Implications of Test-Driven Development A Pilot Study

Reid Kaufmann
Sun Microsystems, Inc.
3450 N. Rock Rd. Suite 401
Wichita, KS 67226
316-315-0382 ext. 228
reid.kaufmann@sun.com

David Janzen
Bethel College
300 E. 27th St.
North Newton, KS 67117
316-284-5259
djanzen@bethelks.edu

ABSTRACT
A Spring 2003 experiment examines the claims that test-driven development or test-first programming improves software quality and programmer confidence. The results indicate support for these claims and inform larger future experiments.

Categories and Subject Descriptors
D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *object-oriented programming.*

General Terms
Management, Measurement, Design, Experimentation.

Keywords
Test-Driven Development, Agile Development, Software Quality

1. INTRODUCTION
Many software professionals such as [1] and [2] advocate test-first programming, claiming it's benefits include faster debugging, greater reliability, increased confidence, and superior design. Test-first programming, or test-driven development, is included as a best practice in agile software development methodologies such as Extreme Programming. Although Extreme Programming has attracted widespread attention, it has not been universally adopted. Much of the evidence supporting the best practices is merely anecdotal. In Spring 2003, a small pilot experiment was conducted at Bethel College to study the effects of test-first programming on software quality, programmer productivity, and programmer confidence. The experiment is described along with results and potential future experiments.

2. THE EXPERIMENT
Two groups of four students each served as the test subjects. These students varied in classification from sophomore to senior, but all were computer science majors having completed at least two programming courses with C++ as the primary language. One group used test-first programming and the other served as a control group by practicing test-last programming.

The students were enrolled in an elective course titled "Software Studio." The objective of the course was to give students an opportunity to gain experience programming cooperatively in a small group on a longer, more sustained project than those completed in the first two programming courses.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

The groups and projects were self-selected. Both groups used the Java programming language to develop graphical game applications. Group 1 chose to write an adventure game that focused on a character advancing through a predetermined plot. This group was the control group, which did not use test-first programming. Group 2, which volunteered to try test-first programming, chose to write an action game in which stick figures fight each other [3].

Since no one in the class had any experience writing games in the Java programming language, the first several weeks of the semester were spent learning how to use the graphics APIs and how to obtain input from the mouse or keyboard. During this period, the students were not expected to write tests, since they were not producing "production" code.

3. METRICS
Design quality and programmer productivity were measured with the internal variables [4] class size, functions per class, McCabe's Cyclomatic Complexity, fan-in, fan-out, information flow, and non-commented lines of code. Several metrics tools were employed to collect metrics including JavaNCSS [5], JDepend [6], JMetric [7], and CCCC [8].

A snapshot of the production code was gathered on three dates: April 10, May 6, and May 9 from each of the groups. Each snapshot was analyzed with all four metrics tools.

Programmer confidence was measured through a survey administered to all students at the end of the semester. Defect rate was not directly measured due to the short time frame of the experiment and the lack of concrete external requirements with which to identify defects. It should be noted however that the test-last group's second code snapshot did not compile.

4. RESULTS AND OBSERVATIONS
Lines of code and observable functionality indicated that the test-first group was considerably more productive. The test-first group produced 50% more code than the test-last group as measured by non-commented lines of code. The cyclomatic complexity numbers [9] indicated that the code produced by both groups was of similar complexity, giving more credibility to using program size as a measure of productivity. Also, it is arguable that the test-first group took on a more complex problem, since their game involved real-time interaction between two players.

Measuring design quality requires the evaluation of numerous metrics. Metrics often are considered to have a threshold for normal values and then anything outside of that range would

cause an “alarm” [10]. A slight variation of this is to designate “three ranges for each metric: safe, flag, and alarm” [4]. This is the method used in CCCC.

Among the available metrics that pertain to design quality are the structural metrics and object-oriented design metrics from CCCC and perhaps cyclomatic complexity. The CCCC metrics indicated that the test-last group code contained a class with more than twice the information flow measure (square of fan-in and fan-out) of any other module between either project. This may be an indicator of a possible design problem and may be a result of the overburdening of a class [11]. JDepend also would have applicable metrics for large projects, but there were not meaningful differences between the two projects.

Using statistics from JavaNCSS, trends in both groups were observed regarding the organization of their code in objects and functions. While both groups were within reasonably safe bounds, it is important to notice how many functions there are per object. If one object is overburdened with too many functions it is a sign of poor design. The test-last group showed a possibly problematic trend of reducing the number of classes with time as the number of functions increased (and as the amount of source code increased).

To evaluate programmer confidence, a survey was given to each member of both groups. The self-reported confidence in their projects' functionality as of May 9th differed. On a scale of 1 to 5 with 5 being the most confident, the test-last group averaged 2.5 while the test-first group averaged 4.75. Also on a 5-point scale, the test-first group was asked how much they thought test-first programming helped with their debugging and design. With an average response of 4.25, they affirmed advantages of test-first programming.

There are clearly problems with this pilot experiment. Obviously the sample and project sizes are both too small. The context is unrealistic because the students knew they would not likely have to maintain their code over time. Neither group completed a sufficient number of tests. Even the test-first group only wrote 16 JUnit assertions.

It is likely that the test-first group’s increased productivity and confidence, as well as the superiority in their project's design and functionality, was a result of greater programming experience. The test-first group had one senior, one junior, and two sophomores, while the test-last group had two juniors and two sophomores.

The average final grade in the Programming 2 class for the test-first group was a whole letter grade higher than the average grade of the test-last group. This indicates a better mastery of programming. In a Java skills and core knowledge test constructed for this experiment, the test-first group scored higher on average than the test-last group. Most of these problems were anticipated prior to conducting this experiment. It is often very difficult to control all variables. In this case, the context of the academic course with set objectives limited the constraints of the experiment.

Since the test-first group demonstrated more programming experience and did not adhere to test-first programming faithfully, it would be completely inappropriate to conclude that test-first programming was the reason their project was more successful than the test-last group. To make this experiment successful, a larger sample population of equally skilled programming groups working on identical projects is needed. Supervision and feedback on the creation of tests would also be helpful so that each group's project could be judged on a more equal basis.

5. REFERENCES

[1] Beck, Kent. Extreme Programming Explained. Addison Wesley Longman, Inc., Reading MS, 2000.

[2] Martin, Robert Cecil. Agile Software Development. Prentice Hall, Inc., Upper Saddle River NJ, 2003.

[3] Stick Figure Karate http://sourceforge.net/projects/sfkarate/

[4] Henderson-Sellers, Brian. Object-oriented Metrics: Measures of Complexity. Prentice Hall, Inc., Upper Saddle River NJ, 1996.

[5] JavaNCSS http://www.kclee.com/clemens/java/javancss/.

[6] JDepend http://www.clarkware.com/software/JDepend.html.

[7] JMetric http://www.it.swin.edu.au/projects/jmetric/default.htm.

[8] CCCC http://cccc.sourceforge.net/.

[9] McCabe, T. J., A complexity measure. IEEE Transactions on Software Engineering. vol. 2. (December 1976), 308-320.

[10] Kitchenham, B. A., S. J. Linkman. Design metrics in practice. Information and Software Technology. vol. 32 no. 4. (1990), 304-310.

[11] Card, David N., Robert L. Glass. Measuring Software Design Quality. Prentice Hall, Inc., Englewood Cliffs NJ, 1990.