



Agile Software Development of Embedded Systems

Version : 1.0
Date : 2006.03.03
Pages : 15

Authors

Maria Siniaalto

Status

Final

Confidentiality

Public

Agile Deliverable D.2.7

Test driven
development:
empirical body of
evidence

Abstract

This document contains a review of the existing empirical body of evidence on test-driven development.



I T E A

INFORMATION TECHNOLOGY
FOR EUROPEAN ADVANCEMENT



Test-Driven Development
Deliverable ID:

Page :2 of 15

1.0
Date : 03.03.06

Status: Final
Confid : Final

TABLE OF CONTENTS

1.	INTRODUCTION	4
2.	TEST-DRIVEN DEVELOPMENT	5
2.1	WRITE TEST	6
2.2	RUN TEST	7
2.3	WRITE CODE.....	7
2.4	REFACTOR	7
3.	EXISTING EMPIRICAL BODY OF EVIDENCE.....	8
3.1	RESEARCHES CONDUCTED IN AN INDUSTRIAL CONTEXT	8
3.2	RESEARCHES CONDUCTED IN A SEMI-INDUSTRIAL CONTEXT	9
3.3	RESEARCHES CONDUCTED IN AN ACADEMIC CONTEXT	10
3.4	LIMITATIONS OF THE STUDIES	12
3.4.1	<i>Applicability of Results to the Industrial Settings.....</i>	<i>12</i>
3.4.2	<i>Internal Validity and Generalizability of the Results.....</i>	<i>12</i>
4.	CONCLUSIONS.....	13
5.	REFERENCES.....	14



Test-Driven Development
Deliverable ID:

Page :3 of 15

1.0
Date : 03.03.06


Status: Final
Confid : Final

CHANGE LOG

Ver s.	Date	Author	Description
0.1	17.1.06	Maria Siniaalto	First draft created
0.2	18.1.06	Maria Siniaalto	First full draft
0.3	21.2.06	Maria Siniaalto	Reviewed by Minna Pikkarainen
1.0	3. 3.06	Maria Siniaalto	Language checked

APPLICABLE DOCUMENT LIST

Ref.	Title, author, source, date, status	Identification


	Test-Driven Development Deliverable ID:	Page :4 of 15
		1.0 Date : 03.03.06
		Status: Final Confid : Final

1. INTRODUCTION

This document contains an introduction to Test-driven development (TDD) and it aims at providing an extensive literature review by gathering all the empirical results related to TDD that have been published in the academic articles and experience reports in scientific forums. This work also acts as a basis for an empirical experiment that is arranged by Agile VTT during October - December 2005.

The results of this literature survey show indications that TDD may help to improve software quality significantly, in terms of decreased fault rates, when employed in an industrial context. The results did not show remarkable differences in the developer productivity in any of the studies included in this work.

The document is structured as follows: First, the TDD technique along with its phases is introduced. Next, the findings of the existing studies are categorized based on the context in which they were conducted. The limitations of the studies are also discussed here. Finally, the results are summarized and conclusions are drawn.

	Test-Driven Development Deliverable ID:	Page :5 of 15
		1.0 Date : 03.03.06
		Status: Final Confid : Final

2. TEST-DRIVEN DEVELOPMENT

Test-driven development (TDD) is the core part of the Agile code development approach derived from Extreme Programming (XP) (Beck 2004) and the principles of the Agile Manifesto (Beck, Beedle, et al.). According to literature, TDD is not all that new; an early reference to the use of TDD is the NASA Project Mercury in the 1960's (Larman and Basili 2003). Several positive effects have been reported to be achievable with TDD. It is proposed to guarantee testability and to reach an extremely high test coverage (Astels 2003), to increase developer confidence (Beck 2003), to help to produce highly cohesive and loosely coupled systems and to enable integration more often which allows larger teams of programmers to work on the same code base, because the code can be checked in more often. It is also said to encourage the explicitness about the scope of the implementation while it helps separating the logical and physical design, and to simplify the design, when only the code needed at a certain time is implemented. (Beck 2001)

Despite its name, TDD is not a testing technique, but rather a development and design technique in which the tests are written prior to the production code (Beck 2001). The tests are added gradually during the implementation process and when the test is passed, the code is refactored to improve the internal structure of the code. The incremental cycle is repeated until all functionality is implemented (Astels 2003). The TDD cycle consists of six fundamental steps:

1. Write a test for a piece of functionality,
2. run all tests to see the new test to fail,
3. write code that passes the tests,
4. run the test to see all pass,
5. refactor the code and
6. run all tests to see the refactoring did not change the external behavior.

The first step involves simply writing a piece of code that tests the desired functionality. The second one is required to validate that the test is correct, i.e. the test must not pass at this point, because the behavior under implementation must not exist as yet. Nonetheless, if the test passes, the test is either not testing the correct behavior or the TDD principles have not been followed. The third step is the writing of the code. However, it should be kept in mind to only write as little code as possible to pass the test (Astels 2003). Next, all tests must be run in order to see that the change has not introduced any problems somewhere else in the system. Once all tests pass, the internal structure of the code should be improved by refactoring. The aforementioned cycle is presented in Figure 1 (adapted from (Ambler 2003; Beck 2003; Abrahamsson, Hanhineva, et al. 2004)), and the following subsections discuss the steps of this cycle in more detail.

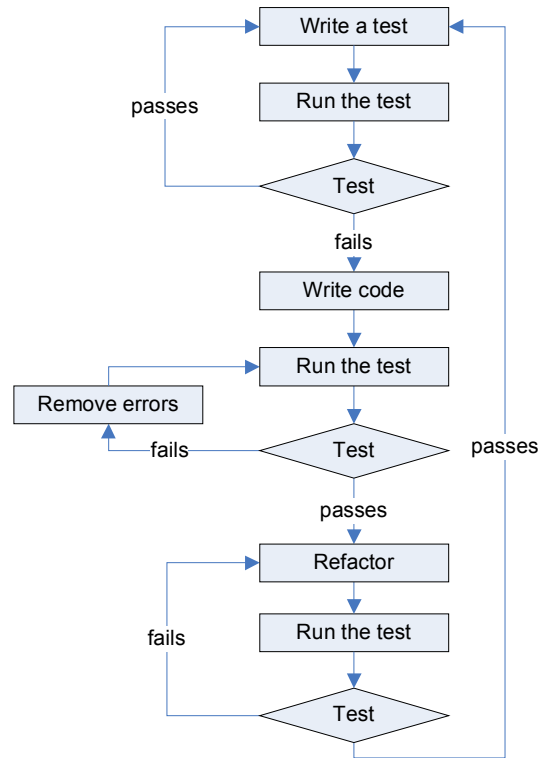



Figure 1. TDD cycle.

2.1 WRITE TEST

Tests in TDD are called programmer tests. They are like unit tests with the difference that they are written for behaviors, not for methods. (Astels 2003) It is important that test are written so that they are order independent, i.e. the result remains the same regardless of the sequence in which the tests are run (Beck 2003).

The tests are grouped into different suites according to what behaviors they test. This allows testing of a particular part of the system without having to run all tests. A test method is the smallest unit and it should test a single behavior while a test case gathers together related tests. It is a common delusion that a test case should contain all tests of a specific class. This misunderstanding is reinforced by most of the IDE plugins, because they usually generate a test case for each class containing test methods for each method. The fixture reuse is allowed by setting preconditions and assumptions identical to each test method with setup() and teardown() methods in each test case. In that way, the tests become repeatable and they can be run identically in the same runtime context time after time. (Astels 2003)

When writing the tests it should be kept in mind that the tests should concentrate on testing the true behaviors, i.e. if a system has to handle multiple inputs, the tests should reflect multiple inputs. It is equally important to refactor the tests during the development cycle, meaning that there should not be a list of 10 input data if a list of 3 items will lead to the same design and implementation decision. (Beck 2003)

	Test-Driven Development Deliverable ID:	Page :7 of 15
		1.0 Date : 03.03.06
		Status: Final Confid : Final

2.2 RUN TEST

The automatic tests should be run after each change of the application code in order to assure that the changes have not introduced errors to the previous version of the code (Astels 2003). The running of the tests is usually performed with the aid of unit testing tools, such as JUnit (2005), which is a simple framework for writing repeatable tests with Java. In the TDD cycle, the test must be run after writing of the test, writing of the code and refactoring.

2.3 WRITE CODE

In TDD, the code writing is actually a process for making the test work, i.e. writing the code that passes the test. Beck proposes three different approaches for doing that: fake it, triangulation, and obvious implementation. The first two are techniques that are less used in concrete development work. Despite it, they may be useful for implementing smaller pieces of a larger and more complex solution, when the developer does not have a clear view about how to implement or abstract the system.(Beck 2003)

"Fake it" may be employed, for example, by replacing the return value of some method with a constant. It provides a quick way to make the test pass. It has a psychological effect while giving the programmer confidence to proceed with refactoring and it also takes care of the scope control by starting from one concrete example and then generalizing from there. The abstract implementation is driven through sensing the duplication between the test and the code. (Beck 2003) "Fake it" implementation can give a push towards the right solution, if the programmer really does not know where to begin to write the code.


Triangulation technique can be used to get to the correct abstraction of the behavior, i.e. the "fake it" solution is not usable anymore. For example, there are at least two different cases in the test method requiring different return values, and obviously, returning of the constant does not satisfy both of them. After reaching the abstract implementation, the other assertion becomes redundant with the first one and it should be eliminated. (Beck 2003)

The obvious implementation is used when the programmer is confident and knows for sure how to implement some operation. Constant practicing of "obvious implementation" can be exhaustive, since it requires constant perfection. When the tests start to fail consecutively, it is recommended to practice the "fake it" or the "triangulation" until confidence returns. (Beck 2003)

2.4 REFACTOR

Refactoring is a process of improving the internal structure by editing the existing working code, without changing its external behavior (Astels 2003). It is essential, because the design integrity of software scatters over time due to the accumulated pressure of modifications, enhancements and bug fixes (Fowler 2002). In TDD it is used to clean up the code after doing the simplest thing possible to make the test pass. Refactoring of the test code is as important as it is to refactor the application code.(Astels 2003)

The idea of refactoring is to carry out the modifications as a series of small steps without introducing new defects into to the system. Refactoring requires a strong suite of automated tests, because it is usually performed manually, and the external behavior of the code may change unintentionally during that process, due to the human factor. (Fowler 2002)

	Test-Driven Development Deliverable ID:	Page : 8 of 15
		1.0 Date : 03.03.06
		Status: Final Confid : Final

3. EXISTING EMPIRICAL BODY OF EVIDENCE

The existing empirical studies on TDD are still quite limited in number. A total of thirteen studies were included in this survey. Seven of the studies were conducted in the University environment: six with undergraduate subjects (Steinberg 2001; Edwards 2003; Kaufmann and Janzen 2003; Pancur, Ciglaric, et al. 2003; Abrahamsson, Hanhineva, et al. 2004; Erdogmus, Morisio, et al. 2005) and one with computer science graduate students (Müller and Hagner 2002). Of the remaining five studies, conducted with professional developers, three were arranged in real industrial settings (Maximilien and Williams 2003; Lui and Chan 2004; Damn, Lundberg, et al. 2005), whereas the last three (Langr 2001; George and Williams 2004; Geras, Smith, et al. 2004) were based on voluntary participation of professional developers.

In addition to these above mentioned studies, a few researches related to computing education and Test-driven development were located. Since they focused on exploring TDD purely as a pedagogical approach to software development and provided very little or no results concerning its effects to the quality or the productivity attributes, they were excluded from this survey. The information presented is gathered from academic articles and experience reports published in scientific forums.

The results of the studies are categorized by the context. Three of the studies are classified as "performed in an industrial context" since they were conducted primarily with professional developers in real industrial settings developing a concrete software product to be delivered. Four studies which were either conducted with professional developers implementing exercises designed for that particular experiment or which were conducted with students working in close-to-industry settings aiming at the delivery of a concrete software product are classified as "performed in a semi-industrial context". The remaining six studies are categorized as "performed in an academic context" because the test subjects were mostly undergraduate students employing TDD to some study specific exercises. Whether the developers were professional is somewhat questionable in (Lui and Chan 2004), but because the developers were not categorized as students, the study is classified as "performed in an industrial context". In some cases, the type of the research is concluded based on its context and focus by the author of this paper.

3.1 RESEARCHES CONDUCTED IN AN INDUSTRIAL CONTEXT

The main findings of the studies performed in industrial settings are gathered in Table 1. It is especially significant that all industrial case studies (Maximilien and Williams 2003; Lui and Chan 2004; Damn, Lundberg, et al. 2005) reported considerably reduced defect rates, as much as 40-50 % (Maximilien and Williams 2003; Williams, Maximilien, et al. 2003). The productivity effects were not that obvious: Damm et al. (Damn, Lundberg, et al. 2005) report TDD significantly decreasing the leadtime while the Maximillien and Williams (Maximilien and Williams 2003) study shows TDD to have no difference or only slight impact to the developer productivity.

The developer experiences were also reported to be positive in studies conducted in industrial settings: The developers were willing to continue the use of TDD (Maximilien and Williams 2003) and even estimated that it will reduce the development time more and more over the time, reaching a 25 % reduction in the third upcoming project (Damn, Lundberg, et al. 2005). Lui and Chan (Lui and Chan 2004) noticed that developers using TDD were able to fix defects found during user acceptance testing much faster.

Table 1. Summary of the studies conducted in an industrial context

Study	Type/ Point of comparison	Number of subjects	Quality effects	Productivity effects	Notes
Lui and Chan (Lui and Chan 2004)	Case study/ Traditional test-last	Not known	Significant reduction in defect density	N/A	Defects were fixed faster with TDD
Maximilien and Williams (Maximilien and Williams 2003), Williams, Maximilien and Vouk (Williams, Maximilien, et al. 2003)	Case study/ Ad-hoc unit testing	9	40-50 % reduction in defect density	No difference/ Minimal impact to productivity with TDD	-
Damm, Lundberg and Olsson (Damm, Lundberg, et al. 2005)	Case Study/ Specialized tool for testing isolated components	Not known	Significantly decreased fault rates	Significantly decreased lead-time	Lead-time estimated to reduce 25 % in the third project

3.2 RESEARCHES CONDUCTED IN A SEMI-INDUSTRIAL CONTEXT

The main findings of studies performed in semi-industrial settings are gathered in Table 2. The quality effects of TDD are not as obvious in this context as with studies performed in industrial settings. However, none of the studies report negative quality effects while George and Williams (George and Williams 2004) report TDD to produce code that passes 18 % more black-box tests and Langr (Langr 2001) reports on significantly improved code quality. When considering productivity effects, George and Williams (George and Williams 2004) noticed that TDD required 16% longer development time while no differences were found in the developer productivity in the experiment conducted by Geras, Smith and Miller (Geras, Smith, et al. 2004).

In the George and Williams (George and Williams 2004) experience, the industrial developers found TDD to improve productivity and to be effective as well as to lead to high test coverage. Langr (Langr 2001) presents that TDD also leads to better maintainability and produces 33 % more tests when compared to the traditional test-last approach. Abrahamsson et al. (Abrahamsson, Hanhineva, et al. 2004) report results differing from the above mentioned: They noticed the test subjects in their experiment indicated strong reluctance to the use of TDD, and found adoption of TDD difficult. The ratio of time the developers spent on developing test and application code was 5.6 % leading to the LOC ratio of test and application code of 7.8 %.

Table 2. Summary of the studies conducted in a semi-industrial context

Study	Type/ Point of comparison	Number of subjects	Quality effects	Productivity effects	Notes
George and Williams (George and Williams 2004)	Controlled experiment/ Traditional test-last	24	TDD passed 18 % more test cases	TDD took 16 % more time	98 % method, 92 % statement and 97 % branch coverage with TDD
Geras, Smith and Miller (Geras, Smith, et al. 2004)	Controlled experiment/ Traditional test-last	14	N/A	No difference	-
Langr (Langr 2001)	Action research/ Traditional test-last	1	Considerable improvement of code quality and better maintainability	N/A	33% more tests with TDD
Abrahamsson et al. (Abrahamsson, Hanhineva, et al. 2004) and Hanhineva (Hanhineva 2004)	Controlled Case Study/ no point of comparison	4	N/A	N/A	The LOC ratio of test and application code is 7.8 %. The ratio of time spent in developing test and application code is 5.6 %

3.3 RESEARCHES CONDUCTED IN AN ACADEMIC CONTEXT

The main findings of studies performed in the academic settings are gathered in Table 3. Three of the studies found no differences in quality when using TDD (Müller and Hagner 2002; Pancur, Ciglaric, et al. 2003; Erdogmus, Morisio, et al. 2005). Though no significant quality improvements were perceived, Erdogmus et al. (Erdogmus, Morisio, et al. 2005) reported TDD to have more consistent quality results. Müller and Hagner (Müller and Hagner 2002) did not find clear differences in quality effects in used techniques either, but they noticed that TDD enables better reuse. Kaufmann and Janzen (Kaufmann and Janzen 2003) found that TDD produces better quality in terms of improved information flow. Steinberg (Steinberg 2001) reported that students practicing TDD produced more cohesive and less coupled code, and were able to correct faults more easily. Nevertheless, Edwards (Edwards 2003) alone was able to give concrete numbers when reporting that 45 % fewer defects were found when using TDD.

When considering the results from the point of view of productivity, the experiments conducted with students indicated that it may increase with TDD in some case: In the Kauffman and Janzen experiment (Kaufmann and Janzen 2003), TDD was measured to produce 50 % more code, where as Pančur et al. (Pancur, Ciglaric, et al. 2003) and Müller and Hagner (Müller and Hagner 2002) did not find any differences. Erdogmus et al. (Erdogmus, Morisio, et al. 2005) reported increased productivity, but no concrete numbers were available. Although they were not able to present accurate numbers, they believed the productivity advantage to be due to a number of synergistic effects, such as better task understanding and focus, faster learning and lower rework effort.




Test-Driven Development
Deliverable ID:

The programmer confidence was found higher among the students using TDD in experiments conducted by Kaufmann and Janzen (Kaufmann and Janzen 2003) and Edwards (Edwards 2003), but the experiment at the University of Ljubljana (Pančur, Ciglaric, et al. 2003) showed results that were quite the opposite: The students felt that TDD was not very effective, but they agreed that developer testing practiced within XP was useful.

Table 3. Summary of the studies conducted in an academic context

Study	Type/ Point of comparison	Number of subjects	Quality effects	Productivity effects	Other
Müller and Hagner (Müller and Hagner 2002)	Controlled experiment/ Traditional test-last	19	No difference	No difference	Better reuse with TDD
Kaufmann and Janzen (Kaufmann and Janzen 2003)	Controlled experiment/ Traditional test-last	8	Improved information flow	TDD produced 50% more code	Higher programmer confidence with TDD
Pančur et al. (Pančur, Ciglaric, et al. 2003)	Controlled experiment/ Iterative test-last	38	No difference	No difference	Students think TDD as not very effective
Erdogmus, Morisio and Torchiano (Erdogmus, Morisio, et al. 2005)	Controlled experiment/ Iterative test-last	24	No difference	TDD more productive	More consistent quality results with TDD
Edwards (Edwards 2003)	Controlled experiment/ No test written previously	59	45 % fewer defects with TDD	N/A	Higher programmer confidence with TDD
Steinberg (Steinberg 2001)	Controlled experiment/ No unit tests written	Not known	More cohesive code with less coupling	N/A	The faults were easier to correct

	Test-Driven Development Deliverable ID:	Page :12 of 15
		1.0 Date : 03.03.06
		Status: Final Confid : Final

3.4 LIMITATIONS OF THE STUDIES

3.4.1 Applicability of Results to the Industrial Settings

Although there are results from three experiments studying TDD in the industrial context, (i.e. test subjects were professional developers aiming at delivering a software product for a business purpose) the environment and settings, in most of the studies, differ significantly from the actual, industrial project settings. This raises some concerns about their applicability to the real industrial context. On the other hand, some of the studies performed in a semi-industrial context (i.e. professional developers implemented exercises designed for some particular experiment or the experiment was conducted with students working in close-to-industry settings aiming at the delivery of a concrete software product), such as (Abrahamsson, Hanhineva, et al. 2004), may provide an adequate analogy to the industrial context. The results of (Lui and Chan 2004) are applicable in the Asian developing countries, and the industrial setting differs notably from the western industrial environment.


Approximately half of the studies included in this survey were performed in the academic context (i.e. test subjects were students implementing exercises specifically designed for the study) and therefore the external validity could be limited. Whether this is the case, can be argued, because the studies comparing students and professionals have concluded that similar improvement trends can be identified among both groups (Runeson 2003) and that students may provide an adequate model of the professionals (Höst, Regnell, et al. 2000).

Another element limiting the industrial applicability of these study results is the scope of the tasks carried out within the experiments: Real software products often consist of several thousands of lines of code and require several developers' work contribution. However, quite many of the experiments consisted of several short tasks and were as small as only a couple of hundred lines of code.

3.4.2 Internal Validity and Generalizability of the Results

The internal validity and generalizability (i.e. external validity) of the study results are considered by the definitions of Yin (Yin 2003). They can be questionable in some studies included in this work due to several factors. One threat is the uncontrolled research environment along with the level of conformance of the subjects to the techniques under study. In this context, "uncontrolled research environment" is used when referring to environment where the researchers have no control over observing whether the Test-driven development technique is really used, i.e. the tests are truly written before implementing the code. This is the case in most of the studies included in this work, though none of them really explains how or whether this threat was overcome.

Another issue limiting the validity and generalizability of the results are the uncontrollable variables in the studies. If the variables cannot be isolated, it is hard to conclude which results are dependent on which variables. For example, in the George and Williams experiment (George and Williams 2004), the results apply to a combination of TDD and pair programming. It is also impossible to compare the results of different studies directly because they had different points of comparison and, as mentioned earlier, some studies employed different software development processes along with TDD. The used metrics were also left undefined in some studies.

	Test-Driven Development Deliverable ID:	Page :13 of 15
		1.0 Date : 03.03.06
		Status: Final Confid : Final

4. CONCLUSIONS

The existing empirical studies provide valuable information about Test-driven development. Nonetheless, the comparison of the studies is difficult, since their focus and the metrics used to describe certain effects varied from one study to another. The limitations of the studies discussed in the previous chapter can depress the significance of their results, especially regarding the industrial applicability. In some studies, the results are also reported with anecdotal evidence.

One of the most important findings is that the software quality seems to improve with TDD. The three studies conducted in the industrial setting, all reported improved quality. The similar effect was not so obvious in the studies conducted in the semi-industrial or academic context, but on the other hand, none of those studies reported decreased quality while five out of ten studies reported indications towards improvement.

The findings related to the developer productivity were contradictory. In the industrial and semi-industrial context, the productivity was mainly evaluated against the time spent on implementing the tasks: The results vary from 16 % increased development time to "significantly decreased project lead-time". The studies performed in the academic context were mainly using the amount of implemented code when evaluating the developer productivity: In two out of six studies, the productivity was not applicable. From the remaining four studies, two reported TDD to be more productive and two did not find any differences to their point of comparison.

The main limitations concerning the applicability of these results to industrial settings are the notable differences with the project settings and scope when comparing to a concrete industrial context. The biggest threat to the internal validity and generalizability of these findings, for one, is the presence of disparate quality and productivity variables. It was also noticed that no researches were found concentrating on the design quality effects on which TDD is proposed to have impact.


Based on the findings of the existing studies, it can be concluded that TDD seems to improve software quality, especially when employed in an industrial context. The findings were not so obvious in the semi-industrial or academic context, but none of those studies reported on decreased quality either. The productivity effects of TDD were not very obvious, and the results vary regardless of the context of the study. However, there were indications that TDD does not necessarily decrease the developer productivity or extend the project lead-times: In some cases, significant productivity improvements were achieved with TDD while only two out of thirteen studies reported on decreased productivity. However, in both of those studies the quality was improved.

The empirical evidence on the practical use of TDD and its impacts on software development are still quite limited. There is definitely a need for further research, especially in the industrial context, in order to be able to generalize the results presented in this paper in a larger context. Another issue noted during this work was the fact that no empirical material reporting on the design quality effects TDD is proposed to have impact on, i.e. coupling and cohesion could be found.



5. REFERENCES

- Abrahamsson, P., A. Hanhineva, et al. (2004). Improving Business Agility Through Technical Solutions: A Case Study on Test-Driven Development in Mobile Software Development. Business Agility and Information Technology Diffusion. R. Baskerville, L. Mathiassen, J. Pries-Heje and J. DeGross. New York, USA, Springer: 227 - 243.
- Ambler, S. W., Introduction to Test Driven Development (TDD), 28.12.2005, <http://www.agiledata.org/essays/tdd.html>
- Astels, D. (2003). Test-Driven Development: A Practical Guide. Upper Saddle River, New Jersey, USA, Prentice Hall.
- Beck, K. (2001). "Aim, fire." IEEE Software **18**(5): 87 - 89.
- Beck, K. (2003). Test-Driven Development By Example. Boston, Massachusetts, USA, Addison-Wesley.
- Beck, K. (2004). Extreme Programming Explained, Second Edition :Embrace Change. Boston, Massachusetts, USA, Addison-Wesley.
- Beck, K., M. Beedle, et al., Manifesto for Agile Software Development, 28.12.2005, <http://www.agilemanifesto.org>
- Damn, L.-O., L. Lundberg, et al. (2005). "Introducing Test Automation and Test-Driven Development: An Experience Report." Electronic Notes in Theoretical Computer Science **116**: 3 - 15.
- Edwards, S. H. (2003). Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance. International Conference on Education and Information Systems: Technologies and Applications, Orlando, Florida, USA, available: <http://web-cat.cs.vt.edu/grader/Edwards-EISTA03.pdf>.
- Erdogmus, H., M. Morisio, et al. (2005). "On the effectiveness of the test-first approach to programming." IEEE Transactions on Software Engineering **31**(3): 226 - 237.
- Fowler, M. (2002). Refactoring. International Conference on Software Engineering, Orlando, Florida, USA, ACM Press.
- George, B. and L. Williams (2004). "A structured experiment of test-driven development." Information and Software Technology **46**: 337 - 342.
- Geras, A., M. Smith, et al. (2004). A prototype empirical evaluation of test driven development. 10th International Symposium on Software Metrics, Chicago, Illinois, USA, IEEE Computer Society.
- Hanhineva, A. (2004). Test-Driven Development in Mobile Java Environment. University of Oulu, The Department of Information Processing Science, Oulu, Finland.
- Höst, M., B. Regnell, et al. (2000). "Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment." Empirical Software Engineering **5**(3): 201 - 214.
- JUnit, JUnit Testing Framework, 29.11.2005, <http://www.junit.org/>
- Kaufmann, R. and D. Janzen (2003). Implications of Test-Driven Development A Pilot Study. ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages and Applications, Anaheim, California, USA, ACM Press.
- Langr, J., Evolution of Test and Code Via Test-First Design, 28.12.2005, <http://www.objectmentor.com/resources/articles/tfd.pdf>
- Larman, G. and V. R. Basili (2003). "Iterative and Incremental Development: A Brief History." IEEE Computer **36**(6): 47 - 56.
- Lui, K. M. and K. C. C. Chan (2004). Test driven development and software process improvement in China. 5th International Conference XP 2004, Garmisch-Partenkirchen, Germany, Springer-Verlag.
- Maximilien, E. M. and L. Williams (2003). Assessing test-driven development at IBM. IEEE 25th International Conference on Software Engineering, Portland, Orlando, USA, IEEE Computer Society.
- Müller, M. M. and O. Hagner (2002). "Experiment about test-first programming." IEE Proceedings **149**(5): 131 - 136.
- Pancur, M., M. Ciglaric, et al. (2003). Towards Empirical Evaluation of Test-Driven Development in a University Environment. EUROCON 2003, Ljubljana, Slovenia, IEEE.

	Test-Driven Development Deliverable ID:	Page :15 of 15
		1.0 Date : 03.03.06
		Status: Final Confid : Final

- Runeson, P. (2003). Using students as Experiment Subjects - An Analysis of Graduate and Freshmen Student Data. Empirical Assessment in Software Engineering, Keele University, Staffordshire, UK, Kluwer Academic Publishers.
- Steinberg, D. H. (2001). The effect of unit tests on entry points, coupling and cohesion in an introductory Java programming course. XP Universe, Raleigh, North Carolina, USA.
- Williams, L., E. M. Maximilien, et al. (2003). Test-driven development as a defect-reduction practice. 14th International Symposium of Software Reliability Engineering, Denver, Colorado, USA, IEEE Computer Society.
- Yin, R. K. (2003). Case Study Research - Design and Methods. 3rd ed. Thousand Oaks, California, USA, SAGE Publications.