On the Effectiveness of Unit Test Automation at Microsoft

Laurie Williams[1], Gunnar Kudrjavets[2], and Nachiappan Nagappan[2]
[1]*Department of Computer Science, North Carolina State University*
williams@ncsu.edu
[2]*Microsoft Corporation*
{gunnarku, nachin}@microsoft.com

Abstract

Instituting an automated unit testing practice across a large software development team can be technically challenging and time consuming. As a result, teams may question the economic value of instituting such a practice. One large Microsoft team consisting of 32 developers transitioned from ad hoc and individualized unit testing practices to the utilization of the NUnit automated unit testing framework by all members of the team. These automated unit tests were typically written by developers after they completed coding functionality, approximately every two to three days. After a period of one year of utilizing this automated unit testing practice on Version 2 of a product, the team realized a 20.9% decrease in test defects at a cost of approximately 30% more development time relative to Version 1 of the product. The product also had a relative decrease in defects found by customers during the first two years of field use. Comparatively, other industrial teams have experienced larger decreases in defects when automated unit tests are written iteratively, as is done with the test driven development practice, for a similar time increase. These results indicate automated unit testing is beneficial but additional quality improvements may be realized if the tests are written iteratively.

1. Introduction

Unit testing[1] has been widely used in commercial software development for decades. But academic research has produced little empirical evidence via a large scale industrial case study on the experiences, costs, and benefits of unit testing. Does automated unit testing produce higher quality code? How does "test last" writing of automated unit testing compare with incremental techniques like test-driven development [2]? These are all open research for both researchers and practitioners.

To address these questions and provide such empirical evidence, in this paper we report on a post hoc analysis of unit testing performed in a Microsoft team. In Version 1 of an application, the team had an ad hoc and individualized unit testing practice. In Version 2 of the product, the team instituted a more unified and systematic automated unit testing process. Version 2 consisted of 350 thousand lines (KLOC) of new predominantly C# source code and associated NUnit[2] automated tests produced by 32 developers. Generally unit testing in the context of this paper consists of white box tests in which the author of the tests is aware of how the code handles errors, can inspect the code to verify that the test handles all code paths, and may test the properties and state of the code.

Our research methodology uses a four-phased approach. We conducted a post hoc data analysis of the code, test, bug, and other associated repositories. We also ran a survey of developers and testers, and the first author interviewed a random sampling of four developers and four testers. Finally, the second author conducted action research as a member of the development team.

The rest of the paper is organized as follows: Section 2 provides an overview of prior research on automated unit testing and test driven development. Section 3 provides the detail of our case study, and Section 4 presents our results. Finally, Sections 5 and 6 distill some lessons learned through the use of automated unit tests and conclude this paper, respectively.

2. Background and Related Work

In this section, we provide information on related work in unit testing and the test-driven development mode of writing automated unit tests.

2.1 Unit Testing

By definition and in practice [18], unit testing is done by the developer, not an independent tester or quality assurance person. Unit testing is based upon a

[1] The IEEE definition of unit testing is the testing of individual hardware or software units or groups of related units [13].

[2] http://www.nunit.org/index.php

structural view into the implementation code. Code coverage tools can be used to provide the programmer with insight into which part of the code structure has been exercised by tests. However, developers often do not use code coverage information in their determination that they have tested enough [18, 21] .

In the C# language, developers can automate unit tests using the Visual Studio or NUnit test framework. An example NUnit test appears in Figure 1. Central to unit testing are the assert statements in the figure which are used to compare actual with expected results.

```
/// <summary>
/// Test the basic functionality of
ConvertHexStringToByteArray function.
/// Verify that given the valid input,
function returns expected output.
/// </summary>
public void
TestConvertHexStringToByteArrayBasic1()
{
     byte[] result =
ConvertHexStringToByteArray("080e0d0a");
     Assert.IsNotNull(result);
     Assert.AreEqual(4, result.Length);
     Assert.AreEqual(0x08, result[0]);
     Assert.AreEqual(0x0e, result[1]);
     Assert.AreEqual(0x0d, result[2]);
     Assert.AreEqual(0x0a, result[3]);
}
```

Figure 1: Example NUnit test

2.2 Test Driven Development

TDD is a practice that has been used sporadically for decades [7, 14]. With this practice, a software engineer cycles minute-by-minute between writing failing automated unit tests and writing implementation code to pass those tests. In this section, we provide an overview of TDD research conducted with industrial teams.

A set of TDD experiments were run with 24 professional programmers at three industrial locations, John Deere, Rolemodel Software, and Ericsson [8, 9]. One group developed code using the TDD practice while the other followed a waterfall-like approach. All programmers practiced pair programming [22], whereby two programmers worked at one computer, collaborating on the same algorithm, code, or test. The experiment's participants were provided the requirements for a short program to automate the scoring of a bowling game in Java [15]. The TDD teams passed 18% more functional black box test cases when compared with the control group teams. The experimental results showed that TDD developers took more time (16%) than the control group developers. However, the variance in the performance of the teams was large and these results are only directional. Additionally, the control group pairs did not generally write any worthwhile automated test cases (though they were instructed to do so), making the comparison uneven. The lack of automated unit tests by the control group may reflect developers reduced desire to write tests once they have completed code and feel a sense of assurance that the code produced works properly.

Case studies were conducted with three development teams at Microsoft (Windows, MSN, and Visual Studio) developed in C++ and C# and that used the TDD practice [3, 17]. Table 1 shows a comparison of the results of these teams relative to a comparable team in the same organization that did not use TDD. The TDD teams realized a significant decrease in defects, from 62% to 91%.

Table 1: Microsoft TDD case studies

	Windows	**MSN**	**Visual Studio**
Test LOC [3] / Source LOC	0.66	0.89	0.35
% block coverage	79%	88%	62%
Development time (person months)	24	46	20
Team size	2	12	5
Relative to pre-TDD:			
Pre-TDD Defects/LOC	X	Y	Z
Decrease in Defects/LOC	.38X	.24Y	.09Z
Increase in development time	25-35%	15%	25-30%

A controlled experiment was conducted with 14 voluntary industrial participants [10] in Canada. Half of the participants used a test-first practice, and half of these used a test-last practice to develop two small applications that took 90-100 minutes, on average, to complete. The research indicated little to no differences in productivity between the methods, but that test-first may induce developers to create more tests and to execute them more frequently.

Another controlled experiment was conducted with 28 practitioners at the Soluziona Software Factory in Spain [5]. Each practitioner completed one programming task using the TDD practice and one task using a test-last practice, each taking approximately five hours. Their research indicated that TDD requires more development time, but that the improved

[3] LOC = lines of code

quality could offset this initial increase in development time. Additionally TDD leads developers to design more precise and accurate test cases.

Finally, an IBM case study [11, 16, 17, 19, 23] was conducted with a nine to 17 person team located in North Carolina, USA; Guadalajara, Mexico; and Martinez, Argentina that had been developing device drivers for over a decade. They have one legacy product, written in C, which has undergone seven releases since late 1998. This legacy product was used as the baseline in the case study in comparison to five years and over ten releases of a Java-implemented product. The team worked from a design and wrote tests incrementally before or while they wrote code and, in the process, developed a significant asset of automated tests. Throughout the five years, the team's ratio of test LOC to source LOC varied from 0.55 to 0.75. The block coverage of their automated unit tests was 95%. The IBM team realized sustained quality improvement of 40% fewer test defects relative to a pre-TDD project and consistently had defect density below industry standards. Additionally, the team focused on running automated performance tests throughout development which resulted in the Java device drivers having significantly better performance relative to the legacy product written in C.

3. Automated Unit Testing by Microsoft Team

In this section, we present an overview of the Microsoft case study. First, we present our research methodology. We then present contextual information about the product and the releases under study, the team, and the automated unit testing practice used. We complete this section with some limitations to our empirical approach.

3.1 Research Methodology

The research involves mining the software repositories (source code, test, and bug), survey deployment, interviews, and action research. The second and third authors mined the repositories and to obtain quantitative metrics, such as lines of source code, test code and number of defects for each releases. Additionally, the second author was part of the development team and could be considered an action researcher. His knowledge of the daily operations of the team is shared throughout this paper.

The third author conducted two anonymous surveys which were administered on an internal Microsoft survey system. One survey was for the developers and the other was for the testers. The purpose of the surveys was to obtain qualitative and quantitative information about the use of automated testing from the developer and test team. The developer survey was offered to 32 developers and answered by 11 (34.4%). The tester survey was offered to 15 testers and answered by two (13%). Due to the low response rate, the tester survey data was not analyzed. Finally, the first author conducted one hour interviews of four developers and four testers on the team. Survey and interview protocols are provided in the appendix.

3.2 The Project

We compare the results of two versions, Version 1 (V1) and Version 2 (V2), of a Microsoft application. During both versions the development was mainly done in C#, though some smaller, specific pieces written in C/C++. Version 1 of the product consisted of 1,000 KLOC and was developed over a period of three years. Similarly, Version 2 of the product consisted of 150 KLOC of changed lines of code and 200 KLOC of new lines (resulting in a V2 code base of 1,200 KLOC) developed over a period of two years. For both releases, we examine a one year period of development which encompasses design, code implementation, testing, and stabilization. V1 and V2 are very different. V1 used external components for performing much functionality, while V2 developers wrote their own libraries to work at the protocol level.

3.3 Team

The development team was co-located in Redmond, Washington, USA. During the V1 timeframe 28 developers contributed to the production code. In V2 this number rose to 32. Additionally, approximately 22 testers were on the V1 team, and 28 testers were on the V2 team. Comparing the two team lists, 14 developers were involved in both V1 and V2, about 50% of the development team.

3.4 Development Practices

In this section, we describe the development practices for the V1 and V2 teams.

3.4.1. Version 1. The development process was quite linear. Program management and senior developers came up with the vision, feature list, and user scenarios in the form of Microsoft Word documents. Based on that information, developers wrote design documents for review. Design documents were published in a form of Word documents which contained diagrams describing the architecture and design of different

components. Diagrams were mainly authored in Microsoft Visio and used Unified Modeling Language (UML) notation. After the developers' design documents were made public to the entire product team the test team developed corresponding test plans and conducted test plan reviews. Primarily, developers participated in design reviews and testers participated in test plan reviews with minimal involvement of testers in design reviews and vice-versa. After the developers' design documents were reviewed, the developers began coding and debugging until a satisfactory level of quality was achieved. This quality level was based primarily on code review feedback and individual developer's gut feeling. Some code was run through a static analyzer. Some code was informally inspected by other team members. Then, functional tests (called build verification tests or BVTs at Microsoft) were run on the code base which includes the newly-developed code. The BVTs were run as both acceptance tests of the newly-completed feature and as regression tests to check whether the new code broke existing code. The new code was checked into the code base when the BVTs pass.

Developers rarely, if ever, wrote any automated test cases. Those who did kept that code on their own machine. The majority of tests written in this manner were one time use to verify certain functionality.

After source code was checked into the source tree, the test team drove the daily build going forward. Testing consisted of running manual ad-hoc and previously-planned testing. The testers planned their test cases while the developers wrote code. They based their tests on the feature specification/user scenarios provided by the product manager. They discovered additional details necessary for planning their tests by further discussions with the product manager and with the developer. They wrote performance, stress, and security tests.

3.3.2 Version 2. Similar to V1, program management and senior developers came up with the vision, feature list, and user scenarios in the form of Microsoft Word documents. Based on that information, developers wrote design documents for review. In the beginning of V2, the practice of writing NUnit automated unit tests was mandated by the development manager. These automated unit tests were called Developer Regression Tests or DRTs. Similar to V1, some code was run through static analysis and/or informal code review. New code had to pass the BVT tests prior to check in. The informal code review included reviewing associated automated unit tests. When appropriate, a reviewer may comment if they thought the developer did not write sufficient unit tests.

The team did not adopt TDD [2] whereby unit tests and code are written incrementally on a minute-by-minute basis. Rather, the team did unit testing after completing the coding of a requirement. Most developers reported writing unit tests every two to three days based upon the code finished in the prior days. Developers focused their unit tests on testing the new functionality, including boundary value and error conditions. Occasionally, developers wrote unit tests to probe performance or security concerns. Most developers indicated that they wrote unit tests until they felt they covered all the necessary scenarios, including both normal and error conditions. Some indicated that their unit test writing may be abbreviated due to time constraints.

Approximately four times the number of automated unit tests were written for V2 than were written for V1. As discussed earlier, the automated unit tests for V1 were ad hoc, maintained on the developer's machine, and were often run only once. Conversely, the V2 unit tests were maintained with the code base such that developers could run other developers' unit tests. Survey results indicated that most developers ran their own unit tests at least once per day and ran the unit tests of other developers at least once per week.

The test line of code to source line of code ratio for V2 was 0.47 and the statement coverage was 33.4% This coverage does not include BVTs and other testing which contributed to a total coverage > 85%.

The testers followed the same process as they did with V1. However, as will be discussed in Section 4.3, the testers noted that they had to work much harder to find defects with V2 than they had to with V1. The testers noted that all V1 tests were run on the V2 code. Additional tests for new V2 functionality were added to their test set. More so with V2 than V1, specific security test cases were planned and executed due to Microsoft's focus on the Secure Software Development Lifecycle [12].

3.5 Limitations of Case Study

Formal, controlled experiments, such as those conducted with students or professionals, over relatively short periods of time are often viewed as "research in the small" [6]. These experiments may suffer from external validity limitations (or perceptions of such). On the other hand, case studies such as ours can be viewed as "research in the typical" [6]. However, concerns with case studies involve the internal validity of the research, or the degree of confidence and generalization in a cause-effect relationship between factors of interest and the observed results [4].

Case studies often cannot yield statistically significant results due to the lack of random sampling. Nonetheless, case studies can provide valuable information on a new technology or practice. By performing multiple case studies and recording the context variables of each case study, researchers can build up knowledge through a family of studies [1] which examine the efficacy of a software development practice, such as automated unit testing.

The results presented in this paper are based upon the work of more than 32 developers and 28 testers over a period of two years. As a result, the quantitative results cannot conclusively be attributed to the use of automated unit testing. On the other hand the results contribute to the body of knowledge of the effectiveness and performance of automated unit testing. Our results, therefore, apply to teams that follow a similar process to that discussed in Section 3.3.2.

4. Results

In this section, we provide the results of our analysis of study data. First, we look at the defects of both releases. Second we provide information about the perception of utility of unit testing by the developers and testers. Finally we compare the results of this teams test-last automated unit testing practice with the results of TDD studies available in the literature.

4.1 Defects

For V2, the team experienced a 20.9% percent decrease in pre-release defects found by the test team and by internal customers, despite having a significant amount of code churn and added features. Of these, as shown in Table 2, the percentage of higher severity (Severity 1 and 2 combined) defects also declined from a total of 65.3% of total defects to 56.9% of total defects. Additionally, as discussed in Section 4.2 and 4.3, the developers and testers note differences in the kinds of defects found in V1 versus V2. Due to the proprietary nature of the system we do not provide absolute number of defects.

Table 2: Defect Severity Distribution

Defect Severity	Version 1 (%)	Version 2 (%)
Severity 1	15.5%	16.8%
Severity 2	49.8%	40.1%
Severity 3	28.7%	18.8%
Severity 4	6.0%	3.4%

Field quality also improved. Product statistics indicate that the quantity of defects found on the product during the first two years increased by a factor of 2.9. However, the customer base for the product increased by at least a factor of 10. Such a large increase in customers would be expected to cause significantly more defects because the larger customer base is likely to use the product in a wider variety of ways, executing different areas of the code base. We therefore view the data as indicating a relative decrease in customer-reported failures.

4.2 Developer Perception

Increased quality was achieved. In addition to the quantitative results provided in Section 4.1, the developers sensed their quality improvement based upon their interactions with the test team. The developers did not maintain accurate effort estimate to gauge the impact (if any) on productivity. Hence, on the survey and in the interviews we asked the developers for their perception of how much overall time it took to write the unit tests. Of the eleven developers that answered this question and four developers' interview, responses ranged from 10% to 200%. The median and mode responses were 30%. An increase in quality can pay for moderate productivity losses. Specifically, the developers noted that they spent less time fixing defects found by testers particularly in the "stabilization phase." During stabilization, no new feature gets developed. The entire focus is on fixing the bugs, ensuring functional correctness, making test cases pass, and gradually increasing the product quality with controlled amount of code changes.

Figure 2 illustrates survey results that provide insight into the developers' perceptions on the value of automated unit testing. Figure 2 provides the questions in descending order of positive feeling by the developers. The developers have the highest degree of positive perception towards writing unit tests to execute a code fix once a defect is found; and about the fact that writing unit tests help them write higher quality code. The developers also felt that writing unit tests helped them write solid code from the start. The developers echoed this sentiment in the interviews by stating that writing automated unit tests had increased their awareness for implementing code for error conditions and for handling boundary cases correctly. The majority of developers felt positive that unit tests help them understand the code when they "inherit" it from others or when they need to debug the code. Only 40% of developers felt that defects were more likely to be found in places that have no unit tests.

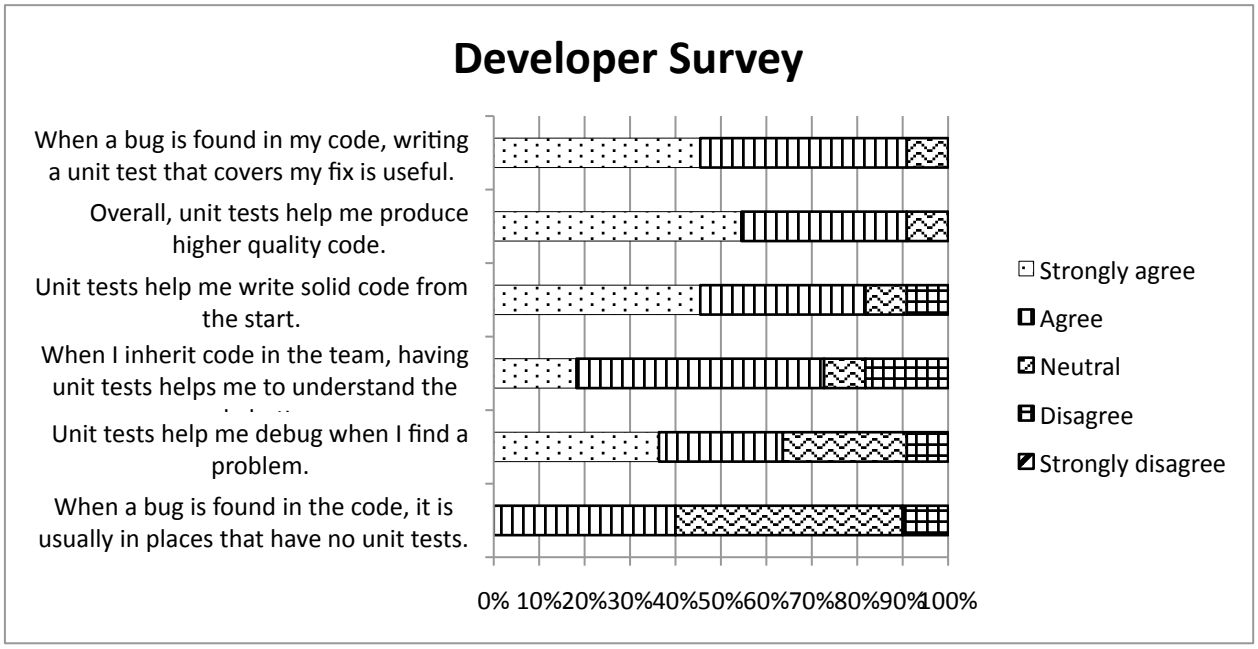

Figure 2: Developer Perception

The developer interviews provided additional insights into their perceptions. In general, the developers feel that unit testing was worth their time and that it helped them to catch the "low hanging bugs" before being tested by the test team. They also felt more comfortable modifying other developer's code and refactoring due to the presence of automated unit tests. The developers felt that the testers were finding more complex defects in V2 than in V1, so the 20% decrease in defects was accompanied by the perception that test was able to do a better job of finding defects customers would find.

4.3 Tester Perception

In the interviews, the testers unanimously noted that the code delivered to them was of higher quality and that they had to work harder to find defects. Some verbatim comments from the interviews were as follows:

Bugs used to come "for free."

It's harder to find bugs. Now, all the obvious bugs are gone.

Ad hoc testing is not as easy as it used to be.

The testers noted that they were able to get more sophisticated and realistic with their testing. They still found defects. However they felt, given their normal time constraints, they were more effective as finding the more comprehensive defects that would have been discovered by customers rather than sticking to an isolated area of new functionality.

4.3 Comparison with Test-Driven Development

The team discussed in this case study realized a 20.9% decrease in test defects in V2 in which they wrote automated unit tests, relative to V1 which did not have automated unit tests. From V1 to V2, product quality improved which the team attributes to the introduction of automated unit tests into their process.

A comparison of case studies of TDD teams, as reported in Section 2, indicates that additional quality improvements may be gained by writing unit tests more incrementally as is done with TDD. The TDD teams had 62% to 91% fewer defects. The incremental nature of writing unit tests via TDD may cause teams to write more tests. The TDD teams had a higher test LOC to source LOC ratio and higher test coverage. These results are consistent with a controlled case study conducted in Finland of three nine-week projects [20]. Results of this case study indicated that test coverage was higher when tests were written incrementally before writing code.

5. Lessons Learned

In addition to the empirical data provided in Section 4, we share some suggestions for other teams

considering transitioning to the use of automated unit testing:

- Management support for unit testing is necessary. Without the leadership realizing the benefits of unit tests and supporting it, the efforts will most likely die after a couple of months if supported only by a few of the enthusiasts.
- There needs to a single tool mentality across the entire product team. The situation where one team uses NUnit, second team some other tool, and a third team is starting to write their own test harness, will not pay off. The "enforcement" of a single tool is enabled through a tool such as NUnit which can be run and monitored as a part of the build process.
- The time for developing unit tests needs to be factored into the development schedule. The entire product team must understand that development may take longer, but the quality will be higher and in the end the final result will be better.
- Unit tests need to be considered as part of the product code, i.e., they need to evolve in parallel with other pieces of the product, their code quality needs to the same as product code quality.
- Testability of the system should be considered as part of architecture and design to understand how it can impact the tests, test effort and effectiveness of the testing process.
- Unit testing coverage needs to be measured. The quantity of test cases is a bad measurement. More reliable is some form of code coverage (class, function, block etc.). Those measurements need to be communicated to the team and be very visible.
- A simple team rule that was used in the Microsoft team: before checking in a change to the code base, all the unit tests are run and verified that all of them pass. Following this rule made a significant difference in the team's ability to prevent bugs from escaping.
- Occasionally some amount of root cause analysis needs to be performed on all the bugs in the product to determine if these bugs can be prevented by writing appropriate unit tests. For example: if public APIs have bugs for not checking parameters then it makes sense to invest some effort into developing a comprehensive unit test suite for all the public APIs.
- The unit testing suite must be open for contributions by both development and test teams. Team members must not feel that unit testing is only the developer's game. Anyone who can write a test case which verifies some aspect of the system, should be allowed to add it to the suite.
- Execution of the unit test suite should be easy. If it is not easy then people might not be motivated to run it. The execution time of the test suite also needs to be monitored carefully and made sure that it stays short (for example in the Microsoft case the tests ran in less than ten minutes).

6. Summary

One large Microsoft team consisting of 32 developers transitioned from ad hoc and individualized unit testing practices to the utilization of the NUnit automated unit testing framework by all members of the team. These automated unit tests were typically written by developers after they completed coding functionality, approximately every two to three days. The tests were daily by the developers and run nightly as part of the build process. Developers who did not include unit tests were prompted to do so by peer code reviewers. The software developers felt positive about their use of automated unit testing via the NUnit framework. Testers indicated their work change dramatically because the "easy" defects were found by the developers or were prevented due to the presence of automated unit tests. The test cases needed to be more complex for defects to be found.

After a period of one year of utilizing this automated unit testing practice on Version 2 of a product, the team realized a 20.9% decrease in test defects. Additionally, customer-reported defects during the first two years of field use increased by 2.9X while the customer base increased by 10X, indicating a relative decrease in customer-reported defects. This quality increase came at a cost of approximately 30% more development time. Comparatively, other teams at Microsoft and IBM have realized larger decreases in defects (62% to 91%) when automated unit tests are written incrementally with TDD, for a similar time increase. The TDD teams had a higher test LOC to source LOC ratio and higher test coverage. These results indicate automated unit testing is beneficial. However, increased quality improvements may result if the unit tests are written more incrementally.

7. Acknowledgements

We thank the Microsoft developers and testers who participated in this case study.

8. References

[1] V. R. Basili, F. Shull, and F. Lanubile, "Building Knowledge Through Families of Experiments," *IEEE Transactions on Software Engineering,* vol. 25, no. 4, pp. 456 - 473, 1999.
[2] K. Beck, *Test Driven Development -- by Example*. Boston: Addison Wesley, 2003.
[3] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development: industrial case studies," in *ACM/IEEE international symposium on International symposium on empirical software engineering*, Rio de Janeiro, Brazil, 2006, pp. 356 - 363
[4] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Design for Research*. Boston: Houghton Mifflin Co., 1963.
[5] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. A. Visaggio, "Evaluating Advantages of Test Driven Development: a Controlled Experiment with Professionals," in *International Symposium on Empirical Software Engineering (ISESE) 2006*, Rio de Jaiero, Brazil, 2006, pp. 364-371.
[6] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*: Brooks/Cole, 1998.
[7] D. Gelperin and W. Hetzel, "Software Quality Engineering," in *Fourth International Conference on Software Testing*, Washington, DC, June 1987.
[8] B. George, "Analysis and Quantification of Test Driven Development Approach MS Thesis," in *Computer Science* Raleigh, NC: North Carolina State University, 2002.
[9] B. George and L. Williams, "An Initial Investigation of Test-Driven Development in Industry," in *ACM Symposium on Applied Computing*, Melbourne, FL, 2003, pp. 1135-1139.
[10] A. Geras, M. Smith, and J. Miller, "A Prototype Empirical Evaluation of Test Driven Development," in *International Symposium on Software Metrics (METRICS)*, Chicago, IL, 2004, pp. 405 - 416.
[11] C.-w. Ho, M. J. Johnson, L. Williams, and E. M. Maximilien, "On Agile Performance Requirements Specification and Testing," in *Agile 2006*, Minneapolis, MN, 2006, pp. 47-52.
[12] M. Howard and S. Lipner, *The Security Development Lifecycle*. Redmond, WA: Microsoft Press, 2006.
[13] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.
[14] C. Larman and V. Basili, "A History of Iterative and Incremental Development," *IEEE Computer,* vol. 36, no. 6, pp. 47-56, 2003.
[15] R. C. Martin and R. S. Koss, "Engineer Notebook: An Extreme Programming Episode," *http://www.objectmentor.com/resources/articles/xpepisode.htm,* no. 2001.
[16] E. M. Maximilien and L. Williams, "Assessing Test-driven Development at IBM," in *International Conference of Software Engineering*, Portland, OR, 2003, pp. 564-569.
[17] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams," *Empirical Software Engineering,* vol. 13, no. 3, pp. 289-302, June 2008.
[18] P. Runeson, "A Survey of Unit Testing Practices," *IEEE Software,* no. pp. 22-29, July/Aug 2006.
[19] J. Sanchez, L. Williams, and M. Maximilien, "A Longitudinal Study of the Test-driven Development Practice in Industry," in *Agile 2007*, Washington, DC, pp. 5-14.
[20] M. Siniaalto and P. Abrahamsson, "A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage," in *International Symposium on Empirical Software Engineering and Measurement*, Rio de Janiero, Brazil, pp. pp. 275-284.
[21] B. Smith and L. Williams, "A Survey on Code Coverage as a Stopping Criterion for Unit Testing," North Carolina State University Technical Report TR-2008-22, 2008.
[22] L. Williams and R. Kessler, *Pair Programming Illuminated*. Reading, Massachusetts: Addison Wesley, 2003.
[23] L. Williams, E. M. Maximilien, and M. Vouk, "Test-Driven Development as a Defect-Reduction Practice," in *IEEE International Symposium on Software Reliability Engineering*, Denver, CO, 2003, pp. 34-45.

Appendix

Here we provide the questions asked of the developers and testers in our surveys.

A.1 Developer Survey

1. When you develop code, do you work from a design document?
   - o Yes
   - o No
2. If you use a design document, is it a high level design document?
   - o Yes
   - o No
3. Approximately how many lines of code (LOC) do you write before you start writing DRTx/unit tests for it? [open ended]
   - o The structure of the code
4. Approximately what percentage of your unit tests are automated? [open ended]
5. What is the typical source lines of code to test lines of code ration for your code? For example, 0.3 would mean that for 100 lines of source code you have 30 lines of test code. [open ended]
6. Which of these do you think about in your unit tests? (check all that apply)
   - o Security
   - o Reliability
   - o Functionality
   - o Performance
   - o Other
7. What stopping criteria do you use to stop writing unit tests? [open ended]

8. How often do you run your unit tests per week? [open ended]
9. Do you run unit tests from other developers/testers?
   - Yes
   - No
10. Overall, how much time do you feel writing unit tests adds to your development time? [open ended]

Note: Six additional questions on developer perception are presented in Section 4.2.

A.2 Developer Interview Prototcol

1. When you develop code, do you work from a design document? If so, can you tell me the form of that document – for example is it high or low level design?
2. Please explain your process when you write code.
3. Depending upon the answer to #2, when do you write unit tests, before you write the code, as you write the code, or after you finish some code? If you write unit tests after you finish writing code, how much code to you finish before you write some tests?
4. Do you automate your unit tests? If so, what technology do you use to automate your tests?
5. On average, about how many lines of code is a typical unit test? About how many methods in the source code are typically executed with each unit test?
6. What do you think about when you write unit tests? (e.g. the structure of the code, the requirement . . .) Based on answer – probe about whether the unit tests are really white box-ish or black box-ish.
7. Do you ever unit test for a non-functional requirement, such as security or performance? If so, how do you go about doing that?
8. How do you decide when you have written enough unit tests?
9. How often do you run your own unit tests?
10. Do you ever run the unit tests from others in our team? The whole team or part of the team? How often?
11. When you have turned your code over to the testing group, and the testers find a problem with your code, do you ever go back and write a unit test to reveal that bug they find? If yes, do you find this useful?
12. Have you ever inherited code from another team member that has had unit tests? Does it help you to learn the code when it has unit tests?
13. How helpful is your bank of unit tests for regression testing? Do you find it to be a safety net – do you feel more courageous when you make a change that the unit tests will tell you if you screwed something up with the change?
14. Do you think unit test helps you write more solid code from the start? Why or why not?
15. Do you think unit tests help you to debug when you do find a problem? Why or why not?
16. Overall, do you think writing unit tests helps you produce a higher quality product? Why or why not?
17. Overall, how much time do you feel writing unit tests adds to your development time?
18. When yvou are pressured for time, do you think you write less unit tests?
19. Do you ever pair program as you write code? Why or why not?

A.3 Tester Interview Protocol

1. Can you tell me about the types of tests you write (e.g. is it integration testing, are they functional in nature, test a specific type of non-functional requirement, test the system as a whole in a typical customer environment)?
2. What do you base your test cases on (e.g. a requirements document, conversations with the developer, conversations with a requirements analyst, conversations with a customer)?
3. When in the development process do you write these tests?
4. How do you decide when you have written enough tests?
5. Do you test for a non-functional requirement, such as security or performance? If so, how do you go about doing that?
6. Do you use these test cases for regression testing? Can you tell me about your regression testing process?
7. Can you tell me about the unit testing practices of the developers you work with?
8. Do you think unit test helps them write more solid code from the start? Do you feel a difference when you accept code with good unit tests into your process? Was it harder to find defects? [How did that make you feel?] Why or why not?
9. Is there any way to tell how many test cases your team ran in V1? In V2?
10. Do you think you found more or less bugs in V2 than V1? Do you think you found different types of defects in V2 than V1? How so?
11. Overall, do you think when the developers write unit tests the team ultimately produces a higher quality product? Why or why not?