Proceedings of the 11th IASTED International Conference
SOFTWARE ENGINEERING AND APPLICATIONS
November 19–21, 2007, Cambridge, MA, USA
ISBN Hardcopy: 978-0-88986-705-5 / CD: 978-0-88986-706-2

EFFECTS OF DEPENDENCY INJECTION ON MAINTAINABILITY

Ekaterina Razina
Intuit, Inc.
2550 Garcia Ave
Mountain View, CA, USA
email: kate_razina@intuit.com

David Janzen
Department or Computer Science
California Polytechnic State University
1 Grand Ave.
San Luis Obispo, CA, USA
email: djanzen@calpoly.edu

ABSTRACT
Software maintenance consumes around 70% of the software life cycle. Improving software maintainability could save software developers significant time and money. This paper examines whether the pattern of dependency injection significantly reduces dependencies of modules in a piece of software, therefore making the software more maintainable. This hypothesis is tested with 20 sets of open source projects from sourceforge.net, where each set contains one project that uses the pattern of dependency injection and one similar project that does not use the pattern. The extent of the dependency injection use in each project is measured by a new Number of DIs metric created specifically for this analysis. Maintainability is measured using coupling and cohesion metrics on each project, then performing statistical analysis on the acquired results. After completing the analysis, no correlation was evident between the use of dependency injection and coupling and cohesion numbers. However, a trend towards lower coupling numbers in projects with a dependency injection count of 10% or more was observed.

KEY WORDS
Maintainability, Dependency Injection, Spring Framework

1. Introduction

Non-functional requirements are important to all software system development. One major non-functional requirement that encompasses many others within itself is maintainability. Maintainability is the ease with which a software system or component can be modified. Modifications may include extensions, porting to different computing systems or improvements. Flexibility, reusability, testability and integrability contribute to modifiability, and therefore are defined as sub attributes of maintainability [12]. Software maintainability is difficult to fully measure because it relies on many factors, some of which are very subjective [1]. However, studies have shown that small, decoupled, highly cohesive modules lead to an increase in maintainability [2, 15]. A new technique of dependency injection attempts to separate programs into smaller, more independent components that can be externally configured [7].

The authors hypothesize that use of the dependency injection pattern significantly reduces dependencies of modules in a piece of software, therefore making the software more maintainable. This hypothesis is tested by calculating coupling and cohesion metrics on 20 sets of projects. Each set contains two projects, one project that uses the pattern of dependency injection and one project that uses similar technologies to the first, yet does not use the pattern of dependency injection. The extent to which the pattern of dependency injection is used will be measured using a new metric developed for this analysis, the Number of DIs metric. This paper is structured as follows. Section 2 explains the pattern of dependency injection and discuss how maintainability is measured in our experiments. Section 3 discusses how data is collected, what tools are used to measure the projects, and what tools are used for the statistical analysis. Section 4 presents the results of the experiments and the analysis of the results. Finally, section 5 provides some conclusions and suggestions for future work.

2. Previous Work

2.1 Dependency Injection

This section briefly introduces dependency injection in order to give the reader the basic idea of the concept that is prevalent throughout this study. This study is concerned with two types of dependency injection: constructor and setter injection. The Spring framework, a framework that every dependency injection project uses in the experiments that follow, allows for constructor injection and setter injection [7].

Dependency injection is a pattern that allows the programmer to inject objects into a class by using a container that is externally configured (often by an XML file), instead of letting the class directly instantiate the objects. This pattern will be explained through a simple example. Suppose we have written a class, *a*, that contains a class *b* object. Class *b* is an implementation of the interface B. This makes class *a* have a dependency on class *b*, as well as interface B, as shown in Figure 1.

Dependency injection removes the dependence of class *a* on class *b* by adding a container and making it responsible for the dependency look up. This container is

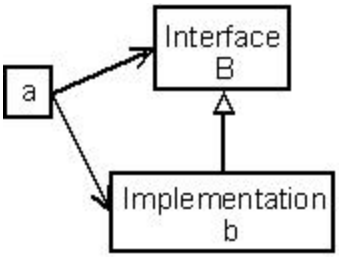

Figure 1. Class Dependencies

usually configured by an XML file. This changes the dependencies as shown in Figure 2.

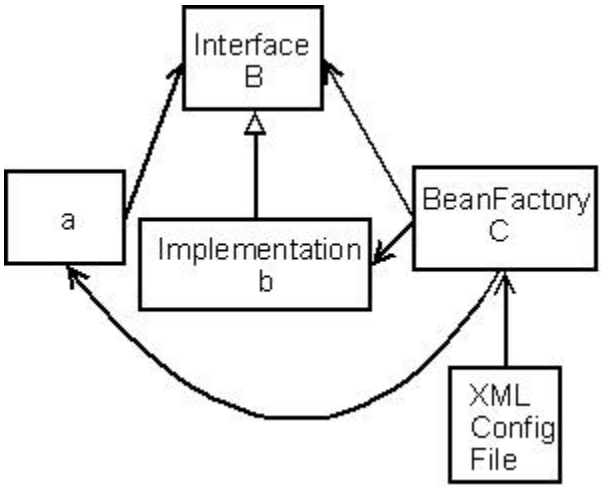

Figure 2. Dependency Injection in Spring

Now, the container is responsible for providing the necessary resources and looking up the necessary resources for class *a*. Also, this allows class *a* to work with any implementation of the B interface eliminating the previous dependency of *a* on *b* [7].

Dependency injection in Spring can be done in two different ways: constructor injection and setter injection. In constructor injection, the components express dependencies via constructor arguments. In setter injection, components express dependencies on configuration values via JavaBean properties.

Any non-trivial application is made up of two or more classes that collaborate with each other to accomplish some task. Traditionally, each object is responsible for obtaining its own reference to the object it wants to collaborate with, forming a dependency. In the long run this leads to programs that are highly coupled and hard to test [18].

When the pattern of dependency injection is used, the responsibility of coordinating collaboration between dependent objects is removed from the objects themselves [18]. This way we can substitute any implementation of interface B into our program with minimal changes to the XML file and no changes to class *a*. If we no longer want to use *b*, but want to use a new class, *c*, we can easily make the substitution without altering class *a*.

Dependency injection also makes testing easier. In Figure 1, testing class *a* inevitably leads us to testing implementation *b*. If we use the pattern of dependency injection in writing our program, we can insert a mock implementation of class *b* into the system and test class *a* in isolation. This allows for easier testing, or testability. Since testability is one of the components of maintainability, it leads us to believe that the pattern of dependency injection improves maintainability.

Walls and Breidenbach [18] state that using the pattern of dependency injection provides us with less coupled modules and code. Logically, we can see that there are fewer dependencies in Figure 2 (we do not count the container, since Spring provides this for us). However, no studies have been done to demonstrate that dependency injection provides significant improvements when it comes to coupling measures. An experiment is needed to provide concrete data to support that the pattern of dependency injection decouples modules and allows for better maintainability.

2.2 Maintainability Measures

2.2.1 Coupling

Coupling is defined as "the degree of interdependence between parts of a design" [6]. An object is coupled to another if one object uses methods or instance variables of another [6]. Excessive coupling between objects makes modules very dependent on each other. This makes the program hard to understand and makes the code hard to reuse. Higher coupling of objects increases the sensitivity to changes in other parts of the design. This also makes maintainability more difficult [14].

In a widely cited paper by Briand et al.[3], the authors, drawing on the existing coupling measurement techniques, come up with a list of seven different ways to measure coupling. The two coupling measures that will be used in our experiments are described below.

- Coupling between objects (CBO) for a class is the count of the number of classes to which it is coupled. If class *a* is coupled to class *b* and *c*, its CBO is two. This definition of coupling includes inheritance [9]. The less an object is coupled to other objects, the more likely it is to be reused in another application. Since reusability is one of the four components of maintainability, CBO should correlate directly to maintainability.

- Response for class (RFC) is a set of methods that can be potentially executed in response to a message received by an object of the class. If a large number of

methods are invoked in response to receiving a message, the testing and debugging becomes more complicated since a greater level of understanding is required. Testability is also one of the four components of maintainability, so RFC should also directly correlate to maintainability.

Lower coupling leads to fewer errors [3] which reduces the testability aspect of maintainability. Lower coupling also allows the modules to be changed easier, which speeds up the flexibility aspect of maintainability [3]. Flexibility and testability are both components of maintainability. Therefore reducing these components reduces the time spent in the maintainability phase.

As Figure 2 demonstrated, dependency injection should intuitively influence coupling by loosening the connection between the interface implementation, *b*, and the class that uses that implementation, *a*. This connection is loosened by relocating it to the configuration file and injecting *b* into *a* via the configuration file. Intuitively, this rearranging, should reduce the number of couplings in the Java code. If the number of couplings is reduced, maintainability should also be reduced.

2.2.2 Cohesion

Cohesion is the degree of similarity of methods [4, 6]. It is the degree to which each part of the module is associated with each other part. Cohesion of methods within a class is often desirable. Cohesive methods promote encapsulation of objects [6]. A lack of cohesion in a class implies that the class should be re-factored into two or more subclasses. Low cohesion increases complexity and therefore, increases the likelihood of errors during the development process [6]. Modules with high cohesion, however, have more reliable and easy to understand code [8]. These modules are also easier to develop, maintain, and reuse and are less fault-prone [11].

The original Lack of Cohesion in Methods (LCOM) [6] metric that was presented by Chidamber and Kemerer is still in use today. They define this metric in the following way:

> Consider a class C_1 with methods M_1, M_2,. . .M_n. Let $\{I_i\}$ = set of instance variables used by methods M_i. There are n such sets $\{I_1\}$,. . . $\{I_n\}$. The degree of similarity of methods is given by $\{I_1\} \bigcup \{I_2\} \bigcup . . . \bigcup \{I_n\}$

LCOM is generally tied to the instance variables and methods of an object; therefore, it is a measure of the attributes of an object.

Clearly, cohesion is related to maintainability. Since low cohesion increases complexity and therefore, the likelihood of errors, maintainability decreases. A study [8] of a 148,000 source line system from a production environment found that routines with the highest coupling to cohesion ratios had seven times as many errors per 1,000 source statements as compared to those with the lowest coupling to cohesion ratios. These were also twenty times as costly to fix.

LCOM, the metric used to measure cohesion in this study, is a broadly used cohesion metric. However, LCOM has been argued to be incomplete and to not accurately measure cohesion [5, 10]. Chae and Kwon found that accessor methods, constructors and destructors do not affect class cohesion; however, LCOM takes these methods into consideration. Kabaili et al. [10] found a weak correlation between cohesion and changeability, leading the authors to believe that LCOM does not reflect the real cohesion of a class.

3. Experiment

3.1 Data Collection

To find out whether the stated hypothesis is true, 20 sets of projects were gathered. Each set contained two projects that used similar technologies. One project used the Spring framework and the other did not use Spring.

In order to gather these projects, sourceforge.net was utilized. Sourceforge.net is an open source community that contains many projects written using various technologies. Here, 20 sets of projects that comprised the data set were gathered.

First, the 20 Spring projects were selected to be used in the study. These projects were all open source, written in Java using the Spring framework as well as other technologies. Some projects used technologies such as Hibernate, MySQL, Tapestry, Struts, and others. Most of the projects were web-based.

Then, 20 projects that did not use Spring, but used similar technologies to their counterpart Spring project were selected. For example, if the Spring project was web-based and used Hibernate, then the non-Spring project was also web-based and used Hibernate. For an ideal experiment, we would have wanted pairs of projects that were implemented from the same set of requirements, one done using the Spring framework and the other done without the use of the Spring framework. However, this ideal scenario was difficult to accomplish.

3.2 Metric Tools Used

3.2.1 CKJM

Because maintainability is correlated to coupling and cohesion, these were the main metrics used to evaluate each project set. A tool called the Chidamber and Kemmerer metric (CKJM) tool was used in the data collection of this paper [16].

The CKJM tool uses Chidamber and Kemmerer metrics such as Coupling between Objects (CBO), Response for Class (RFC), Lack of Cohesion (LCOM), as well as other complexity metrics. We will mostly focus on the

CBO, RFC, and LCOM measures since these measures relate directly to maintainability. Also, CBO and RFC measure coupling, which, from Figure 1, appears to be what dependency injection reduces. The simple examples in Figures 1 and 2 were implemented to verify that the CBO and RFC results come out as hypothesized. The RFC and CBO values in the Spring example were reduced by one as compared to the non-Spring example. The code verified what the figures show, therefore there is reason to believe that Spring will have an effect on CBO and RFC numbers of larger projects.

3.2.2 DI Metric

Besides using the CKJM tool to collect project metrics, a new metric tool was written to calculate Number of DIs. This tool will be used to examine the number of times the dependency injection pattern is used in each Spring project. The only thing we know about the Spring projects gathered for this experiment is that they use Spring. However, Spring is a framework with many components. The use of the Spring framework does not guarantee the use of dependency injection. The Number of DIs metric will show to what extent each Spring project actually uses dependency injection.

The numbers produced by this tool will be divided by the sum of the CBO numbers of each Spring project. CBO represents the number of classes to which a class is coupled. Dependency injection reduces the number of classes to which a class is coupled through the configuration file. When we use the pattern of dependency injection, we move the couplings between objects that would have been in Java code instead to an XML file. The provided container uses the XML file to connect the Java objects together. Because of this, the sum of the CBO numbers is the sum of all the couplings that could potentially be done using dependency injection. Dividing the number of dependency injections in a project by the sum of the CBOs (DI / Σ CBO) will allow us to normalize the dependency injection numbers of each project so we can compare them to each other.

3.3 Analysis

Minitab statistical software was used to perform an ANOVA analysis on the data. The purpose of this analysis is to quantify the relationship between several independent variables and a dependent criterion variable (dependency injection). This general analysis predicts whether the dependency injection factor has an effect on the metrics of each project.

Using the results of all the projects put together, this analysis shows if the presence of Spring, or dependency injection, has an effect on any of the metrics. If the outcome of this test is a p-value less than or equal to 0.05, then the presence of dependency injection has an effect on the metric we are testing.

Following the ANOVA general linear data analysis, a t-test analysis is used. A t-test assesses whether the means of two groups are statistically different from each other [17]. This test judges how significant the variability of the means in two datasets is with respect to the distribution of their values in a bell curve. For this analysis, a two-sample two-tailed t-test was used.

The t-test outputs a probability value or p-value. This value represents the probability of getting a value of the test statistic by chance alone. If the p-value produced by the t-test is 0.05 or less, we can say that we are 95% or more confident that the results in these tests did not simply occur due to random chance [17]. Therefore, we are only interested in metric results with a p-value less than or equal to 0.05.

4. Results

The ANOVA analysis was performed on the cumulative CBO and RFC results using Minitab Statistical Software. ANOVA is used to determine if the differences between the Spring and non-Spring project's CBO and RFC values are statistically significant [13]. The ANOVA results for CBO produced a p-value of 0.00. This tells us that Spring contributes to the difference in CBO numbers. However, this difference could not be calculated because the type of project also contributed to the difference in CBO. We cannot yet say if the CBO is lower or higher for the Spring versus the non-Spring project. The gathered projects were all very different. This variance affected the CBO numbers. Similar results were found for RFC and LCOM.

Table 1 lists how many Spring projects performed better for each metric as compared to their non-Spring project counterpart. A project is considered to have done better on a metric, if it had a lower value for the CBO and RFC metrics and that value was statistically significant. A project was considered to have done better if it had a higher LCOM value. Since LCOM is an inverse metric and measures the lack of cohesion, a higher LCOM value is more desirable. A project was counted if its t-test values for that metric were less than or equal to 0.05.

During our t-test analysis we found that no obvious correlation exists between the presence of dependency injection and the Chidamber and Kemmerer metrics. Table 1 shows a nearly equal number of project sets that had the Spring project with lower RFC and CBO numbers, as the number of non-Spring projects with lower RFC and CBO numbers. There were five sets of projects of the twenty sets for which the Spring project had a lower average CBO value. There were six sets of projects of the twenty sets for which the non-Spring project had a lower average CBO value. The other nine projects sets did not demonstrate statistically significant results.

The results from Table 1 disprove the hypothesis of this paper. However, an interesting trend was noticed during the analysis of the experiment results. The project sets that had lower average CBO and average RFC tended to

Table 1. Cumulative Summary

	CBO	RFC	LCOM
Spring	5	6	4
No-Spring	6	7	5

have higher numbers of DI/Σ CBO. Upon further examination of the five Spring projects that exhibited lower average CBO with a CBO p-value of less than or equal to 0.05, four had a DI/Σ CBO of over 10%. Table 2 shows this result. The Proj column lists the project's set numbers for which the Spring project had a lower average CBO as compared to the non-Spring project. The %DI column lists the results of the equation DI/Σ CBO *100. From this table, we can see a trend: projects with a high percentage of dependency injections (greater than 10%) tend to have lower average CBO.

Table 2. Spring Lower Average CBO / Higher DI Percentage

Proj	%DI
3	3.1
9	41.7
11	10.43
13	10.09
14	19.39

Of the six Spring projects that had higher average CBO than their counterparts, all six had DI/Σ CBO numbers of less than 10% as seen in Table 3. The project set number for which the non-Spring projects had a lower average CBO is listed in the left hand column. The right hand column lists the percentage of dependnecy injection of the Spring project from that set. The projects listed in this table have less than 10% dependency injection.

Table 3. Spring Higher Average CBO / Lower DI Percentage

Proj	%DI
4	9.61
8	2.22
12	5.61
15	3.56
18	1.12
19	3.90

The results presented in Table 2 and Table 3 are significant findings. However, since eleven of our twenty projects exhibit this characteristic, we cannot conclusively state that a higher percent of dependency injection will lower the CBO numbers or vice-versa. The other nine projects did not have statistically signifacnt average CBO and RFC numbers. Further analysis of such projects should be done. Simular results were found for RFC.

5. Conclusion

Mintainability of a software product is a big problem that often consumes 60% to 80% of the software life cycle. This problem is familiar to software developers and has existed for years, with no sign of relief in sight. Even though a complete solution to this problem does not exist, ways to measure code and predict maintainability do exist. Some of the measures that predict maintainability are coupling and cohesion metrics.

This paper examines if the pattern of dependency injection significantly reduces dependencies of modules in a piece of software, therefore making the software more maintainable. We tested this hypothesis by collecting 20 sets of projects and calculating three metrics–CBO, RFC, and LCOM–on those projects.

The experiment results were unable to substantiate this hypothesis. There does not appear to be a trend in lower coupling or higher cohesion measures with or without the presence of dependency injection. However, a trend of lower coupling in projects with higher dependency injection percentage (more than 10 %) was evident. We cannot conclude that such a trend persists through all the Spring projects due to the amount of projects that exhibited this trend (a fourth of all the projects with a CBO/RFC p-value of less than or equal to 0.05). However, further analysis of this should be done.

Even though the coupling metrics failed to produce lower numbers for the projects that used dependency injection, it is still possible that the use of dependency injection allows us to write more maintainable software. The use of the XML file to configure objects allows us to have more configurable couplings. The XML file maintains most of the project couplings in one place. In order to change a coupling, a developer would simply edit the XML file, instead of figuring out what Java file needs changing. Since the Spring framework allows all couplings to be located in one XML file, we can easily manage and change them.

We could produce an experiment to measure whether some couplings are more configurable than others. This experiment could involve two similar projects, one written in Spring and the other written without the use of Spring. Two developers would be given a task of altering each project. We could measure how long each developer takes to perform the task and how many files he has to alter. We would also measure the code for number of lines, and number of defects. Ideally, we would run this experiment multiple times on different projects. By analyzing these statistics we could figure out if the couplings created by dependency injection are easier to work with and change.

One drawback to keeping most of the couplings in the XML file is that eventually, when the project becomes

large, the XML file will also become large. If the XML file is large, more hours will be spent on maintaining it. Because of this, it may not be optimal to use dependency injection in all types of projects.

The results of the study do not confirm the hypothesis because no trend in the metric measurements was discovered. In order to truly measure maintainability and the effects of dependency injection we would have to construct a more controlled study. However, a trend of lower coupling in projects with higher dependency injection percentage (more than 10 %) was evident. We cannot conclude that such a trend persists through all the Spring projects due to the amount of projects that exhibited this trend (a fourth of all the projects with a CBO/RFC p-value of less than or equal to 0.05). However, further analysis of this should be done.

References

[1] K. K. Addarwall, Y. Singh, and J. K. Chhabra. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Conference*, Seattle, USA, 2002, 235-244.

[2] E. Arisholm. Dynamic coupling measures for object-oriented software. *IEEE Symposium on Software Metrics*, 30(8), 2002, 33-34.

[3] L. Briand, J. Daly, and J. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 24(1), 1999, 91-121.

[4] L. Briand, J. Daly, and J. Wust. A unified framework for cohesion measurement in object oriented systems. *Empirical Software Engineering: An International Journal*, 3(1), 1998, 65-117.

[5] H. S Chae and Y. R Kwon. A cohesion measure for classes in object-oriented systems. In *Fifth International Software Metrics Symposium*, Maryland, USA, 1998, 158-166.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), 1994, 476-493.

[7] Martin Fowler. Inversion of control containers and the dependency injection pattern. Technical report, Thought Works, 2004.

[8] N. Gupta and P. Rao. Program execution based module cohesion measurement. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering*, 2001, 144-153.

[9] Brian Henderson-Sellers. *Object-Oriented Metrcis*. (New Jersey: Prentice Hall, 1996).

[10] H. Kabaili, R. K. Keller, and F. Lustman. Cohesion as changeability indicator in object-oriented systems. In *European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal, 2001, 39-46.

[11] Y Lee and K Chang. Software maintenance: Reusability and maintainability metrics for object-oriented software. In *Proceedings of the 38th Annual on Southeast Regional Conference ACM-SE*, South Carolina, USA, 2000, 88-94.

[12] M. Mari and N. Eila. The impact of maintainability on component-based software systems. In *Euromicro Conference*, 2003, 25-32.

[13] Minitab. Minitab. http://www.minitab.com/, 2007.

[14] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *22nd IEEE International Conference on Software Maintenance*, Pansylvania, USA, 2006, 469-478.

[15] C. Rajaraman and M.R. Lyu. Reliability and maintainability related software coupling metrics in c++ programs. In *Third International Symposium on Software Reliability*, North Carolina, USA, 1992, 303-311.

[16] D Spinellis. Chidamber and kemerer java metrics. http://www.spinellis.gr/sw/ckjm/, 2005.

[17] William Trochim. Research methods knowledge base. http://www.socialresearchmethods.net/kb/statt.php, 2006.

[18] Craig Walls and Ryan Breidenbach. *Spring In Action*. (Colorado: Manning Publications, 2005).