAmountOfWhitePaperInStock());
        }

        /**
         * Simple not null test on getter
         */
        public void testGetPaperCurrentlyInOrder()
        {
                assertNotNull(pmndiUnderTest.getPaperCurrentlyInOrder());
        }
}
```

It consist out of 108 TLOC containing 16 calls to JUnit assert methods. There are no lines of configuration code needed to initialize the dependencies (they are all referenced and initialized in the `PaperManagerNonDi` class).

According to the JUnit test report the success rate of this test is 62.50%, the reason why it is not 100% is that all tests that test methods depending on `PaperCompanyA` are failing since it is not available.

Class model.papermanager.impl.PaperManagerNonDiTest

| Name | Tests | Errors | Failures | Time(s) | Time Stamp | Host |
|---|---|---|---|---|---|---|
| PaperManagerNonDiTest | 8 | 3 | 0 | 0.750 | 2007-08-20T18:55:26 | RICARDO-XP |

Tests

| Name | Status | Type | Time(s) |
|---|---|---|---|
| testBrownPaperUsedInProductionProcess | Error | PaperCompanyA not available!<br>java.lang.RuntimeException: PaperCompanyA not available!<br>at model.paperprovider.impl.PaperCompanyA.orderPaper(PaperCompanyA.java:28)<br>at model.papermanager.impl.PaperManagerNonDi.checkBrownPaperInStock(PaperManagerNonDi.java:124)<br>at model.papermanager.impl.PaperManagerNonDi.brownPaperUsedInProductionProcess(PaperManagerNonDi.java:44)<br>at model.papermanager.impl.PaperManagerNonDiTest.testBrownPaperUsedInProductionProcess(PaperManagerNonDiTest.java:50) | 0.234 |
| testWhitePaperUsedInProductionProcess | Error | PaperCompanyA not available!<br>java.lang.RuntimeException: PaperCompanyA not available!<br>at model.paperprovider.impl.PaperCompanyA.orderPaper(PaperCompanyA.java:28)<br>at model.papermanager.impl.PaperManagerNonDi.checkWhitePaperInStock(PaperManagerNonDi.java:109)<br>at model.papermanager.impl.PaperManagerNonDi.whitePaperUsedInProductionProcess(PaperManagerNonDi.java:51)<br>at model.papermanager.impl.PaperManagerNonDiTest.testWhitePaperUsedInProductionProcess(PaperManagerNonDiTest.java:75) | 0.000 |
| testGetAmountOfBrownPaperInStock | Success | | 0.000 |
| testGetAmountOfWhitePaperInStock | Success | | 0.000 |
| testPaperOrderReceivedFromPaperProvider | Error | PaperCompanyA not available!<br>java.lang.RuntimeException: PaperCompanyA not available!<br>at model.paperprovider.impl.PaperCompanyA.orderPaper(PaperCompanyA.java:28)<br>at model.papermanager.impl.PaperManagerNonDi.checkBrownPaperInStock(PaperManagerNonDi.java:124)<br>at model.papermanager.impl.PaperManagerNonDi.brownPaperUsedInProductionProcess(PaperManagerNonDi.java:44)<br>at<br>model.papermanager.impl.PaperManagerNonDiTest.testPaperOrderReceivedFromPaperProvider(PaperManagerNonDiTest.java:117) | 0.000 |
| testSetAmountOfBrownPaperInStock | Success | | 0.000 |
| testSetAmountOfWhitePaperInStock | Success | | 0.000 |
| testGetPaperCurrentlyInOrder | Success | | 0.000 |

Properties »

Figure 7: Failing tests

The Cobertura testcoverage report tells us that 93% of the lines of code and 83% of all branches in the `PaperManagerNonDi` class are tested.

| Classes in this File | Line Coverage | Branch Coverage | Complexity |
|---|---|---|---|
| PaperManagerNonDi | 93% 38/41 | 83% 5/6 | 0 |

Figure 8: Code test coverage

This is also due to the `PaperCompanyA` class failing during tests.

```
66          @Override
67          public void paperOrderReceivedFromPaperProvider(AbstractPaperOrder apo) throws UnknownPaperOrderException
68          {
69  1               if (paperOrdered.contains(apo))
70              {
71                      // Dispatch paper company
72  0                   paperCompanyA.confirmReceivedPaperOrder(apo);
73                      // Delete from local history
74  0                   paperOrdered.remove(apo);
75              }
76              else
77              {
78  1                   throw new UnknownPaperOrderException("Order did not exist in paper orders");
79              }
80  0       }
```

Figure 9: Lines not executed during test(s)

The test cannot get around this failing component since it cannot be replaced with another `PaperProviderInterface` implementation.

## Note about the Metrics tool for Eclipse

While writing the code for the concrete `PaperManagerNonDi` class it became clear that the Metrics tool for Eclipse does correctly calculate the testability metric McCabe cyclometic complexity. For example the following method is given a value of 2 for testability:

```
private void checkWhitePaperInStock()
{
        if (getAmountOfWhitePaperInStock() < ORDER_WHITEPAPER_BELOW)
        {
                WhitePaperOrder wpo = new WhitePaperOrder();
```

```
                wpo.setId(getNewOrderId());
                wpo.setAmount(200);
                wpo.setOrderDate(new Date(System.currentTimeMillis()));
                paperCompanyA.orderPaper(wpo);
        }
}
```

Which is a correct value since there are two possible paths in this method's code structure; the amount of paper in stock is too low or not (in the form of the if-statement).

Another thing about the Metrics tool that actually is something good to know when interpreting the total lines of code (TLOC) metric is that it counts statements that continue on the next line as two lines of code, when it is actually only one statement. This became clear after automatically formatting the code with the help of Eclipse. Eclipse by default wraps long statements on more lines as an effort to make them better readable. Because this can be confusing we will not make use of auto formatting keeping each statement on not more than one line.
The example below shows how both statements, that are exactly the same and take equal amount of effort to write, are counted differently which can make the TLOC metric not valid if we use it as an indication of the amount of effort needed to write a component or test.

Counted as one line of code:

```
paperCompanyA.confirmReceivedPaperOrder(apo);
```

Counted as two lines of code:

```
paperCompanyA.
      confirmReceivedPaperOrder(apo);
```

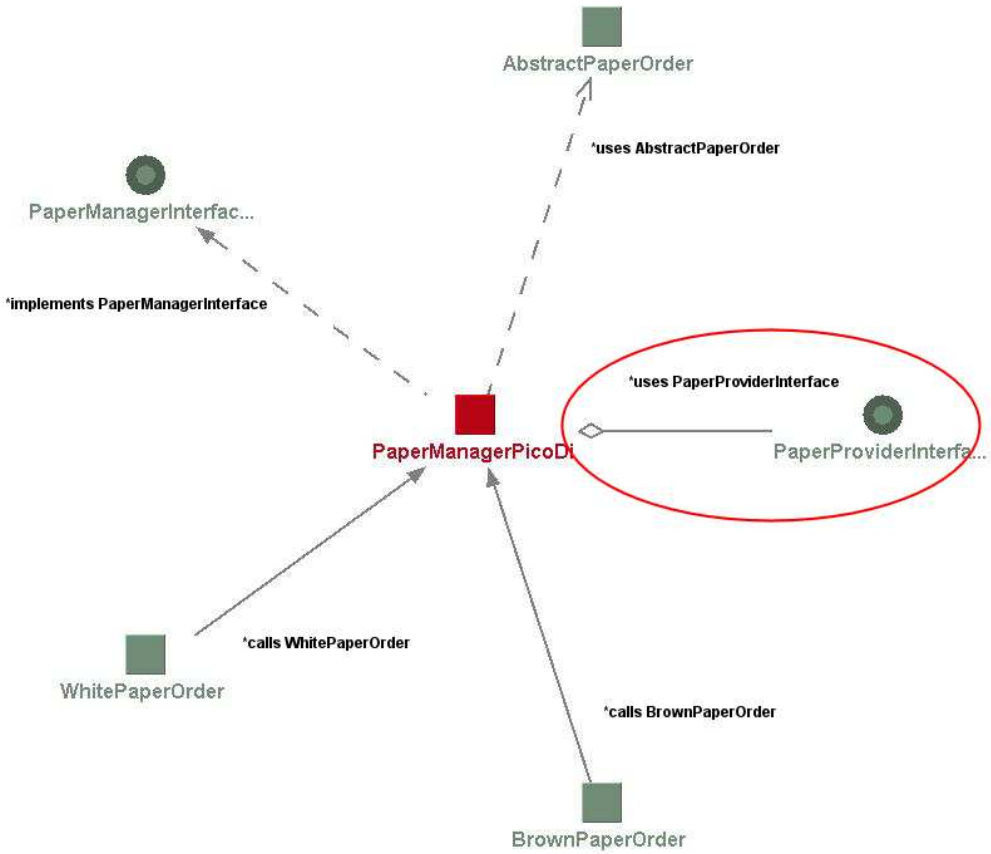
### *Implemented with PicoContainer dependency injection*

The dependency we will focus on when applying the dependency injection principle is the `PaperManagerInterface` implementation depending on a `PaperProviderInterface` implementation. We will use PicoContainer to be responsible for managing the desired dependencies with the help of dependency injection.

The `PaperManagerPicoDi` class makes it clear that it doesn't contain a reference to a specific `PaperProviderInterface` implementation. It is only aware of the `PaperProviderInterface` abstraction. Which can be seen in the import block..

```
import model.papermanager.PaperManagerInterface;
import model.paperorder.AbstractPaperOrder;
import model.paperorder.BrownPaperOrder;
import model.paperorder.UnknownPaperOrderException;
import model.paperorder.WhitePaperOrder;
import model.paperprovider.PaperProviderInterface;
```

…as well as in the 'IBM Structural Analysis for Java' toolkit visualization:



The complete implementation of the `PaperManagerPicoDi` class is given below

## PaperManagerPicoDi.java

```
package model.papermanager.impl;

import java.util.Collections;
import java.util.Date;
import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;

import model.papermanager.PaperManagerInterface;
import model.paperorder.AbstractPaperOrder;
import model.paperorder.BrownPaperOrder;
import model.paperorder.UnknownPaperOrderException;
import model.paperorder.WhitePaperOrder;
import model.paperprovider.PaperProviderInterface;

/**
 * An implementation of the PaperManagerInterface making use of PicoContainer
dependency injection
 * @author Ricardo Lindooren
 */
public class PaperManagerPicoDi implements PaperManagerInterface
{
	private PaperProviderInterface paperProvider;
	private int amountOfBrownPaperInStock;
	private int amountOfWhitePaperInStock;
	private SortedSet<AbstractPaperOrder> paperOrdered;
	private int ORDER_WHITEPAPER_BELOW = 100;
	private int ORDER_BROWNPAPER_BELOW = 100;
	private long lastOrderId;

	/**
	 * Constructor initializing the reference to a paper company
	 */
	public PaperManagerPicoDi(PaperProviderInterface currentPaperProvider)
	{
		paperProvider = currentPaperProvider;
		paperOrdered = Collections.synchronizedSortedSet(new
TreeSet<AbstractPaperOrder>());
		lastOrderId = 0;
	}

	@Override
	public void brownPaperUsedInProductionProcess(int amountOfMeters)
	{
		amountOfBrownPaperInStock -= amountOfMeters;
		checkBrownPaperInStock();
	}

	@Override
	public void whitePaperUsedInProductionProcess(int amountOfMeters)
	{
		amountOfWhitePaperInStock -= amountOfMeters;
		checkWhitePaperInStock();
	}

	@Override
	public int getAmountOfBrownPaperInStock()
	{
		return amountOfBrownPaperInStock;
	}

	@Override
	public int getAmountOfWhitePaperInStock()
	{
		return amountOfWhitePaperInStock;
	}

	@Override
	public void paperOrderReceivedFromPaperProvider(AbstractPaperOrder apo) throws
UnknownPaperOrderException
	{
```

```
                if (paperOrdered.contains(apo))
                {
                        // Dispatch paper company
                        paperProvider.confirmReceivedPaperOrder(apo);
                        // Delete from local history
                        paperOrdered.remove(apo);
                }
                else
                {
                        throw new UnknownPaperOrderException("Order did not exist in
paper orders");
                }
        }

        @Override
        public void setAmountOfBrownPaperInStock(int amountOfMeters)
        {
                amountOfBrownPaperInStock = amountOfMeters;
        }

        @Override
        public void setAmountOfWhitePaperInStock(int amountOfMeters)
        {
                amountOfWhitePaperInStock = amountOfMeters;
        }

        @Override
        public Set<AbstractPaperOrder> getPaperCurrentlyInOrder()
        {
                return paperOrdered;
        }

        /**
         * Checks and orders white paper when needed
         */
        private void checkWhitePaperInStock()
        {
                if (getAmountOfWhitePaperInStock() < ORDER_WHITEPAPER_BELOW)
                {
                        WhitePaperOrder wpo = new WhitePaperOrder();
                        wpo.setId(getNewOrderId());
                        wpo.setAmount(200);
                        wpo.setOrderDate(new Date(System.currentTimeMillis()));
                        // Keep in local history
                        paperOrdered.add(wpo);
                        // Dispatch to paper company
                        paperProvider.orderPaper(wpo);
                }
        }

        /**
         * Checks and orders brown paper when needed
         */
        private void checkBrownPaperInStock()
        {
                if (getAmountOfBrownPaperInStock() < ORDER_BROWNPAPER_BELOW)
                {
                        BrownPaperOrder bpo = new BrownPaperOrder();
                        bpo.setId(getNewOrderId());
                        bpo.setAmount(200);
                        bpo.setOrderDate(new Date(System.currentTimeMillis()));
                        // Keep in local history
                        paperOrdered.add(bpo);
                        // Dispatch to paper company
                        paperProvider.orderPaper(bpo);
                }
        }

        /**
         * Creates a new order Id
         * @return last order id + 1
         */
        private synchronized Long getNewOrderId()
        {
                return new Long(lastOrderId + 1);
        }
```

```
}
```

The main difference in source code between the `PaperManagerNonDi.java` and `PaperManagerPicoDi.java` classes is that the constructor of the last now contains an argument. This is because PicoContainer is based on the constructor dependency injection strategy. Which means that the dependency is injected during instantiation of the class.

The `PaperManagerPicoDi` implementation consists out of 106 TLOC. And the average McCabe complexity is 1,25.

The fact that the `PaperManagerInterface` dependency is injectable means that we should be able to inject a Mock implementation improving testability of this class (compared to the testability of the `PaperManagerNonDi` class).
The JUnit testcase for this component is given below.

### PaperManagerPicoDiTest.java

```
package model.papermanager.impl;

import java.util.Iterator;

import junit.framework.TestCase;
import model.paperorder.AbstractPaperOrder;
import model.paperorder.BrownPaperOrder;
import model.paperorder.UnknownPaperOrderException;
import model.paperorder.WhitePaperOrder;
import model.paperprovider.impl.PaperCompanyMock;

import org.picocontainer.MutablePicoContainer;
import org.picocontainer.defaults.DefaultPicoContainer;

/**
 * JUnite testcase for class PaperManagerNonDi
 *
 * @see PaperManagerNonDi
 * @author Ricardo Lindooren
 */
public class PaperManagerPicoDiTest extends TestCase
{
	private PaperManagerPicoDi pmpdiUnderTest;
	private PaperCompanyMock pctm;

	private int brownPaperInStockToStartWith = 500;
	private int whitePaperInStockToStartWith = 500;
	private int orderBrownPaperBelow = 100;
	private int orderWhitePaperBelow = 100;
	private int brownPaperAmountThatShouldBeOrdered = 200;
	private int whitePaperAmountThatShouldBeOrdered = 200;

	@Override
	protected void setUp() throws Exception
	{
		super.setUp();

		/* The PicoContainer configuration code */

		// Use the PicoContainer logic to manage the dependencies
		MutablePicoContainer picoContainer = new DefaultPicoContainer();

		// Register the used PaperCompany implementation (MOCK OBJECT)
		picoContainer.registerComponentImplementation("PaperCompany",
PaperCompanyMock.class);

		// Register the used PaperManager implementation
		picoContainer.registerComponentImplementation("PaperManager",
PaperManagerPicoDi.class);
```

```
                // Get the paper manager with the needed dependencies by PicoContainer
                pmpdiUnderTest = (PaperManagerPicoDi)
picoContainer.getComponentInstance("PaperManager");

                // Get the paper company test mock
                pctm = (PaperCompanyMock)
picoContainer.getComponentInstance("PaperCompany");


        pmpdiUnderTest.setAmountOfBrownPaperInStock(brownPaperInStockToStartWith);

        pmpdiUnderTest.setAmountOfWhitePaperInStock(whitePaperInStockToStartWith);
        }

        public void testBrownPaperUsedInProductionProcess()
        {
                /* Test if amount in stock decreases correctly */
                int usedAmountOfMeters = 5;
                pmpdiUnderTest.brownPaperUsedInProductionProcess(usedAmountOfMeters);
                assertEquals(brownPaperInStockToStartWith - usedAmountOfMeters,
pmpdiUnderTest.getAmountOfBrownPaperInStock());

                /* Test if paper manager orders paper from paper company */
                // Clear all pending orders
                pmpdiUnderTest.getPaperCurrentlyInOrder().clear();

                // Use paper so that a new order has to be placed
                pmpdiUnderTest.setAmountOfBrownPaperInStock(orderBrownPaperBelow);
                pmpdiUnderTest.brownPaperUsedInProductionProcess(1);
                assertEquals("There should be an order", false,
pmpdiUnderTest.getPaperCurrentlyInOrder().isEmpty());

                // Check order validity
                Iterator<AbstractPaperOrder> paperOrderIterator =
pmpdiUnderTest.getPaperCurrentlyInOrder().iterator();
                AbstractPaperOrder apo = paperOrderIterator.next();
                assertTrue(apo instanceof BrownPaperOrder);
                assertEquals(brownPaperAmountThatShouldBeOrdered, apo.getAmount());

                /* Possible to test if PaperCompanyA has been called! */
                assertFalse(pctm.checkCurrentlyProcessedPaperOrders().isEmpty());
                Iterator<AbstractPaperOrder> paperOrderIterator2 =
pctm.checkCurrentlyProcessedPaperOrders().iterator();
                AbstractPaperOrder apo2 = paperOrderIterator2.next();
                assertEquals(0, apo2.compareTo(apo));
        }

        public void testWhitePaperUsedInProductionProcess()
        {
                /* Test if amount in stock decreases correctly */
                int usedAmountOfMeters = 5;
                pmpdiUnderTest.whitePaperUsedInProductionProcess(usedAmountOfMeters);
                assertEquals(whitePaperInStockToStartWith - usedAmountOfMeters,
pmpdiUnderTest.getAmountOfWhitePaperInStock());

                /* Test if paper manager orders paper from paper company */
                // Clear all pending orders
                pmpdiUnderTest.getPaperCurrentlyInOrder().clear();

                // Use paper so that a new order has to be placed
                pmpdiUnderTest.setAmountOfWhitePaperInStock(orderWhitePaperBelow);
                pmpdiUnderTest.whitePaperUsedInProductionProcess(1);
                assertEquals("There should be an order", false,
pmpdiUnderTest.getPaperCurrentlyInOrder().isEmpty());

                // Check order validity
                Iterator<AbstractPaperOrder> paperOrderIterator =
pmpdiUnderTest.getPaperCurrentlyInOrder().iterator();
                AbstractPaperOrder apo = paperOrderIterator.next();
                assertTrue(apo instanceof WhitePaperOrder);
                assertEquals(whitePaperAmountThatShouldBeOrdered, apo.getAmount());

                /* Possible to test if PaperCompanyA has been called! */
                assertFalse(pctm.checkCurrentlyProcessedPaperOrders().isEmpty());
                Iterator<AbstractPaperOrder> paperOrderIterator2 =
pctm.checkCurrentlyProcessedPaperOrders().iterator();
```

```
                AbstractPaperOrder apo2 = paperOrderIterator2.next();
                assertEquals(0, apo2.compareTo(apo));
        }

        public void testGetAmountOfBrownPaperInStock()
        {
                assertEquals(brownPaperInStockToStartWith,
pmpdiUnderTest.getAmountOfBrownPaperInStock());
        }

        public void testGetAmountOfWhitePaperInStock()
        {
                assertEquals(whitePaperInStockToStartWith,
pmpdiUnderTest.getAmountOfWhitePaperInStock());
        }

        public void testPaperOrderReceivedFromPaperProvider()
        {
                WhitePaperOrder wpo = new WhitePaperOrder();
                // fake Id
                wpo.setId(new Long(324));
                wpo.setAmount(1);

                UnknownPaperOrderException upoex = null;
                try
                {
                        pmpdiUnderTest.paperOrderReceivedFromPaperProvider(wpo);
                }
                catch(UnknownPaperOrderException ex)
                {
                        upoex = ex;
                }
                assertNotNull("Unknown order should throw an exception", upoex);

                // Use paper so that a new order has to be placed
                pmpdiUnderTest.setAmountOfBrownPaperInStock(orderBrownPaperBelow);
                pmpdiUnderTest.brownPaperUsedInProductionProcess(1);
                Iterator<AbstractPaperOrder> paperOrderIterator =
pmpdiUnderTest.getPaperCurrentlyInOrder().iterator();
                AbstractPaperOrder apo = paperOrderIterator.next();

                /* Possible to test if PaperCompanyA has been called! */
                assertFalse(pctm.checkCurrentlyProcessedPaperOrders().isEmpty());


                UnknownPaperOrderException upoex2 = null;
                try
                {
                        pmpdiUnderTest.paperOrderReceivedFromPaperProvider(apo);
                }
                catch(UnknownPaperOrderException ex)
                {
                        upoex2 = ex;
                }

                assertNull(upoex2);
                assertTrue(pmpdiUnderTest.getPaperCurrentlyInOrder().isEmpty());

                /* Possible to test if PaperCompanyA has been called! */
                assertTrue(pctm.checkCurrentlyProcessedPaperOrders().isEmpty());
        }

        /**
         * Simple getter/setter test
         */
        public void testSetAmountOfBrownPaperInStock()
        {
                int testValue = 10;
                pmpdiUnderTest.setAmountOfBrownPaperInStock(testValue);
                assertEquals(testValue, pmpdiUnderTest.getAmountOfBrownPaperInStock());
        }

        /**
         * Simple getter/setter test
         */
        public void testSetAmountOfWhitePaperInStock()
```

```
        {
                int testValue = 20;
                pmpdiUnderTest.setAmountOfWhitePaperInStock(testValue);
                assertEquals(testValue, pmpdiUnderTest.getAmountOfWhitePaperInStock());
        }

        /**
         * Simple not null test on getter
         */
        public void testGetPaperCurrentlyInOrder()
        {
                assertNotNull(pmpdiUnderTest.getPaperCurrentlyInOrder());
        }
}
```

This testcase contains 126 TLOC. Of which 5 LOC are used to configure the dependencies managed by PicoContainer. It also contains 22 calls to JUnit assert methods. The reason for the difference in the amount of TLOC compared with the `PaperManagerNonDiTest` JUnit testcase is that this testcase contains PicoContainer configuration code as well as more test code (calls to JUnit assert methods) used to test values in the `PaperCompanyMock` class that's injected in `PaperManagerPicoDi` during the setup before each test.

According to the JUnit test report the success rate of this test is 100%. The Cobertura test coverage report shows that all LOC and branches are executed during the test.

| Classes in this Package | Line Coverage | Branch Coverage | Complexity |
|---|---|---|---|
| PaperManagerPicoDi | 100% 42/42 | 100% 6/6 | 0 |

Figure 10: Code test coverage

The code of the mock object used to test the behavior of `PaperManagerPicoDi` class more rigorously is given below.

## PaperCompanyMock.java

```
package model.paperprovider.impl;

import java.util.Collections;
import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;

import model.paperorder.AbstractPaperOrder;
import model.paperorder.UnknownPaperOrderException;
import model.paperprovider.PaperProviderInterface;

public class PaperCompanyMock implements PaperProviderInterface
{
        private SortedSet<AbstractPaperOrder> paperOrdered;

        public PaperCompanyMock()
        {
                paperOrdered = Collections.synchronizedSortedSet(new
TreeSet<AbstractPaperOrder>());
        }

        @Override
        public Set<AbstractPaperOrder> checkCurrentlyProcessedPaperOrders()
        {
                return paperOrdered;
        }

        @Override
        public void confirmReceivedPaperOrder(AbstractPaperOrder apo)
                        throws UnknownPaperOrderException
        {
                if (paperOrdered.contains(apo))
```

```
                {
                        paperOrdered.remove(apo);
                }
                else
                {
                        throw new UnknownPaperOrderException("Order did not exist in
paper orders");
                }
        }

        @Override
        public void orderPaper(AbstractPaperOrder apo)
        {
                paperOrdered.add(apo);
        }
}
```

### *Implemented with Java EE5 dependency injection*

The Java Enterprise Edition 5 implementation is in the form of a so called Enterprise Java Bean (EJB). In this most recent edition of their enterprise Java framework Sun's has attempted to make it more easy for developers to develop EJB's. Annotations give developers the possibility to manage dependencies instead of having to make use of XML configuration files.

With the `@EJB` annotation it is possible to let a Java application server inject an implementing bean:

```
@Stateless
public class PaperManagerBean implements PaperManagerInterface
{
	@EJB
	private PaperProviderInterface paperProvider;
```

The `@Stateless` annotation indicates that the application server should manage the life cycle of this EJB as a Stateless Session Bean. Meaning that its instantiation is not bound to a specific client that makes use of it (a Statefull Sesion Bean on the other hand is always bound to a single client session).

The `@EJB` annotation example given above proved un-testable with a JUnit testcase. The reason for this was that a managed environment (like a Java application server) is needed to realize this form of field dependency injection.
Fortunately the `@EJB` annotation also works on setter methods:

```
@EJB
public void setPaperProvider(PaperProviderInterface
paperProviderImplementation)
{
	paperProvider = paperProviderImplementation;
}
```

The complete implementation for this class is given below.

### PaperManagerBean.java

```
package model.papermanager.impl;

import java.util.Collections;
import java.util.Date;
import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;

import javax.ejb.EJB;
import javax.ejb.Stateless;

import model.papermanager.PaperManagerInterface;
import model.paperorder.AbstractPaperOrder;
import model.paperorder.BrownPaperOrder;
import model.paperorder.UnknownPaperOrderException;
import model.paperorder.WhitePaperOrder;
import model.paperprovider.PaperProviderInterface;

/**
 * An implementation of the PaperManagerInterface making use of EJB dependency
injection
```

```
 * @author Ricardo Lindooren
 */
@Stateless
public class PaperManagerBean implements PaperManagerInterface
{
	private PaperProviderInterface paperProvider;

	private int amountOfBrownPaperInStock;
	private int amountOfWhitePaperInStock;
	private SortedSet<AbstractPaperOrder> paperOrdered;
	private int ORDER_WHITEPAPER_BELOW = 100;
	private int ORDER_BROWNPAPER_BELOW = 100;
	private long lastOrderId;

	/**
	 * Non argument constructor
	 */
	public PaperManagerBean()
	{
		paperOrdered = Collections.synchronizedSortedSet(new
TreeSet<AbstractPaperOrder>());
		lastOrderId = 0;
	}

	@Override
	public void brownPaperUsedInProductionProcess(int amountOfMeters)
	{
		amountOfBrownPaperInStock -= amountOfMeters;
		checkBrownPaperInStock();
	}

	@Override
	public void whitePaperUsedInProductionProcess(int amountOfMeters)
	{
		amountOfWhitePaperInStock -= amountOfMeters;
		checkWhitePaperInStock();
	}

	@Override
	public int getAmountOfBrownPaperInStock()
	{
		return amountOfBrownPaperInStock;
	}

	@Override
	public int getAmountOfWhitePaperInStock()
	{
		return amountOfWhitePaperInStock;
	}

	@Override
	public void paperOrderReceivedFromPaperProvider(AbstractPaperOrder apo) throws
UnknownPaperOrderException
	{
		if (paperOrdered.contains(apo))
		{
			// Dispatch paper company
			paperProvider.confirmReceivedPaperOrder(apo);
			// Delete from local history
			paperOrdered.remove(apo);
		}
		else
		{
			throw new UnknownPaperOrderException("Order did not exist in
paper orders");
		}
	}

	@Override
	public void setAmountOfBrownPaperInStock(int amountOfMeters)
	{
		amountOfBrownPaperInStock = amountOfMeters;
	}

	@Override
	public void setAmountOfWhitePaperInStock(int amountOfMeters)
```

```
        {
                amountOfWhitePaperInStock = amountOfMeters;
        }

        @Override
        public Set<AbstractPaperOrder> getPaperCurrentlyInOrder()
        {
                return paperOrdered;
        }

        /**
         * Checks and orders white paper when needed
         */
        private void checkWhitePaperInStock()
        {
                if (getAmountOfWhitePaperInStock() < ORDER_WHITEPAPER_BELOW)
                {
                        WhitePaperOrder wpo = new WhitePaperOrder();
                        wpo.setId(getNewOrderId());
                        wpo.setAmount(200);
                        wpo.setOrderDate(new Date(System.currentTimeMillis()));
                        // Keep in local history
                        paperOrdered.add(wpo);
                        // Dispatch to paper company
                        paperProvider.orderPaper(wpo);
                }
        }

        /**
         * Checks and orders brown paper when needed
         */
        private void checkBrownPaperInStock()
        {
                if (getAmountOfBrownPaperInStock() < ORDER_BROWNPAPER_BELOW)
                {
                        BrownPaperOrder bpo = new BrownPaperOrder();
                        bpo.setId(getNewOrderId());
                        bpo.setAmount(200);
                        bpo.setOrderDate(new Date(System.currentTimeMillis()));
                        // Keep in local history
                        paperOrdered.add(bpo);
                        // Dispatch to paper company
                        paperProvider.orderPaper(bpo);
                }
        }

        /**
         * Creates a new order Id
         * @return last order id + 1
         */
        private synchronized Long getNewOrderId()
        {
                return new Long(lastOrderId + 1);
        }

        @EJB
        public void setPaperProvider(PaperProviderInterface
paperProviderImplementation)
        {
                paperProvider = paperProviderImplementation;
        }
}
```

This class source file contains 113 TLOC and the average McCabe cyclomatic complexity is 1,23. It has only two real differences with the `PaperManagerPicoDi` class. These are that it has a non-argument constructor because it doesn't use the constructor injection strategy (having a non-argument constructor is also mandatory for an EJB). Instead it has an extra setter method because it uses the setter injection strategy.

The setter injection makes it possible to inject the needed dependency from within a testcase. The testcase used is given below.

### PaparManagerBeanTest.java

```
package model.papermanager.impl;

import java.util.Iterator;

import junit.framework.TestCase;
import model.paperorder.AbstractPaperOrder;
import model.paperorder.BrownPaperOrder;
import model.paperorder.UnknownPaperOrderException;
import model.paperorder.WhitePaperOrder;
import model.paperprovider.impl.PaperCompanyMock;

/**
 * JUnite testcase for class PaperManagerBean
 *
 * @see PaperManagerNonDi
 * @author Ricardo Lindooren
 */
public class PaperManagerBeanTest extends TestCase
{
	private PaperManagerBean pmbUnderTest;
	private PaperCompanyMock pctm;

	private int brownPaperInStockToStartWith = 500;
	private int whitePaperInStockToStartWith = 500;
	private int orderBrownPaperBelow = 100;
	private int orderWhitePaperBelow = 100;
	private int brownPaperAmountThatShouldBeOrdered = 200;
	private int whitePaperAmountThatShouldBeOrdered = 200;

	@Override
	protected void setUp() throws Exception
	{
		super.setUp();

		pctm = new PaperCompanyMock();

		pmbUnderTest = new PaperManagerBean();
		pmbUnderTest.setPaperProvider(pctm);


	pmbUnderTest.setAmountOfBrownPaperInStock(brownPaperInStockToStartWith);

	pmbUnderTest.setAmountOfWhitePaperInStock(whitePaperInStockToStartWith);
	}

	public void testBrownPaperUsedInProductionProcess()
	{
		/* Test if amount in stock decreases correctly */
		int usedAmountOfMeters = 5;
		pmbUnderTest.brownPaperUsedInProductionProcess(usedAmountOfMeters);
		assertEquals(brownPaperInStockToStartWith - usedAmountOfMeters,
pmbUnderTest.getAmountOfBrownPaperInStock());

		/* Test if paper manager orders paper from paper company */
		// Clear all pending orders
		pmbUnderTest.getPaperCurrentlyInOrder().clear();

		// Use paper so that a new order has to be placed
		pmbUnderTest.setAmountOfBrownPaperInStock(orderBrownPaperBelow);
		pmbUnderTest.brownPaperUsedInProductionProcess(1);
		assertEquals("There should be an order", false,
pmbUnderTest.getPaperCurrentlyInOrder().isEmpty());

		// Check order validity
		Iterator<AbstractPaperOrder> paperOrderIterator =
pmbUnderTest.getPaperCurrentlyInOrder().iterator();
		AbstractPaperOrder apo = paperOrderIterator.next();
		assertTrue(apo instanceof BrownPaperOrder);
		assertEquals(brownPaperAmountThatShouldBeOrdered, apo.getAmount());
```

```
                /* Possible to test if PaperCompanyA has been called! */
                assertFalse(pctm.checkCurrentlyProcessedPaperOrders().isEmpty());
                Iterator<AbstractPaperOrder> paperOrderIterator2 =
pctm.checkCurrentlyProcessedPaperOrders().iterator();
                AbstractPaperOrder apo2 = paperOrderIterator2.next();
                assertEquals(0, apo2.compareTo(apo));
        }

        public void testWhitePaperUsedInProductionProcess()
        {
                /* Test if amount in stock decreases correctly */
                int usedAmountOfMeters = 5;
                pmbUnderTest.whitePaperUsedInProductionProcess(usedAmountOfMeters);
                assertEquals(whitePaperInStockToStartWith - usedAmountOfMeters,
pmbUnderTest.getAmountOfWhitePaperInStock());

                /* Test if paper manager orders paper from paper company */
                // Clear all pending orders
                pmbUnderTest.getPaperCurrentlyInOrder().clear();

                // Use paper so that a new order has to be placed
                pmbUnderTest.setAmountOfWhitePaperInStock(orderWhitePaperBelow);
                pmbUnderTest.whitePaperUsedInProductionProcess(1);
                assertEquals("There should be an order", false,
pmbUnderTest.getPaperCurrentlyInOrder().isEmpty());

                // Check order validity
                Iterator<AbstractPaperOrder> paperOrderIterator =
pmbUnderTest.getPaperCurrentlyInOrder().iterator();
                AbstractPaperOrder apo = paperOrderIterator.next();
                assertTrue(apo instanceof WhitePaperOrder);
                assertEquals(whitePaperAmountThatShouldBeOrdered, apo.getAmount());

                /* Possible to test if PaperCompanyA has been called! */
                assertFalse(pctm.checkCurrentlyProcessedPaperOrders().isEmpty());
                Iterator<AbstractPaperOrder> paperOrderIterator2 =
pctm.checkCurrentlyProcessedPaperOrders().iterator();
                AbstractPaperOrder apo2 = paperOrderIterator2.next();
                assertEquals(0, apo2.compareTo(apo));
        }

        public void testGetAmountOfBrownPaperInStock()
        {
                assertEquals(brownPaperInStockToStartWith,
pmbUnderTest.getAmountOfBrownPaperInStock());
        }

        public void testGetAmountOfWhitePaperInStock()
        {
                assertEquals(whitePaperInStockToStartWith,
pmbUnderTest.getAmountOfWhitePaperInStock());
        }

        public void testPaperOrderReceivedFromPaperProvider()
        {
                WhitePaperOrder wpo = new WhitePaperOrder();
                // fake Id
                wpo.setId(new Long(324));
                wpo.setAmount(1);

                UnknownPaperOrderException upoex = null;
                try
                {
                        pmbUnderTest.paperOrderReceivedFromPaperProvider(wpo);
                }
                catch(UnknownPaperOrderException ex)
                {
                        upoex = ex;
                }
                assertNotNull("Unknown order should throw an exception", upoex);

                // Use paper so that a new order has to be placed
                pmbUnderTest.setAmountOfBrownPaperInStock(orderBrownPaperBelow);
                pmbUnderTest.brownPaperUsedInProductionProcess(1);
                Iterator<AbstractPaperOrder> paperOrderIterator =
pmbUnderTest.getPaperCurrentlyInOrder().iterator();
```

```
                AbstractPaperOrder apo = paperOrderIterator.next();

                /* Possible to test if PaperCompanyA has been called! */
                assertFalse(pctm.checkCurrentlyProcessedPaperOrders().isEmpty());


                UnknownPaperOrderException upoex2 = null;
                try
                {
                        pmbUnderTest.paperOrderReceivedFromPaperProvider(apo);
                }
                catch(UnknownPaperOrderException ex)
                {
                        upoex2 = ex;
                }

                assertNull(upoex2);
                assertTrue(pmbUnderTest.getPaperCurrentlyInOrder().isEmpty());

                /* Possible to test if PaperCompanyA has been called! */
                assertTrue(pctm.checkCurrentlyProcessedPaperOrders().isEmpty());
        }

        /**
         * Simple getter/setter test
         */
        public void testSetAmountOfBrownPaperInStock()
        {
                int testValue = 10;
                pmbUnderTest.setAmountOfBrownPaperInStock(testValue);
                assertEquals(testValue, pmbUnderTest.getAmountOfBrownPaperInStock());
        }

        /**
         * Simple getter/setter test
         */
        public void testSetAmountOfWhitePaperInStock()
        {
                int testValue = 20;
                pmbUnderTest.setAmountOfWhitePaperInStock(testValue);
                assertEquals(testValue, pmbUnderTest.getAmountOfWhitePaperInStock());
        }

        /**
         * Simple not null test on getter
         */
        public void testGetPaperCurrentlyInOrder()
        {
                assertNotNull(pmbUnderTest.getPaperCurrentlyInOrder());
        }
}
```

This testcase contains 122 TLOC of which 3 LOC are used to set up the dependencies. It also contains 22 calls to JUnit assert methods and uses the same mock object also used to test the PicoContainer implementation.
Both the JUnit and Cobertura reports indicates it also scores the same as the `PaperManagerPicoDiTest` testcase; 100% success and a complete coverage of all lines and branches:

| Classes in this Package | Line Coverage | Branch Coverage | Complexity |
|---|---|---|---|
| PaperManagerBean | 100% 43/43 | 100% 6/6 | 0 |

Figure 11: Code test coverage

A note about testing an EJB class with the JUnit test framework is that by default the `@EJB` annotations are not understood and thus the test fails completely. To overcome this problem it is possible to make the needed Java EE5 libraries available. In this

case we used j2ee.jar and javaee.jar that are distributed with the community Sun Java application server called GlassFish.

### Note about the Metrics tool for Eclipse

The average cyclometic complexity of the `PaperManagerBean` class (1,23) is lower than that of the `PaperManagerPicoDi` class (1,25). The reason for this that the Metrics tool for Eclipse calculates the McCabe complexity per method so the average is based on the amount of methods and their indication of testability. The `PaperManagerBean` has one method more (the setter method used for injecting the `PaperProviderInterface` implementation) with a cyclomatic complexity of 1. This causes the difference for this metric although the logic defined in both classes doesn't really differ from each other.

Another thing maybe good to know when interpreting the TLOC metric is that the Metrics tool counts annotations as a line of code. And while investigating this, I also came to the conclusion that braces on a new line are also counted as a line of code.

So the next example is counted as 5 LOC:

```
@EJB
public void setPaperProvider(PaperProviderInterface
paperProviderImplementation)
{
      paperProvider = paperProviderImplementation;
}
```

And commenting out the annotation and placing everything on one line is counted as one line of code (LOC = 1):

```
//@EJB
public void setPaperProvider(PaperProviderInterface
paperProviderImplementation){paperProvider =
paperProviderImplementation;}
```

Therefore to give the LOC metric the same meaning for all java source files we use formatting as in the '5 LOC' example above. Meaning; method-signature on one line, braces on a new line and each statement on a new line.

### *What about the other dependencies?*

In the paper manager example the `PaperManagerInterface` implementing classes depend on a `PaperProviderInterface` implementation. But the paper manager contains more dependencies.

If we look at the concrete class `PaperManagerPicoDi` for example; it implements the `PaperManagerInterface` with the goal to manage its `PaperProviderInterface` dependency with dependency injection managed by PicoContainer. But since it communicates in means of paper order objects with a `PaperProviderInterface` implementation it also depends on the classes `AbstractPaperOrder`, `WhitePaperOrder` and `BrownPaperOrder`.

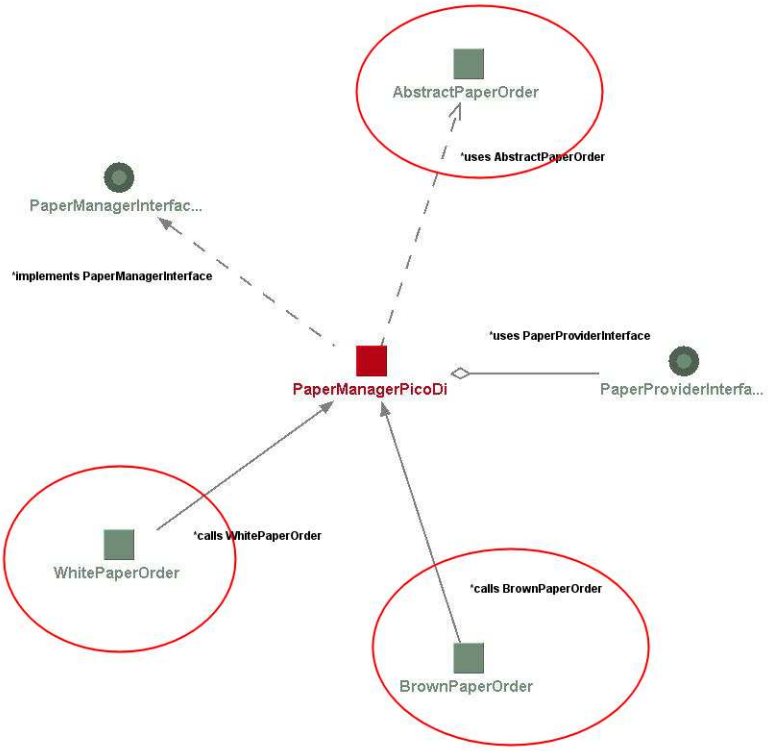



Figure 12: Still existing dependencies

One might question this design right away. But for this research it was desirable to create an example project containing different kind of dependencies, ignoring the fact that this might result in a bad design.

The result of this design is that both `PaperManagerInterface` and `PaperProviderInterface` implementations depend on the paper order classes. As said; to keep communication between both implementations generic `AbstractPaperOrder` objects are communicated between them both. The signature of the `PaperProviderInterface` `orderPaper` method demonstrates this:

```
public void orderPaper(AbstractPaperOrder apo);
```

Instead of an abstract class that needs to be extended by another class (in this case the `WhitePaperOrder` and `BrownPaperOrder` classes) an interface could also have been used, for example named `PaperOrderInterface`. The reason for choosing the abstract class approach is because the actual order implementation classes do not differ from each other, so the abstract class is used to contain the shared implementation logic.

But if an `AbstractPaperOrder` is communicated why does the `PaperManagerPicoDi` implementation also depend on the `WhitePaperOrder` and `BrownPaperOrder` classes? The reason is that the `PaperManagerPicoDi` class contains logic for creating instances of the correct order.

Dependency injection in this case seemed unusable because the `PaperManagerPicoDi` class creates new instances itself whenever they are needed. It is interesting, as a side exploration, to find out if it is possible to eliminate these dependencies as well.

A factory component can be used to create the correct instances of an `AbstractPaperOrder` without the `PaperManagerPicoDi` knowing if it is an instance of `WhitePaperOrder` or `BrownPaperOrder`. The factory could then contain the following two methods for example;

```
public AbstractPaperOrder getNewBrownPaperOrder();

public AbstractPaperOrder getNewWhitePaperOrder();
```

Unfortunatly the `PaperManagerPicoDi` class then depends on this factory component. Thinking further it also proved to be possible to create an interface for such a factory component. A `PaperManagerInterface` implementation can then be extended with a possibility to inject an implementation of the factory interface. It then only depends on the `PaperProviderInterface` and the `PaperOrderFactoryInterface` as well as the `AbstractPaperOrder` class, but no longer on the extending classes `WhitePaperOrder` and `BrownPaperOrder`.

Unfortunately this approach cannot be used at the other end of the communication; the `PaperProviderInterface` implementations still need to find out what the exact implementing paper order class is. This could for example be done by using Java's `instanceof`:

```
if (apo instanceof BrownPaperOrder)
{
    // logic for processing an order for brown paper
}
else if (apo instanceof WhitePaperOrder)
{
    // logic for processing an order for white paper
}
```

So there seems no escape for a `PaperProviderInterface` implementation when it comes to depending on `AbstractPaperOrder` extending classes. Next to that the

`instanceof` example doesn't look very sophisticated; this solution feels more like fixing a bad design.

Another approach to break free from the `AbstractPaperOrder`, `WhitePaperOrder` and `BrownPaperOrder` class dependencies is to not communicate objects at all in this case. A design principle which is also promoted by the Law of Demeter (sometimes also called 'law of good design' [Lieberherr88]). In short the Law of Demeter (LoD) describes how components should communicate with each other; this can be summarized as 'Only talk to your immediate friends'. Components should know as little possible of the structure of the software product. This means that components should for example not be aware of subcomponents..

Applying the LoD on the paper manager to break with the paper order dependencies can for example look like this when we focus on the ordering part of the `PaperProviderInterface` abstraction:

```
public void orderBrownPaper(int meters);

public void orderWhitePaper(int meters);
```

instead of:

```
public void orderPaper(AbstractPaperOrder apo);
```

[Lieberherr88] gives a more thorough description of the principles of the LoD but also the trade-offs; the possible increase in number of methods and arguments for methods, which can result in making maintenance more difficult. In the paper manager example this is for example possible when the company also wants to be able to order green paper; another method is then needed in the `PaperProviderInterface` abstraction; `orderGreenPaper(int meters)`.

A short but good and more complete example of the effects of implementing the LoD is given in [Bock??].

# References


| | |
|---|---|
| [Zeller05] | Andreas Zeller – Why Programs Fail |
| [SWEBOK04] | Software Engineering Body Of Knowledge |
| [Gelperin88] | David Gelperin et. al – The Growth of Software Testing |
| [Kaner99] | Cem Kaner et. al – Test Types and their Place in the Software Development Process |
| [Binder94] | Robert V. Binder – Design for Testability |
| [Dijkstra69] | Edsger W. Dijkstra – Notes on Structured Programming |
| [Jackson03] | Daniel Jackson – Module Dependences in Software Design |
| [Nene05] | Dhananjay Nene – A beginners guide to Dependency Injection |
| [Martin96] | Robert C. Martin – The Dependency Inversion Principle |
| [McConnel04] | Steve McConnell – Code complete (2nd Edition) |
| [Lieberherr88] | K. Lieberherr et. al – Object-Oriented Programming: An Objective Sense of Style |
| [Bock??] | David Bock – The Paperboy, The Wallet, and The Law Of Demeter |
| [Bruntink03] | Magiel Bruntink – Testability of Object-Oriented Systems: a Metrics-based Approach |
| [Weiskotten06] | Jeremy Weiskotten – Dependency Injection & Testable Objects |
| [Christensen03] | Henrik Baerbak Christensen – Systematic Testing should bot be a Topic in the Computer Science Curriculum! |
| [Whittaker00] | James A. Whittaker – What Is Software Testing? And Why Is It So Hard? |
| [Fowler04] | Martin Fowler – Inversion of Control Containers and the Dependency Injection pattern |
| [Wikipedia-DI] | http://en.wikipedia.org/wiki/Dependency_injection |
| [Wikipedia-Complexiteitsgraad] | http://nl.wikipedia.org/wiki/Complexiteitsgraad |
| [CodeHausPico] | http://docs.codehaus.org/display/PICO/IoC+Types |