Assessing Test-Driven Development at IBM

E. Michael Maximilien
IBM Corp. and NCSU
5505 Six Forks Road
Raleigh, NC 27609

maxim@us.ibm.com

Laurie Williams
North Carolina State University
Department of Computer Science
Raleigh, NC 27695-8207

williams@csc.ncsu.edu

Abstract

In a software development group of IBM Retail Store Solutions, we built a non-trivial software system based on a stable standard specification using a disciplined, rigorous unit testing and build approach based on the test-driven development (TDD) practice. Using this practice, we reduced our defect rate by about 50 percent compared to a similar system that was built using an ad-hoc unit testing approach. The project completed on time with minimal development productivity impact. Additionally, the suite of automated unit test cases created via TDD is a reusable and extendable asset that will continue to improve quality over the lifetime of the software system. The test suite will be the basis for quality checks and will serve as a quality contract between all members of the team.

1 Introduction

IBM Retail Store Solutions (RSS) is located primarily in Raleigh, North Carolina. As one of the founding members of the Java for Point of Sale (JavaPOS) specification, IBM participated in the creation of the standard and specification with Sun, NCR and Epson. The JavaPOS specification defines a set of JavaBeans (software services) to allow access to point of sale (POS) devices (e.g. printers, cash drawers, magnetic stripe readers, bar code readers or scanners) in Java applications. The specification defines a set of properties, methods, and events applicable for each device class and the semantic model for the behavior of the devices. Over the past three years, the RSS division has implemented the JavaPOS specification for a wide array of devices on various operating systems (e.g. Windows, Linux, and IBM's own retail operating system 4690-OS[1]).

Though the development team has a broad experience with the JavaPOS[2] specification and the POS devices, we have noticed that for each revision of the deliverable, the defect rate after Functional Verification Test (FVT) was not being reduced as we had expected. As a result, the development and management teams were open to new approaches to development. We proposed a development practice based on the eXtreme Programming (XP) [1] test-driven development (TDD) [2] approach. With TDD, all major public classes of the system have a corresponding unit test class to test the public interface, that is, the contract of that class [8] with other classes (e.g. parameters to method, semantics of method, pre- and post-conditions to method). This practice was used by the new JavaPOS[3] development and test teams. The team was made up of nine full-time engineers. Five of the engineers were located in Raleigh (including the team lead); four were located in Guadalajara, Mexico. Additionally, part-time resources for project management and for performance were dedicated to the team.

In this paper, we examine the efficacy of the TDD approach, to alleviating the recurrent quality and testing problems, of the new JavaPOS project. The remainder of this paper is organized as follows. Section 2 provides an overview of traditional and TDD unit testing techniques. Section 3 discusses our view of expected gains and business risks upon transitioning to TDD. Section 4 discusses our experiences with TDD in our software development organization. Section 5 presents our results and lessons learned. Finally, Section 6 summarizes our findings and future work.

2 Unit Testing

This section provides an overview of traditional unit testing and the emerging test-driven development practice.

2.1 Prior Unit Test Approaches

[1] http://www2.clearlake.ibm.com/store/support/html/driver.htm

[2] IBM, NCR Epson and Sun http://javapos.com

[3] We use "new JavaPOS" for the JavaPOS release where we used the TDD development process. JavaPOS Legacy is the moniker we use for the previous releases.

The prior approach to unit testing at IBM RSS was an ad-hoc approach. The developer coded a prototype of the important classes and then created a design via UML class and sequence diagrams [6]. 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 coding phases. Real unit testing then followed as a post-coding activity. One of the following unit test approaches was chosen:

- After enough coding was done, an interactive tool was created by the developer that permitted the execution of the important classes.
- Unit testing was executed using an interactive scripting language or tool, such as jython[4], which allows manual interactive exercising of the classes by creating objects and calling their methods.
- Unit testing was done by the creation of independent ad-hoc driver classes that test specific important classes or portions of the system which have clear external interfaces.

In all cases, the unit test process was not disciplined and was done as an afterthought. More often than not, no unit tests were created, especially when the schedule was tight, the developer got side tracked with problems from previous projects, or when new requirements that were not clearly understood surfaced. Most of the unit tests developed were also thrown away and not executed during the FVT phase or when a new release of the software was developed.

2.2 Test-Driven Development

With TDD, before writing implementation code, the developer writes automated unit test cases for the new functionality they are about to implement. After writing test cases that generally will not even compile, the developers write implementation code to pass these test cases. The 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 control because he or she is continuously making small design and implementation decisions and increasing functionality at a relatively consistent 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.

As an example of how the unit tests are structured is to consider a typical method that takes one input parameter, returns an output value, and could throw an exception. Such a method would then have a unit test for (1) a valid parameter value which expects a valid return and (2) an invalid value causing appropriate exception; boundary values are typically selected. The unit test method could be longer than the method itself. Often the unit tests are not exhaustive but rather test the typical expected behavior with valid parameters and a few negative paths. Furthermore, additional unit test methods are sometimes added for each method when its behavior is dependent on the object being in a particular state. In these cases, the unit test starts with the correct method calls that put the object in the correct state.

Some professed benefits to TDD are discussed below:

- In any process, there exists a gap between decision (design developed) and feedback (functionality and performance obtained by implementing that design). The success of TDD can be attributed to reducing that gap, as the fine granular test-then-code cycle gives constant feedback to the developer. An often-cited tenet of software engineering, in concert with the "Cost of Change" [3], is that the longer a defect remains in a software system the more difficult and costly it is to remove. With TDD, defects are identified very quickly and the source of the defect is more easily determined.
- 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 include the following: (1) production of a reliable system, (2) improvement of the quality of the test effort, (3) reduction of the test effort and (4) minimization of the schedule [5].
- The TDD test cases create a thorough regression test bed. By continuously running these automated test cases, one can easily identify if a new change breaks anything in the existing system. This test bed should also allow smooth integration of new functionality into the code base.

With XP, developers do little or no up-front design before embarking in tight TDD cycles consisting of test case generation followed by code implementation. However, many of the benefits listed above can be realized in essentially any development process simply by shifting from *unit test after implementing* to *unit test before implementing*.

3 Assessing Expected Gains and Risks

In our past experiences, the ad-hoc approach to unit testing usually leads to last minute or even no testing at all. Therefore, when we set out to build the new implementation of the JavaPOS services, the management team was open to a new unit testing practice as long as we could articulate our expected long term advantages. Since this was the first time that IBM RSS had taken such an approach to software development, there were many

[4] http://www.jython.org

unknowns and questions that the management and development teams wanted answered:

- *Defect Rate.* How will this rigorous approach affect the defect rate in the short term (current release) and the longer term (future releases)?
- *Productivity.* What is the impact to developer productivity (lines of code (LOC) per person-month)?
- *Test Frequency.* What will be the ratio of interactive vs. automated tests? How often will each of these types of test be run?
- *Design.* Does the use of the TDD practice yield systems that have a more robust design? We assess the robustness of design by examining the ease of handling late requirements and supporting new devices and services.
- *Integration.* Does TDD and its resulting automated regression test assets allow for smoother code integration?

Many in the development and management team were concerned that this rigorous approach would impact productivity so much that we would not be able to keep our schedule dates. Further, there was some resistance from the developers at first, since many were not only new to TDD but also some were somewhat unfamiliar with Java. All but two of the nine full-time developers were novices to the JavaPOS specification and the targeted POS devices. The domain knowledge of the developers had to be built during the design and development phases.

To alleviate the productivity concerns, we decided to be very careful in the scheduling phase. Since we had an existing similar system, we decided to measure the LOC of the existing system and to extrapolate and predict the LOC of the new system. We used an average productivity rate of 400 LOC per person-month as per the finding of an internal audit of our process by an IBM consultant. This productivity rate was determined to be justified for our development team by studying past historical data. This rate included time for design, unit testing, and code implementation. This rate is also appropriate for the type of software that we develop in which essential money [4] is at risk.

4 IBM RSS Experiences

In the past, unit test was usually an afterthought after code had been developed and was *working*. With TDD, test cases are developed up front as a means to reduce ambiguity and to validate the requirements, which for the JavaPOS comes in the form of a full detail standard specification. We found that such up-front testing drives a good understanding of the requirements and an up-front design. In XP projects, such up-front testing proceeds without any "big design up front," commonly referred to as BDUF [1]. However in our system, the requirements were stable, and we chose to do up-front design via UML class and sequence diagrams. This design activity was interspersed with the up-front unit test creation.

After creating a "spike" [1] 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 *complete testing* as ensuring that the public interface and semantics (the behavior of the method as defined in the JavaPOS specification) of each method were tested utilizing the JUnit[5] unit testing framework. 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 achieve 80 percent of the important classes covered by automated unit testing. By automated we mean requiring no human interaction and thus unit test that can be batched and executed automatically. Each design document included a unit test section that listed all important classes and the public methods that would be tested. Some unit tests would also contain methods that tested particular variations on behavior, e.g. the printer has an asynchronous printing capability and the regular print methods behave differently in synchronous vs. asynchronous mode.

To guarantee that all unit tests would be run by all members of the team, we decided to set up automated build and test systems both locally (in Raleigh) and remotely (in Guadalajara). Daily, these systems would extract all the code from the library build and run all the unit tests. The Apache ANT[6] build tool was used. After each automated build/test run, 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 locally and remotely. Eventually, we decided to alternate the build between locations and to only run the build tests once a day. Figure 1 summarizes the development and test process used by the team.

[5] http://junit.org

[6] Apache Software Foundation http://www.apache.org/jakarta/ant

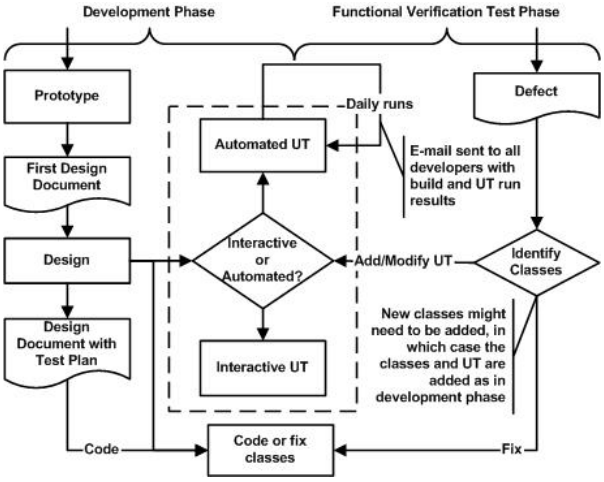

Figure 1: Summary of development and test process (UT = Unit Test)

For every project at IBM RSS, various metrics are collected during the FVT phase to keep track of the test progress and to predict the start of regression and the end of the test. We define the start of regression as the date when 100% of the FVT tests have been attempted. (This does not imply that the defects from these attempted tests are all resolved.) We predict 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 based on historical data. From these predictions, a defect and test progression curve is calculated with weekly points that forecast the number of executed tests cases and the expected number of defects for each week of the project. For the new JavaPOS project, we entered test with 71.4 KLOC of new code and 34 KLOC of JUnit code; the total predicted defects for our process was 286 defects or 4 errors/KLOC. Generally, new development projects within RSS are estimated at 8 errors/KLOC; the test team demonstrated their confidence in the TDD approach by utilizing a 50% reduction in their estimate.

Another set of data that we collected during the development phase is the number of KLOC from all different modules (source and test code) and the number of JUnit tests for each module. For approximately 71 KLOC, we wrote approximately 2390 automated unit test cases. Additionally, over 100 automated JUnit performance test cases were written.

The new JavaPOS team experienced approximately a 50% reduction in defect density in FVT. Figure 2 displays the defect density for a comparable JavaPOS legacy implementation that was recently updated; TDD was not used. On this project, the realized defect density was consistently higher than the projected density and averaged 7.0 errors/KLOC. The development and test teams for this legacy project were fully experienced with the specification, code and devices (this is the third release of that project). Comparatively, Figure 3 displays the defect density for the relatively inexperienced new JavaPOS team that employed the TDD practice. Our actual defect density averaged 3.7 errors/KLOC. We attribute this quality increase to the use of the TDD practice.

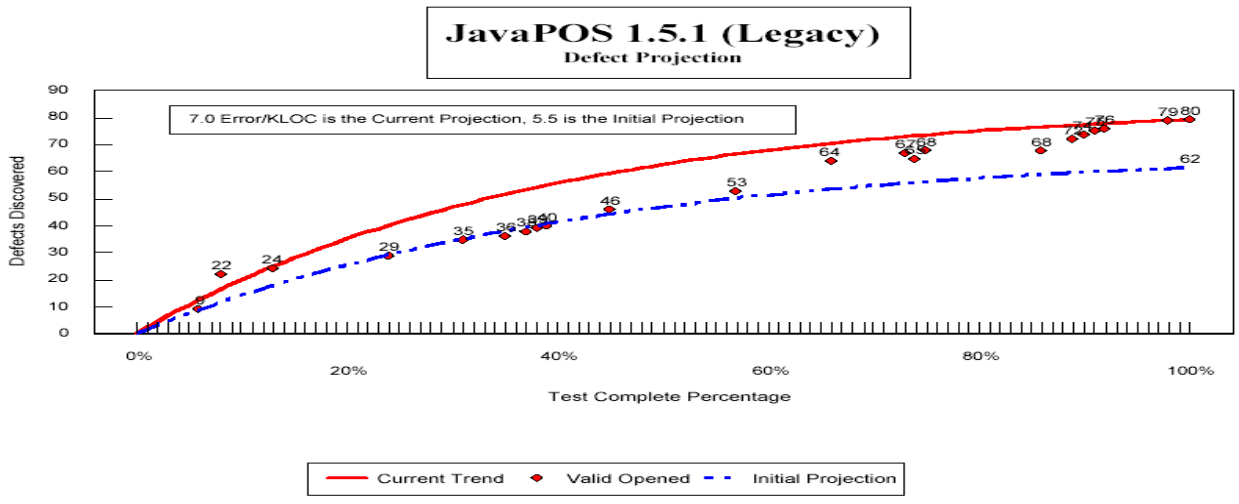

Figure 2: FVT Defect Projection - Ad-Hoc Testing Project

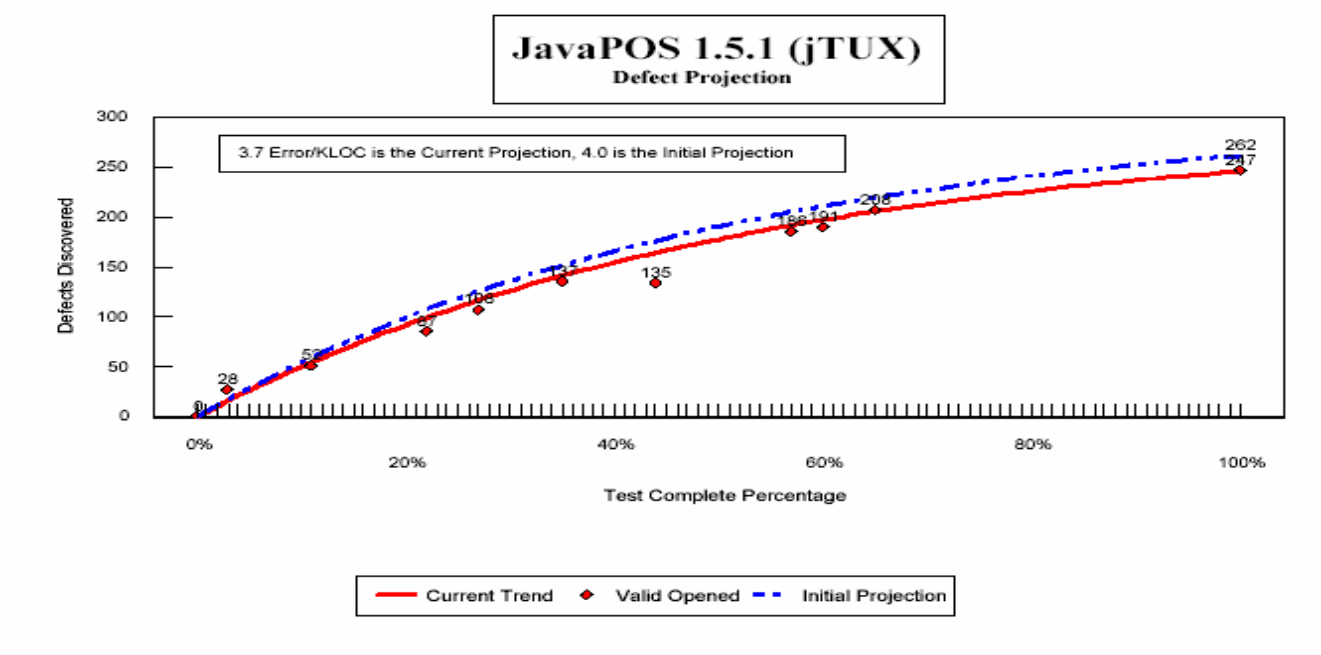

Figure 3: FVT Defect Projection - TDD Project

5 Results and Lessons Learned

In this section, we first share the business results from our adoption of TDD. Next, we share our lessons and suggestions for team transitioning to TDD. Our business results are now discussed:

- *Defect Rate.* With TDD, unit testing actually happens. In our prior approaches, testing was an afterthought. With TDD, unit testing is an integral part of code development. As a result, we achieved a dramatic 50% improvement in the FVT defect rate of our system.
- *Productivity.* Because the project is still in the final phases of regression test, we cannot compute exact productivity numbers. However, we know the project is on schedule. We expect our productivity numbers to be at or slightly below the 400 LOC/person-month estimate. Other studies have also found a slight decrease in developer productivity when employing the TDD practice [7, 9]. We also must attribute some credit toward our productivity to the new use of Microsoft Project Central[7], which many believe has improved our project management practices and kept the developers consistent about making visible progress on their tasks.
- *Test Frequency.* We wrote approximately 2500 automated tests and 400 interactive tests. Eighty-six percent of the tests were automated, exceeding our 80% target. The interactive tests were rarely run; the automated tests were run daily.
- *Design.* We believe that the TDD practice aided us in producing a product that would more easily incorporate late changes. A couple of devices (albeit, not overly complex devices) were added to the product about two-thirds of the way through the schedule.
- *Integration.* In the past, we only integrated code into the product once we were close to FVT. The daily integration certainly saved us from late integration problems. The success (or failure) of the daily integration served as the heart beat of the project and minimized risk because problems surfaced much earlier.
- *Morale.* The developers are very positive about the TDD practice and have continued its use.

Additionally we have gained experiences with transitioning to TDD we wish to share with other teams. We list these in order of importance.

- Start the TDD from the beginning of project. Set the expectation that team members that fervently apply the unit test paradigm may initially appear less productive or frustrated at the time it takes to create the unit tests. Assure them that the practice will have ultimately minimal impact to their productivity.
- For a team new to TDD, introduce automated build test integration towards the second third of the development phase—not too early but not too late. If this is a brand new project, adding the automated build test towards the second third of the development

[7] http://www.microsoft.com/office/project/default.asp

schedule allows the team adjusts to and become familiar with TDD. Prior to the automated build test integration, each developer should run all the test cases on their own machine.

- Convince the development team to add new tests every time a problem is found, no matter when the problem is found. Thus, at least one new test should confirm the presence of and removal of each valid, opened defect. By doing so, the unit test suites improve during the development and test phases. This part of the process needs to be communicated early and enforced by reminders and monitoring of unit test count.
- Get the test team involved and knowledgeable about the TDD approach. The test team should not accept new development release if the unit tests are failing.
- Hold a thorough review of an initial unit test plan, setting an ambitious goal of having the greatest number of automated tests, since automated tests can be easily integrated and run automatically without necessitating human intervention. We used 80% automated tests as a minimum goal. We cannot overemphasize the importance of constantly running in a daily automatic build; tests run should become the heartbeat of the system as well as a means to track progress of the development. This also gives a level of confidence to the team when new features are added. If new classes and tests are added and the next build test is successful, the team can have a higher confidence in the change.
- Encourage fast unit test execution and efficient unit test design. Test execution speed is very important since when all tests are integrated the complete execution can become quite long for a decent project and constant test execution. Results are important early and often, to provide feedback on the current state of the system. Further, the faster the execution of the tests the more likely developers themselves will run the tests without waiting for the automated build tests results.
- Take all opportunities to encourage development team to add unit tests to their code. This can be done by monitoring which subsystems have more unit tests relative to how big that subsystem is and acknowledging the developer of the subsystem.

6 Summary and Future Work

The new JavaPOS development team in the IBM Retail Store Solutions organization transitioned from an ad-hoc to a TDD unit testing practice. Through the introduction of this practice the relatively inexperienced team realized about 50% reduction in FVT defect density when compared with an experienced team who used an ad-hoc testing approach for a similar product. They achieved these results with minimal impact to developer productivity. Additionally, the suite of automated unit test cases created via TDD is 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.

7 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 Mike Hyde and Dale Heeks from the FVT team for creating Figures 2 and 3. We also want to thank the IBM RSS management teams for their willingness to try a new approach to development without prior data on whether this would be effective or not. We also thank the NCSU Software Engineering reading group for their comments and review of initial drafts of this paper. Finally, we want to acknowledge Kent Beck for devising and documenting the TDD approach and for co-authoring the JUnit testing framework.

8 References

[1] Beck, K., *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.

[2] Beck, K., *Test Driven Development -- by Example*. Boston: Addison Wesley, 2003.

[3] Boehm, B. W., *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[4] Cockburn, A., *Agile Software Development*. Reading, Massachusetts: Addison Wesley Longman, 2001.

[5] Dustin, E., Rashka, J., and Paul, J., *Automated Software Testing*. Reading, Massachusetts: Addison Wesley, 1999.

[6] Fowler, M., *UML Distilled*. Reading, Massachusetts: Addison Wesley, 2000.

[7] George, B. and Williams, L., "An Initial Investigation of Test-Driven Development in Industry," ACM SAC, Mel, FL, 2003.

[8] Meyer, B., "Applying Design by Contract," *IEEE Computer*, vol. 25, pp. 40-51, October 1992.

[9] Muller, M. M. and Hagner, O., "Experiment about Test-first Programming," presented at Conference on Empirical Assessment in Software Engineering (EASE), 2002.