Delft University of Technology

Computer Science

Bachelor thesis

Dependency Injection vs Dedicated Test

Author:
Irati Casi

Supervisor:
Mauricio Aniche

July 7, 2018

Abstract

This document is an analytical research about the relationship between the use of the dependency injection principle and the use of unit testing in object-oriented programming. Source code and javadoc of the tool is available in `https://github.com/iraticasi/dip_vs_testability` and in `https://zenodo.org/record/1307178#.W0DoanYzZhE`.

1 Introduction

In todays business world, software is everywhere and its quality is becoming fundamental. Along with this, the knowledge of the importance and benefits of software testing is increasing. Testing not only improves the quality of the software but also helps measuring its reliability. With this in mind, now we should pay attention to what makes our code easier to test.
As each project is unique, there is no such thing as "the best method for testing". However, software testing best practices can ease our work. Several researches have been made about this topic and testing experts have shared their insights.
Dependency injection is said to improve testing since it enables mock practice to perform testing in isolation. This paper aims to discover whether the assertion holds. For this purpose, an analysis of various java projects will be performed in search of a relationship between the use of the principle and how the class is being tested.

2 Background

Dependency Injection Principle in object-oriented programming is a design principle whereby objects are supplied to a class, instead of being created inside the class.
A simple example of a class that does not follow the principle could be:

```
class A{
   B b;
   A(){
      this.b = new B();
   }
}
```

And the alternative using dependency injection is the following:

```
class A{
   B b;
   A(B b){
      this.b = b;
   }
}
```

Mock objects simulate the behavior of real objects. They are usually created to test some other object in isolation. In a unit test, mock objects can simulate the behavior of complex, real objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test.

Testability can be defined as *the degree to which a software artifact supports testing in a given test context.* It involves two components: the effort and effectiveness of software tests .

When objects are not created inside the class, mock objects can be injected.

Therefore, it is reasonable to believe that dependency injection simplifies unit testing.

3 Related work

As mention in section 1, we can find testing good practice that has been shared on the internet. In 2008, Google shared a guide to write testable code [1] that mentions 4 flaws: "Constructor does Real Work", "Digging into Collaborators", "Brittle Global State & Singletons" and "Class Does Too Much". It also provides warning signs and alternatives.

There are also several researches about how different factors affect testability: Gao, J., Gupta, K., Gupta, S., & Shim, S. (2002) [2] focus on how to build testable software components by increasing their testability ; Bruntink, M., & van Deursen, A. (2006) [3] investigate testability factors in object-oriented software systems; Voas, J., & Kassab, L. (1999) [4] study the effects of assertions; Ma, L., Zhang, C., Yu, B., & Sato, H. (2015) [5] analyze the influence of code visibility...

Some researchers focus on dependency injection effects: Yang, H. Y., Tempero, E., & Melton, H. (2008) [6] examine the use of dependency injection and its impact in software quality attributes such as extensibility, modifiability, testability and reusability; Razina, E., & Janzen, D. S. (2007) [7] study its effects on maintainability...

Mengkeang Veng (2014) [8] studies the effects of dependency injection and mocking on software and testing and attempt to answer four research questions: To what extent does dependency injection improve code structure design? To what extent does dependency injection facilitate software testing? To what extent does mock facilitate software testing? And to what extent do dependency injection and mock facilitate software testing. In order to answer those questions, an experiment is carried out in which two systems are compared to measure the extent of facilitation of the dependency injection on software. The results demonstrate that the dependency injection does not seem to improve the code design if comparing on the selected metrics and that the testability of the two systems is similar.

4 Methodology

In order to discover if dependency injection improves testability, the study will collect various projects and analyze their classes in search of a relationship between following the principle and how they are being tested. Our hypothesis is that classes that follow the DIP are better tested in isolation than those classes that do not.

This can be called a "analytical research" in which the researcher has to use facts or information already available and analyze them to make a evaluation of the material.

To reach the overall goal, three main research decisions can be identified:

1. How to measure the use of dependency injection in a class.
2. How to measure in which degree a class is being tested in isolation.
3. How to collect the projects.

In the following subsections, each decision will be covered one by one.

4.1 DIP use measurement

As mention in section 2, a class that follows the DIP instead of creating the objects, get the instances passed as parameters. Thus, in order to measure the use of this principle, we need to analyze creations of objects and to this effect, we need to detect the `new` keyword. In this paper, we are going to refer to these creations as "dependencies".

"Dependency" is a creation of an object inside a class. So if an object B is created inside class A, class A depends on class B.

Among all the dependencies, there are some that can not be consider to violate the DIP. These include some trivial java dependencies like lists, sets or dates from the `java.util` package. Besides, dependencies of the same package are under our control (unless these classes violate the DIP), so we are not going to consider those either. Now, we can classify these exceptions as "internal dependencies" and the rest as "external dependencies".

"External dependency" is a dependency on:

- a class that is neither from the same package nor from `java.util`
- a class that has external dependencies itself (recursive)

"Internal dependency" is a dependency that is not "external".

Similarly, we can define "external classes" and "internal classes":

"External class" is a class with external dependencies. Thus, it violates the DIP.

"Internal class" is a class without external dependencies. Thus, it follows the DIP.

4.2 Testing measurement

The next decision is how to measure in which degree a class is being tested in isolation. Among all the test, we are interested in unit tests since they are those that are supposedly improved by the use of the DIP. For this measure, we propose two metrics.

- **If the class has a dedicated unit test class.** By convention, the unit test classes are named with the name of the class to be tested followed by the suffix "Test" or "Tests". An easy way to check if a class has a unit test is to search for a class with that format. That is why we are going to consider a class to have a dedicated test if there exists a class named `<classname>Test.java` or `<classname>Tests.java`.

- **The code coverage of the class.** This metric would be more precise, since the existence of a dedicated test does not necessarily mean that the class is being properly tested. There are different tools available that allow us to register the number of lines executed while testing, some of them integrate easily with maven.

In this research, only the first metric is implemented, but the idea is to extend it with code coverage as we discuss in section 8.

4.3 Collect projects

If we want to collect projects in an easy and cheap way, our best option is Github. Github counts with millions of open source projects in multiple programming languages. But how can we select a few projects between millions? Our idea is to obtain them from *The Apache Software Foundation*, since it counts with hundreds of open source well-know projects. Besides, many of them are written in Java and all of them use Maven, which will be convenient for future analysis of the code coverage.

5 Implementation

Once it has been defined how to measure the use of DIP and the degree in which a class is being tested and how to collect the projects, we can implement the tool. The tool is written in java in the package `com.github.iraticasi.testability` and it is divided in three subpackages:

`analyzer` provides analysis for the use of the dependency injection principle.

`report` contains implementation for report generation about the correlation between the use of dependency injection and testing.

`collector` provides a apache project collector.

In the following subsections, each package will be briefly explained. For more specific information, you can take a look to the javadoc or the source code.

5.1 Analyzer package

This package counts with two classes:

`Analyzer` : instances can be constructed from the directory of a project.
Its method `analyze()` analyze the project and return a list of `ClassInfo`.
Its method `makeReport(String file)` create a CSV report named `<file>` with the format shown in Table 1.

`ClassInfo` : It represents a class of the project and holds all the required information that can be obtained using the methods: `getName()`, `getPkg()`, `getDependencies()` and `hasExternalDependencies()`.

5.2 Report package

The class `TestChecker` is responsible for generating reports with statistics of the existence of dedicated tests. An instance of this class can be created from a folder with the target projects and it includes two public methods:

`externalReport(String file)` creates a CSV report of external dependencies. For each project, it computes statistics of the relationship between classes with external dependencies

Package	Class	External dependencies
`package name`	`class name`	Y/N
...	...	...

Table 1: Format of external dependencies report

and classes with dedicated test. The report follows the format shown in Table 2 and its fields are explain after that.

- *Project name*: The name of the analyzed project.
- *# external classes with test*: Number of "external" classes that have a dedicated test.
- *# external classes without test*: Number of "external" classes that do not have a dedicated test.
- *# internal classes with test*: Number of "internal" classes that have a dedicated test.
- *# internal classes without test*: Number of "internal" classes that do not have a dedicated test.

`librariesReport(String file, String[] libraries)` is implemented to generate a report that shows the relation between a class having dependencies on some given libraries and having a dedicated test. This functionality is not used in this research, it is only implemented for future work as discussed in section 8. The report follows the format shown in Table 3 and its fields are explain hereunder:

- *Library name*: The name of the library.
- *Library dep with test*: Number of classes that have dependencies on classes of that library and a dedicated test.
- *Library dep without test*: Number of classes that do not have dependencies on classes of that library but have with a dedicated test.
- *No library dep with test*: Number of classes that do not have dependencies on classes of that library nor a dedicated test.
- *No library dep without test*: Number of classes that does not have dependencies on classes of that library but have a dedicated test.

5.3 Collector package

This package is only created in order to collect projects automatically using the github API. The class `ProjectorCollector` clones a given number of apache projects written in java in the folder `apache_projects`.

6 Results

The analysis has been carried out for 18 projects with a total of 29,013 classes, the resulting report is shown in **table 4**.

6.1 Results by project

Figure 1 represents the dependency injection use per project. The percentage of classes that follow the principle goes from 19% (`beam`) to a maximum of 53% (`incubator-skywalking`). We can observe for example that 27% of `flink` classes and 24% of cassandra classes follow the principle.
Figure 2 represents the use of dedicated

Project name	# external classes with tests	# external classes without tests	# internal classes with tests	# internal classes with tests
...	...	...	...	...

Table 2: Format of test and external dependencies correlation report.

Library name	library dep with test	library dep without test	no library dep with test	no Library dep no test
...	...	...	...	...

Table 3: Format of test and libraries use correlation report.

test per project. The percentage of classes that have a dedicated test goes from 0.1% (`hive`) to 44% (`beam`). If we focus on the projects mentioned above, 24.6% of `flink` classes and 19.9% of cassandra classes have a dedicated test.

Figure 3 shows the percentages of classes that count with a dedicated test per project differentiating external and internal classes. Red bars show the percentage of external classes with tests and green bars show the percentage of internal classes with tests. For example, in `flink` 28.8% of external classes and 13.2% of internal classes have a dedicated test. Contrary to what we expect, in every project the percentage of external classes that have a dedicated test is greater than the percentage of internal classes that have a dedicated test.

6.2 Overall results

Lastly, in **table 5** we have the external report overall results. Only 12.2% of classes have a dedicated test and 29.7% follow the dependency injection principle according to our definition.

Figure 4 shows that 14.99% of external classes and only 5.74% of internal classes have a dedicated test.

7 Discussion

Our hypothesis was that classes that follow the DIP are better tested in isolation than those classes that do not. Therefore, we thought that the percentage of internal classes that have a dedicated test was going to be greater than the percentage of external classes that have a dedicated test. Unexpectedly, results show the opposite. They also show a low use of unit testing and dependency injection according to our definition .

Unit testing finds problems early in the development cycle, facilities changes and simplifies integration, but the tests can be more complex than the actual code [9]. Thus, we conjecture that developers decide to focus on integration testing.

On the other hand, interdependence between classes is unavoidable. Coupling of object implementations is not always bad and sometimes is better to create a dependency in the dependent class than to code the loosely-coupled resolution of the dependency (injection, service location, factory). That can be a the reason why so few classes follow the principle. Yet, the

assumptions made above do not explain why "external classes" are slightly more tested so we have come to the following suppositions:

- A significant proportion of classes that have no dependencies are simple by nature so developers do not create unit test for them.
- Apart from java.util and classes from the same package, there might be other classes that we did not consider that should not be injected either.

8 Conclusion

We have perform an analysis in search of a relationship between the use of DI and the use of unit testing. The obtained results indicate a low use of both DIP and unit testing and, contrary to what we expected, that classes that follow the principle are slightly more tested.
For future work, code coverage will make a more accurate analysis and will provide us a better understanding of unit testing use. Another approach would be try to detect which libraries dependencies make classes that have them be less testable. Using the library report of the `report` package, you can analyze the existence of dedicated test in classes that have dependencies on a given library.

References

[1] Jonathan Wolter, Russ Ruffer, Miko Hevery (2008). *Guide: Writing Testable Code.*

[2] Gao, J., Gupta, K., Gupta, S., & Shim, S. (2002, February). *On building testable software components.* In International Conference on COTS-Based Software Systems (pp. 108-121). Springer, Berlin, Heidelberg.

[3] Bruntink, M., & van Deursen, A. (2006). *An empirical study into class testability.* Journal of systems and software, 79(9), 1219-1232.

[4] Voas, J., & Kassab, L. (1999). *Using assertions to make untestable software more testable.* Software Quality Professional, 1(4), 31.

[5] Ma, L., Zhang, C., Yu, B., & Sato, H. (2015, May). *An empirical study on effects of code visibility on code coverage of software testing.* In Automation of Software Test (AST), 2015 IEEE/ACM 10th International Workshop on (pp. 80-84). IEEE.

[6] Yang, H. Y., Tempero, E., & Melton, H. (2008, March). *An empirical study into use of dependency injection in java. In Software Engineering, 2008.* ASWEC 2008. 19th Australian Conference on (pp. 239-247). IEEE.

[7] Razina, E., & Janzen, D. S. (2007). *Effects of dependency injection on maintainability.* In Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA (p. 7).

[8] Mengkeang Veng (2014). *Dependency Injection and Mock on Software and Testing.* Master thesis, Department of Informatics and Media, Uppsala University.

[9] James O Coplien (2014). *Why Most Unit Testing is Waste.* RBCS.

Project name	**# external classes with tests**	**# external classes without tests**	**# internal classes with tests**	**# internal classes without tests**
incubator-weex	45	597	8	230
hive	2	4133	3	1339
zeppelin	112	195	9	115
rocketmq	80	230	13	266
zookeeper	52	201	14	109
jmeter	35	718	4	332
groovy	51	733	10	428
tomcat	3	1284	0	675
incubator-dubbo	194	285	53	279
kafka	334	433	32	378
storm	152	1128	21	494
beam	587	626	72	213
incubator-skywalking	110	776	50	934
flink	870	2146	147	970
cordova-android	1	21	1	15
incubator-heron	125	361	34	246
hbase	5	2526	0	747
cassandra	299	939	24	359

Table 4: External report results.

	external classes	**internal** classes	total
with test	3057	495	3552 (12.2%)
without test	17332	8129	25461 (87.8%)
total	20389 (70.3%)	8624 (29.7%)	29013

Table 5: External report overall results.

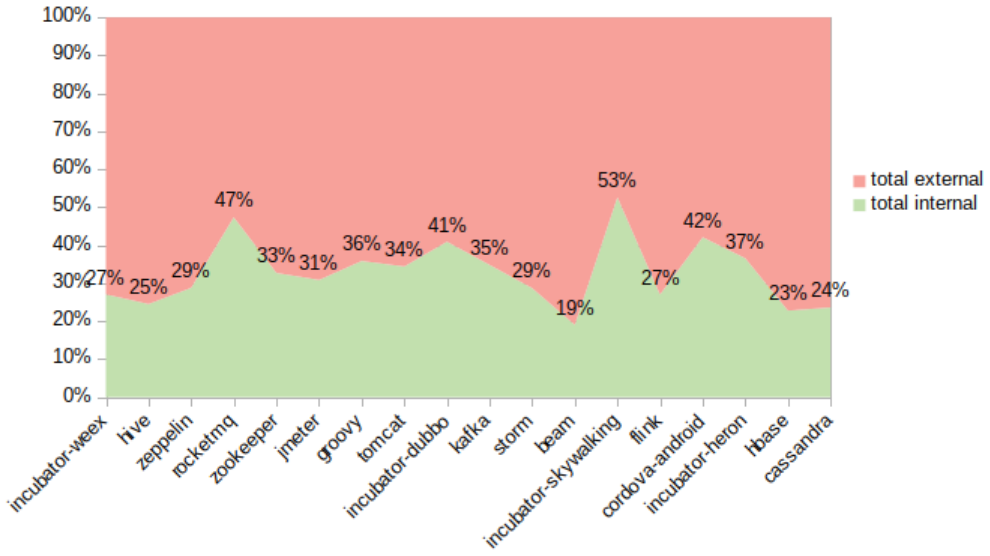

Figure 1: Dependecy injection principle use.

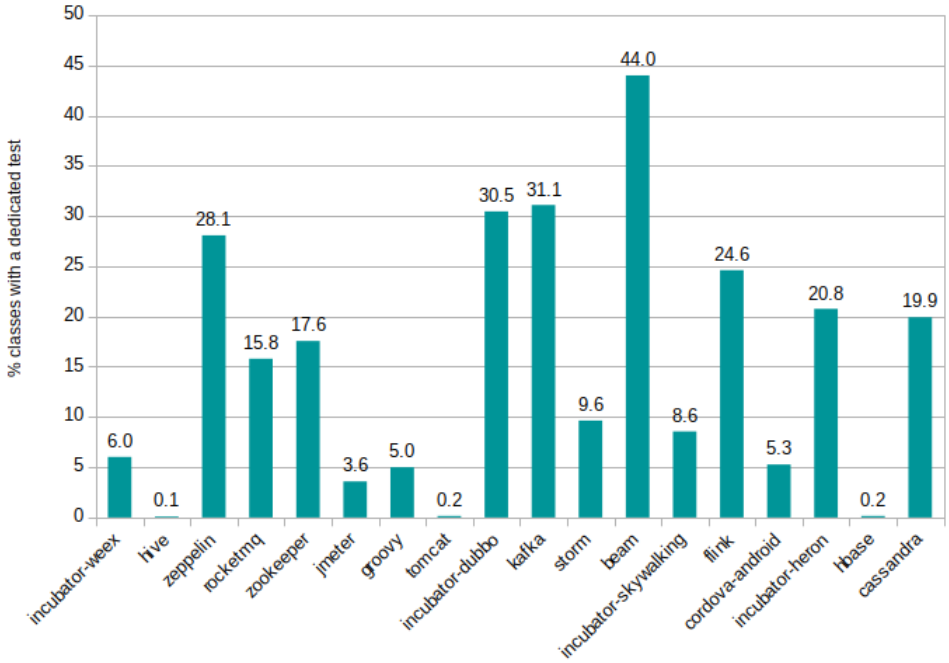

Figure 2: Use of dedicated test.

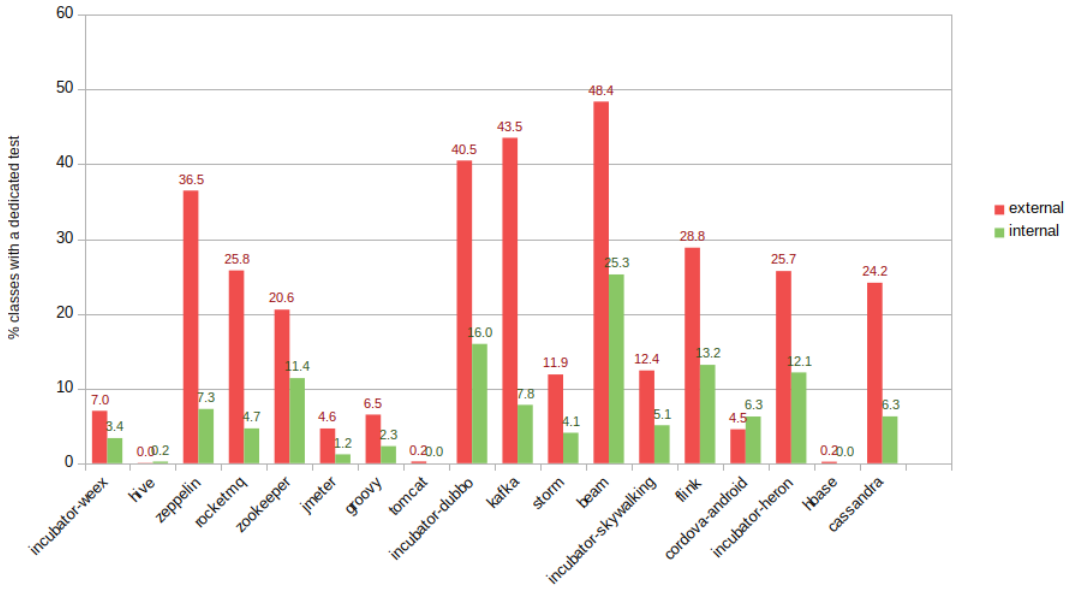

Figure 3: Dependency injection principle use on external/internal classes per project.

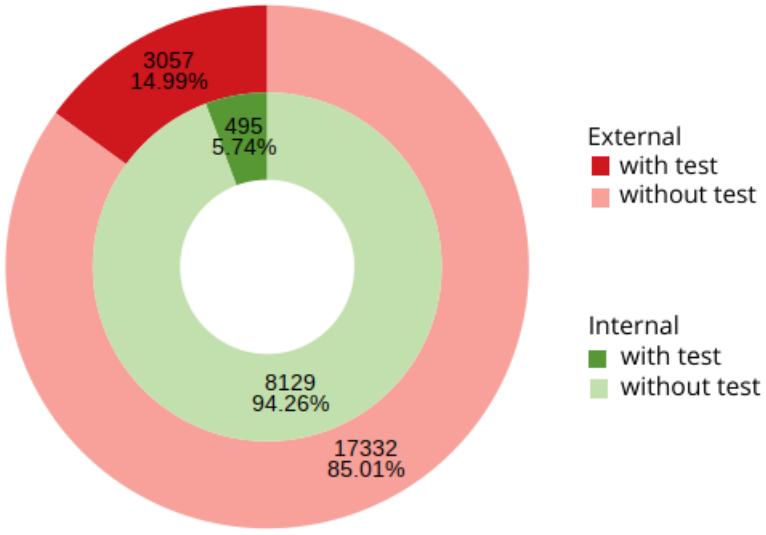

Figure 4: Dependency injection principle use on all external/internal classes .