Integrating Unit Testing Into A Software Development Team's Process

Randy A. Ynchausti
President, Monster Consulting, Inc.
845 South Main Street, Suite 24
Bountiful, Utah 84010 USA
+1 801 243 0694
randy@monsterconsulting.com

ABSTRACT

Unit testing was integrated into the software development process of a five-member programming team using a test-during-coding training module. The training approach and module are briefly described. Individual and pair developer performance was measured before and after the training module was presented. The improvements in quality achieved by the team ranged from 38% to 267% fewer defects.

Keywords

extreme programming, software development process, test-during-coding, unit testing

1 INTRODUCTION

Recent attention focused on extreme programming has heightened the awareness of unit testing as an important process element for software development. Unit testing is the common name for a set of practices that result in increased confidence and reliability in developed source code. Some key practices of unit testing are:

- Unit tests are automated so they are easy to run and re-run to validate the production software system.
- Unit tests are created for every class in the production system.
- Unit tests are implemented for every method that could break in the production system.
- Unit tests are written before or in conjunction with writing the production code.
- Unit tests are coded in the most simple and direct way possible.
- Unit tests are created so that nuclear concepts and constructs are expressed only once.
- Each unit test returns a value indicating that the test passed or failed.
- All unit tests must pass before new code is released.
- Unit tests are maintained with the production code and used by every developer working on the software.

An effective way of integrating these practices into a development team's process is by teaching them as part of a training module that provides coding exercises to illustrate and reinforce their value. Developers willingly continue to use them once they are convinced of their value. This work describes how these practices were integrated into a software development team's process successfully and highlights the benefits and results that were achieved.

2 TARGET SOFTWARE DEVELOPMENT TEAM

The target software development team consisted of five software engineers. Four of these engineers worked as two sets of pair programmers. The fifth engineer worked individually. The average age of team members was approximately 33 years. The average length of service for team members was over seven years.

Past studies of extreme programming led the team to adopt the practice of pair programming, more than one year prior to the work reported herein, as a means for improving product quality and cross-training. During that period, the team also spent time developing a test framework for unit testing. However, they concluded that the architecture of their software did not lend itself to unit testing and thereby discontinued further development and use of their testing framework.

The diversity of this software development team and their work assignments presented some unique challenges for training and unit testing, but also enhanced the value of the observations and measured results and benefits achieved.

3 TEST-DURING-CODING TRAINING MODULE

In order to symbolize and emphasize the desired end-result, the training module was titled the "Test-During-Coding Training Module." The test-during-coding training module focused on accomplishing several objectives, including: 1) cope with the change that results when implementing new development practices and methodologies; 2) understand why the test-during-coding approach is a prerequisite for high-quality, professional software development; 3) teach the test-during-coding process; 4) assess and analyze the benefits and drawbacks of the test-during-coding approach for the target software development team. Some of the key elements of the test-during-coding training module were:

- Base-line coding game
- Lost traveler video
- Process improvement principles
- Test-during-coding process
- Practice coding game
- Production code guidelines that facilitate testing
- Post-training coding game
- Training module assessment

The elements designed to accomplish the first and second objectives were the lost traveler video and process improvement principles. The lost traveler video element is an eight-minute video presentation that provides an outstanding metaphor for software development and teaches the value of adopting and practicing the sage advice of the industry experts. The process improvement principles element is primarily based on John P. Kotter's eight-stage process for leading change [1]. During this element software developers work through the first four stages to develop a sense of urgency regarding unit testing, develop a sense of teamwork, and understand how unit testing will impact the entire team and development process.

The elements designed to accomplish the third objective were the test-during-coding process, practice coding game, and production code guidelines that facilitate testing. These elements teach developers which production code to unit test and how to test it during development. The practice coding game is an exercise that allows developers their first opportunity to flex their unit testing muscles. After developers have practiced unit testing by playing the practice coding game, coding principles and approaches that facilitate unit testing were reinforced in the production code guidelines that facilitate testing element.

The elements designed to accomplish the fourth objective were the base-line coding game and post-training coding game. The base-line coding game has individual and pair programmers use their current best practices to write a program that solves the problem posed in the game. The post-training coding game has individual and pair programmers use the skills learned during the test-during-coding training module to write a program that solves the problem posed in that game. Comparison of the program and development process metrics for the solutions to both coding games allows the benefits and drawbacks of unit testing to be assessed and analyzed for the target software development team. Data and a discussion from that comparison are provided below.

4 BASE-LINE AND POST TRAINING CODING GAME COMPARISON

Figure 1 provides a graph comparing the program size for the base-line and post-training coding game solutions submitted by each individual or programmer pair. Data for the post-training solution with and without unit test code is included. The post-training coding game solution for Developer 1 was approximately 80% larger than the solution for the base-line coding game. The Developer 2-3 pair produced a post-training coding game solution that is approximately 63% larger than the solution produced for the base-line coding game. The Developer 4-5 pair submitted a solution to the post-training coding game that was more than 200% larger than the solution produced for the base-line coding game.

Figure 1. Program Size Comparison for the Base-Line and Post-Training Coding Games

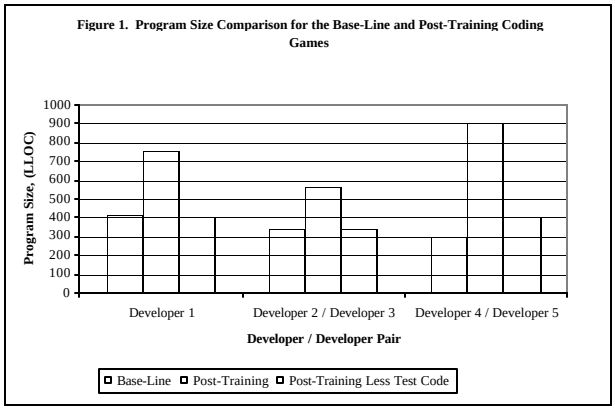
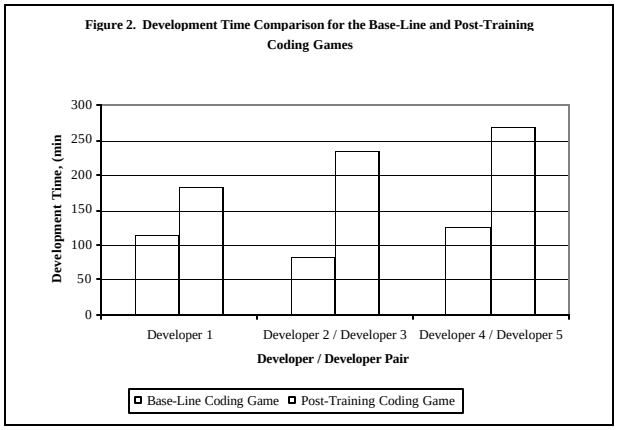


Figure 1 also shows that the program size of the solution for both coding games is essentially equivalent for Developer 1 and the Developer 2-3 pair. The Developer 4-5 pair produced a solution containing approximately 33% more code for the post-training coding game.

Figure 2 shows the development time taken by the individual and pair programmers to complete both coding games. Developer 1 required approximately 60% more time to complete the post-training coding game than the base-line coding game. The Developer 2-3 pair required approximately 187% more time to complete the post-training coding game as compared to the base-line coding game. The Developer 4-5 pair required approximately 116% more time to complete the post-training coding game as compared to the base-line coding game.

Many software development managers, and even developers, feel the additional development time is a high price to pay for essentially the same amount of production code. However, this line of thinking is problematic in that it neglects the amount and value of the unit testing code. The unit tests prove that the production code works. A product with a high level of coverage from unit tests allows developers to evolve the software more rapidly and with more confidence [2].

Figure 2. Development Time Comparison for the Base-Line and Post-Training Coding Games



This confidence comes from being able to change or improve any part of the production code and then retest the entire system. Software development teams that have unit tests are able to modify or replace problematic code without great concern that they will introduce subtle bugs and undesirable side effects [3]. Therefore, the production code becomes less brittle and stale because any part of the system can be refactored and improved as it is being worked on.

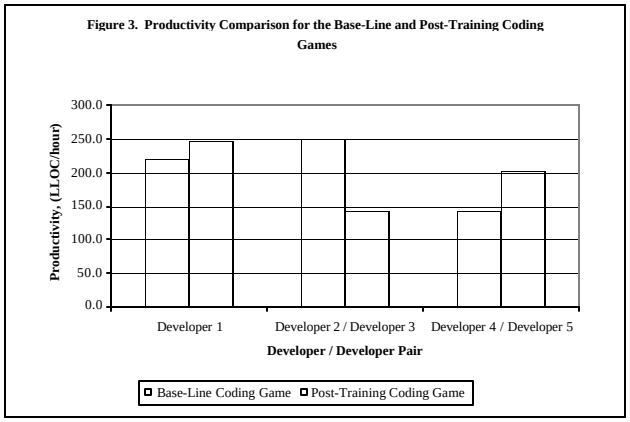
One of the ways to help managers and developers understand the impact on coding when adopting a test-during-coding approach is to track the code productivity rates for the software development team members. A simple productivity metric used in this study was the number of logical lines of source code produced divided by the development time required to produce that code. Consider the comparison, illustrated in Figure 3, of the productivity of each individual and pair programmers for the base-line coding game and the post-training coding game.

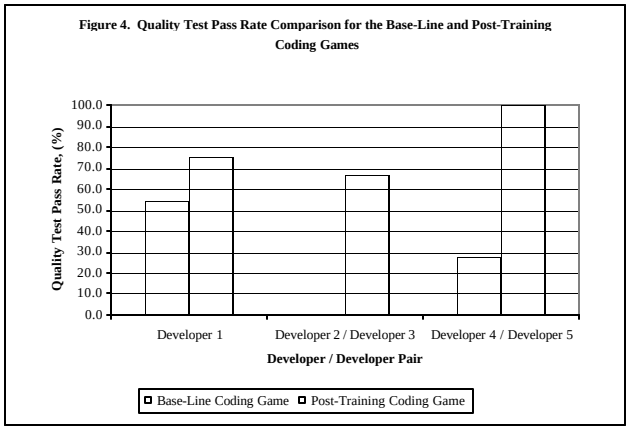
The productivity of the developer working alone improved more than 12% and the Developer 4-5 pair improved by more than 40%. Understanding that an improvement in productivity is often achieved using test-during-coding processes will help most managers and developers overcome the feeling that unit testing excessively slows the development of production code.

The Developer 2-3 pair, however, did not exhibit the normal increase in productivity. Their productivity actually decreased by about 43%. The reasons for this will be examined after presenting the productivity data in Figure 4.

Figure 4 shows the quality test pass rate percentage for each of the individual and pair developers of the target software development team for the base-line and post-training coding games. The quality improvement achieved by the individual developer, Developer 1, was approximately 38%. The quality improvement for the Developer 2-3 pair was infinite since they did not pass a single quality test for the base-line coding game. The quality improvement recorded by the Developer 4-5 pair was about 267%. Furthermore, the Developer 4-5 pair was able to achieve a perfect quality test passing rate of 100%.

Figure 3. Productivity Comparison for the Base-Line and Post-Training Coding Games





Contrasting the data of Figure 2, Figure 3 and Figure 4 for the Developer 2-3 pair provides an understanding of why their productivity went down when unit testing. Notice that the Developer 2-3 pair produced the base-line solution in less time (approximately 35% -- Figure 2) than did the Developer 4-5 pair even though the size of their base-line program was larger (about 15% -- Figure 3). Additionally, the Developer 2-3 pair produced a solution to the base-line coding game that did not pass a single quality test (Figure 4). The net effect was that the Developer 2-3 pair wrote an application that ran, but did not produce the correct results even for the test data that was supplied in the coding game assignment. Therefore, their low quality product negated any of the value of their high productivity rate measured for that coding game. This provides a striking example of the value of unit testing. The Developer 2-3 pair improved the quality of their software solutions to the training module coding games infinitely because of the test-during-coding training and implementing unit testing into their software development process.

The time spent by a software development team writing unit tests during coding is usually much less than the time spent to correct only a fraction of the defects remaining in the system after development. Almost all software developers spend time testing their code before releasing it. Spending this time writing unit tests that are always available to check and re-check the quality of the production code is more valuable than typical debugging or manual testing. As an illustration of this, consider how much more time the Developer 2-3 pair would have required to isolate and correct the defects which prevented any of the quality tests to pass. During the training module, when the pair learned that none of the quality tests passed, they spent over an hour to find a problem with the index of a loop. The correction they made only allowed 27% of the quality tests to pass. Additional time was required to correct the other defects in their code. A unit test could have prevented the index problem and would have taken only a few minutes to generate. Additional unit tests could have been written with the remaining portion of that hour.

5 BENEFITS AND IMPROVEMENTS

One of the most important aspects of this work was to overcome the negative impressions held by the target software development team regarding the test-during-coding approach. During planning meetings before beginning the training module, members of the target development team were very candid about their feelings that their software architectures were not testable. Additionally, the target software development team had tried to implement test-during-coding processes, but these attempts did not produce lasting changes. This experience reinforced the belief that their software could not be tested. In providing training module content that dealt with the impact of change on people, each member of the development team was prepared to overcome the past experiences and worked hard to implement test-during-coding processes in a relatively short period of time.

The training module was structured to implement test-during-coding into the team's software development process in a few days. The training module was completed in three working days, but was taught over a four-day period where only a fraction of the first and last day was used.

Every member of the target software development team was convinced of the benefits and improvements that would be achieved by implementing the test-during-coding processes. One of the most avid developers who suggested that their architecture was not testable recommended to corporate managers that the test-during-coding processes be adopted throughout the entire corporation.

In general terms, the initial impact of the test-during-coding process for the target software development team is that they will spend between 60% and 100% more time implementing 100% more code in the form of unit tests. This result correlates well with the recommendation that good software processes will allow developers to spend 25% to 50% of their time developing tests [4]. As they practice these skills, the time required to write unit tests during development will likely decrease.

It is expected that the quality of the commercial software will mimic the improvements measured using the base-line and post-training coding games. Therefore, improvements in quality ranging from approximately 38% to about 267% fewer defects are expected. This improvement in quality is staggering and well worth the additional effort of implementing test-during-coding

processes.

6 CONCLUSIONS

Implementing unit testing using test-during-coding processes was very successful for the target software development team. Team members were able to overcome their past experiences and reservations regarding the testability of their products. Coding games at the beginning and end of the training module allowed the initial benefits of the test-during-coding approach to be measured. Members of the target development team can expect to spend up to 100% more time implementing unit tests in conjunction with the production code being written. Improvements of up to 267% fewer defects can be achieved through the test-during-coding processes.

REFERENCES

1. Kotter, J. P. "Leading Change," Harvard Business School Press, 1996, ISBN 0875847471
2. Jeffries, R. E. "Extreme Testing – Why aggressive software development calls for radical testing efforts," Software Testing & Quality Engineering, April 1999
3. Cockburn, A. "Balancing Lightness with Sufficiency," American Programmer Editorial Guidelines, Online at http://members.aol.com/acockburn/papers/barelysufficient.htm
4. Beck, K. "Simple Smalltalk Testing: With Patterns," Online at http://www.xprograming.com/testfram.htm