

Test-Driven Development as a Defect-Reduction Practice

Laurie Williams¹, E. Michael Maximilien², Mladen Vouk¹

¹North Carolina State University, Department of Computer Science
{williams, vouk}@csc.ncsu.edu

²IBM Corporation and North Carolina State University
maxim@us.ibm.com

Abstract

Test-driven development is a software development practice that has been used sporadically for decades. With this practice, test cases (preferably automated) are incrementally written before production code is implemented. Test-driven development has recently re-emerged as a critical enabling practice of the Extreme Programming software development methodology. We ran a case study of this practice at IBM. In the process, a thorough suite of automated test cases was produced after UML design. In this case study, we found that the code developed using a test-driven development practice showed, during functional verification and regression tests, approximately 40% fewer defects than a baseline prior product developed in a more traditional fashion. The productivity of the team was not impacted by the additional focus on producing automated test cases. This test suite will aid in future enhancements and maintenance of this code. The case study and the results are discussed in detail.

1. Introduction

Test Driven Development (TDD) [4] is a software development practice that has been used sporadically for decades [13]; an early reference of its use is the NASA Project Mercury in the early 1960's [27]. The practice has gained added visibility recently as a critical enabling practice of Extreme Programming (XP) [1, 3, 25, 26]. With TDD, before implementing production code, the developer writes automated unit test cases for the new functionality they are about to implement. After writing test cases, the developers produce code to pass these test cases. The process is essentially "opportunistic" in nature [8]. A developer writes a few test cases, implements the

code, writes a few test cases, implements the code, and so on. The work is kept within the developer's intellectual bounds because he or she is continuously making small design and implementation decisions and increasing the functionality at a manageable rate. New functionality is not considered properly implemented unless these new (unit) test cases, and every other unit test case written for the code base, run properly.

In this paper, we examine the efficacy of the TDD practice as a means for reducing defects in a software-intensive system. We assessed the efficacy of the TDD technique with an IBM software development group. The group develops mission-critical software for its customers in a domain that demands high availability, correctness, and reliability. "Essential" money [7] and customer relations are at risk for IBM's customers if the software is not available, correct, and reliable. "Discretionary" money [7] and convenience are at risk for the recipients of the computer-dependant service provided by the IBM product. In our case study, we quantitatively examined the efficacy of the TDD as it relates to **defect density reduction** before a black-box functional verification test (FVT) run by an external testing group after completion of production code. We also comment on the use of the TDD practice in the context of **more robust design** and in the context of the role automated regression tests have in **smoother code integration**.

Section 2 provides background on the TDD unit testing techniques. Section 3 provides an overview of other TDD studies. Section 4 presents details of our case study. Section 5 presents the results of our analyses. Section 6 summarizes our findings. Finally, a detailed code example appears in the appendix.

2. Test-Driven Development

Software development processes and methods have been studied for decades. Despite that, we still do not have reliable tools for ensuring that complicated software systems intended for high-confidence tasks are free from faults and operational failures. Faults¹ may result from

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

¹ An error committed by a person becomes a fault in a software artifact such as specification, design, code, etc. This fault, unless caught, propagates as a defect in the

root causes that range from knowledge errors, to communication errors, to incomplete analysis errors, to transcription errors. Exacerbating the problem is the ever-growing expectations of the end-users and the growth in the complexity of the tasks. There are essentially three ways of dealing with faults:

1. **Fault-avoidance** is achieved through appropriate specification, design, implementation, and maintenance activities intended to avoid faults in the first place. This includes use of advanced software construction methods, formal methods and re-use of reliable self-describing software building blocks (objects), and active knowledge domain support.

2. **Fault-elimination** is the analytical compensation for errors committed during specification, design and implementation. It manifests as verification, validation and testing.

3. **Fault-tolerance** is the run-time compensation for any residual problems, out-of-specification changes in the operational environment [30], user errors, etc.

Absolute fault-avoidance may not be economical or feasible. The next best thing is to eliminate faults as soon as they occur, certainly before they propagate into the operational phase of the system, e.g., [5]. Given the error-proneness of humans, it is prudent to revisit software artifacts one or more times to ensure that our understanding of the issues, our designs, and our implementations are mutually conformant and correct. Process feed-back and feed-forward loops for problem detection and fault-elimination [5, 10] are beneficial. These loops may be over different releases of the product, over individual phases of a single release, and/or over individual tasks. Reliable implementation of very tight fault-elimination loops, especially those that are not just reactive (i.e., that result from a problem that needs to be corrected), but are also proactive (forward error correction – preventive activities and dynamic process improvement) are generally associated with high Capability Maturity Model (CMM) levels [31, 32]. Additionally, the earlier one finds an error, the less expensive it is to fix [5, 19, 33].

executable code. When a defective piece of code is executed, it puts the software/system into an error-state. Finally, this error-state can become a visible anomaly or failure when the program is executed. [22] IEEE, "IEEE Standard 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software," 1988, [23] IEEE, "IEEE Standard 982.2-1988, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," 1988, [24] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.

This section provides an overview of the test-driven development practice as a front-end, efficient fault detection and elimination technique due to its tight feedback loops. We also provide some information on TDD scripts.

2.1. Overview

Given functional requirements, TDD software engineers develop production code through rapid iterations of the following:

- ⊄ Writing a small number of automated unit test cases;
- ⊄ Running the new unit test cases to ensure they fail (since there is no code to run yet);
- ⊄ Implementing code which should allow the new unit test cases to pass;
- ⊄ Re-running the new unit test cases to ensure they now pass with the new code; and
- ⊄ Periodically (e.g., once a day provided the code base is small enough) re-running all the test cases in the code base to ensure the new code does not cause did not break any previously-running test cases. This, of course, is regression testing.

This highly-iterative cycle is used to develop a piece of new functionality (generally not more than one day's work). Since all of the test cases must successfully pass before new code is added to the code base, there can be some level of confidence that the new code did not introduce a fault, or mask a fault in the current code base. In the XP-implementation of the practice, TDD encompasses both design and unit testing. In XP, software engineers do not precede code implementation with any formal designs. In fact, some believe that strict adherence to TDD can help to minimize, if not eliminate, the need for upfront design [14]. However, the TDD practice is flexible and can be adapted to any process methodology, including those that specify low-level (detailed) upfront design phases. Many of the benefits that will be discussed in section 2.3 can be realized in essentially any development process by shifting from unit test development after implementing to unit test development before implementing with tight iterations [8, 20, 21, 31].

2.2. TDD Test Scripts

An important aspect of the TDD practice is the use of tools and a framework for the creation of the object-oriented unit tests. The defacto standard unit testing framework is the xUnit² framework, originally devised by Beck and Gamma. In its Java incarnation, or JUnit, unit test cases are added one per public class, usually with the

² <http://xprogramming.com/software.htm>

name `<ClassName>TestCase`. For each public method of the class, there is at least one corresponding `test<Method>` that tests the contract of that method. Multiple `test<Method>`s are added when testing different behavior of the method, for instance passing incorrect parameter arguments to test negative paths. Prior to the execution of each `test<Method>` a `setUp` method is executed in the `<ClassName>TestCase` class where the test fixtures are initialized, e.g. creating instances of `<ClassName>` which are used to execute the test methods on. At the end of the `test<Method>` execution, a `tearDown` method is executed on the `<ClassName>TestCase` class to reset the fixtures. `TestCase` classes are usually logically grouped into `TestSuite` classes, which allow execution of the `TestCase` classes in batches.

Example Java code with corresponding JUnit test scripts can be found in the Appendix.

2.3. Suggested Benefits of TDD

Some possible benefits of TDD are:

Efficiency. The fine granularity of test-then-code cycle gives continuous feedback to the developer. With TDD, faults and/or defects are identified very early and quickly as new code is added to the system, and the source of the problem is more easily determined. We contend that the efficiency of fault/defect removal and the corresponding reduction in the debug time compensates for the additional time spent writing and executing test cases. In net, TDD does not have a detrimental effect on the productivity of the software developer.

Test Assets. TDD entices programmers to write code that is automatically testable, such as having functions/methods returning a value which can be checked against expected results. Benefits of automated testing, such as TDD testing, include: (1) production of a more reliable system, (2) improvement of the quality of the testing effort, (3) reduction of the testing effort, and (4) minimization of the schedule [9]. The automated unit test cases written with TDD are valuable assets to the project. Subsequently, when the code is enhanced or maintained, running the automated unit tests may be used for the identification of newly introduced defects, i.e., for regression testing.

Reducing Defect Injection. Debugging and software maintenance is often viewed as a low-cost activity in which working code defect is “patched” to alter its properties, and specifications and designs are neither examined nor updated [17]. Unfortunately, such fixes and “small” code changes may be nearly 40 times more error prone than new development [18], and often new faults are injected during the debugging and maintenance. The TDD test cases are a high-granularity low-level regression

test. By continuously running these automated test cases, one can find out whether a change breaks the existing system. The ease of running the automated test cases after changes are made should allow smooth integration of new functionality into the code base and reduce the likelihood that fixes and maintenance introduce new permanent defects.

3. Related Work

Recently, researchers have started to conduct studies on the effectiveness of the TDD practice. Muller and Hagner conducted a structured experiment comparing TDD with traditional programming [29]. The experiment, conducted with 19 graduate students, measured the effectiveness of TDD in terms of (1) development time, (2) reliability, and (3) understandability. The researcher divided the experimental subjects into two groups, TDD and control; each group solved the same task. The task was to complete a program in which the specification was given along with the necessary design and method declarations; the students completed the body of the necessary methods. The researchers set up the programming in this manner to facilitate automated acceptance testing and reliability analysis.

The TDD students wrote their test cases while the code was written, as described above; the control group students wrote automated test cases after completing the code. The experiment occurred in four phases: (1) an initial implementation phase (IP); (2) an evaluation phase in which the researchers ran tests and provided feedback to the students; (3) an acceptance test phase (AP) during which the students fixed their identified IP defects; and (4) post-experiment analysis. The researchers found no difference between the groups in overall development time. The TDD group had lower reliability after IP and higher reliability after AP. Based on these results the researchers concluded that writing programs in test-first manner neither leads to quicker development nor provides an increase in quality.

Another set of experiments were run with 24 professional programmers at three industrial locations [14, 15]. One group developed code using the TDD practice while the other a waterfall-like approach. All programmers practiced pair programming [34], whereby two programmers worked at one computer, collaborating on the same algorithm, code or test. The experiment participants were provided the requirements for a short program to automate the scoring of a bowling game [28]. When comparing the groups, the researchers found the following:

TDD teams passed 18% more functional black box test cases when compared with the control group teams.

The experiment results showed that TDD developers took more time (16%) than 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.

Qualitatively, this research also found the TDD approach facilitates simpler design, and the lack of upfront design is not a hindrance. However, transitioning to the TDD mindset is difficult for some.

The results of both of these studies need to be viewed within the limitations of the experiments conducted. In both cases, the sample size was small, the students in the first study and several of the professional programmers had limited experience with TDD, and the results were blurred by large variance. Further controlled studies on a larger scale in industry and academia could strengthen or disprove these findings.

4. Case Study

Experimentation in software engineering can be difficult. 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” [11]. These experiments may suffer from the external validity limitations (or perceptions of such). On the other hand, case studies can be viewed as “research in the typical” [11]. 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 [6]. There is also an apprehension with case studies of the ability to make a valid comparison between the baseline and the new treatment, since the same project is generally not replicated. Finally, case studies often cannot yield statistically significant results due to a small sample size. 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 experiments [2] which examine the efficacy of a new practice. We add to the knowledge about the TDD practice by performing a case study. We studied efficacy of TDD as a defect-reduction practice within an IBM development group.

This IBM group has been developing device drivers for over a decade. They have one legacy product which has undergone seven releases since late 1998. This legacy product was used as the baseline in our case study. In 2002, the group developed device drivers on a new platform. In our case study, we compare the seventh release on the legacy platform with the first release on the new platform. Because of its longevity, the legacy system handles more classes of devices on more platforms with more vendors than the new system. Hence, while not a

true control group, the legacy software still can provide a valuable relative insight into the performance of the TDD methodology.

4.1. Context

All participating IBM software engineers on both projects had a minimum of a bachelor’s degree in computer science, electrical or computer engineering. Two had master’s degrees. The seventh release legacy team consisted of five collocated full-time employees with significant experience in the programming language of choice (Java and C++) and the domain. The new product team was made up of nine full-time engineers, five in a US location and four in Mexico. Additionally, part-time resources for project management and for system performance analysis were allocated to the team. No one on the new team knew TDD beforehand, and three were somewhat unfamiliar with Java. All but two of the nine full-time developers were novices to the targeted devices. The domain knowledge of the developers had to be built during the design and development phases. A comparison of the two teams is summarized in Table 1. In general, we think that the differences between the teams present challenges that make the results we observed even more interesting and difficult to achieve.

Table 1: Team and Product Comparison

	Legacy 7 th Iteration	New 1 st Release
Team Size (Developers)	5	9
Team Experience (Language and Domain)	Experienced	Some Inexperienced
Collocation	Collocated	Distributed
Code Size (KLOC) New; Base; Total	6.6; 49.9; 56.5	64.6; 9.0; 73.6
Language	Java/C++	Java
Unit Testing	Ad hoc	TDD
Technical Leadership	Shared resource	Dedicated coach

4.2. Unit Testing Practices

The unit testing approach of the legacy group can be classified as ad-hoc. A developer would code a prototype of the important classes and then create a design via UML class and sequence diagrams [12]. We define *important classes* to be utility classes, classes which collaborate with other classes, and classes that are expected to be reused. This design was then followed by an implementation stage that sometimes caused design

changes, and thus some iteration between the design and the coding phases. Unit testing then followed as a post-coding activity. One of the following unit test approaches was usually chosen:

a) After enough coding was done, an interactive tool was created by the developer that permitted the execution of the important classes.

b) Unit testing was executed using an interactive scripting language or tool, such as `jython`³, which allows manual interactive exercising of the classes by creating objects and calling their methods.

c) Unit testing was done by the creation of independent ad-hoc driver classes that tested specific important classes or portions of the system which have clear external interfaces.

In all cases, the unit test process was not too formal and disciplined. More often than not, there were resource and schedule limitations that constrained the number of test cases developed and run. Most of the unit tests developed were also not reused during the subsequent Functional Verification Test (FVT) phase, when a defect was found, or when a new release of the software was developed.

With TDD, test cases were developed up front as a means of reducing ambiguity and to validate the requirements, which for this team was a full detail standard specification. We found that such up-front testing drives a good understanding of the requirements. It is important to note that in XP projects, up-front testing proceeds without any such “big design up front,” commonly referred to as BDUF [3]. However in our system, the requirements were stable, and we chose to do up-front design via UML class and sequence diagrams. This may be a significant factor as to why TDD appears to have performed exceptionally well in this case study. This design activity was interspersed with the up-front unit test creation. After creating a “spike” [3] of the system by implementing an end-to-end service for one device, each logical portion of the system was layered and completely designed using UML class and sequence diagrams.

For each important class, we enforced complete unit testing. We define *important classes* to be utility classes, classes which collaborate with other classes, and classes that are expected to be reused. We define *complete testing* as ensuring that the public interface and semantics (the behavior of the method as defined in the specification) of each method were tested utilizing the JUnit⁴ unit testing framework. Each design document included a unit testing section that listed all important classes and public methods that would be tested. For each public class, we had an associated public test class; for

each public method in the class we had an associated public test method in the corresponding unit test class. (See example in the Appendix.) Our goal was to cover 80 percent of the important classes by automated unit testing. Some unit tests also contained methods that tested particular variations of the behavior, e.g., the printer device has an asynchronous printing capability and the regular print methods behaved differently in synchronous and asynchronous modes.

To guarantee that all unit tests would be run by all members of the team, we decided to set up an automated build and test systems in both “new project” geographical locations. Daily, these systems would extract all the code from the library build and run all the unit tests. The Apache ANT⁵ build tool was used. After each automated build/test run cycle, an email was sent to all members of the teams listing all the tests that successfully ran and any errors found. This automated build and test served us as a daily integration and validation for the team. At first this build test was run multiple times a day in both locations. Eventually, we decided to alternate the build between locations and to only run the build tests once a day.

With both the legacy and new projects, when the majority of the device driver code was implemented and passed all their unit tests and those in the code base, the device drivers were sent to FVT. The external FVT team (different from both development legacy and “new project” teams) had written black box test cases based on the functional system specification. More than half of the FVT tests were automated in part (requiring human intervention to declare pass/fail); the remaining tests were split fairly evenly between fully automated and fully manual.

Defects identified from running these test cases were communicated to the code developers via a defect tracking system. The defects were then categorized by device. As illustrated in Figure 1, defects were assigned a severity code based on the nature of the defect and on how many other FVT tests the failure blocked. Once 100% FVT tests have been attempted, all test cases are re-run by the FVT team in a regression test. (This does not imply that the defects from these attempted tests are all resolved.) Figure 2 summarizes the development and test process used by the “new project” team.

5. Results

Every project in this IBM division, collects a variety of metrics. Among other items, the process calls for tracking of the testing progress and for predicting test characteristics. For example, approximately quarter of the way into the project (but before substantial amount of code had been developed), the team predicted the

³ <http://www.jython.org>

⁴ <http://junit.org>

⁵ <http://www.apache.org/jakarta/ant>

productivity, the number of new and changed lines of code in the project, and the number of total defects that will be found during the FVT. Historical, organization-specific models were used to do that.

One of the most interesting findings was that the *defect rate (defects or faults per KLOC)*, i.e., observed fault

density, of the code entering FVT/regression test appeared to be significantly better for the “new” system when compared with the legacy system. This observation is summarized in Figure 3. The new product appears to exhibit approximately a 40% lower defect density.

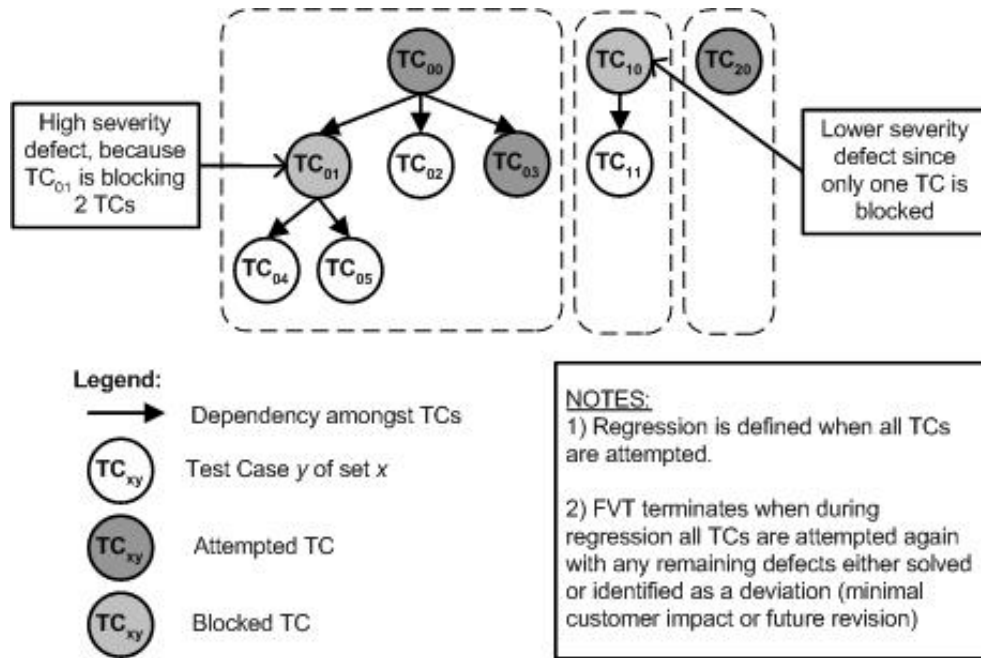


Figure 1: Illustration of test case blocking severity

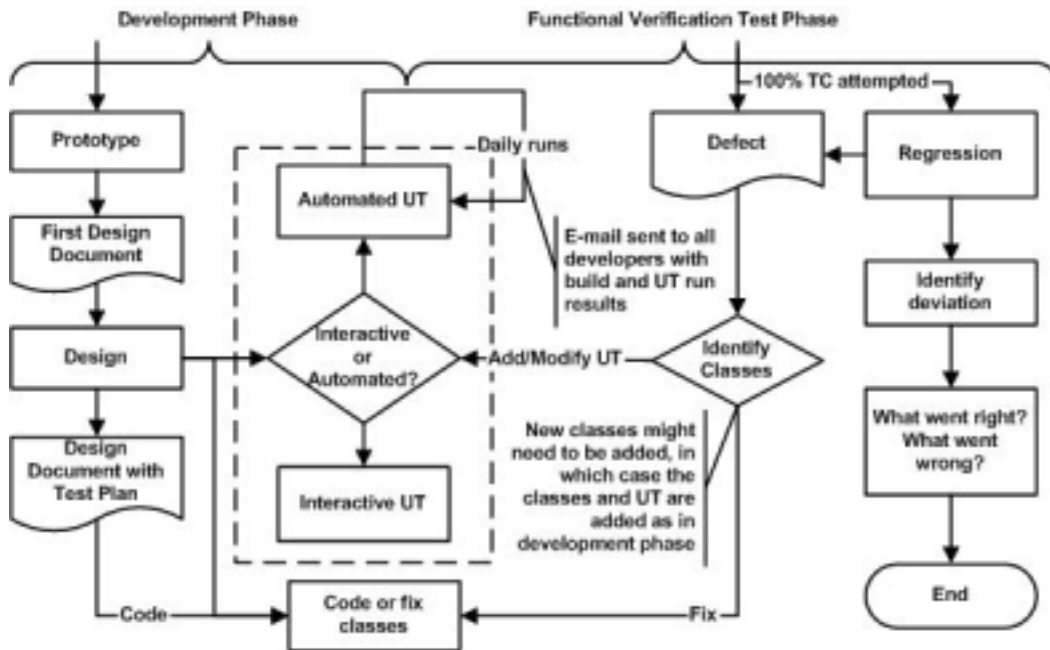


Figure 2: Summary of development and test process (UT = Unit Test)

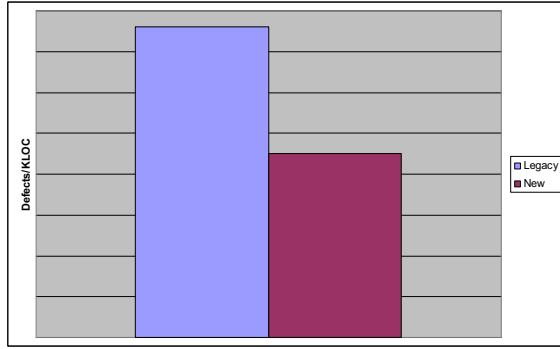


Figure 3: FVT/Regression Defect Density

The severity distribution of the defects (faults) was essentially equal in the two cases. This is shown in Table 2. Severity 1 defects are the most critical; Severity 4 defects are non-critical.

Table 2: Defect Severity Distribution

	Legacy 7 th Iteration	New 1 st Iteration
Severity 1	3%	2%
Severity 2	25%	25%
Severity 3	70%	65%
Severity 4	2%	8%

To help understand the defect or fault density differences, we turn to some testing issues – specifically to the number of test-cases run. One thing to note is that if a device was supported by both the legacy and the “new product” code, the FVT test cases for that device were identical. A “substitutability” requirement for the new system was to “pass all the legacy system FVT tests”. An identical FVT/regression exit criterion was used for both projects. This criterion identifies the percentage of FVT and regression test cases that must be attempted/passed and the percentage of defects that may remain unresolved based on the severity level.

However, it must be noted that in absolute terms, the legacy product was tested using about twice as many test-runs when compared with the “new product”. The reason is device diversity. Specifically:

- ∄ The devices on the legacy product had to run on two platforms (Windows and Linux). The “new system” devices needed to work only on Linux. (*numberOfOS*)
- ∄ The legacy product worked on more hardware platforms than the “new product”. Test cases needed to be re-run for each platform. (*numberOfSystemFamily*)
- ∄ For each class of device (e.g. the printer class of device), the legacy product supported more

brands/models of devices. As a result the same set of tests was often run multiple times on various but perhaps similar devices. (*deviceClass*, *numberModels*, *TCforDevice*)

Also for each class of device there is a percentage of the test cases that are only ran once because they were common for all devices. Hence, the number of test cases needed for a class of device could be reduced by this factor. (*commonTCFactor* is used to account for this effect)

The total number of test cases (TC) run on each product may be approximated by the following formula:

$$TC = \frac{(\text{numberModels} * TCforDevice * \text{commonTCFactor}) * \text{numberOfOS}}{\text{numberOfSystemFamily}}$$

Table 3 illustrates the results. Because the “new product” was less expansive, only half of the effort was needed for FVT, and only about half as many test-cases were run. Yet, the testing uncovered about twice as many defects per test case. Was the “new code” more defective or were the TDD based test-cases more efficient?

For the legacy system, both the factor *numberOfOS* and the factor *numberOfSystemFamily* were 2 or more. In the new product both of these factors were 1. Also more brands/models were supported by the legacy. This means that significantly more test cases needed to be run on the legacy code (requiring more FVT effort) than on the “new project” code to meet the same FVT/regression criteria. When test cases are repeated for multiple hardware/software platforms, these test cases often execute the same lines of code. This drives up the ratio of test cases per LOC for the legacy product. Similarly, these multiple executions drive down the ratio of defects to number of test cases. Since most of the test cases ran without incident. Re-running of these incident-free test cases on multiple platforms decreases the legacy Test Cases per LOC ratio.

Table 3: Legacy vs. New Project Comparison

	Legacy 7 th Iteration	New 1 st Iteration
FVT Effort	E	0.49E
Test Cases Run	TC	0.48TC
Test Cases/Total LOC	TCL	0.36TCL
Defects/Test Case	DTC	1.8DTC
Defects/LOC	DFL	0.61DFL

While the developers spend more time writing the test cases, TDD reduces the time they spend in debugging. The time previously spent on relatively unpredictable debugging was traded off for controlled test creation and

execution. Other studies have found a slight decrease in developer productivity when employing the TDD practice [15, 16, 29]. In our case, the productivity was roughly the same in both cases.

While legacy product unit testing was an ad-hoc process, primarily manually, the new project benefited from automation of unit test-cases since they could be used as part of the FVT regression test-suite. For the new project, we entered test with 64.6 KLOC of new code and 34 KLOC of JUnit code. This is slightly more than 0.5 lines of test code for every line of implementation code. Anecdotally, TDD results in at least this much test code, often as much as 1:1 test/implementation code ratio. Approximately 2390 automated unit test cases were written. Additionally, over 100 automated JUnit performance test cases were written. We also wrote approximately 400 interactive tests. Eighty-six percent of the tests were automated, exceeding our 80% target. The interactive tests were rarely run; the automated tests ran daily.

When FVT or regression testing revealed a defect, developers augmented the TDD test suite by adding new test to reveal the presence of the defect found. Additionally running the extensive automated test suite after defect fixes were implemented gave added confidence that the fix did not introduce new defects.

We believe that the TDD practice aided us in producing a product that more easily incorporated later changes. A couple of devices (albeit, not overly complex devices) were added to the product about two-thirds of the way through the schedule without major perturbations.

In the past, we only integrated code into the “legacy” product once we were close to FVT. In the TDD approach, the daily integration certainly saved us from late integration problems. The success (or failure) of the daily integration served as the heartbeat of the project and minimized risk because problems surfaced much earlier. In addition, the developers are very positive about the TDD practice and have continued its use.

6. Summary and Future Work

A development team in IBM transitioned from an ad-hoc to a TDD unit testing practice. The IBM team utilized the TDD practice after a thorough UML design process. Through the introduction of this practice a relatively inexperienced team realized about 40% reduction in FVT detected defect density of new/changed code when compared with an experienced team who used an ad-hoc testing approach for a similar product. They achieved this result with minimal impact to developer productivity. Additionally, the suite of automated unit test cases created via TDD became a reusable and extendable asset that will continue to improve quality over the lifetime of the software system. The test suite will also be the basis for

quality checks and will serve as a quality contract between all members of the team.

Through the TDD practice, a significant suite of regression test cases are created and the code is developed in a “testable” manner. In our current research, we are exploiting these characteristics of TDD-developed projects to enable an extension of XP to encompass a measure of reliability. We are enhancing the TDD practice to include explicit estimation of the probability that the software system performs according to its requirements based on a specified usage profile. [35]

Acknowledgements

We would like to thank the Raleigh and Guadalajara teams for an outstanding job executing this new process and for their trust that this new process of up-front unit testing would pay off. The results and actual execution of the ideas came from their hard work. We especially acknowledge Dale Heeks from the FVT team. We also want to thank the IBM management teams for their willingness to try a new approach to development without prior data on whether this would be effective or not. Finally, we would like to thank the NCSU software engineering reading group for their helpful suggestions.

This work is supported in part by NSF Award 9901004 and the NC State Center for Advanced Computing and Communication.

References

- [1] K. Auer and R. Miller, *XP Applied*. Reading, Massachusetts: Addison Wesley, 2001.
- [2] V. R. Basili, F. Shull, and F. Lanubile, "Building Knowledge Through Families of Experiments," *IEEE Transactions on Software Engineering*, vol. 25, pp. 456 - 473, 1999.
- [3] K. Beck, *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.
- [4] K. Beck, *Test Driven Development: By Example*: Addison Wesley, 2002.
- [5] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.
- [6] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Design for Research*. Boston: Houghton Mifflin Co., 1963.
- [7] A. Cockburn, *Agile Software Development*. Reading, Massachusetts: Addison Wesley Longman, 2001.
- [8] B. Curtis, "Three Problems Overcome with Behavioral Models of the Software Development Process (Panel)," presented at International Conference on Software Engineering, Pittsburgh, PA, 1989.

- [9] E. Dustin, J. Rashka, and J. Paul, *Automated Software Testing*. Reading, Massachusetts: Addison Wesley, 1999.
- [10] S. E. Elmaghraby, E. I. Baxter, and M. A. Vouk, "An Approach to the Modeling and Analysis of Software Production Processes," *Intl. Trans. Operational Res.*, vol. 2, pp. 117-135, 1995.
- [11] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Brooks/Cole Pub Co., 1998.
- [12] M. Fowler, *UML Distilled*. Reading, Massachusetts: Addison Wesley, 2000.
- [13] D. Gelperin and W. Hetzel, "Software Quality Engineering," presented at Fourth International Conference on Software Testing, Washington D.C., June 1987.
- [14] B. George, "Analysis and Quantification of Test Driven Development Approach MS Thesis," in *Computer Science*. Raleigh, NC: North Carolina State University, 2002.
- [15] B. George and L. Williams, "An Initial Investigation of Test-Driven Development in Industry," presented at ACM Symposium on Applied Computing, Melbourne, FL, 2003.
- [16] B. George and L. Williams, "A Structured Experiment of Test-Driven Development," *Information and Software Technology (IST)*, to appear 2003.
- [17] D. Hamlet and J. Maybee, *The Engineering of Software*. Boston: Addison Wesley, 2001.
- [18] W. S. Humphrey, *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley, 1989.
- [19] W. S. Humphrey, *A Discipline for Software Engineering*. Reading, Massachusetts: Addison Wesley Longman, Inc, 1995.
- [20] IEEE, "ANSI/IEEE Standard 1008-1987, IEEE Standard for Software Unit Testing," 1986.
- [21] IEEE, "IEEE Std. 1012-1986, IEEE Standard Software Verification and Validation Plans," 1986.
- [22] IEEE, "IEEE Standard 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software," 1988.
- [23] IEEE, "IEEE Standard 982.2-1988, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," 1988.
- [24] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.
- [25] R. Jeffries, A. Anderson, and C. Hendrickson, *Extreme Programming Installed*. Upper Saddle River, NJ: Addison Wesley, 2001.
- [26] R. E. Jeffries, "Extreme Testing," in *Software Testing and Quality Engineering*, vol. 1, 1999, pp. 22-27.
- [27] C. Larman and V. Basili, "A History of Iterative and Incremental Development," *IEEE Computer*, to appear June 2003.
- [28] R. C. Martin and R. S. Koss, "Engineer Notebook: An Extreme Programming Episode," http://www.objectmentor.com/resources/articles/xpepis_ode.htm, 2001.
- [29] M. M. Müller and O. Hagner, "Experiment about Test-first Programming," presented at Conference on Empirical Assessment in Software Engineering (EASE), 2002.
- [30] J. D. Musa, *Software Reliability Engineering*. New York: McGraw-Hill, 1998.
- [31] M. C. Paulk, B. Curtis, and M. B. Chrisis, "Capability Maturity Model for Software Version 1.1," Software Engineering Institute CMU/SEI-93-TR, February 24, 1993 1993.
- [32] T. Potok and M. Vouk, "The Effects of the Business Model on the Object-Oriented Software Development Productivity," *IBM Systems Journal*, vol. 36, pp. 140-161, 1997.
- [33] I. Sommerville, *Software Engineering*, Sixth ed. Harlow, England: Addison-Wesley, 2001.
- [34] L. Williams and R. Kessler, *Pair Programming Illuminated*. Reading, Massachusetts: Addison Wesley, 2003.
- [35] L. Williams, L. Wang, and M. Vouk, "'Good Enough' Reliability for Extreme Programming," presented at Fast Abstract, International Symposium on Software Reliability Engineering, Annapolis, MD, 2002.

Appendix A: Test-Driven Development/JUnit Example

The vast majority of software engineers that practice TDD utilize a test framework from the open source xUnit family. See <http://www.xprogramming.com/software.htm> where over 60 versions of the testing framework are available for Java, C++, Ruby, Scheme, XML, and many other languages. The testing framework was originally authored by Kent Beck and Erich Gamma in Java; the Java version has won the 2002 JavaOne “Best Java Performance/Testing Tool” award. Software engineers around the world have ported this framework to the other languages. Much of the success of the tools can be attributed to its ease of use and short learning curve. To begin using the tool, a developer puts the downloaded code for the testing classes/methods into his or her hierarchy and CLASSPATH. Test case creation proceeds by writing test methods that inherit from the testing classes.

In this section, we present some example Java/JUnit TDD code scripts. The class diagram for the sample application is shown below in Figure 4. In parallel to this implementation code hierarchy, corresponding test classes were created. These three test classes are called `DeviceFactoryTestCase`, `DeviceEventTestCase`, and `LightBulbDeviceTestCase`. Each of these classes have `setUp` and `tearDown` methods. The `setUp` method runs before each test method; the `tearDown` method runs after each method. For example, for the `DeviceEventTestCase` class, these methods are as follows:

```
protected void setUp()
{
    deviceEvent = new DeviceEvent( this,
        DEFAULT_TYPE);
}

protected void tearDown()
{ deviceEvent = null; }
```

The `setUp` method creates and initializes an instance of the `DeviceEvent` class. The `tearDown` method clears the instance.

These same methods are more involved with the `LightBulbDeviceTestCase` class:

```
protected void setUp() throws
                    DeviceException
{
    deviceFactory = new DeviceFactory();
```

```
    lbd = deviceFactory.
        createLightBulbDevice();
    dl = this.new TestDL();
}

protected void tearDown() throws
DeviceException
{
    deviceFactory = null;

    try
    {
        lbd.close();
        lbd = null;
    }
    catch( DeviceException de )
    {
        //Don't care since this is thrown
        //if device was not opened
    }

    dl = null;
    deviceEvent = null;
}
```

Ideally, each method in the production code has at least one corresponding test method. This correspondence is illustrated with an in Table 4. In the left hand column of the table is a production code method. A test method written for this method is in the right hand column. (Notice that often the test code is longer than the production code.) The test cases are structured such that several types of assert statements (for Java: `assertEquals`, `assertTrue`, `assertNull`, `assertNotNull`) are used to compare actual results to expected results. If the assert returns false, the test case fails. If the tested method throws exceptions then the test method declares these exceptions in its signature. Since test methods will generally verify the positive path for the method then any thrown exception will fail the test. Negative path test methods will not declare exceptions in their signatures since for such test methods; exception will be expected and thus caught.

Also, the JUnit `setUp` and `tearDown` methods can also declare that they throw some exception, especially when the test fixture could potentially cause an exception. Any such occurrence will result in the failure of the test for which the `setUp` or `tearDown` was executing.

The tool reports the % of test cases that pass and details on the failing test cases. Many version of the tool have a GUI interface. In the GUI interface, a bold green bar across the interface indicates 100% test cases passed. A red bar is displayed when even a single test case fails.

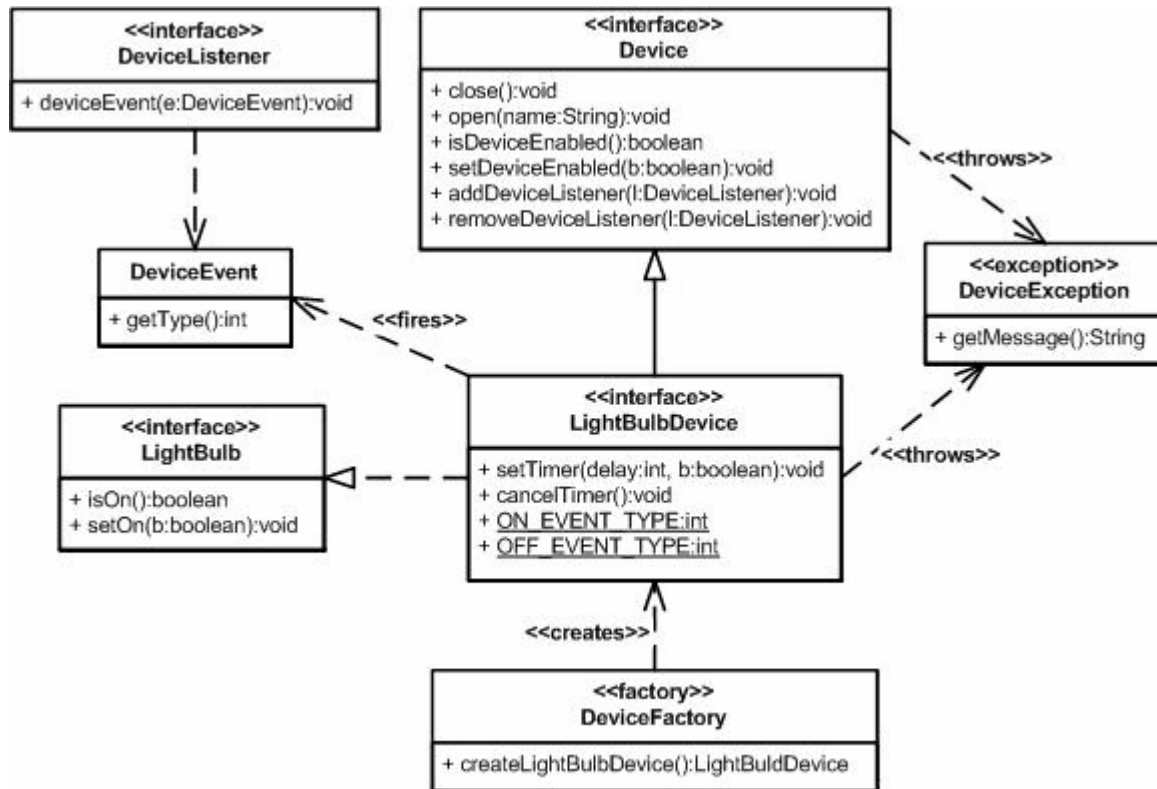


Figure 4: Class Diagram for Light Bulb Program

Implementation Method	Test Method
Class: DeviceEvent <pre>public int getType() { return type; }</pre>	Class: DeviceEventTestCase <pre>public void testGetType() { deviceEvent = new DeviceEvent(this, 1); assertTrue(deviceEvent.getType() == 1); assertTrue(deviceEvent.getSource() == this); }</pre>
Class: DeviceFactory <pre>public LightBulbDevice createLightBulbDevice() { return new DefaultLightBulbDevice(); }</pre>	Class: DeviceFactoryTestCase <pre>public void testCreateLightBulbDevice() { LightBulbDevice lb0 = deviceFactory.createLightBulbDevice(); assertTrue(lb0 != null); LightBulbDevice lb1 = deviceFactory.createLightBulbDevice(); assertTrue(lb1 != null); assertTrue(lb0 != lb1); }</pre>

Implementation Method	Test Method
<pre> Class: DeviceFactory public void open(String name) throws DeviceException { if(name == null) throw new DeviceException("Invalid argument to open method"); if(opened) throw new DeviceException("Device already opened"); openImp(name); opened = true; } </pre>	<pre> Class: LightBulbDeviceTestCase public void testOpen_null() throws DeviceException { try { lbd.open(null); fail("Expected failure since name is null"); } catch(DeviceException de) {} } </pre>
<pre> Class: DeviceFactory Same method as above </pre>	<pre> Class: LightBulbDeviceTestCase public void testOpen_Multiple() throws DeviceException { lbd.open("Light"); try { lbd.open("Light2"); fail("Should not be able to open device twice"); } catch(DeviceException de) {} } </pre>
<pre> Class: DeviceFactory Same method as above plus: public void setDeviceEnabled(boolean b) throws DeviceException { if(!opened) throw new DeviceException("Cannot enable or disable a closed device"); if(b) enableImp(); else disableImp(); enabled = b; } public void setOn(boolean b) throws DeviceException { checkOpenedEnabled(); lightBulb.setOn(b); } </pre>	<pre> Class: LightBulbDeviceTestCase public void testIsOn() throws DeviceException { lbd.open("Light"); lbd.setDeviceEnabled(true); if(lbd.isOn()) { lbd.setOn(false); assertTrue(lbd.isOn() == false); } else { lbd.setOn(true); assertTrue(lbd.isOn()); } assertTrue(lbd.isOn() == lbd.isOn()); } </pre>

Table 4: Code Example