Does Test-Driven Development Improve the Program Code? Alarming Results from a Comparative Case Study

Maria Siniaalto[1] and Pekka Abrahamsson[2]

[1] F-Secure Oyj,
Elektroniikkatie 3, FIN-90570 Oulu, Finland
Maria.Siniaalto@f-secure.com
[2] VTT Technical Research Centre of Finland,
P.O. Box 1100, FIN-90571 Oulu, Finland
Pekka.Abrahamsson@vtt.fi

Abstract. It is suggested that test-driven development (TDD) is one of the most fundamental practices in agile software development, which produces loosely coupled and highly cohesive code. However, how the TDD impacts on the structure of the program code have not been widely studied. This paper presents the results from a comparative case study of five small scale software development projects where the effect of TDD on program design was studied using both traditional and package level metrics. The empirical results reveal that an unwanted side effect can be that some parts of the code may deteriorate. In addition, the differences in the program code, between TDD and the iterative test-last development, were not as clear as expected. This raises the question as to whether the possible benefits of TDD are greater than the possible downsides. Moreover, it additionally questions whether the same benefits could be achieved just by emphasizing unit-level testing activities.

Keywords: Test-Driven Development, Test-first Programming, Test-first Development, Agile Software Development, Software Quality.

1 Introduction

Test-driven development (TDD) is one of the core elements of Extreme Programming (XP) method [1]. The use of the TDD is said to yield several benefits. It is claimed to improve test coverage [2] and to produce loosely coupled and highly cohesive systems [3]. It is also believed to encourage the implementation scope to be more explicit [3] and to enable more frequent integration [4]. On the other hand, it is claimed that rapid changes may cause expensive breakage in tests and that the lack of application or testing skills may produce inadequate test coverage [5]. TDD has also received criticism over not being very suitable for systems such as multithreaded applications or security software, since it cannot mechanically demonstrate that their goals have been met [6]. However, the scientific empirical evidence behind all of these claims is currently sparse, and thus it is difficult to

B. Meyer, J.R. Nawrocki, and B. Walter (Eds.): CEE-SET 2007, LNCS 5082, pp. 143–156, 2008.

draw meaningful conclusions. The studies dealing with TDD have mainly focused on developer productivity and external code quality, whereas the TDD's impacts on program code have received less attention. The existing empirical evidence supports the claim that TDD yields improved external quality (see a recent summary of the TDD studies in [7]). However, it is not clear what has been the baseline for the comparison in those studies, e.g. did any unit level tests exist previously? The results of the studies, which address TDD's design impact, are presented in Section 2 in more detail.

Despite the lack of solid empirical evidence, both the industry and academia are keenly adopting test-driven development approaches. The purpose of this study is to investigate whether and how the structure of the program code changes or improves with the use of TDD. Five semi-industrial software development projects, containing both students and professionals as research subjects, were involved in the comparison of TDD and iterative test-last (ITL) approaches. Two of the projects used iterative test-last (ITL) development technique and three utilized TDD. The metrics used for evaluating the code are the traditional and widely-used suite of Chidamber and Kemerer (CK-metrics) [8] strengthened with McCabe's cyclomatic complexity metric [9]. To obtain a balance, the dependency management metrics proposed by Martin [10] for studying the code's package structure, were chosen as well.

The results of this study partially contradict the current literature. In particular, the case empirical evidence shows that TDD does not improve all the areas of the program design as expected. The results imply that TDD may produce less complex code, but on the other hand, the package structure may become more difficult to change and maintain.

The remainder of the paper is organized as follows. In the following section, the related work will be introduced. This is followed by an introduction to the metrics used to study the impact of TDD on program structure. Section 4 presents the empirical results, outlines the research design used to complete the study in a scientifically valid manner as well as detailing the threats to the validity of the empirical results. Section 5 discusses the novelty of the results in the light of existing studies and identifies the implications of the presented results. The conclusion section summarizes the principal results and proposes the potential future research avenues.

2 Related Work

In this section, the existing empirical evidence on the TDD's impact on the program design is presented. A total of five studies is included.

Janzen and Saiedian [11] compared the TDD and the ITL approaches using students as research subjects. They calculated several structural and object-oriented metrics in order to evaluate the differences in the internal quality of the software. As most of these results were within acceptable limits, there were some concerns regarding the complexity and coupling in the TDD code.

Kaufmann and Janzen [12] conducted a controlled experiment with students as research subjects comparing the design quality attributes of the TDD and the test-last approaches (whether the test-last was used iteratively is not known). The design quality was assessed with several structural and object-oriented metrics. They did not find any differences in the code complexities, but they report that there were indications that the

design quality of the TDD code was superior. However, they also admit that this finding may be due to the better programming skills of the subjects applying the TDD.

Steinberg [13] reports on the findings of the use of unit testing in the TDD style in an XP study group. Although, Steinberg concentrates on discussing the results from an educational point of view, his study also provides concrete experiences about the effects of the novel use of TDD and is thereby included in this study. He notices that the students tended to write more cohesive code when using TDD and the coupling was looser, since the objects had more clearly defined responsibilities. Evidence to support these last two claims was not provided, however.

In our initial study [7], we explored the effect of TDD on program design in semi-industrial setting comparing two ITL and one TDD projects. The design impact was evaluated using traditional object-oriented metrics. The initial results indicated that TDD does not always produce highly cohesive code. However, we concluded that the cohesion results might have been affected by the fact that all the developers in the TDD project were less experienced when compared to the subjects in the ITL projects.

Müller [14] studied the effect of test-driven development on program code. He included five TDD software systems of which three were student projects and compared them with three open source-based conventional software systems. He assessed the impact of TDD on the resulting code with Chidamber and Kemerer's [8] object-oriented metric suite and his own newly developed metric called assignment controllability. Müller reports that CK-metrics did not show any impact on the use of TDD but that the new assignment controllability metric showed a difference i.e. the number of methods where all assignments are completely controllable is higher for systems developed by TDD.

3 Metrics to Study Changes in Program Structure

The demand for quality software has resulted in a large set of different metrics some of which have been validated and some have not. Many of these metrics have been a subject of criticism and their empirical validity has been questioned. There is an ongoing debate on which metrics are the best indicators of the software quality and whether some particular metric even maps to the quality attribute it is supposed to represent. However, in many cases the authors presenting the criticism have not been able to propose a metric that would have solved the problem and would thereby have been widely adopted. Due to these reasons, we wish to emphasize that the aim of this study is not to validate or comment on the validity of any particular metric. The aim is to study whether and how TDD affects the code and its structure, and therefore both traditional and novel metrics were chosen for this study.

3.1 Traditional Metrics

The object-oriented metric suite, proposed by Chidamber and Kemerer [8], and McCabe's Cyclomatic complexity [9], were chosen as traditional representatives since they have been and still are widely used. CK-metrics, validated in [15], measure the different aspects of object-oriented construct. The suite contains six individual

metrics: weighted methods per class (WMC), depth of inheritance tree (DIT), number of children (NOC), coupling between objects (CBO), response for a class (RFC) and lack of cohesion in methods (LCOM). The suite was strengthened with Henderson-Sellers's lack of cohesion (LCOM*) [16. McCabe's' cyclomatic complexity measures the number of independent paths through a program module, and it is proposed to profile system's testability and maintainability. Although it is one of the most used and accepted of the static software metrics, it has also received criticism e.g. [17].

WMC measures the number of methods in a class and it is proposed to predict how much time and effort is required to maintain the class. DIT measures the depth of each class within its hierarchy, and its result shows how many ancestor classes can potentially affect this class. NOC presents the number of subclasses for each class. CBO presents the number of classes to which the class is coupled and it can be used as an indicator of whether the class hierarchy is losing its integrity. RFC presents the number of methods that can be executed in response to a message to the class. It is proposed as an indicator of the complexity and testing effort. LCOM assesses the similarity of the class methods by comparing their instance variable use pairwise. It is proposed to identify classes that are likely to behave in a less predictable way, because they are trying to achieve many different objectives. The biggest flaws of LCOM are that it indicates a lack of cohesion only when fewer than half of the paired methods use the same instance variables and the fact that a zero value does not necessarily indicate good cohesion, though a large value suggests poor cohesion [15, 16]. LCOM* measures the correlation between the methods and the local instance variables of a class. A low value of LCOM* indicates high cohesion and a well-designed class. A cohesive class tends to provide a high degree of encapsulation.

3.2 Dependency Management Metrics

The dependency management metrics proposed by Martin [10] measure and characterize the dependency structure of the packages. The suite includes: afferent coupling (C_a), efferent coupling (C_e), instability (I), abstractness (A) and normalized distance from the main sequence (D'). C_a counts the number of classes outside the package that depend on the classes inside the package whiles C_e counts the number of classes inside the package that depend on the classes outside the package. These two values are used when the instability (I) of the package is assessed. It has a specific range [0,1] with value 0 indicating a maximal stability and value 1 indicating maximal instability, i.e. no other package depends on this package. A package with lots of incoming dependencies is regarded as stable, because it requires a lot of work to reconcile the possible changes with all the dependent packages. The abstractness is simply measured by the ratio of abstract classes in a package to the total number of classes in a package.

Martin proposes that the package should be as abstract as it is stable so that the stability does not prevent the class from being extended. The main sequence presents this ideal ratio of stability and abstractness. Fig. 1 presents the main sequence and the zones of exclusion around (0,0) and (1,1). Packages that fall into the zone of pain are very difficult to change because they are extremely stable and cannot be extended since they are not abstract. The zone of uselessness contains packages that are abstract enough but useless since they have no dependents. The packages that remain near the

main sequence are considered to balance their abstractness and instability well. D' has the range [0,1] and it indicates how far a package is from this main sequence. Value 0 indicates that the package is directly on the main sequence whereas value 1 indicates that it is as far away as possible.

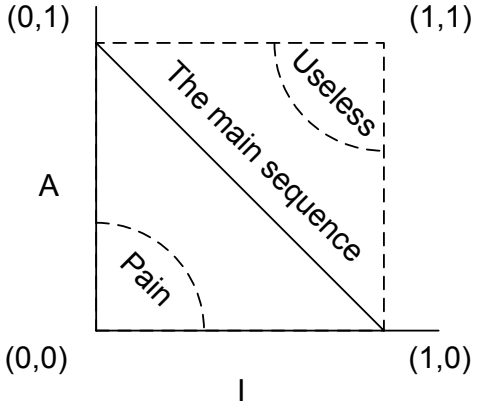


Fig. 1. Distance from the main sequence [10]

4 Empirical Results from a Comparative Case Study

The comparative empirical evaluation of TDD in five small scale software development case studies aims at exploring the effects of TDD on program codes. The layout of the research design for the study is presented first and is followed by the empirical results. The threats to validity are then identified and subsequently addressed.

4.1 Research Design

The research method for the three case projects is the controlled case study approach [18], which combines aspects of experiments, case studies and action research. It is especially designed for studying agile methodologies, and it involves conducting a project with a business priority of delivering a functioning end product to a customer, in close-to-industry settings. At the same time, the measurement data is collected for rapid feedback, process improvement and research purposes. The development is performed in controlled settings and may involve both students and professionals as developers.

All the case projects had the aim of delivery of a concrete software product to a real customer. Two of the projects used ITL development and three utilized TDD: Every project team worked in a shared co-located development environment during the project. The projects were not simultaneous. All the projects continued for nine weeks and followed an agile software development method, Mobile-D™ [19], which provided a coherent framework for this study to compare ITL and TDD. Mobile-D™ is an agile method, which is empirically composed over a series of software development projects in 2003-2006. The method is based on two-month production rhythm, which is divided in five sub phases. Each of the sub phases takes from one to two

weeks in calendar time. These phases are called set-up, core functionality one, core functionality two, stabilize and wrap-up & release. Mobile-D™ adopts most of the Extreme Programming practices, Scrum management practices and RUP phases for life-cycle coverage. The method is described in pattern-format and can be downloaded from `http://agile.vtt.fi`. The code development took place in controlled settings using the same Mobile-D™ practices in all the projects. The only difference was that projects 1 and 2 used ITL and projects 3, 4 and 5 used TDD. The implementations were realized with Java programming language. The difficulty of implementation was at the same level in all the projects, as they all were quite simple systems whose main functionalities were to enable data storing and retrieving.

Table 1 provides a summary of the parts of the projects of which they are not convergent to each other.

Table 1. Summary of the case projects

	Case 1	Case 2	Case 3	Case 4	Case 5
# of developers	4	5	4	2	2
Developer type[1]	S	S	S	P	S
Dev. technique	ITL	ITL	TDD	TDD	TDD
Iterations	6	6	6	4	4
Product type	Intranet	Mobile	Intranet	Internet	Intranet
Total Product size (LOC)	7700	7000	5800	5000	8900 (3100 new)

The development teams of the projects 1, 2, 3 and 5 consisted of 5-6th year Master's students. All the team members in projects 1 and 2, which used ITL, had some industrial coding experience, while only one of the developers in project 3 and none of the developers in project 5, which both used TDD, had previously worked in industrial settings. However, all the subjects in project 3 and 5 were either Software Engineering graduates or had a personal interest in programming. The team members in case project 4, which also used TDD, were professional, experienced developers whose normal daily work includes teaching and development assignments in academic settings. The developers were told and encouraged to write tests in all the projects regardless of the development technique used. In addition, in projects 3, 4 and 5 the use of TDD was stated as mandatory. Intranet applications were implemented in projects 1, 3 and 5: in project 1 for managing research data and in projects 3 and 5 for project management purposes. Case project 5 was a follow-up of case project 3. All the systems consisted of server side and graphical user interface. A stock market browsing system to be used via mobile device was implemented in project 2. The biggest part concentrated on the server side and the mobile part mainly handled connecting to the server and presenting the retrieved data. Internet application for information storing was realized in project 4. The implementation contained a server side and a graphical user interface. However, to make the comparison of TDD and

[1] S= Student, P= Professional.

ITL even, graphical user interfaces and the mobile client application part in project 2 were excluded from the evaluation.

4.2 Results

The results of the traditional and dependency management metrics are presented in the following subsections. These results are discussed in more detail in section 5.

4.2.1 Traditional Metrics

The results of the traditional metrics are presented in Appendix 1a and 1b. The significance of the differences between TDD and ITL were evaluated using the Mann-Whitney U-test (Table 2). WMC, RFC and McCabe's cyclomatic complexity were used to assess the complexity of the code in this study. The WMC values do not differ significantly between the development approaches while the RFC values seem to be lower with TDD. The U-test confirms this distinction as statistically significant ($p<0.05$). The McCabe's cyclomatic complexity results are also lower with TDD and are supported by statistical analysis as well.

The inheritance was studied using DIT and NOC. The DIT values are higher in cases in which TDD was used. This difference is statistically significant. The results of NOC cannot make any difference between the development methods and they are quite surprising, as there are only a few outliners for each case.

The coupling was measured using CBO metric. The results do not differ significantly between the development methods used, and the values are fairly low in all the cases.

The LCOM of CK-suite and LCOM* by Henderson-Sellers were used to find out the cohesion characteristics of the code. The original LCOM does not reveal any differences whereas the new LCOM* seem to be higher in TDD cases, though the difference is not statistically significant.

Table 2. Mann-Whitney p-values for the results of the traditional metrics[2]

	WMC	DIT	NOC	RFC	CBO	LCOM	Cyclomatic Complexity	LCOM*
p	0,389	**0,001**	0,396	**0,018**	0,678	0,535	**0.000**	0,061

4.2.2 Dependency Management Metrics

The measures of the dependency management metrics are presented in Appendix 2. The AC value is much higher in the TDD cases 3 and 5 meaning that there are more classes outside the package that depend on the classes inside the package. Case 4 presents an exception to that, and therefore it cannot be concluded that TDD would produce code with high afferent coupling. The EC values of all TDD cases are significantly lower than the corresponding ITL cases, meaning that in TDD cases there are fewer classes inside the package that depend on the classes outside the package. The instability results show that the package structure of TDD code is more stable than the ITL code. This means that the ITL packages are less dependent on other

[2] The values where the difference is statistically significant are presented in bold.

packages. The measure of abstractness gives only slight indications that TDD may produce packages with a higher level of abstraction. The results of the normalized distance from the main sequence clearly differ between the ITL and TDD cases. Fig. 2 presents the scatter plot of the packages case-wise. The packages in the TDD code seem to come closer to the zone of pain meaning that they are more stable but yet not abstract. However, it should be noticed that the number of packages is much smaller in the ITL cases and that can obviously bias these results.

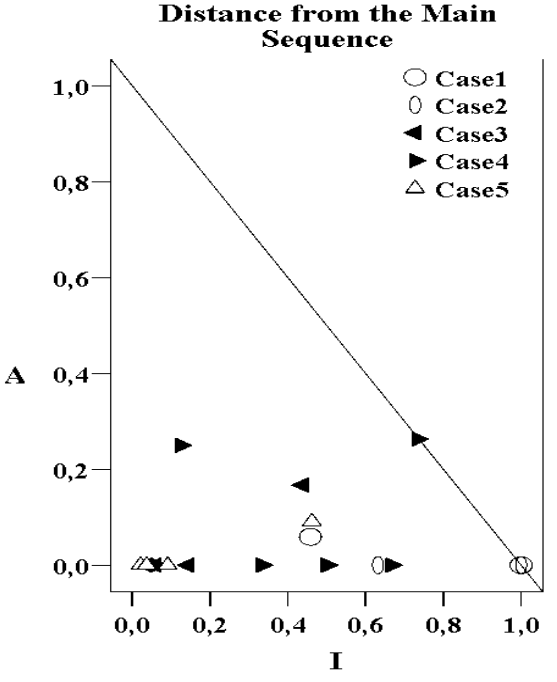


Fig. 2. The distance from the main sequence

4.3 Threats to Validity

The interpretation of results is always more complex when students are used as study subjects. However, the development environment was explicitly designed to relate closely to that of an industrial development setting with strict time-to-market pressures, regular working hours and, significantly, the developers were implementing a real product to be delivered to a real customer. Höst [20] and Runeson [21] suggest that students may provide an adequate model of professionals as similar improvement trends may be identified between both groups. In addition, in industry, teams usually have a mixed set of experiences and skills. The teams in the case projects included in this study represented a similar mix. In addition one TDD team consisted of professional developers only in this study.

The differences between product types and concepts place a threat to internal validity of this study. We excluded graphical user interfaces and the mobile client application part in case 2 to keep the comparison of the projects even. We also proceed in the belief that all product implementations present a similar level of difficulty, as the basic functionality in all the systems was simple data storage and retrieval, which was realized using a model-view-controller structure. The fact, that project 5 is a follow-up of project 3, can somewhat bias the results. However, in project 5, the implementation concentrated on totally new functionalities and the amount of new code is significant in proportion to the total size of the code.

To control the subjects' conformance to implement the tests and to use TDD correctly a person responsible for testing was appointed in all the projects. That person's responsibilities included monitoring the testing implementation. The developers were not aware that the program design was to be compared as the measurements were compiled after the project endings, which reduces possible observation effects. Another limitation relates to the size of the software product as well as the distribution of the project work. All the case projects had less than 10 000 lines of code, their development took around 1000 person hours and a single team in one location developed the products. The impact of TDD on program design, however, should be visible from the very start and thereby present in all of the studied projects.

5 Discussion

The traditional metrics indicated statistically significant differences in DIT, RFC and cyclomatic complexity. These findings partly contradict the findings of Müller [14] and Janzen and Saiedian [11]: Müller reports that in his study, none of the CK-metrics showed differences between the development approaches used, whereas Janzen and Saiedian noticed that the cyclomatic complexity was worse with TDD. Even though DIT was increased with TDD and the difference statistically significant, it should be noted that the level of inheritance was very low in all the cases included in this study regardless of the development approach used. Therefore, it is too early to draw conclusions that TDD encourages to greater use of inheritance. In addition, the target programs were quite small in all the cases resulting in a limited code base, which may be one reason for the low inheritance.

Both, the RFC and cyclomatic complexity, were lower with TDD which may indicate that TDD helps produce less complex code. In this context, it should be noted that the corresponding values for ITL cases were not poor- TDD values just were slightly better. Other traditional metrics did not reveal statistically significant differences. Although the medians of LCOM* results were higher, the statistical significance of this difference is not high according to the U-test.

The results of the dependency management metrics indicate that TDD may cause the software packages to become more stable. The results imply that TDD produces fewer classes inside the package that depend on the classes outside the package. This affects the instability result meaning that TDD produces more packages that are not dependent on other packages but have many dependents. It can be argued that this makes them more difficult to alter *a posteriori*. On the other hand, the high dependency on other packages and the lack of dependents is not desirable either, because it

could cause the packages to change more easily. The measure of abstractness gives only slight indications that TDD may produce packages with a higher level of abstraction, although the difference is not significant. The normalized distance from the main sequence, which measures the ratio of instability and abstractness, differed clearly from the proposed ideal ratio, as it indicated that the TDD packages are too stable in proportion to their abstractness. Both these findings lead us to conclude that the package structure of the code produced with TDD may be difficult to change and maintain, because it is likely to be concrete and have many dependents. This finding contradicts the claims in the literature. On the other hand, the number of packages was clearly higher in the cases in which TDD was used, and is likely to affect the results. i.e. in the ITL cases, there were only two packages in both, while in the TDD cases the corresponding values were 4, 8 and 4. Case 5 is based on the "legacy" code of the case 3, and this is probably one reason for the similarity between the results of these two cases. However, these findings indicate that TDD may result in a greater number of packages that are very concrete in relation to their stability. The fact that the results were similar in all the TDD cases regardless of the professionalism and developers' experience is also significant.

6 Conclusion

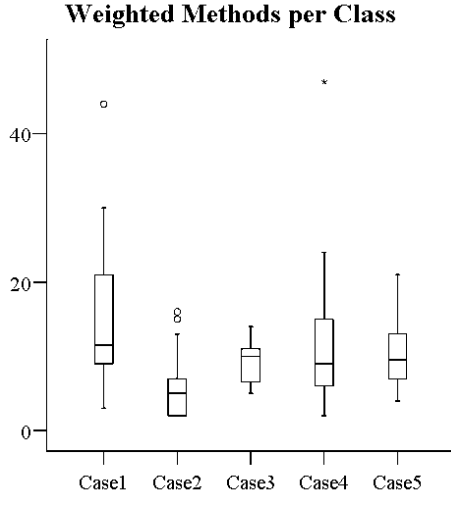
Test-driven development is claimed to be one of the most important practices of agile development, and to address many problems at once. The current empirical research has mainly focused on exploring the external quality effects of TDD. Despite the fact that very little is known about its internal quality effects, academia and industry are eagerly adopting the practice. This study aims at contributing to the empirical body of knowledge by examining the effect of TDD on program design.

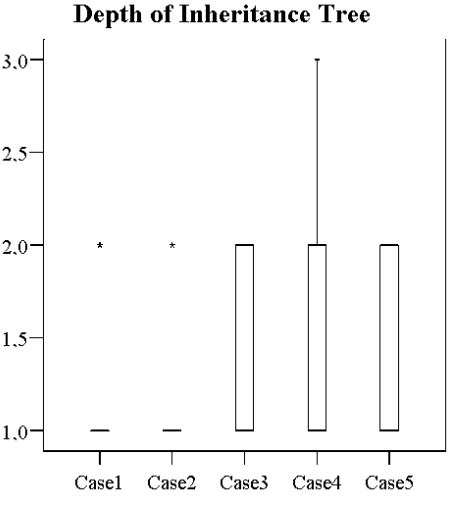
We studied the effect of TDD in five different software projects with students and professionals as research subjects. The results provide some warning that the benefits of TDD are not automatic and as self-evident as expected. Some of the findings imply that TDD may produce a less complex code while other findings indicate the opposite as there are indications that TDD may produce package structures that are more difficult to change. The existing empirical evidence supports the claim that TDD yields improve external quality, especially when employed in an industrial context. This finding clearly conflicts with the case study which identifies certain risks in the adoption of TDD. Therefore, the present authors query whether the reported external quality benefits can be achieved with a more traditional approach to unit-level testing or whether they are really due to TDD itself. We intend to use the results of this study as a baseline for further empirical studies, with experienced developers employing TDD in industrial settings. Our aim is to increase the understanding of test-driven development in different real-life development settings and thereby contribute to the growing body of evidence in the area of agile software development in general and test-driven development in particular. We maintain that whether TDD ultimately improves program design, remains to be answered.

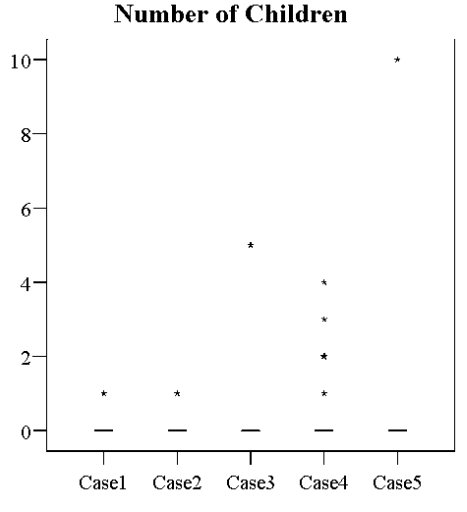
References

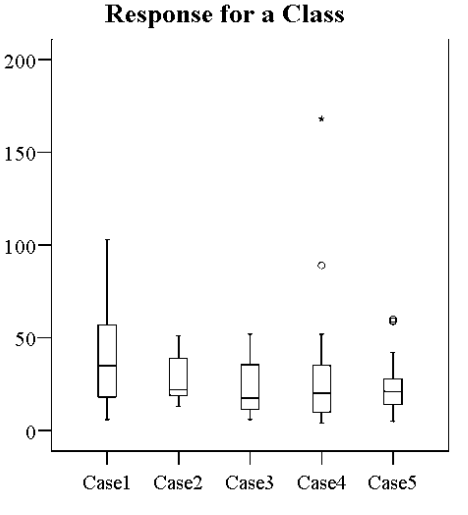
1. Beck, K.: Extreme Programming Explained, 2nd edn. Embrace Change. Addison-Wesley, Boston (2004)
2. Astels, D.: Test-Driven Development: A Practical Guide. Prentice Hall, Upper Saddle River (2003)
3. Beck, K.: Aim, fire. IEEE Software 18(5), 87–89 (2001)
4. Beck, K.: Test-Driven Development By Example. Addison-Wesley, Boston (2003)
5. Boehm, B., Turner, R.: Balancing Agility and Discipline - A Guide for the Perplexed. Addison-Wesley, Reading (2004)
6. Stephens, M., Rosenberg, D.: Extreme Programming Refactored: The Case Against XP. Apress, Berkeley (2003)
7. Siniaalto, M., Abrahamsson, P.: A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage. In: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), pp. 275–284. IEEE Press, New York (2007)
8. Chidamber, S.R., Kemerer, C.F.: A metrics Suite for Object Oriented Design. IEEE Trans.Software Eng. 20(6), 476–493 (1994)
9. McCabe, T.J.: A Complexity Measure. IEEE Trans.Software Eng. 2(4), 308–320 (1976)
10. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Pearson Education, Upper Saddle River (2003)
11. Janzen, D.S., Saiedian, H.: On the Influence of Test-Driven Development on Software Design. In: 19th Conference on Software Engineering Education and Training (CSEET 2006), pp. 141–148. IEEE Press, New York (2006)
12. Kaufmann, R., Janzen, D.: Implications of Test-Driven Development A Pilot Study. In: 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2003), pp. 298–299. ACM, New York (2003)
13. Steinberg, D.H.: The effect of unit tests on entry points, coupling and cohesion in an introductory Java programming course. XP Universe (2001)
14. Müller, M.M.: The Effect of Test-Driven Development on Program Code. In: Abrahamsson, P., Marchesi, M., Succi, G. (eds.) XP 2006. LNCS, vol. 4044, pp. 94–103. Springer, Heidelberg (2006)
15. Basili, V.R., Melo, W.L.: A validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Trans.Software Eng. 22(10), 751–761 (1996)
16. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. Prentice Hall, Upper Saddle River (1996)
17. Shepperd, M.: A critique of cyclomatic complexity as a softwaremetric. Software Engineering Journal (1988)
18. Salo, O., Abrahamsson, P.: Empirical Evaluation of Agile Software Development: The Controlled Case Study Approach. In: Bomarius, F., Iida, H. (eds.) PROFES 2004. LNCS, vol. 3009, pp. 408–423. Springer, Heidelberg (2004)
19. Ihme, T., Abrahamsson, P.: Agile Architecting: The Use of Architectural Patterns in Mobile Java Applications. International Journal of Agile Manufacturing 8(2), 97–112 (2005)
20. Höst, M., Regnell, B., Wohlin, C.: Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. Empirical Software Engineering 5(3), 201–214 (2000)
21. Runeson, P.: Using students as Experiment Subjects - An Analysis of Graduate and Freshmen Student Data. In: Empirical Assessment in Software Engineering (EASE 2003) (2003)

Appendix 1a: The Results of Traditional Metrics

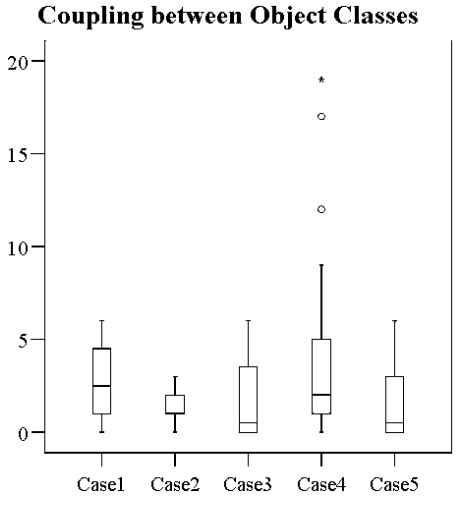


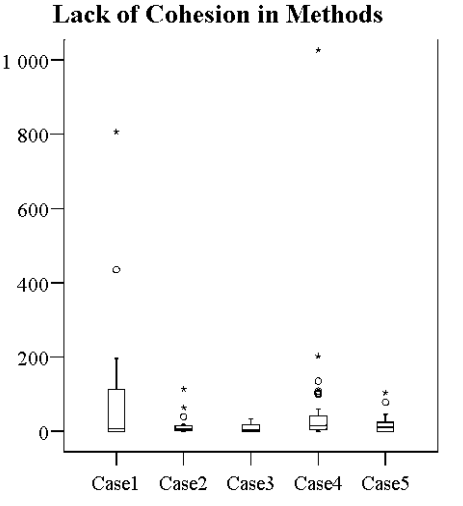


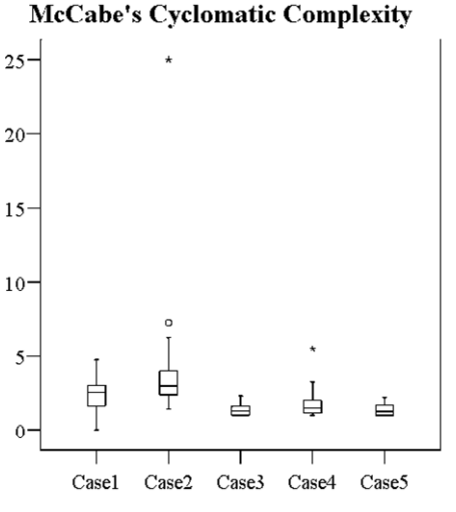


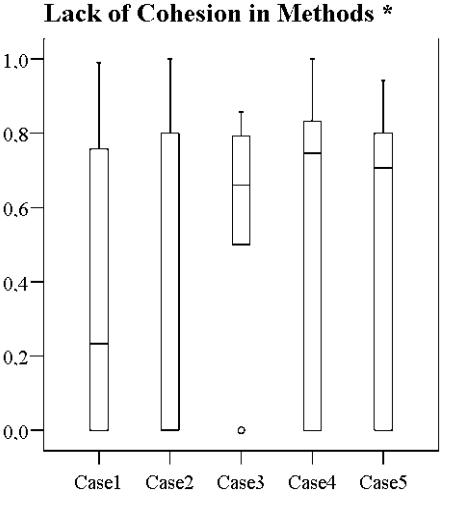


Appendix 1b: The Results of Traditional Metrics









Appendix 2: The Results of Dependency Management Metrics