Continuous Test-Driven Development
A Novel Agile Software Development Practice and Supporting Tool

Lech Madeyski[1] and Marcin Kawalerowicz[2]
[1]*Wroclaw University of Technology, Wyb.Wyspianskiego 27, 50-370 Wroclaw, Poland*
[2]*Opole University of Technology, ul. Sosnkowskiego 31, 45-272 Opole, Poland*

Keywords: Continuous Testing, Test-Driven Development, TDD, Continuous Test-driven Development, CTDD, Agile.

Abstract: Continuous testing is a technique in modern software development in which the source code is constantly unit tested in the background and there is no need for the developer to perform the tests manually. We propose an extension to this technique that combines it with well-established software engineering practice called Test-Driven Development (TDD). In our practice, that we called Continuous Test-Driven Development (CTDD), software developer writes the tests first and is not forced to perform them manually. We hope to reduce the time waste resulting from manual test execution in highly test driven development scenario. In this article we describe the CTDD practice and the tool that we intend to use to support and evaluate the CTDD practice in a real world software development project.

1 INTRODUCTION

In 2001 a group of forward thinking software developers published "Manifesto for Agile Software Development" (Beck et al., 2001). It proposes a set of 12 principles that the authors recommend to follow. It is not without reason that the first principle of the Agile Manifesto is that the highest priority is to satisfy the customer, while the continuous delivery of valuable software is a way to achieve it. This principle is in the center of our research. We are targeting the concept of test first programming, rediscovered by Beck in the eXtreme Programming (XP) methodology (Beck, 1999; Beck and Andres, 2004). One of the XP concepts is that the development is driven by tests. From this concept Test-Driven Development (TDD) practice arose. It is proposed that using TDD one can achieve better test coverage (Astels, 2003) and development confidence (Beck, 2002). Empirical studies by Madeyski (Madeyski, 2010a) showed that TDD is better in producing loosely coupled software in comparison with traditional test last software development practice.

In this paper we propose the extension of TDD, called Continuous Test-Driven Development (CTDD), in which we combine TDD with continuous testing (Saff and Ernst, 2003) to solve the problem of time consuming test running during the development.

Furthermore, we show the current state of tools around continuous testing and present the tool we use to empirically evaluate the CTDD practice (AutoTest.NET4CTDD), an open source continuous testing plug-in for Microsoft Visual Studio Integrated Development Environment (IDE), that we modified to perform an empirical study in industrial environment on commercial software development project.

We have also performed a preliminary evaluation (pre-test) of the AutoTest.NET4CTDD tool using a survey inspired by Technology Acceptance Model (TAM) (Davis, 1989; Venkatesh and Davis, 2000). The survey was performed in a team of professional software engineers from software development company located in Poland.

2 BACKGROUND

In the subsequent subsections we describe two pillars of the Continuous Test-Driven Development practice proposed in this paper, namely continuous testing and Test-Driven Development practice.

2.1 Test-Driven Development

TDD constitutes an incremental development practice which is based on selecting and understanding a requirement, specifying a piece of functionality as

Madeyski L. and Kawalerowicz M..
Continuous Test-Driven Development - A Novel Agile Software Development Practice and Supporting Tool.
DOI: 10.5220/0004587202600267
In *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2013), pages 260-267
ISBN: 978-989-8565-62-4

a test, making sure that the test can potentially fail, then writing the production code that will satisfy the test condition (i.e. following one of the green bar patterns), refactoring (if necessary) to improve the internal structure of the code, and ensuring that tests pass, as shown in Figure 1.

TDD provides feedback through tests, and simplicity of the internal structure of the code through rigorous refactoring. The tests are supposed to be run frequently, in the course of writing the production code, thus driving the development process. The technique is usually supported by frameworks to write and run automated tests (e.g. JUnit (Gamma and Beck, 2013; Tahchiev et al., 2010), NUnit (Osherove, 2009), CppUnit, PyUnit and XMLUnit (Hamill, 2004)). A good practical approach to TDD is provided in (Freeman and Pryce, 2009), (Koskela, 2007), (Newkirk and Vorontsov, 2004).

TDD has gained recent attention in professional settings (Beck and Andres, 2004; Koskela, 2007; Astels, 2003; Williams et al., 2003; Maximilien and Williams, 2003; Canfora et al., 2006; Bhat and Nagappan, 2006; Sanchez et al., 2007; Nagappan et al., 2008; Janzen and Saiedian, 2008) and has made first inroads into software engineering education (Edwards, 2003a; Edwards, 2003b; Melnik and Maurer, 2005; Müller and Hagner, 2002; Pančur et al., 2003; Erdogmus et al., 2005; Flohr and Schneider, 2006; Madeyski, 2005; Madeyski, 2006; Gupta and Jalote, 2007; Huang and Holcombe, 2009). For example, Madeyski in his monograph on empirical evaluation and meta-analysis of the effects of TDD (Madeyski, 2010a) points out that TDD leads to code that is loosely coupled. As a consequence of Constantine's law, one may claim that TDD supports software maintenance due to loosely coupled code that is less error-prone and less vulnerable to problems arising from normal programming activities, e.g. modifications due to design changes or maintenance (Endres and Rombach, 2003). This important result, although not codified as a law, has serious consequences with respect to software development and maintenance costs.

2.2 Continuous Testing

Continuous Testing (CT) was introduced by Saff and Ernst (Saff and Ernst, 2003) as a mean to reduce the time waste for running the tests. The idea was coined also by Gamma and Beck (Gamma and Beck, 2003) as one of the features of a plugin they described was ability to automatically run all the tests for a project every time the project was built (they called this feature "auto-testing"). One of the goals of TDD is to run the tests often. But while running the tests often the developer needs to interrupt his work often to physically run the tests. Modern IDEs like Eclipse or Visual Studio provide the possibility to keep the codebase in compiled state. This practice is called sometimes continuous compilation (automatic compilation, automatic build). This approach eliminates the waste from compiling the code manually after writing some source code by providing the functionality in IDE to perform the build in background while the developer is writing and/or saving the file. CT relies on this approach and goes one step further by performing the tests while the developer works. There is no need to interrupt the work to run the tests. The tests are ran automatically in the background and feedback is provided to the developer immediately. Saff reports that the waste that is eliminated by doing so is between 92 and 98% (Saff and Ernst, 2003) and has significant effect on success in completing programming tasks (Saff and Ernst, 2004). Good practical approach to CT is given in (Rady and Coffin, 2011) and (Duvall et al., 2007).

There are numerous plug-ins for various IDEs and other tools on the market that enable CT. The tools are mostly provided as plug-ins for modern IDEs. The first tools were developed for Eclipse IDE and Java. Analogous tools were developed for Visual Studio and .NET. There are also tools for other languages like Ruby. Some of the tools supporting CT are Infinitest (open source Eclipse and IntelliJ plug-in), JUnit Max (Eclipse CT plugin), Contester (Eclipse plug-in from the students of Software Engineering Society at Wroclaw University of Technology), NCrunch (NCrunch is very rich commercial continuous testing plug-in for Visual Studio), Autotest (continuous testing for Ruby), Continuous Testing for VS (commercial Visual Studio plug-in), AutoTest.NET (see section 4.2 for details), Mighty Moose (packaged version of AutoTest.NET).

Nowadays CT becomes recognized by the IDE creators themselves. The newest version of Microsoft Visual Studio 2012 comes with continuous testing build-in. This option is available only in the two highest and most expensive versions of Visual Studio 2012 that is Premium and Ultimate.

Hence, the value of the CT practice became recognized by the industry and our aim is to go even further and to take advantage of the synergy of the both ideas (TDD and CT) combined together into an agile software development practice, as well as to collect an empirical evidence on the usefulness of the proposed practice in industrial settings.

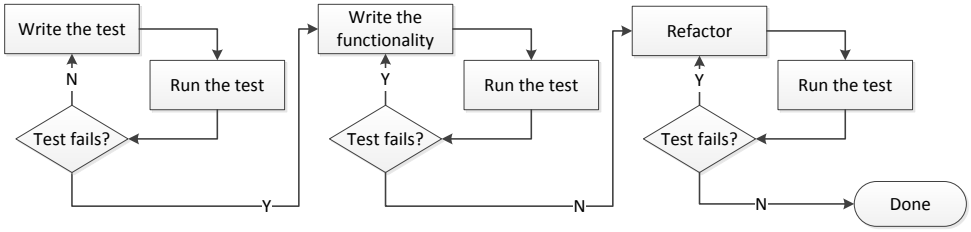

Figure 1: Test-Driven Development activities.

3 CONTINUOUS TEST-DRIVEN DEVELOPMENT

In the aforementioned papers concerning CT we will not find discussion of relationship between CT and TDD. The only exception we know of is a MSc thesis by Olejnik supervised by the first author but their research paper is not even submitted yet.

As the software industry employs millions of people worldwide, even small increases in their productivity could be worth billions of dollars a year. Hence, we decided to investigate a possible synergy of the both aforementioned ideas (TDD and CT) and combine them into an agile software development practice which would demonstrate this synergy. Furthermore, our aim is to provide a preliminary empirical evaluation of the proposed practice which we call Continuous Test-Driven Development (CTDD).

As in TDD the developer writes a potentially failing test for the functionality that not yet exists. Unlike in the TDD there is no need to run the test. The test is run under the hood by an IDE and the test feedback is provided to the developer. The developer can start right away to create the functionality. When he is done (or even while he is typing) the tests will be run in the background and the feedback will be provided. There is no need to manually start the tests to ensure the functionality passes the tests. That is the case for refactoring activity too. There is no need to perform the tests manually. The tests are continuously performed in the background.

Figure 1 shows the common TDD strategy called Red-Green-Refactor. The test is written. The test is run but it fails (or even the code does not compile). We have so called red bar. The functionality is written (possibly faked) as quickly as possible. The tests are run until the green bar is reached. The code is refactored and the tests are run until the code reaches desired form.

Figure 2 shows the enhanced Continuous Test-Driven Development (CTDD) technique. All the steps involving manual test executions are removed because they are no longer being necessary. All the tests are run in the background and feedback is provided straight forward and friction free to the developer. All other steps are still necessary (including the refactoring). It is quite important to deliver the test results to the developer without introducing additional noise to the development process. The developer should not be interrupted in his effort to produce good software but he needs to be provided with visual indicator that the process works as expected or not.

4 USING AutoTest.NET4CTDD TO GATHER EMPIRICAL DATA IN INDUSTRIAL SETTINGS

This section introduces a measurement infrastructure we prepared in order to evaluate the CTDD practice in professional settings.

4.1 Measurement Infrastructure

Figure 3 shows the software/hardware infrastructure assembled to gather the empirical data.

Developers are using Microsoft Windows 7 computers with Visual Studio 2010 installed. To support the continuous testing AutoTest.NET4CTDD tool is used (more information in Section 4.2). Measurement data are gathered automatically using web services and stored in Microsoft SQL Server database. To protect our study from malfunctioning networks or database we added fall-back logging capacities that use local developer machine hard drives and text files. We additionally store all the data in CSV flat file/s. The data gathered will be than assessed using "Evaluator" showed on Figure 3. We purposefully left the description of this part of our infrastructure because it is being developed at the time of writing this paper. Some inspirations are drawn from Zorro (Kou et al., 2010), user action logger (Müller and Höfer, 2007) and ActivitySensor (Madeyski and Szala, 2007).

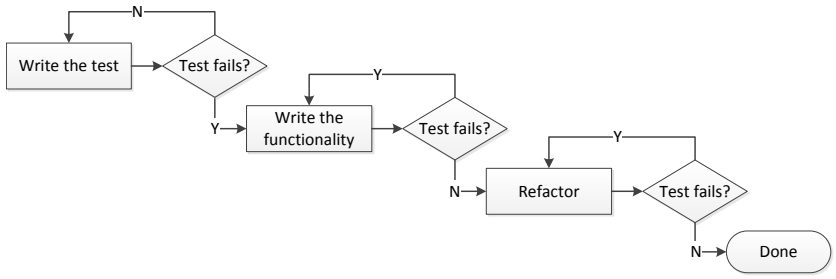

Figure 2: Continuous Test-Driven Development activities.

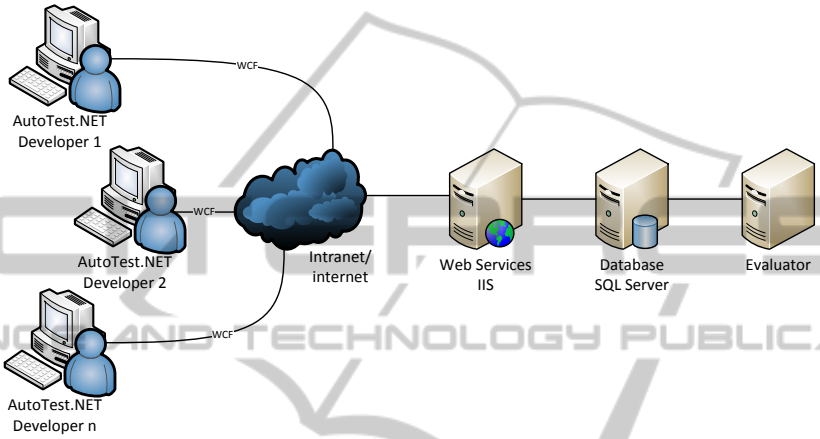

Figure 3: Measurement infrastructure architecture.

4.2 AutoTest.NET4CTDD Description

AutoTest.NET was originally a .NET implementation of Ruby Autotest framework. It was first started by James Avary and hosted on Google Code and then maintained by Svein Arne Ackenhausen on GitHub.

AutoTest.NET does not run all the tests all the time. Such would have devastating influence on large systems. First of all the tests are run only if something significant changes - the developer saves the file. Second not all the tests are run every time. AutoTest.NET is intelligent enough to detect what changes influence what tests and run only the tests that are relevant to the change made. The tests are run in the background and the report is provided in a friction free manner. The development is not interrupted with any dialogue windows. The system runs the tests in the background. Nevertheless for large systems with a lot of tests defined it might require high level of hardware equipment. The computers the developers use need to be fast. Along with the preliminary evaluation of AutoTest.NET4CTDD discussed in Section 6 we have asked the developers if their IDE performs noticeably slower with AutoTest.NET4CDD activated - majority of developers reported no change in performance (only 14% reported above average drop of performance). But it is worth mentioning that the evaluation was performed on average to small projects.

To prepare for the empirical evaluation of the CTDD practice in industrial environment where Microsoft technologies are required, we have chosen AutoTest.NET open source plug-in. We were able to fork it on GitHub add new functionality required to gather measurement data (using Visual Studio 2012 CT functionality would lock us to the newest IDE version and prevent us from extending it with measurements). We have called it AutoTest.NET4CTDD (abbreviation of AutoTest.NET for CTDD). The first version of our plug-in is able to save the various measurements, e.g. related to each build or test run. In the investigated environment, with AutoTest.NET4TDD installed, the empirical data will be stored in a dedicated database and on the developer machine. We built into our version of AutoTest.NET (i.e. AutoTest.NET4CTDD) a client for simple web server that we installed on one of the servers in the company. For web services Microsoft Windows Communication Foundation (WCF) framework is used.

AutoTest.NET4TDD is stored in "Impressive Code" repository as a fork of the original AutoTest.NET plug-in on GitHub. The central web page for our plug-in is hosted under http://madeyski.e-informatyka.pl/tools/autotestdotnet4ctdd/.

Our Plug-in is under constant development and

new functionalities are planned. It is planned to enrich AutoTest.NET4CTDD with functionality derived from ActivitySensor Eclipse plug-in (Madeyski and Szala, 2007).

5 RESEARCH GOAL, QUESTIONS AND METRICS

The overall goal we are going to reach in the empirical study we are preparing to is thorough evaluation of the CTDD impact on software development efficiency. However, the intermediate goal we are going to address in this paper, a set of questions that define this goal and the measurements needed to answer these questions (as required by the GQM method (Basili et al., 1994)) are as follows:

Goal: The evaluation of the new CTDD tool acceptance by professional software developers.

Question 1: What is the Perceived Usefulness of the AutoTest.NET4CTDD Continuous Testing Tool?

Metrics:

M1.1 (better code actuator) — AutoTest.NET4CTDD will help to me produce better code;

M1.2 (efficient work actuator) — AutoTest.NET4CTDD will help to work faster and efficient;

M1.3 (test coverage actuator) — AutoTest.NET4CTDD will help to maintain better test coverage;

M1.4 (subjective perceived usefulness) — Do I want to use AutoTest.NET4CTDD in my projects?

Question 2: What is the Perceived Ease of Use of the AutoTest.NET4CTDD Continuous Testing Tool?

Metrics:

M2.1 (ease of install) — I find the installation process easy and straightforward;

M2.2 (ease of configuration and use) — AutoTest.NET4CTDD is easy to configure and use;

M2.3 (tool discoverability) — Use of AutoTest.NET4CTDD is self-explanatory;

M2.4 (tool performance) — Visual Studio performs noticeably slower with AutoTest.NET4CTDD plug-in activated;

M2.5 (quality of feedback) — AutoTest.NET4CTDD feedback window is useful.

Question 3: Are the developers intending to use the AutoTest.NET4CTDD Continuous Testing Tool?

Metrics:

M3.1 (use intention) — Assuming I had access to, I intend to use it;

M3.2 (use prediction) — Given that I had access to, I predict that I would use it;

M3.3 (commitment level) — My personal level of commitment to using AutoTest.NET4CTDD is low;

M3.4 (intention level) — My personal intention to use AutoTest.NET4CTDD is high.

Question 4: How is the tool perceived in terms of subjective norm?

Metrics:

M4.1 (approval level) — Most people that are important in my professional career would approve of my use of AutoTest.NET4CTDD;

M4.2 (recommendation level) — Most people that are important in my professional career would tend to encourage my use of AutoTest.NET4CTDD.

Question 5: How is the tool perceived in terms of organizational usefulness?

Metrics:

M5.1 (success ratio) — My use of AutoTest.NET4CTDD would make my organization more successful;

M5.2 (benefit ratio) — My use of AutoTest.NET4CTDD would be beneficial for my organization.

Table 1 shows 7-point Likert scale used to measure the metrics.

Table 1: Survey scale.

Answer	Level
Strongly disagree	1
Disagree	2
Disagree somewhat	3
Undecided	4
Agree somewhat	5
Agree	6
Strongy agree	7

We have used the tool described above and performed an empirical acceptance study on it.

6 A PRELIMINARY EVALUATION OF AutoTest.NET4CTDD BY DEVELOPERS

User acceptance of novel technologies and tools is an important field of research. Although many models have been proposed to explain and predict the use of a tool, the Technology Acceptance Model (Davis, 1989; Venkatesh and Davis, 2000) has been the one which has captured the most attention of the Information Systems community.

A survey based on Technology Acceptance Model was carried out among a small group of six professional developers at one of the software development

Table 2: TAM Survey results of the AutoTest.NET4CTDD Continuous Testing Tool.

Question	Mode	Median	Q1 (25th percentile)	Q3 (75th percentile)
Perceived Usefulness:				
better code actuator	5	5	5	5.75
efficient work actuator	3	4.5	3.25	5
test coverage actuator	6	6	5.25	6.75
subjective perceived usefulness	5	5	4.25	5.75
Perceived Ease of Use:				
ease of install	7	7	7	7
ease of configuration and use	4	4.5	4	5.75
discoverability	7	6.5	6	7
tool performance	4	4	2.5	4
quality of feedback	7	7	6.25	7
Intention to Use:				
use intention	5	5	5	5
use prediction	5	5	5	5.75
commitment level	4	4	4	4.75
intention level	4	4.5	4	6.5
Subjective Norm				
approval level	4	5	4	6.75
recommendation level	4	5.5	4.25	6
Organizational Usefulness:				
success ratio	5	5	5	5.75
benefit ratio	5	5	5	5

companies in Opole, Poland. The developers were asked to install the AutoTest.NET4CTDD tool, to use it on a real project and to provide a preliminary evaluation of the tool using our web survey. We asked the developers for example if they think they will be more productive and efficient using the tool, if the tool in their opinion will provide better way to maintain better test coverage, it the tool is easy to use and useful and if the want to use it at all and how committed to use it they are. Table 2 shows results of the survey.

The questions were divided into 5 groups: Perceived Usefulness (PU), Perceived Ease of Use (EOU), Intention to Use (ITU), Subjective Norm (SN), Organizational Usefulness (OU) of the AutoTest.NET4CTDD Continuous Testing Tool. The PU and EOU are drawn from first version of TAM model (Davis, 1989). SN is drawn from the second version of TAM model (Venkatesh and Davis, 2000). PU and EOU are determinants of the intention to use AutoTest.NET4CTDD Continuous Testing Tool among developers. SN is determinant of intention. ITU and OU are variations of determinants describing the direct intention to use (ITU) and an organizational impact of the AutoTest.NET4CTDD Continuous Testing Tool (OU).

After the questionnaire is completed, each item may be analysed separately. Given the Likert Scale's ordinal basis, it makes sense to summarize the central tendency of responses from a Likert scale by using the mode and the median, with variability measured by quartiles or percentiles i.e. first quartile Q1 (25th percentile which splits lower 25% of data), second quartile which is median, third quartile Q3 (75th percentile which splits lower 75%).

Both, mode and median (if we transform answers of negated questions) show that the tool is perceived as rather useful (although not necessarily will help to work faster) and easy to use (although not necessarily easy to configure). The preliminary intention of developers to use our tool is somewhere between positive and sitting on the fence. The subjective norm and organizational usefulness seem to be discernible although slightly. The most positive results regard the usefulness of the AutoTest.NET4CTDD feedback window and the easiness of the installation process. In both cases mode and median are equal to 7 (strongly agree). The former result is promising the usefulness of the fast feedback from the tool is the key concept which could determine the usefulness of the CTDD development practice.

7 DISCUSSION, CONCLUSIONS AND FUTURE WORK

In this paper we have described a novel combination of TDD and CT that we called CTDD. It seems plausible that the CTDD practice as an enhancement of the TDD practice and supported by proper tool (we proposed and made available such a tool) has a chance

not only to gain attention in professional settings, but also impact the development speed and software quality. These kinds of effects have been investigated with regard to the TDD practice, e.g. (Madeyski, 2010a; Madeyski, 2010b; Madeyski and Szala, 2007; Madeyski, 2006; Madeyski, 2005). We showed that the possible synergy effect between TDD and CT can be useful and is worth empirical investigation.

In our empirical study that will came after this paper we will measure the benefits that the CTDD practice deliver in the real-world software project. We will use our AutoTest.NET4CTDD tool (which is an open source tool, available to download) to constitute the "C" that we added to the "TDD" to form CTDD. Performed tool acceptance study showed promising results for further research. The developers found the tool rather useful and easy to use. Extending the AutoTest.NET proved to be relatively easy task. We were able to add the functionality for gathering measurements quite fast. We have developed infrastructure to confidently collect the measurement data. We are certain that the tool gives us enough flexibility to add new functionality in the future during the course of our studies over CTDD.

ACKNOWLEDGEMENTS

Marcin Kawalerowicz is a fellow of the "PhD Scholarships - an investment in faculty of Opole province" project. The scholarschip is co-financed by the European Union under the European Social Fund.

REFERENCES

Astels, D. (2003). *Test Driven development: A Practical Guide*. Prentice Hall Professional Technical Reference.

Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, MA, USA.

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley, Boston, MA, USA.

Beck, K. and Andres, C. (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, MA, USA, 2nd edition.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development. http://agilemanifesto.org/.

Bhat, T. and Nagappan, N. (2006). Evaluating the efficacy of test-driven development: industrial case studies. In *ISESE'06: ACM/IEEE International Symposium on Empirical Software Engineering*, pages 356–363, New York, NY, USA. ACM Press.

Canfora, G., Cimitile, A., Garcia, F., Piattini, M., and Visaggio, C. A. (2006). Evaluating advantages of test driven development: a controlled experiment with professionals. In *ISESE'06: ACM/IEEE International Symposium on Empirical Software Engineering*, pages 364–371, New York, NY, USA. ACM Press.

Davis, F. D. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):319–340.

Duvall, P., Matyas, S. M., and Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional.

Edwards, S. H. (2003a). Rethinking computer science education from a test-first perspective. In *OOPSLA'03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 148–155, New York, NY, USA. ACM.

Edwards, S. H. (2003b). Teaching software testing: automatic grading meets test-first coding. In *OOPSLA'03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 318–319, New York, NY, USA. ACM.

Endres, A. and Rombach, D. (2003). *A Handbook of Software and Systems Engineering*. Addison-Wesley.

Erdogmus, H., Morisio, M., and Torchiano, M. (2005). On the Effectiveness of the Test-First Approach to Programming. *IEEE Transactions on Software Engineering*, 31(3):226–237.

Flohr, T. and Schneider, T. (2006). Lessons Learned from an XP Experiment with Students: Test-First Need More Teachings. In Münch, J. and Vierimaa, M., editors, *PROFES'06: Product Focused Software Process Improvement*, volume 4034 of *Lecture Notes in Computer Science*, pages 305–318, Berlin, Heidelberg. Springer.

Freeman, S. and Pryce, N. (2009). *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 1st edition.

Gamma, E. and Beck, K. (2003). *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Gamma, E. and Beck, K. (2013). JUnit. http://www.junit.org/ Accessed Jan 2013.

Gupta, A. and Jalote, P. (2007). An experimental evaluation of the effectiveness and efficiency of the test driven development. In *ESEM'07: International Symposium on Empirical Software Engineering and Measurement*, pages 285–294, Washington, DC, USA. IEEE Computer Society.

Hamill, P. (2004). *Unit test frameworks*. O'Reilly.

Huang, L. and Holcombe, M. (2009). Empirical investigation towards the effectiveness of Test First programming. *Information and Software Technology*, 51(1):182–194.

Janzen, D. and Saiedian, H. (March–April 2008). Does Test-Driven Development Really Improve Software Design Quality? *IEEE Software*, 25(2):77–84.

Koskela, L. (2007). *Test driven: practical tdd and acceptance tdd for java developers*. Manning Publications Co., Greenwich, CT, USA.

Kou, H., Johnson, P. M., and Erdogmus, H. (2010). Operational definition and automated inference of test-driven development with zorro. *Automated Software Engineering*, 17(1):57–85.

Madeyski, L. (2005). Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality. In Zieliński, K. and Szmuc, T., editors, *Software Engineering: Evolution and Emerging Technologies*, volume 130 of *Frontiers in Artificial Intelligence and Applications*, pages 113–123. IOS Press.

Madeyski, L. (2006). The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design – An Experiment. *Lecture Notes in Computer Science*, 4034:278–289.

Madeyski, L. (2010a). *Test-Driven Development - An Empirical Evaluation of Agile Practice*. Springer.

Madeyski, L. (2010b). The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information and Software Technology*, 52(2):169–184.

Madeyski, L. and Szala, L. (2007). The impact of test-driven development on software development productivity - an empirical study. In Abrahamsson, P., Baddoo, N., Margaria, T., and Messnarz, R., editors, *Software Process Improvement*, volume 4764 of *Lecture Notes in Computer Science*, pages 200–211. Springer Berlin Heidelberg.

Maximilien, E. M. and Williams, L. (2003). Assessing test-driven development at IBM. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 564–569, Washington, DC, USA. IEEE Computer Society.

Melnik, G. and Maurer, F. (2005). A cross-program investigation of students' perceptions of agile methods. In *ICSE'05: International Conference on Software Engineering*, pages 481–488.

Müller, M. M. and Hagner, O. (2002). Experiment about test-first programming. *IEE Procedings-Software*, 149(5):131–136.

Müller, M. M. and Höfer, A. (2007). The effect of experience on the test-driven development process. *Empirical Software Engineering*, 12(6):593–615.

Nagappan, N., Maximilien, E. M., Bhat, T., and Williams, L. (2008). Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3).

Newkirk, J. W. and Vorontsov, A. A. (2004). *Test-Driven Development in Microsoft .Net*. Microsoft Press, Redmond, WA, USA.

Osherove, R. (2009). *The Art of Unit Testing: With Examples in .Net*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

Pančur, M., Ciglarič, M., Trampuš, M., and Vidmar, T. (2003). Towards empirical evaluation of test-driven development in a university environment. In *EUROCON'03: International Conference on Computer as a Tool*, pages 83–86.

Rady, B. and Coffin, R. (2011). *Continuous Testing: with Ruby, Rails, and JavaScript*. Pragmatic Bookshelf, 1st edition.

Saff, D. and Ernst, M. D. (2003). Reducing wasted development time via continuous testing. In *Fourteenth International Symposium on Software Reliability Engineering*, pages 281–292, Denver, CO.

Saff, D. and Ernst, M. D. (2004). An experimental evaluation of continuous testing during development. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 76–85, Boston, MA, USA.

Sanchez, J. C., Williams, L., and Maximilien, E. M. (2007). On the Sustained Use of a Test-Driven Development Practice at IBM. In *AGILE'07: Conference on Agile Software Development*, pages 5–14, Washington, DC, USA. IEEE Computer Society.

Tahchiev, P., Leme, F., Massol, V., and Gregory, G. (2010). *JUnit in Action*. Manning Publications, Greenwich, CT, USA, 2nd edition.

Venkatesh, V. and Davis, F. D. (2000). A theoretical extension of the technology acceptance model: Four longitudinal field studies. *Management science*, 46(2):186–204.

Williams, L., Maximilien, E. M., and Vouk, M. (2003). Test-Driven Development as a Defect-Reduction Practice. In *ISSRE'03: International Symposium on Software Reliability Engineering*, pages 34–48, Washington, DC, USA. IEEE Computer Society.