Closing the Defect Reduction Gap Between Software Inspection and Test-Driven Development: Applying Mutation Analysis to Iterative Test-First Programming

Jerod Wilkerson
University of Arizona

Follow this and additional works at: http://aisel.aisnet.org/amcis2007

Recommended Citation
http://aisel.aisnet.org/amcis2007/389

This material is brought to you by the Americas Conference on Information Systems (AMCIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in AMCIS 2007 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.
CLOSING THE DEFECT REDUCTION GAP BETWEEN SOFTWARE INSPECTION AND TEST-DRIVEN DEVELOPMENT: APPLYING MUTATION ANALYSIS TO ITERATIVE TEST-FIRST PROGRAMMING

Jerod W. Wilkerson
University of Arizona
jwilkers@email.arizona.edu

Abstract
After demonstrating the existence of a defect reduction gap between software inspection and test-driven development (TDD), this dissertation will present and evaluate a method of applying mutation analysis to iterative, test-first (ITF) programming methods such as TDD. The application of mutation analysis will add important stopping criteria to ITF programming, while capitalizing on the iterative and incremental nature of ITF to improve the performance of mutation analysis. The application of mutation analysis to ITF should provide a method for individual software developers to gain many of the software defect reduction benefits of software inspection without the expense, organizational change, and management support required to successfully implement software inspection. The research will be conducted using a design science approach to create and evaluate two new artifacts: an enhanced test-driven development method and a prototype mutation analysis system designed to capitalize on the iterative and incremental nature of ITF.

Keywords: Software Inspection, Test-Driven Development, Mutation Analysis, Unit Testing, Defect

Introduction
All software development organizations face the difficult problem of producing high-quality, low-defect software on-time and on-budget. A recent government study (2002) estimated that software defects are costing the U.S. economy approximately $59.5 billion per year (about 0.6 percent of the total annual U.S. Gross Domestic Product). This dissertation will assist in addressing this epidemic of software defects and budget and schedule overruns by comparing the software defect rates and relative costs of two methods of software defect reduction: software inspection and test-driven development (TDD), and by exploring and analyzing a method of applying mutation analysis to iterative test-first programming methods, such as TDD, to improve defect reduction.

Software inspection has been the focus of over 400 academic research papers since its introduction by Michael Fagan (1976). Software inspection is a formal method of inspecting code (and other software artifacts) to identify defects. This method has been in use for over 30 years and has been found to be very effective at reducing software defects. Fagan reported software defect reduction rates between 66 and 82% (1976). However, software inspection is expensive and is often not used because of a lack of support from the programmers whose code is to be inspected.

TDD is an iterative, test-first (ITF) programming practice in which programmers write unit tests before program code. New tests are written before features are added or changed, and the new features or changes are considered complete only when
the new tests and any previously written tests succeed. Although the results have been mixed, some research has shown that TDD can reduce software defects by between 18 and 50% (George et al. 2004; Maximilien et al. 2003), with the added benefit of eliminating defects at an earlier stage of development than software inspection.

Although the software development industry has been adopting TDD in recent years as part of the Extreme Programming software development methodology, no study prior to the start of this dissertation had compared the relative costs and benefits of these methods. An initial study conducted as part of this dissertation used a quasi-experimental design in a university laboratory environment to compare the defect reduction benefits and initial development costs of software inspection and TDD. The study found that software inspection is significantly more effective at reducing software defects than TDD, but that it is also significantly more expensive. This added expense, combined with a reluctance of many software developers to have their code subjected to software inspection, highlights the desirability of improving unit-testing based methods (such as TDD) to obtain the defect reduction benefits of software inspection with lower cost and with increased support of the development staff.

Mutation analysis is a technique that has shown promise in improving the software defect reduction benefits of unit testing. Mutation analysis involves the automated creation of several slightly modified versions of a computer program, and execution of these modified versions against a working automated test suite as a means of evaluating the completeness and correctness of the tests. Modified versions of a program that do not result in test failures highlight potential missing or incorrect test cases—and often software defects that were undetected by the existing test suite.

Mutation analysis is inexpensive from a human labor point-of-view, so it has the potential to improve the defect reduction performance of TDD and other unit testing based techniques while maintaining an initial development cost advantage over software inspection. However, the large number of modified versions of a program required by the approach, shown experimentally to be a quadratic function of the number of lines of code under test (Offutt et al. 1996), has resulted in performance issues that have prevented the widespread application of mutation analysis in industrial software development practice.

After demonstrating the existence of a defect reduction gap between software inspection and test-driven development, this dissertation will present and evaluate a method of applying mutation analysis to ITF programming methods, capitalizing on features of ITF to improve the performance of mutation analysis by reducing the number of mutations required to be generated and executed during any ITF iteration. The application of mutation analysis to ITF should provide a method for individual software developers to gain many of the software defect reduction benefits of software inspection without the expense, organizational change, and management support required to successfully implement software inspection. Applying mutation analysis to ITF will also add an important method of determining test adequacy in ITF, providing software developers with an improved method of determining when to consider a given iteration of the ITF process complete.

**Literature Review**

Due to space limitations, this literature review does not include literature relating to the already completed comparison of software inspection and TDD, and only a brief review of TDD and test adequacy criteria is included.

**Test-Driven Development**

TDD is the most widespread and well-known ITF programming practice. TDD is a software development practice that involves the writing of automated unit tests before program code, followed by coding which is deemed complete when the new tests and all previously written tests succeed. The following is a summary of the steps involved in TDD, derived from (Beck 2002, p. 84):

1. Pick some small piece of functionality to implement.
2. Create automated unit test(s) for the chosen functionality.
3. Run the new test(s) to verify that they fail (because the functionality they are intended to test has not been implemented).
4. Write code to get the test(s) to pass.
5. Refactor the new code as necessary to eliminate any duplication introduced in step 4.
6. Return to step 1.
Figure 1 provides an illustration of the steps. The light colored arrow between ‘Refactor’ and ‘Pick Functionality’ indicates that the process stops when there is no functionality left to be implemented.

Studies assessing the defect reduction benefits of TDD have produced mixed results, likely because of an under-specification of the method and the resulting difficulty in determining whether it is being properly implemented. Specific areas where additional specification of the method is needed include the selection of the functionality to be implemented in each iteration (step 1) and the determination of when an iteration should be deemed complete and a new iteration started (step 6).

**Test Adequacy Criteria**

Zhu et al. (1997) provide a detailed overview of several unit testing adequacy criteria useful in defining stopping criteria for unit testing of software systems. They define three main categories of adequacy criteria: 1) structural testing, which includes several variations of code coverage—statement coverage, branch coverage, path coverage, and several variations of each, 2) fault-based, with mutation analysis being the primary method in this category, and 3) error-based, which includes boundary testing and a few other methods.

**Mutation Analysis**

The initial ideas for mutation analysis were formulated by Richard Lipton during his graduate studies, and were first published in 1978 (DeMillo et al. 1978). Mutation analysis is an error-seeding approach of test adequacy measurement that automatically inserts errors into a program by generating slightly mutated versions of the code (called “mutants”). Mutants are generated with the use of mutation operators which specify rules for transforming the code to generate mutants. The adequacy measure used in mutation analysis to determine whether a test set is sufficient for testing a piece of software is the mutation adequacy score (MAS) defined by the formula: 

\[
\text{MAS} = \frac{D}{M - E},
\]

where D is the number of mutants detected by existing tests, M is the total number of mutants generated, and E is the number of equivalent mutants (mutants resulting in a program that is semantically equivalent to the original).

The mutation analysis method is based on assumptions from two underlying theories. The first is the Competent Programmer Hypothesis, which assumes that the program being analyzed was developed by a competent programmer and as a result, is close to being correct. If this assumption is true, most small deviations from the program (mutants) should be less correct than the original program, and should be detectible by automated tests. The second underlying theory is the Coupling Effect Hypothesis, which states that complex errors are coupled to simple errors, so test data that detects simple errors will also detect complex errors. This theory is used as the reasoning for limiting mutation analysis to first-order mutants, which are mutants that have been derived from the original program and not from another mutant. In an empirical evaluation of the Coupling Effect Hypothesis, Offutt (1992) demonstrated that a test set that was effective at identifying first-order mutants was also effective at identifying second-order mutants. However, it is not clear that second-order mutants are representative of the complex faults existing in a software system.

Mutation analysis requires the generation and execution of numerous mutated versions of the software under test. Offutt et al. (1996) showed experimentally that the number of mutants (generated from the set of mutation operators used in a majority of previous mutation analysis research) is a quadratic function of the number of lines of code in the program. As a result,
Applying Mutation Analysis to Iterative Test-First Programming

Much of the mutation analysis research has focused on improving the performance of the method. Untch (1995) defined the following three categories for classifying mutation analysis performance improvement strategies: do fewer, do smarter, and do faster.

Various strategies have been proposed and evaluated. Offutt et al. (1996) performed an experimental evaluation of N-Selective mutation—a ‘do fewer’ approach where the N operators that produce the most mutants are not used during mutation analysis to reduce computation time. They found in simulation testing involving 10 test programs that 6-Selective mutation (eliminating 6 of the 22 traditional mutation operators) resulted in only a .28% reduction in mutation analysis score (from 99.99% to 99.71%), and that while the number of mutants generated using the reduced set of operators was still quadratic as a function of program size, there was a substantial performance savings. They further concluded that 5 of the 22 operators are sufficient for mutation analysis, resulting in an average mutation analysis score over the 10 test programs of 99.5%, with the resulting computation cost being reduced to a linear function of the number of data references in the program.

Two primary ‘do faster’ approaches have been proposed for mutation analysis: mutant schemata, and compiler supported mutant patches. The concept of mutant schemata was presented by Untch et al. (1993). This method involves the automated creation of a single “meta-program” to represent all desired mutations of the original code. The use of a single program eliminates the need to compile multiple mutated versions of the code, and therefore, results in a cost savings during mutant generation. The meta-program is configured during each run of the test suite to represent a different mutated version of the original program. Although this is a promising approach to making mutation analysis practical for industrial use, it has not been implemented in any currently available version of a Java mutation analysis system.

DeMillo et al. (1991) presented the concept of using compiler generated patches to improve the efficiency of mutant generation. This method involves the modification of a programming language compiler to allow the compiler to create “patches” of object or assembly code instructions that can be applied by a patch applicator tool to a compiled version of a program to represent individual mutants. Although the method shows promise for improving mutation analysis performance, it has not been widely used because of the added complexity of modifying a compiler and the requirement that mutation analysis systems based on this approach cannot operate with a traditional compiler.

Several software systems have been developed to automatically create mutants, and to execute an existing test suite against those mutated program versions, reporting both the mutation adequacy score and lists of mutants that are and are not covered by existing tests. The most widely used mutation analysis system is Mothra (King et al. 1991), a research based system for analysis of Fortran 77 programs. Mothra is the mutation system used in a large majority of prior mutation analysis research. The two primary systems for mutation analysis of Java programs are MuJava (Offutt et al. 2004), and Jester (Moore 2001). MuJava supports the set of mutation operators found by Offutt et al. (1996) to be sufficient for mutation analysis, and is a Java version of Mothra. MuClipse (an Eclipse plugin version of MuJava) supports use of MuJava from the Eclipse integrated Java development environment using JUnit test cases. MuClipse represents an important evolution of the MuJava system because of the widespread usage of both Eclipse and JUnit. Although MuJava is useful for mutation analysis research, performance issues prevent this and other mutation systems from being heavily used in industrial software development practice.

Jester supports a very simplistic version of mutation analysis, but does not support the mutation operators found to be effective by the academic mutation analysis research community. Until the introduction of MuClipse, Jester was the only Java mutation analysis system that supported JUnit test cases.

Purpose and Research Objectives

The main purpose of this research is to use a design science research approach as described by Hevner et al. (2004) to improve the defect reduction performance of iterative, test-first programming methods (such as TDD), with an eventual goal of matching or exceeding the defect reduction capability of software inspection. As a first step in achieving this goal, this dissertation explores the application of mutation analysis to TDD. Beck (2002, p. 86) makes a brief reference to using

1 http://ise.gmu.edu/~ofut/mujava/
2 http://jester.sourceforge.net/
3 http://muclipse.sourceforge.net/
4 http://www.eclipse.org/
5 http://www.junit.org
mutation analysis with TDD using the Jester mutation analysis system. However, he does not address any of the issues related to how mutation analysis is best applied to TDD, or how to effectively use mutation analysis with its inherent performance issues as part of the rapid, iterative programming method that TDD is intended to be. Exploring these issues is one of the primary objectives of this dissertation.

The combination of TDD and mutation analysis should have a synergistic effect, with the iterative and incremental nature of TDD (and other ITF methods) providing a means to improve performance of mutation analysis, while the addition of mutation analysis to TDD should add a useful method of assessing test adequacy for each iteration of TDD. This assessment of test adequacy can be used by software developers to determine when an iteration should be deemed complete.

The following are the specific contributions to knowledge expected to be provided by this dissertation:

1. A comparison of the software defect reduction benefits and initial development costs of software inspection and TDD (a specific instantiation of an ITF programming method).
2. Explicit specification of a benchmark for evaluating and comparing ITF defect reduction capabilities.
3. Identification and evaluation of techniques to improve mutation analysis performance when used with an ITF programming method.
4. Addition of useful stopping criteria to be used with ITF programming methods.
5. Criteria for choosing functionality (requirements and design features) to be implemented in a given ITF iteration.

Two design science artifacts will be developed and evaluated as part of this research. The first is an ITF programming method that combines iterative, test-first programming with mutation analysis. This method will be described as an extension of TDD, the current leading ITF programming method. Figure 2 illustrates the extended TDD method. Comparing figure 2 with the illustration of the current TDD process in figure 1 shows that iteration stopping criteria has been added, and that an iteration continues in the extended process until the stopping criteria is satisfied. The second design science artifact will be a proof-of-concept mutation analysis system, developed as an extension of MuJava, to demonstrate the feasibility and expected benefits of the approach.

![Figure 2: Enhanced Test-Driven Development Steps](image)

**Research Approach**

The research approach for the first part of the dissertation (the comparison of software inspection to TDD) was a quasiexperiment using a two-by-two factorial design. This part of the dissertation has been completed, and is not described here. The research approach for the second part of the dissertation will be based on the principles of design science research.
described by Hevner et al. (2004). First, a proof-of-concept mutation analysis system will be developed as an extension to MuJava. The system will enable a 'do smarter' application of mutation analysis to ITF programming by adding the following features:

1. Ability to limit mutants to lines of code shown by statement coverage to be covered by automated tests.
2. Ability to limit mutants to branches shown by branch coverage to be covered by automated tests.
3. Ability to limit mutants to code that was either changed or is dependent upon code that was changed since the last ITF iteration.
4. Ability for users to mark both equivalent-mutants and live-mutants corrected during an ITF iteration.
5. Ability to track mutants between ITF iterations.
6. Ability to limit mutants to those not already marked as equivalent and those not previously reported and left uncorrected from a previous ITF iteration.

After development of the system, a simulation test will be performed to assess any cost savings from the new features. An analysis will also be performed to identify an effective method of using the system to reduce mutation analysis cost without significantly reducing the power of the mutation analysis method. The focus of this part of the analysis will be on identifying mutation and code coverage based stopping criteria for the iterations of TDD, and on providing guidance for how to choose the functionality to be implemented in each iteration to maximize the performance improvement of the new features without reducing the power of the mutation analysis method.

A usability study of the system and an experimental comparison of both the new system with MuJava, and the enhanced TDD method with software inspection and traditional TDD will be performed after completion of the dissertation.

References