

On the Sustained Use of a Test-Driven Development Practice at IBM

Julio Cesar Sanchez¹, Laurie Williams², and E. Michael Maximilien¹

¹IBM Corporation

{juliosan, maxim}@{mx1, us}.ibm.com

²Department of Computer Science, North Carolina State University
williams@ncsu.edu

Abstract

Test-Driven Development (TDD) is an agile practice that is widely accepted and advocated by most agile methods and methodologists. In this paper, we report on a post hoc analysis of the results of an IBM team who has sustained use of TDD for 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. The IBM team realized sustained quality improvement relative to a pre-TDD project and consistently had defect density below industry standards. As a result, our data indicate that the TDD practice can aid in the production of high quality products. This quality improvement would compensate for the moderate perceived productivity losses. Additionally, our data indicates that the use of TDD may decrease the degree to which code complexity increases as software ages, as measured by cyclomatic complexity metric.

1. Introduction

Both the initial adoption and the “staying power” of a new technology adoption (in terms of new processes and new hardware/software) in an organization can be tenuous. The presence of a champion¹ of the new technology can aid in promoting and bringing to the forefront the relative advantage and the results of the technology transfer [22]. Some other factors that can affect the ability of an organization to effectively assimilate a new technology include, the culture of the team when technology is introduced and cause changes, as well as the perception by the individuals of the usefulness, ease of use, and the relative advantage over the old technology [23].

The staying power of test-driven development (TDD) [4] is evidenced by our five years of sustained

use of TDD by the Point of Sale (POS) device driver development team at IBM. With TDD, as defined by Beck [4], a software engineer cycles minute-by-minute between writing unit tests and writing code. Practiced in that fashion TDD is considered as much (or more) of a design process as a testing process, no formal design precedes these cycles. Before checking in code and its associated tests to a code base, software engineers run all the unit tests in the code base. While the TDD practice surfaced contemporarily as part of XP, the practice has been used for decades [10, 17] and is often used by non-XP teams (for example at Microsoft [5]) as part of the team’s software development process.

In this paper, we report on a post hoc analysis of the IBM team. This team has been using a TDD practice since 2001 and has produced ten releases of one small-to-medium-scale software project written in Java™. The team does create an initial UML-based design for portions of the system, and generally writes unit tests incrementally during (not before) development, as will be discussed. Specifically, we will examine the following:

- How does the TDD practice aid in the production of a high quality product?
- How much testing is necessary to realize the benefits of TDD?
- How does the use of TDD impact productivity?
- How does TDD impact the inevitable complexity increase of code as it ages?

Our research methodology includes post hoc data analysis; a survey of developers and testers; and action research on the part of the first and third authors.

The rest of the paper is organized as follows: Section 2 provides an overview of prior research on TDD. Section 3 provides the detail of our longitudinal case study, and Section 4 presents our results. Finally, Sections 5 and 6 distill some lessons learned through almost five years of TDD use and conclude this paper.

¹ A champion is a respected person who supports the introduction of the new technology and is willing to remove obstacles to its adoption

2. Prior Research on TDD

There exists various empirical studies on the effectiveness of TDD conducted in both industrial and in academic settings. In this section, we summarize the results of several of these studies. The results presented in this paper are based upon a case study that has been conducted over a much longer period than any of the results previously reported. As far as we know, we believe that this longitudinal study of TDD is a first of its kind in the literature.

2.1 Industrial Studies

A set of experiments were run with 24 professional programmers at three industrial locations, John Deere, Rolemodel Software, and Ericsson [11, 12]. One group developed code using the TDD practice while the other followed a waterfall-like approach. All programmers practiced pair programming [24], 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 [18]. 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 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.

Case studies were conducted of two development teams at Microsoft (Windows with C++ code and MSN with C++ and C#) that used the TDD practice [5]. 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.

Table 1: Microsoft TDD case studies

	Windows	MSN
Test LOC ² /Source LOC	0.66	0.89
% block coverage	79%	88%
Development time (person months)	24	46
Team size	2	12
Relative to pre-TDD:		
Decrease in Defects/LOC	38%	24%
Increase in development time	25-35%	15%

² LOC = lines of code

A controlled experiment was conducted with 14 voluntary industrial participants [13] 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 [7]. 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 quality could offset this initial increase in development time. Additionally TDD leads developers to design more precise and accurate test cases.

2.2 Academic Studies

Müller and Hagner conducted a controlled experiment comparing TDD with traditional programming [21]. The experiment, conducted with 19 graduate students, measured the effectiveness of TDD in terms of development time and reliability. The researchers 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 researchers concluded that writing programs in test-first manner neither leads to quicker development nor provides an increase in quality.

Janzen and Saiedian conducted a controlled experiment with ten students in an undergraduate software engineering class to examine the effects of TDD on internal software design quality [15]. Students designed and built an HTML pretty-print system that took between 74 and 190 person-hours. One group used the TDD practice, one group used a test-last practice, and a third group did not write any automated tests. The results indicate that TDD can be an effective design approach, improving object decomposition, test coverage, external quality, productivity, and confidence.

Erdogmus et al. conducted a controlled experiment of third-year undergraduate students who were taking an eight-week intensive Java course at Politecnico di Torino in Italy. Twenty-four students implemented a

bowling game [18] in Java. Eleven completed the task using a test-first practice and 13 used a test-last practice. Students were encouraged to complete all work in the same programming laboratory. The test-first students on average wrote more tests and, in turn, students who wrote more tests tended to be more productive.

3. TDD Use by IBM Team

In this section, we present an overview of the IBM case study. First, we will present contextual information about the product and the releases under study, the team, and the TDD practice used. We complete this section with our research methodology as well as listing some limitations to our empirical approach.

3.1 The Project

The project is the development of IBM's JavaPOS³-compliant device drivers. The project consists of the creation of middleware for devices in the POS domain. The POS devices are varied and include most devices necessary for checking out a customer at a typical store. Examples of such devices are:

- cash drawer (or till) that allows the store to keep cash and programmatically open the drawer;
- scanner to quickly find the unique identifier (or SKU) of an item and look-up its price, rebates, description, and other information;
- Magnetic Swipe Reader (MSR) to collect payments in the form of credit and debit cards;
- specialized keyboard with keys for various POS functions in a store, e.g., calculate total; and
- specialized printers to issue receipts of transactions formatted with store logo and information designed to take into account local laws and policies.

In total the JavaPOS technical standard specification⁴ amounts to 24 different POS devices each with an average of 52 properties, 24 methods, and five events. IBM JavaPOS implementation covers 14 devices.

The purpose of the IBM team is to implement the JavaPOS specification for IBM's POS hardware devices and platforms (and from now on noted as IBM's POS drivers or IBM's drivers). In the POS domain,

any failure of a device at a retail check out lane can potentially impact the retailer's bottom line. IBM's customers, therefore, have "essential money" at risk if the JavaPOS device software middleware exhibits failures when deployed. As such, the IBM POS device driver team must use procedures to ensure this a high level of quality.

Since the first release in 1998, the JavaPOS specification has undergone ten major versions. The IBM team has kept the release of the IBM POS drivers in sync with the JavaPOS versions by producing multiple releases over the years since the first one in 2001. In this paper, we analyze data from all of these releases.

In Release 1 of the IBM JavaPOS POS drivers, the initial framework and main devices of the specification were implemented following the TDD practice. The subsequent releases typically added new devices but also implemented new functionality introduced in the specification. Every release involved some changes in the developers, project management, and test teams. This paper focuses on ten releases of the product: Release 1 in the fourth quarter of 2002 through Release 10 in the fourth quarter of 2006. Development for Release 1 began one year prior. We have documented the details of this project and the results of the first release in [19, 25].

3.2 Team

The development team initially consisted of nine full-time engineers, five in Raleigh, NC, USA and four in Guadalajara, Mexico. Eventually, engineers in Martinez, Argentina joined the team. The team size grew to as many as 17 for Release 7, as shown in Figure 1. All team members had a minimum of a bachelor's degree in computer science, electrical, or computer engineering. Some had Master's degrees. Additionally, part-time resources for architect, project management and for system performance analysis were allocated. In the first couple releases of the IBM drivers, no one on the team had experience with TDD before the first release, and three were unfamiliar with Java. The domain knowledge of the developers had to be built during the design and development phases.

Figure 2 shows the quantity of code added in each release. The product increased from approximately 41 thousand lines of code (KLOC) in Release 1 to approximately 114 KLOC in Release 10.

³ <http://www.javapos.org>; standard developed jointly by leading retailers, hardware and software companies

⁴ defined in in UnifiedPOS standard v1.9.
<http://www.nrf-arts.org/UnifiedPOS/>

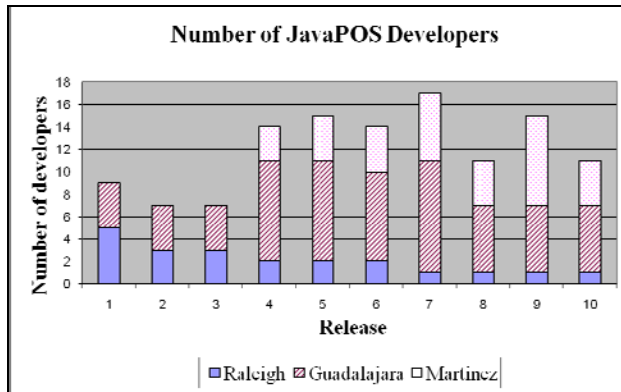


Figure 1: Number of JavaPOS developers

In Release 7, a major effort was placed on refactoring test code, hence the drop in the number of lines of test code. The refactoring was focused on the redundancy of the test cases in the POSPrinter driver. Since IBM supports three different types of printers, often there were three sets of very similar test cases. The refactoring removed the redundancy and facilitated the integration and execution of new tests.

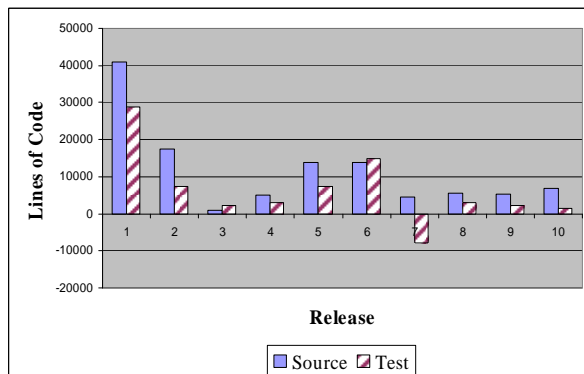


Figure 2: Newly added lines of code

3.3 Testing Practices

In this section, we describe our unit and functional verification testing practices.

3.3.1 Unit testing practices. With TDD, test cases were developed incrementally as a means of reducing ambiguity and to validate the requirements. The requirements were documented in a full detailed specification. 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 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 (e.g., the behavior of the method as defined in the specification) of each method were tested utilizing the JUnit⁵ 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. 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 in asynchronous modes.

To guarantee that all unit tests would be run by all members of the team, we set up an automated build and test systems in the geographical locations where the development teams were located. These systems would extract all the code from the library build and run all the unit tests daily. The Apache Ant⁶ 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 checkpoints for the team.

Each of the implemented devices has a set of exposed methods, properties, and events (following the JavaBeans⁷ specification). The strategy for implementing the unit tests is to implement a test class for each device with test methods for each of the exposed functions. These tests can generally be divided into two primary categories: automated and interactive. Automated tests can be executed with the appropriate device attached and interactive tests require human interactions to be completed; for instance, interrogating the human user to plug or unplug a device. Over time the unit tests were refactored to push common code into super classes as well as into utility classes.

In addition, the team also implemented two other kinds of tests. First, a series of performance-oriented unit tests; these are designed and instrumented to capture performance metrics on the devices at various levels of the software system. The goals of these tests are to give a sense of the performance of the drivers as they are being developed. Our approach and results are documented in [14]. Second, the team also

⁵ <http://junit.org>

⁶ <http://ant.apache.org/>

⁷ <http://java.sun.com/products/javabeans>

implemented unit tests and functional tests for the supporting classes that enable the drivers to be cross-bus as well as enable the complete driver package to support many different hardware platforms and operating systems, e.g., to enable support for different bus connectivities such as the open USB and RS-232 buses as well as IBM's proprietary RS-485 bus.

3.3.2 Functional verification testing. When the majority of the device driver code was implemented and passing their own unit tests and those in the code base, the device drivers were sent to functional verification test (FVT). The external FVT team had written black box test cases based on the functional system specification and on conversations with developers. More than half of the FVT tests were automated in part (requiring human intervention to declare pass or fail) using TCL or Jacl⁸ scripts; the remaining tests were split 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. 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.)

3.4 Research Methodology

The first author mined the source code repository and the defect tracking system to obtain quantitative metrics, such as lines of source code, cyclomatic complexity, and number of defects for each of the ten releases. Additionally, the first and third authors were part of the device driver team and could be considered action researchers⁹. Their knowledge of the daily operations of the team is shared throughout this paper.

The second author conducted two web-based surveys using the SurveyMonkey¹⁰ tool. One survey was for the developers and the other was for the testers. The purpose of the surveys was to obtain qualitative information about the use of TDD from the team. The developer survey was offered to 13 developers and answered by 11. The tester survey was offered to eight testers and answered by seven. Primarily, the survey results are discussed in Section 4.

Finally, the first author qualitatively and subjectively provided his assessment of perceived complexity of devices based upon his extensive observations of how difficult the device driver was to implement.

⁸ Jacl is a Java-compatible version of TCL

⁹ The first author is still part of the development team but the third author moved on to another position at IBM Research

¹⁰ www.surveymonkey.com

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” [9]. 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” [9]. 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 [6]. The results presented in this paper are based upon the work of more than 20 developers over the course of five years. As a result, the quantitative results cannot conclusively be exclusively attributed to the use of TDD. On the other hand the results contribute to the growing body of knowledge of the effectiveness and performance of the TDD practice.

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 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 the efficacy of TDD within an IBM development group over five years and multiple releases. This team chose to do up-front design via UML class and sequence diagrams and test cases were written incrementally as the code was being written, not incrementally before code was written. Our results, therefore, apply to teams that follow a similar process.

4. Results

In this section, we provide the results of our analysis of study data. First, we look at the defect density of the releases. Second, we investigate the impact of TDD on the team. And finally, we look into the details of how the team applied TDD as well as the evolution of the system's complexity.

4.1 Defect Density

In our survey, 100% of the developers indicated that writing tests helped them produce a higher quality product. The developer's perception was shown to be correct. We cannot reveal specific proprietary information (e.g., exact defects/KLOC) about the quality of the device drivers. However, as will be

discussed, our results indicate that both the external and the internal defect density across all releases are significantly lower than the industry averages. The defect density trends are shown in Figure 3 for both internally- and externally-discovered defects. Our LOC measure is non-commented source lines of code.

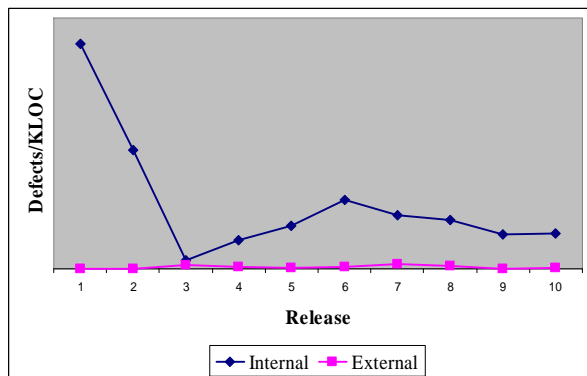


Figure 3: Defects/KLOC

(Y-axis suppressed to protect proprietary data)

The internally-discovered defects were primarily discovered in functional, regression, and beta testing before the official release of the drivers. We had previously reported [19, 25] that Release 1’s internal defect density was 40% lower than a prior device driver project developed in a more traditional, non-TDD fashion. The prior product already had an internal defect density superior to the benchmark published by the Bangalore SPIN Benchmarking group (SPIN) [1] of 8.2 defects/thousand lines of code (KLOC). Internal defect density decreased with Release 2 and Release 3, then stabilizing in future releases and still remaining below the internal defect density of Release 1 (and thereby, much lower than the prior product and the industry standard).

The externally-discovered defects were found and reported by customers. We compare the external defect density to industry averages. Capers Jones [16] reports an average of 0.495 post release defects/function point. To convert defects/function point to defects/KLOC, we use a function point conversion estimate of 60 LOC/function point, published by QSM¹¹ to obtain an industry average of 8.25 defects/KLOC.

While the overall product had a low defect density, the results of individual device drivers written by developers that embraced TDD to varying degrees provide additional information. In Table 2, we provide data about the five devices of the 14 devices supported by IBM with the highest defect density. In the first column is the perceived complexity by the first author

based upon his extensive observations of how difficult the device driver was to implement. The second column provides the relative cyclomatic complexity¹² of the device driver relative to the average cyclomatic complexity of the full set of device drivers (the system average = 1.0). The third column is the ratio of test LOC to source LOC to be compared against this ratio for the full set of device drivers. Finally, the fourth and fifth columns provide the number of manual tests and automated tests for the device, respectively. This measure indicates both the degree to which the device requires manual intervention for testing and to which the developer embraced TDD.

There are some observable trends from the first author for the devices in this “Bottom 5” list:

- The device and/or design is complex.
- Not enough tests have been defined relative to other devices.
- The developer avoided running the manual tests and spending the time to review the output of the tests.
- The devices were developed by inexperienced engineers.

Table 2: “Bottom 5” Devices with Highest Defect Density

Perceive Complex	Relative Cyclom Complex	Test LOC/ Src LOC	Manual Tests	Automat Tests
Easy	1.03	0.54	10	116
Medium	1.24	0.09	1	31
Difficult	0.95	0.59	39	350
Medium	1.00	0.22	9	12
Difficult	1.10	0.76	5	232

Similarly, we provide measures for the five devices of the 14 devices supported by IBM with the lowest defect density. There are some observable trends for the devices in this “Top 5” list:

- The device is well designed.
- There are sufficient tests defined.
- The devices are all developed by experienced engineers.

¹¹ <http://www.qsm.com/FPGearing.html>

¹² Cyclomatic complexity is a measure of the number of linearly independent paths through a program [9] and is often used as an internal measure of the actual complexity of program code. The higher the number the more complex the software. Absolute numbers are not provided to protect proprietary information.

Table 3: “Top 5” Devices with Lowest Defect Density

Perceiv Compl	Relative Cyclom Complex	Test LOC/ Src LOC	Manual Tests	Automat Tests
Medium	1.29	0.63	9	134
Medium	0.96	0.88	3	271
Difficult	0.85	1.12	39	168
Medium	1.20	0.13	3	141
Medium	1.06	0.43	0	113

4.2 Details of Testing Effort

How much testing is required to achieve a quality improvement such as was realized by the IBM team? Though an inexact measure, Figure 4 shows the result of computing the ratio of test LOC to source LOC. The average ratio across all ten releases is 0.61.

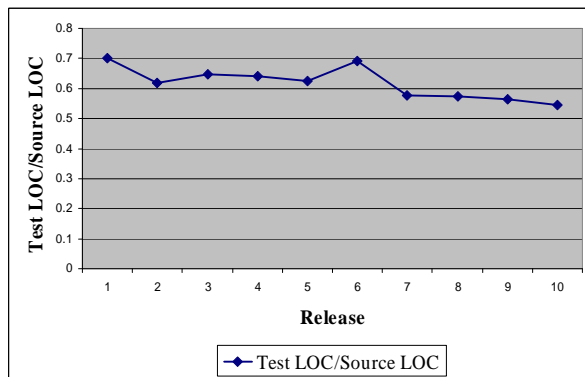


Figure 4: Test LOC/Source LOC

As indicated earlier, the IBM team did not strictly follow Beck’s “no design/incrementally write tests before code” TDD practice. The IBM team worked from a design document., in the form of UML design diagrams. In our survey, three developers indicated they wrote tests before code, seven wrote tests as they wrote the code, and one wrote tests after finishing about half the code. Most developers (10 of 11) said they both ran their own tests and the teams’ tests at least once per day. Developers indicated little time was spent maintaining legacy tests, at most one to three hours per week.

4.3 Developer Perception

Increase quality was achieved. But, what was the impact on productivity? The team did not keep time records. However, on the survey we asked the developers for their perception of how much overall time it took to write the unit tests (Question D8 in the Appendix). Of the ten developers that answered this

question, one said it took no extra time, one said it took less than 5% extra time, two said between 5% and 10%, one said between 11% and 20%, two said between 21% and 25%, and three said more than 25%. An increase in quality can pay for moderate productivity losses. According to one study [8], a reduction of at least 15% in defect density economically justified productivity loss of 15% or lower. Our overall defect density decreased by more than 15%. Additionally, developers indicated that the use of TDD got easier over time:

At the beginning, [TDD] demanded more time... Later it was easy. It pays back when we see robust code, with better quality.

I’ve never used unit testing in this way, I was very impressed with the use of this practice and how it helps to avoid problem injection. In a project like JavaPOS, where there are thousands of LOCs it can be easy to change code that can impact some other functionality. With TDD we know on a daily basis if something goes wrong because of the build notification that is sent via email. Personally, I wholeheartedly recommend the use of this good practice of unit testing; I could even say it is a must!

One way to understand this perception is that with each release, the developers became better at writing the tests and at TDD in general. Specifically, in time the code produced is more testable and with refactoring and good design common test code can make new tests easier to write.

In the open-ended responses, both the developers and the testers indicated that the presence of unit tests allows them to find problems more easily and to avoid regression defects. Additionally, a majority of the functional verification and system testers (four of seven) felt the quality of the software products delivered from the developers into test was better than other products they had tested. Of the remaining testers, two had never tested anything but JavaPOS (so they had no comparison point), and one felt quality was equal to other projects he/she had tested. None felt quality was worse. One tester indicated:

Mainly, the fix for a JavaPOS defect generally does not break anything else and that fix does solves the problem described in the defect description; it is really an exception to have to re-open a defect due to a defective fix. As far as I have seen, testing other products of the same sort there is an important risk that a fix breaks something else or the fix does not solve the problem.

The perception of the increase in quality and decrease in the frustration and time associated with regression defects would contribute to the “staying power” of the TDD practice with this team.

4.4 Code Complexity

Often the complexity of software, as measured by cyclomatic complexity, will increase with each revision [20]. Conversely our data, shown graphically in Figure 5, indicate very little increase in cyclomatic complexity over the course of 10 releases. As Test LOC/Source LOC decreases slightly in Releases 7-10, the complexity increases slightly. Our data suggests that the sustained use of TDD may reduce the increase in complexity as software ages.

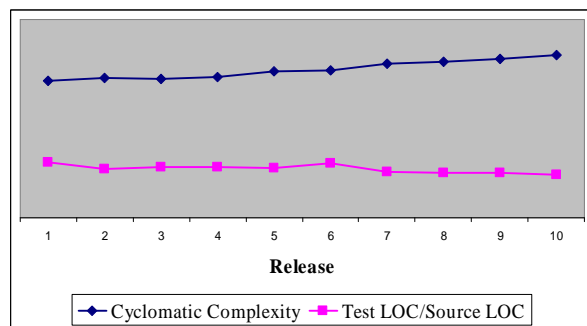


Figure 5: Complexity Trends

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 TDD based upon our five years of experience applying TDD:

- *A passionate champion for the practice and associated tools are necessary.* The champion needs to be willing to spend the extra time to convince and help all team members and project management. Over time, the duties of the champion become easier as other members of the team will start helping newer members and as project managers see tangible results from the TDD practice, e.g., lower defect densities.
- *The JUnit framework is a well structured framework for unit test* relative to the ad hoc, “throw away” kind of testing we had done in the past. Improvements from what we had previously done involve the ease of using the *assert* statements and the automated check of expected results. The suite of tests become an asset to the project and can be run repeatedly, every build.

- *Not all tests can be automated.* In particular, our projects have devices that require manual intervention. For example, a person must swipe a card for the magnetic stripe reader or visually inspect what is printed on a receipt.
- *We extended the JUnit framework to handle manual intervention.* We defined a new signature named *itest*. An *itest* might require manual intervention, but otherwise is similar to a JUnit test. We also modified the JUnit execution framework and added common methods to interact with the users. This allows the JUnit execution to interrogate (with timeout) whenever an *itest* is executed.
- *Set measurable objectives.* Our objective was 80% code coverage.
- *Create a good design and structure of the code (including tests code) using object-oriented principles (OOP).* The use of OOP leads toward good design habits, such as the use of the model-view-controller pattern that will allow more of the code to be tested in an automated fashion. Maximizing the automated tests is desirable because manual tests will not be run as often.
- *Execute all the tests prior to checking code into the code base.*
- *Institute a nightly build process that include running all the unit tests* to ensure that the tests are run at least once/day. There is recommended to have a dedicated person to maintain the automated build systems, in our case it takes 2 hours approx per week.
- *Create a new test whenever a defect is detected internally or externally.* The developer can learn about the type of test that should have been written to prevent the defect from escaping to test or to the customer and can be more confident in the solution.
- *Take time to refactor TDD tests every so often.* With time TDD tests will be duplicated which will make management and currency of tests more difficult (as functionality changes or evolve). Taking time to refactor tests is a good investment that can help alleviate future releases development frustrations and improve the TDD test bank.

6. Conclusions

In this paper, we present the results of a longitudinal case study of an IBM team that has been practicing TDD for ten releases over a five-year period. Our results indicate that the use of TDD can aid a team in developing a higher quality product. The quality improvement was not only evident in our metrics but also to the developers and to the product testers. As noted in Section 1, perception and tangible results indicating the usefulness of the practice are essential for

its “staying power”, as has been done TDD for this IBM team. The developers indicated there could be some productivity decreases, but the product lifecycle quality improvement would compensate for moderate perceived productivity losses. Additionally, the use of TDD may decrease the degree to which code complexity increases as software ages.

7. Acknowledgements

We thank the IBM device driver development team members in Raleigh, NC, Guadalajara, Mexico, and Martinez, Argentina. We would also like to thank members of the NC State University *RealSearch* research group for their help on the paper.

8. References

- [1] Bangalore Benchmarking Special Interest Group, "Benchmarking of Software Engineering Practices at High Maturity Organizations," Bangalore Software Process Improvement Network, 2001.
- [2] 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.
- [3] K. Beck, *Extreme Programming Explained: Embrace Change*. Reading, Mass.: Addison-Wesley, 2000.
- [4] K. Beck, *Test Driven Development -- by Example*. Boston: Addison Wesley, 2003.
- [5] 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
- [6] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Design for Research*. Boston: Houghton Mifflin Co., 1963.
- [7] 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 Janeiro, Brazil, 2006, pp. 364-371.
- [8] H. Erdogmus and L. Williams, "The Economics of Software Development by Pair Programmers," *The Engineering Economist*, vol. 48, no. 4, pp. 283-319, 2003.
- [9] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*: Brooks/Cole, 1998.
- [10] D. Gelperin and W. Hetzel, "Software Quality Engineering," in *Fourth International Conference on Software Testing*, Washington, DC, June 1987.
- [11] B. George, "Analysis and Quantification of Test Driven Development Approach MS Thesis," in *Computer Science* Raleigh, NC: North Carolina State University, 2002.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] D. S. Janzen and H. Saiedian, "On the Influence of Test-Driven Development on Software Design," in *Conference on Software Engineering Education and Training (CSEET)*, Turtle Bay, Hawaii, 2006, pp. 141-148.
- [16] C. Jones, *Software Assessments, Benchmarks, and Best Practices*. Boston, MA: Addison Wesley, 2000.
- [17] C. Larman and V. Basili, "A History of Iterative and Incremental Development," *IEEE Computer*, vol. 36, no. 6, pp. 47-56, June 2003.
- [18] R. C. Martin and R. S. Koss, "Engineer Notebook: An Extreme Programming Episode," <http://www.objectmentor.com/resources/articles/xpepisode.htm>, no. 2001.
- [19] E. M. Maximilien and L. Williams, "Assessing Test-driven Development at IBM," in *International Conference of Software Engineering*, Portland, OR, 2003, pp. 564-569.
- [20] A. Monden, S. Sato, and K. Matsumoto, "Capturing industrial experiences of software maintenance using product metrics," in *5th World Multi-Conference on Systemics, Cybernetics and Informatics*, Orlando, FL, 2001, pp. pp. 394 - 399.
- [21] M. M. Müller and O. Hagner, "Experiment about Test-first Programming," *IEE Proceedings Software*, vol. 149, no. 5, pp. 131-136, 2002.
- [22] S. L. Pfleeger, "Understanding and improving technology transfer in software engineering," *Journal of Systems and Software*, vol. 47, no. 2-3, pp. 111-124, July 1999.
- [23] C. K. Riemenschneider, B. C. Hardgrave, and F. D. Davis, "Explaining software developer acceptance of methodologies: a comparison of five theoretical models," *IEEE Trans. Software Eng.*, vol. 28, no. 12, pp. 1135 - 1145, December 2002.
- [24] L. Williams and R. Kessler, *Pair Programming Illuminated*. Reading, Massachusetts: Addison Wesley, 2003.
- [25] 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

- D1. When you develop code, do you work from a design document?
- No
 - Yes, specify what kind of document(s) (e.g., sequence diagram)
- D2. When do you write automated unit tests?
- before you write the code
 - as you write the code
 - after you finish some code? If you write unit tests after you finish writing code, how much code do you finish before you write some tests?
- D3. What do you base your unit tests on? (check all that apply)
- The structure of the code
 - The requirement document
 - Important scenarios
 - Other (specify)
- D4. How do you decide when you have written enough unit tests?
- Code coverage
 - Gut feel
 - Run out of time
 - Other (specify)
- D5. How often do you run your own unit tests?
- Once/day
 - More than once/day
 - Less than once/day
 - Other (specify)
- D6. How often do you run the unit tests from others in our team?
- Once/day
 - More than once/day
 - Less than once/day
 - Other (specify)
- D7. Overall, do you think writing unit tests helps you produce a higher quality product?
- No
 - Yes
 - Comment
- D8. From a product lifecycle perspective (e.g. during development + defect fix time when your code is in test + defect fix time when your code is in the field), how much time do you feel writing unit tests adds to your time?
- It doesn't
 - <5%
 - 5%-10%
 - 11%-20%
 - 21%-25%
 - More than 25%
- D9. Comment on whether you think your bank of unit tests is helpful for regression testing, makes you feel

more courageous when you make a change, or any other cost/benefit of automated unit testing?

D10. Comment on the effort and payoff required to maintain "legacy" unit test code. Also comment on how long you have worked on this project/what releases.

A.2 Tester Survey

- T1. How long have you been a tester for JavaPOS device drivers?
- Three years or more
 - One to two years
 - Less than one year
- T2. What types of tests do you write (check all that apply):
- integration test
 - functional test
 - system test
- T3. What do you base your test cases on (check all that apply):
- requirements document
 - conversations with the develop
 - conversations with a requirements analyst
 - conversations with a customer
 - other (please specify)
- T4. Do you have any entry criteria before you will accept code into test? Please explain.
- T5. Based upon your experience, how can you compare the quality of JavaPOS device drivers coming into test versus other products you have tested?
- Better
 - Worse
 - About the same
 - I've never tested anything but JavaPOS
- T6. Based on your answer to #5, can you explain the differences you see based upon your knowledge of the development process and/or the product itself?
- T7. Based upon your experience, how can you compare the quality of JavaPOS device drivers delivered to the customer?
- Better
 - Worse
 - About the same
 - I've never tested anything but JavaPOS
- T8. Do you automate your tests?
- No
 - Yes, using the following technology (specify):
- T9. If you are pressured for time and can't run all the tests you planned, how do you decide which test cases to run?
- T10. Please provide any additional comments on your observations as a JavaPOS device driver tester.