Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development

Jerod W. Wilkerson, Jay F. Nunamaker Jr., and Rick Mercer

Abstract—This study is a quasi experiment comparing the software defect rates and implementation costs of two methods of software defect reduction: code inspection and test-driven development. We divided participants, consisting of junior and senior computer science students at a large Southwestern university, into four groups using a two-by-two, between-subjects, factorial design and asked them to complete the same programming assignment using either test-driven development, code inspection, both, or neither. We compared resulting defect counts and implementation costs across groups. We found that code inspection is more effective than test-driven development at reducing defects, but that code inspection is also more expensive. We also found that test-driven development was no more effective at reducing defects than traditional programming methods.

Index Terms—Agile programming, code inspections and walk throughs, reliability, test-driven development, testing strategies, empirical study.

1 Introduction

Software development organizations face the difficult problem of producing high-quality, low-defect software on time and on-budget. A US government study [1] estimated that software defects are costing the US economy approximately $59.5 billion per year. Several high-profile software failures have helped to focus attention on this problem, including the failure of the Los Angeles airport's air traffic control system in 2004, the Northeast power blackout in 2003 (with an estimated cost of $6-$10 billion), and two failed NASA Mars missions in 1999 and 2000 (with a combined cost of $320 million).

Various approaches to addressing this epidemic of budget and schedule overruns and software defects have been proposed in the academic literature and applied in software development practice. Two of these approaches are software inspection and test-driven development (TDD). Both have advantages and disadvantages, and both are capable of reducing software defects [2], [3], [4], [5].

Software inspection has been the focus of more than 400 academic research papers since its introduction by Fagan in 1976 [6]. Software inspection is a formal method of inspecting software artifacts to identify defects. This method has been in use for more than 30 years and has been found to be very effective at reducing software defects. Fagan reported software defect reduction rates between 66 and 82 percent [6].

While any software development artifact may be inspected, most of the software inspection literature deals with the inspection of program code. In this study, we limit inspections to program code, and we refer to these inspections as code inspections.

TDD is a relatively new software development practice in which unit tests are written before program code. New tests are written before features are added or changed, and new features or changes are generally considered complete only when the new tests and any previously written tests succeed. TDD usually involves the use of unit-testing frameworks (such as JUnit[1] for Java development) to support the development of automated unit tests and to allow tests to be executed frequently as new features or modifications are introduced. Although results have been mixed, some research has shown that TDD can reduce software defects by between 18 and 50 percent [2], [3], with one study showing a reduction of up to 91 percent [7], with the added benefit of eliminating defects at an earlier stage of development than code inspection.

TDD is normally described as a method of software design, and as such, has benefits that go beyond testing and defect reduction. However, in this study we limit our analysis to a comparison of the defect reduction benefits of the methods and do not consider other benefits of either approach.

Existing research does not sufficiently assess whether TDD is a useful supplement or a viable alternative to code inspection for purposes of reducing software defects. Previous research has compared the defect reduction benefits of code inspection and software testing—much of which is summarized by Runeson et al. [8]. However, the current high adoption rates of TDD indicate the timeliness and value of specific comparisons of code inspection and TDD. The focus

- *J.W. Wilkerson is with the Sam and Irene Black School of Business, Pennsylvania State University, Erie, 281 Burke Center, 5101 Jordan Road, Erie, PA 16563. E-mail: jww16@psu.edu.*
- *J.F. Nunamaker Jr. is with the Department of Management Information Systems, University of Arizona, 1130 E. Helen St., Tucson, AZ 85721. E-mail: jnunamaker@CMI.arizona.edu.*
- *R. Mercer is with the Department of Computer Science, University of Arizona, 1040 E. 4th St., Tucson, AZ 85721. E-mail: mercer@cs.arizona.edu.*

Manuscript received 11 Jan. 2010; revised 2 Aug. 2010; accepted 21 Dec. 2010; published online 14 Apr. 2011.
Recommended for acceptance by A.A. Porter.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-01-0011.
Digital Object Identifier no. 10.1109/TSE.2011.46.

1. http://www.junit.org.

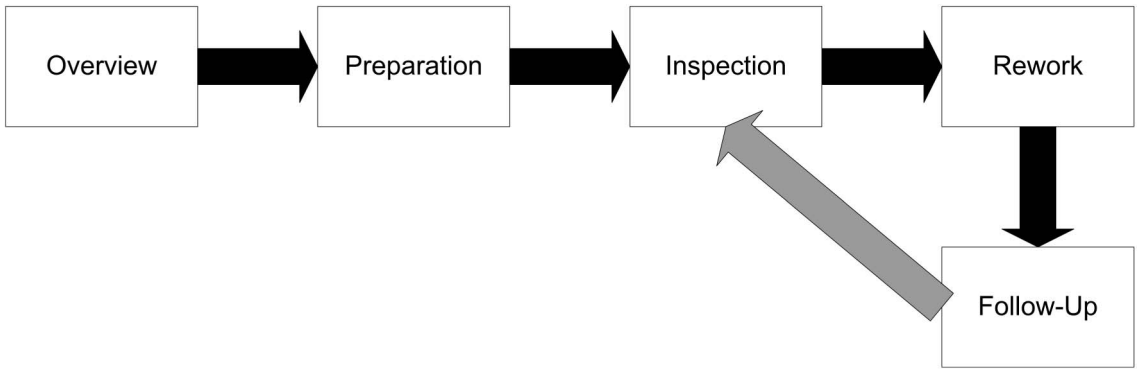


Fig. 1. Software inspection process.

of this study is a comparison of the defect rates and relative costs of these two methods of software defect reduction. In the study, we seek to answer the following two main questions:

1. *Which of these two methods is most effective at reducing software defects?*
2. *What are the relative costs of these software defect reduction methods*?

The software inspection literature typically uses the term "defect" to mean "fault." This literature contains a well-established practice of categorizing defects as either "major" or "minor" [9], [10]. In this paper, we compare the effectiveness of code inspection and TDD in removing "major defects."

The remainder of this paper is organized as follows: The next section discusses the relevant research related to this study. Section 3 describes the purpose and research questions. Section 4 describes the experiment and the procedures followed. Sections 5 and 6 describe the results and implications of the findings, and Section 7 concludes with a discussion of the study's contributions and limitations.

2 Related Research

Since software inspection and TDD have not previously been compared, we have divided the discussion of related work into two main sections: Software Inspection and Test-Driven Development. These sections provide a discussion of the prior research on each method that is relevant to the current study's comparison of methods. The Software Inspection section also includes a summary of findings from prior comparisons of code inspection and traditional testing methods.

2.1 Software Inspection

Fagan introduced the concept of formal software inspection in 1976 while working at IBM. His original techniques are still in widespread use and are commonly called "Fagan Inspections." Fagan Inspections can be used to inspect the software artifacts produced by all phases of a software development project. Inspection teams normally consist of three to five participants, including a moderator, the author of the artifact to be inspected, and one to three inspectors. The moderator may also participate in the inspection of the artifact. Fagan Inspections consist of the following phases: Overview (may be omitted for code inspections), Preparation, Inspection, Rework, and Follow-Up, as shown in Fig. 1. The gray arrow between Follow-Up and Inspection indicates that a reinspection is optional—at the moderator's discretion.

In the Overview phase, the author provides an overview of the area of the system being addressed by the inspection, followed by a detailed explanation of the artifact(s) to be inspected. Copies of the artifact(s) to be inspected and copies of other materials (such as requirements documents and design specifications) are distributed to inspection participants. During the Preparation phase, participants independently study the materials received during the Overview phase in preparation for the inspection meeting. During the Inspection phase, a reader explains the artifact being inspected, covering each piece of logic and every branch of code at least once. During the reading process, inspectors identify errors, which the moderator records. The author corrects the errors during the Rework phase and all corrections are verified during the Follow-Up phase. The Follow-Up phase may be either a reinspection of the artifact or a verification performed only by the moderator.

Fagan [6], reported defect yield rates between 66 and 82 percent, where the total number of defects in the product prior to inspection (t) is

$$t = i + a + u, \tag{1}$$

where "i" is the number of defects found by inspection, "a" is the number of defects found during acceptance testing, and "u" is the number of defects found during the first six months of use of the product. The defect detection (yield) rate (y) is

$$y = i/t * 100. \tag{2}$$

Two papers, [11], [12], summarize much of the existing software inspection literature, including variations in how software inspections are performed. Software inspection variations differ mainly in the reading technique used in the Inspection phase of the review. Reading techniques include Ad Hoc Reading [13], Checklist-Based Reading [6], [9], [10], [14], Reading by Stepwise Refinement [15], Usage-Based Reading [16], [17], [18], and Scenario (or Perspective)-Based Reading [19], [20], [21], [22]. Several comparison studies of reading techniques have also been performed [23], [24], [25], [26], [27].

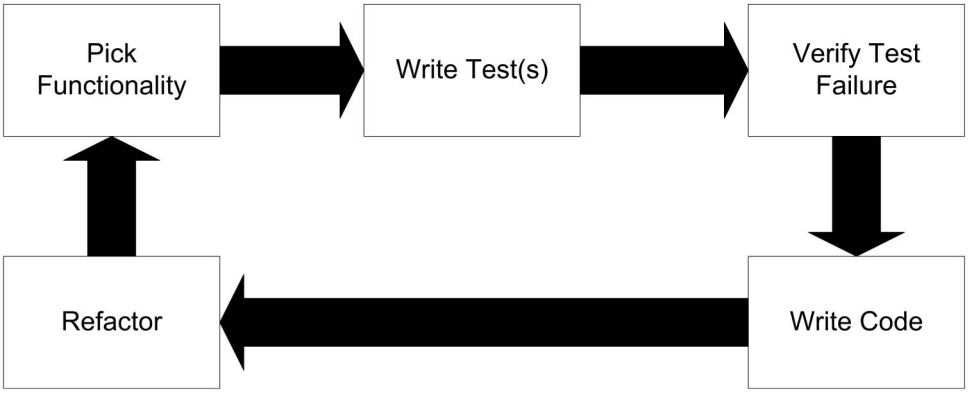


Fig. 2. Test-driven development process.

Porter and Votta [25] and Porter et al. [26] performed experiments comparing Ad Hoc Reading, Checklist-Based Reading, and Scenario-Based Reading for software requirements inspections using both student and professional inspectors. Inspectors using Ad Hoc Reading were not given specific guidelines to direct their search for defects. Inspectors using Checklist-Based Reading were given a checklist to guide their search for defects. Each Scenario-Based Reading inspector was given one of the following primary responsibilities to fulfill during the search for defects: 1) search for data type inconsistencies, 2) search for incorrect functionality, and 3) search for ambiguities or missing functionality. Porter et al. found that Scenario-Based Reading was the most effective reading method, producing improvements over both Ad Hoc Reading and Checklist-Based Reading from 21 to 38 percent for professional inspectors and from 35 to 51 percent for student inspectors. They attributed this improvement to efficiency gains resulting from a reduction in overlap between the types of defects for which each inspector was searching.

Another important advance in the state of software inspections was the application of group support systems (GSS) to the inspection process. Years of prior research have shown that the use of GSS can improve meeting efficiency. Efficiency improvements have been attributed to reductions in dominance of the meeting by one or a few participants, reductions in distractions associated with traditional meetings, improved group memory, and the ability to support distributed collaborative work [28], [29]. Johnson [30] notes that the application of GSS to software inspection can overcome obstacles encountered with paper-based inspections, thereby improving the efficiency of the inspection process. Van Genuchten et al. [31] also found that the benefits of GSS can be realized in code inspection meetings. Other studies [32], [33], [34], [35], [36] have also found improvements in the software inspection process as a result of GSS.

Several studies have compared code inspection with more traditional forms of testing. Runeson et al. [8] summarize nine studies comparing the effectiveness and efficiency of code inspection and software testing in finding code defects. They concluded that "the data doesn't support a scientific conclusion as to which technique is superior, but from a practical perspective it seems that testing is more effective than code inspections." Boehm [37] analyzed four studies comparing code inspection and unit testing, and found that code inspection is more effective and efficient at identifying up to 80 percent of code defects.

2.2 Test-Driven Development

TDD is a software development practice that involves the writing of automated unit tests before program code. The subsequent coding is deemed complete only when the new tests and all previously written tests succeed. TDD, as defined by Beck [38], consists of five steps. These steps, which are illustrated in Fig. 2, are completed iteratively until the software is complete. Successful execution of all unit tests is verified after both the "Write Code" and "Refactor" steps.

Prior studies have evaluated the effectiveness of TDD, and have obtained varied defect reduction results. Müller and Hagner [39] compared test-first programming to traditional programming in an experiment involving 19 university students. The researchers concluded that test-first programming did not increase program reliability or accelerate the development effort.

In a pair of studies by Maximilien and Williams [3] and George and Williams [2], the researchers found that TDD resulted in higher code quality when compared to traditional programming. Maximilien and Williams performed a case study at IBM on a software development team that developed a Java-based point-of-sale system. The team adopted TDD at the beginning of their project and produced 50 percent fewer defects than a more experienced IBM team that had previously developed a similar system using traditional development methods. Although the case study lacked the experimental control necessary to establish a causal relationship, the development team attributed their success to the use of the TDD approach. In another set of four case studies (one performed at IBM and three at Microsoft), the use of TDD resulted in between 39 and 91 percent fewer defects [7].

George and Williams [2] conducted a set of controlled experiments with 24 professional pair programmers. One group of pair programmers used a TDD approach while the other group used a traditional waterfall approach. The researchers found that the TDD group passed 18 percent more black-box tests and spent 16 percent more time developing the code than the traditional group. They also reported that the pairs who used a traditional waterfall

approach often did not write the required automated test cases at the end of the development cycle.

Erdogmus et al. [40] conducted an experiment designed to test a theory postulated in previous studies [2], [3]: that the cause of higher quality software associated with TDD is an increased number of automated unit tests written by programmers using TDD. They found that TDD does result in more tests, and that more tests result in higher productivity, but not higher quality.

In summary, some prior research has found TDD to be an effective defect reduction method [2], [3], [7] and some has not [39], [40]. Section 6 contains a potential explanation for this variability.

2.3 Summary

Code inspection is almost exclusively a software defect reduction method, whereas TDD has several purported benefits—only one of which is software defect reduction. We are primarily interested in comparing software defect reduction methods, so our focus is on the software defect reduction capabilities of the two methods and how these capabilities compare on defect reduction effectiveness and cost. One of the main differences between code inspection and TDD is the point in the software development process in which defects are identified and eliminated. Code inspection identifies defects at the end of a development cycle, allowing programmers to fix defects previously introduced, whereas TDD identifies and removes defects during the development process at the point in the process where the defects are introduced. Earlier elimination of defects is a benefit of TDD that can have significant cost savings [37]. It should be noted, however, that software inspection can be performed on analysis and design documents in addition to code, thereby moving the benefits of inspection to an earlier stage of software development.

3 Purpose and Research Questions

The purpose of this study is to compare the defect rates and relative costs of code inspection and TDD. Specifically, we seek to answer the following research questions:

1. Which software defect reduction method is most effective at reducing software defects?
2. Are there interaction effects associated with the combined use of these methods?
3. What are the relative costs of these software defect reduction methods?

The previously cited literature indicates that both methods can be effective at reducing software defects. However, TDD is a relatively new method, whereas code inspection has been refined through more than 30 years of research. Prior research has clearly defined the key factors involved with successfully implementing code inspection, such as optimal software review rates [9], [10], [14], [41] and inspector training requirements [10], whereas TDD is not as clearly defined due to its lack of maturity.

Currently, the defect reduction results for TDD have been mixed, with most reported reductions being below 50 percent [3]. Defect reduction from code inspection has consistently been reported at above 50 percent since Fagan's introduction of the method in 1976. This leads to a hypothesis that code inspection is more effective than TDD at reducing defects.

H1. *Code inspection is more effective than TDD at reducing software defects.*

Code inspection and TDD have fundamental differences that likely result in each method finding defects that the other method misses. With TDD, the same programmer who writes the unit tests also writes the code. Therefore, any misconceptions held by the programmer about the requirements of the system will result in the programmer writing incorrect tests and incorrect code to pass the tests. These "requirement misconception" defects are less likely in code that undergoes inspection because it is unlikely that all of the inspectors will have the same misconceptions about the requirements that the programmer has—especially if the requirements document has also been inspected for defects.

Although susceptible to requirement misconception defects, TDD encourages the writing of a large number of unit tests—some of which may test conditions inspectors overlook during the inspection process. This effect would likely be more noticeable when using inexperienced inspectors, but could occur with any inspectors. These differences between the methods indicate that each method will find defects that the other method misses. This leads to a hypothesis that the combined use of the methods is more effective than either method alone.

H2. *The combined use of code inspection and TDD is more effective than either method alone.*

The existing literature does not support a hypothesis as to which method has the lowest implementation cost. However, the nature of the cost differs between the two methods. The cost from TDD results from programmers spending additional time writing tests. The cost from code inspection results from both the time spent by the inspectors and the time spent by programmers correcting identified defects. These differences lead to a hypothesis that the methods differ in implementation cost—measured as the cost of developing software using that method of defect reduction.

H3. *Code inspection and TDD differ in implementation cost.*

4 Method

We evaluated the research questions in a quasi experiment using a two-by-two, between-subjects, factorial design. Participants in each research group independently completed a programming assignment according to the same specification using either inspection, TDD, both (Inspection+TDD), or neither. The two independent variables (factors) in the study are whether Inspection was used and whether TDD was used as part of the development method. The Inspection and Inspection+TDD groups constituted the Inspection factor and the TDD and Inspection+TDD groups constituted the TDD factor. The group that used neither TDD nor Inspection was the control group.

The programming assignment involved the creation of part of a spam filter using the Java programming language.

TABLE 1
Descriptive Statistics of
Noncommentary Source Statements Submitted by Group

Group	Mean	Std. Dev.	Min	Max
Control	581.00	59.42	491	689
TDD	519.67	54.78	463	619
Inspection	545.67	70.03	447	634
Inspection+TDD	580.14	87.31	451	735
Total	554.45	69.79	447	735

We gave the participants detailed specifications and some prewritten code and instructed them to use the Java API to read-in an XML configuration file containing the rules, allowed-list, and blocked-list for a spam filter. This information was to be represented with a set of Java objects. We required participants to maintain a specific API in the code to enable the use of prewritten JUnit tests as part of the defect counting process described in Section 4.3.1.

The final projects submitted by the participants contained an average of 554 noncommentary source statements (NCSS), including 261 NCSS that we provided to participants in a starter project. Table 1 provides descriptive statistics of the number of NCSS submitted, summarized by research group.

4.1 Participants

Participants were undergraduate (mostly junior or senior) computer science students from an object-oriented programming and design course at a large Southwestern US university. We invited all students from the class to participate in the study. Participation included taking a pretest to assess their Java programming and object-oriented design knowledge and permitting the use of data generated from their completion of a class assignment. All 58 students in the class agreed to participate but data were only collected for the 40 with the highest pretest scores. We entered the students who agreed to participate into a $100 cash drawing, but we did not compensate them in any other way. Because the programming assignment was required of all students in the class (whether they agreed to be research participants or not) and the assignment was a graded part of the class we recruited them from, we believe that they had a reasonably high motivation to perform well on the assignment.

TABLE 2
Reasons for Participant Exclusion

Reason for Exclusion	Number Excluded
Dropped the class for which the programming assignment was a requirement.	2
Failed to complete the programming assignment.	4
Cheated on the assignment (submitted the same solution as another participant).	2
Submitted code that was not testable.	3

TABLE 3
Participants Excluded by Group

Assigned Group	Number Excluded	Number Remaining
Control	3	7
TDD	1	9
Inspection	4	6
Both	3	7
Total	11	29

Each of the 40 participants was objectively assigned to one of the four research groups—with 10 participants assigned to each group—by a genetic algorithm that attempted to minimize the difference between the groups in both pretest average scores and standard deviation. The algorithm was very successful in producing equalized groups without researcher intervention in the grouping; however, several participants had to be excluded from the study for reasons described in Table 2. Table 3 shows the effect of these exclusions on group size. These exclusions resulted in unequal research groups, so we used pretest score as a control variable during data analysis.

4.2 Experimental Procedures

Before the start of the experiment, all participants received training on TDD and the use of JUnit through in-class lectures, a reading assignment, and a graded programming assignment. At the start of the experiment, we gave participants a detailed specification and instructed them to individually write Java code to satisfy the specification requirements. We gave participants two weeks to complete the project in two separate one-week iterations.

To maintain experimental control during this out-of-lab, multiple-week experiment, we analyzed the resulting code using MOSS[2]—an algorithm for detecting software plagiarism. We excluded two participants for suspected plagiarism, as shown in Table 2.

4.2.1 Software Inspection

All participants in the Inspection and Inspection+TDD groups had their code inspected by a single team of three inspectors. We then gave these participants one week to resolve the major defects found by inspection. We refer to these inspections as "Method Inspections." Inspectors were students but were not participants in the study. Inspections were performed according to Fagan's method [6], [9] with three exceptions. First, we did not invite authors to participate in the inspection process. The inspection process took two weeks to complete because of the large number of inspections performed, and inviting authors to the inspection meetings would have given authors whose code was inspected early in the process extra time to correct their defects. This would have been unfair to the students whose code was inspected later since the assignment was graded and included as part of their course grade. Inviting authors

2. http://theory.stanford.edu/aiken/moss.

to inspection meetings may also have increased the effect of inspection order on both dependent variables. As a result, the moderator also assumed the role of the authors in the inspection meetings.

Second, the inspectors used a collaborative inspection logging tool for both the inspection preparation and the meetings. Each inspector logged issues in the collaborative tool as the issues were found. This allowed the inspectors to see what had previously been logged and to spend their preparation time finding issues that had not already been found by another inspector. Use of the tool also allowed more time in the inspection meetings to find new issues rather than using the meeting time to report and log issues found during Preparation. The collaborative tool also reduced meeting distractions, allowing inspectors to remain focused on finding new issues [42].

Third, the inspectors used the Scenario-Based Reading approach described by Porter et al. [26]. We assigned each inspector a role of searching for one of the following defect types: missing functionality, incorrect functionality, or incorrect Java coding. We instructed the inspectors not to limit themselves to these roles, but to spend extra effort finding all of the defects that would fall within their assigned role.

We instructed the inspector who was assigned to search for incorrect Java coding to use the Java inspection checklist created by Fox [43] to guide the search for defects. Although we also made the checklist available to the other inspectors, searching for this type of defect was not their primary role. We annotated the checklist items that were most likely to uncover major defects with the words "Major" or "Possible Major" and instructed the inspectors to focus their efforts on these items in addition to their assigned role.

We gave inspectors 4 hours of training—approximately 1.5 hours on the inspection process and 2.5 hours on XML processing with Java (the subject matter under inspection). We instructed the inspectors to spend 1 hour preparing for each inspection, and we held inspection meetings to within a few minutes of 1 hour. We limited the number of inspection meetings to two per day to avoid reduced productivity due to fatigue, as noted by Fagan [9] and Gilb and Graham [10]. We also controlled the maximum inspection rate, which has long been known to be a critical factor in inspection effectiveness [9], [10], [14], [41]. Fagan recommends a maximum inspection rate of 125 noncommentary source statements per hour [9], whereas Humphrey recommends a maximum rate of 300 lines of code (LOC) per hour [14]. The mean inspection rate in this study was 180 NCSS/hour with a maximum rate of 395 NCSS/hour. Although the maximum rate was slightly above Humphrey's recommendation, this rate seems justified considering that the inspectors were inspecting multiple copies of code written to the same specification and, as a result, became very familiar with the subject matter of the inspections.

Inspectors categorized the issues they found as being either "major" or "minor" and we instructed them to focus their efforts on major issues, as recommended by Gilb and Graham [10]. After all inspections were completed, we gave each author an issue report containing all of the issues logged by the inspectors. We then gave the authors one week to resolve all major defects and to return the issue report with each defect categorized by the author into one of the following categories: Resolved, Ignored, Not a Defect, or Other. We required authors to write an explanation for any issue categorized as either "Not a Defect" or "Other."

4.2.2 Test-Driven Development

Prior to the start of this experiment, all participants were given classroom instruction on the use of JUnit and TDD. Formal instruction consisted of one 75-minute classroom lecture on JUnit and TDD. Participants were also shown in-class demos during other lectures that demonstrated test-first programming. All participants (and other students in the class) completed a one-week programming assignment prior to the start of the experiment in which they developed an interactive game using JUnit and TDD and had to submit both code and JUnit tests for grading.

We instructed participants in the TDD and Inspection+TDD groups to develop automated JUnit tests and program code iteratively while completing the programming assignment. We instructed them to write JUnit tests first when creating new functionality or modifying existing functionality, and to use the passage of the tests as an indication that the functionality was complete and correct. We also instructed participants in the Inspection+TDD group to use TDD during correction of the defects identified during inspection.

4.3 Measurement

Most research on defect reduction methods has reported "yield" as a measure of method effectiveness, where "yield" is the ratio of the number of defects found by the method to the total number of defects in the software artifact prior to inspection [6], [14]. However, "yield" cannot be reliably calculated for TDD because the TDD method eliminates defects at the point of introduction into the code, making it impossible to reliably count the number of defects eliminated by the method. Therefore, we used the number of defects remaining after application of the method as a substitute for "yield." We used the cost of development of the software using the assigned method, as a second dependent variable.

4.3.1 Defects Remaining

We defined the total number of defects remaining as the summation of the number of major defects found by code inspection after the application of the defect reduction method and the number of failed automated acceptance tests representing unique defects not found by inspection (out of 58 JUnit tests covering all requirements). This is consistent with the measure used by Fagan [6], with the exception of the exclusion of the number of defects identified during the first six months of actual use of the software.

We included code inspection as part of the defect counting procedure, even though code inspection was one of the two factors under investigation in the study, because a careful review of the literature indicates that code inspection has been the most heavily researched and widely accepted method of counting defects since Fagan's introduction of the method in 1976 [6]. To address a potential bias in favor of code inspection resulting from the use of

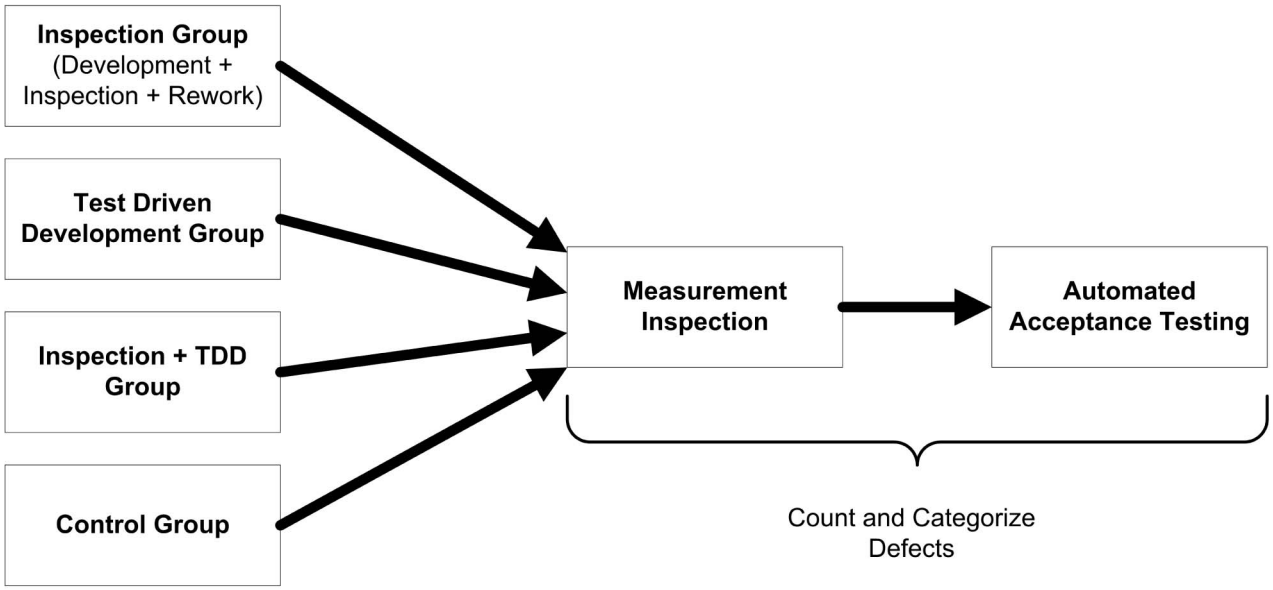


Fig. 3. Treatment conditions and defect counting.

code inspection as both a treatment condition and a defect counting method, we also report "defects remaining" results from acceptance testing (excluding counts from inspection) in a supplemental document, which can be found in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2011.46. The hypothesis testing results were the same in this case—hypothesis H1 is supported with both the combined method of defect counting and the "acceptance testing only" method.

Fig. 3 illustrates our defect counting procedure and how it relates to the treatment conditions tested in the study. The four boxes along the left edge of the diagram represent the four treatment conditions. These are the four software development methods under investigation in the study. For both the Inspection group and the Inspection+TDD group, a code inspection and subsequent rework to resolve the issues identified during the inspection was performed as part of the development method. Then each group, whether code inspection was part of the development method or not, was subjected to a separate code inspection as part of the defect counting procedure. We refer to the defect counting inspections as measurement inspections. A separate inspection team performed the measurement inspections to prevent bias in favor of the Inspection and Inspection+TDD groups. The same method was used for the measurement inspections as was used for the method inspections, except that only two inspectors in addition to the moderator were used due to resource constraints.

We executed the automated JUnit tests after completion of the measurement inspections and added to the defect counts, any test failures representing defects not already found by inspection. We wrote the automated tests before the start of the experiment and made minor adjustments and additions before the final test run.

One of the automated tests executed the code against a sample configuration file that we provided to the participants at the start of the experiment. The passage of this test, which we will refer to as the baseline test, indicates that the code executes correctly against a standard configuration file. We used the passage of this test to indicate that the code is "mostly correct." We wrote all other tests as modifications of this baseline test to identify specific defects. If the baseline test failed, we could not assume that the code was mostly correct, and therefore we could not rule out the possibility that some unexpected condition (other than what the tests were intended to check) was causing failures within the test suite. For example, if the baseline test failed, it could mean that the configuration file was not being read into the program correctly. This would result in almost all of the tests failing because the configuration file was not properly read and not because the code contained the defects the tests were intended to check. As a result, we only considered code to be testable if it passed the baseline test. We made minor changes to some projects to make the code testable, and in these cases, we logged each required change as a defect. Three participants submitted code that would have required extensive changes to make it testable according to the aforementioned definition. We could not be sure that these changes would not alter the author's original intent, so we excluded these participants from the study, as shown in Table 2.

Two adjustments to the resulting defect counts were necessary to arrive at the final number of defects remaining in the code. First, the inspection moderator performed an audit of defects identified by inspection and eliminated false positives. This would have resulted in an understatement of the effect of code inspection if the moderator inadvertently eliminated any real defects, making it less likely to find our reported result.

Second, in several cases, authors either ignored or attempted but did not correct defects that were identified by the method inspections. In an industrial setting, the inspection process would have included an iterative "Follow-Up" phase (see Fig. 1) for the purpose of catching these uncorrected defects and ensuring that they were

corrected before the inspection process completed. However, due to time and resource constraints, we could not allow iterative cycles of Follow-Up, Inspection, and Rework, to ensure that all identified major defects were eventually corrected.

In a well-functioning inspection process with experienced inspectors and an experienced inspection moderator, most (if not all) of these identified defects would be corrected by this process. In this type of process, the only identified defects that are likely to be left uncorrected are those for which a fix was attempted, and the fix appears to the inspector(s) performing the follow-up inspection to be correct, when in fact it is not correct. We assume that these cases are rare, and that we can therefore increase the accuracy of the results by subtracting any uncorrected defects from the final defect counts. However, because of uncertainty about how rare these cases are, we report results for both the adjusted and unadjusted defect counts. The unadjusted defect counts still include the elimination of false positives by the moderator.

4.3.2 Cost

We report the total cost of each method in total man hours. However, we also report man hours separately for inspectors and programmers in the code inspection case to allow for the application of these findings where programmer rates and inspector rates differ.

The total cost for the code inspection method is the sum of the original development hours spent by the author of the code, the hours spent by the software inspectors and moderator (preparation time plus meeting time), and the hours spent by the author correcting defects identified by inspection. The total cost for the TDD method is the sum of the development hours used to write both the automated tests and the code.

Author hours were self-reported in a spreadsheet that was submitted with the code. Inspector preparation hours were also self-reported. The inspection moderator tracked and recorded inspection meeting time. All hours were recorded in 15-minute increments.

4.4 Threats to Internal Validity

We considered the possibility that unexpected factors may have biased the results. We considered the following potential threats to internal validity:

1. selection bias,
2. mortality bias,
3. maturation bias,
4. order bias, and
5. implementation bias.

Selection bias refers to the possibility that participants in the study were divided unequally into research groups, and as a result, the findings are at least partly due to these differences and not to the effects of the treatment conditions. Although many differences in the participants could potentially contribute to a selection bias, Java programming ability seems to be the most likely cause of selection bias in this study. We accounted for this possibility by using a quasi-experimental design with participants assigned to groups by pretest score using the aforementioned genetic algorithm.

Mortality bias refers to the possibility that the groups became unequal after the start of the study as a result of participants either dropping out or being eliminated. We experienced a high mortality rate—starting with 40 participants and ending with 29. However, we used the pretest score to measure and control for this effect. We also used a T-Test to compare the pretest means of those who remained in the study (23.90) and those who did not (22.18) and found the difference not to be statistically significant at the 0.05 level of alpha.

Maturation bias is the result of participants learning at unequal rates within groups during the experiment. Due to the nature of the experiment, our results may include effects of a maturation bias. As a normal part of the inspection process, we gave participants in both the Inspection and Inspection+TDD groups an opportunity to correct defects identified in their code approximately two weeks after submitting the original code, but participants in the control and TDD groups did not have this opportunity. All participants were enrolled in an object-oriented programming and design course during the experiment and may have gained knowledge during any two-week period of the course that would have made them better programmers and less likely to produce defects. Only the participants whose code was inspected as part of the development method had an opportunity to use any knowledge gained to improve their code, and since this potential maturation effect was not measured, we are unable to eliminate or quantify the possible effects of a maturation bias.

Order bias is an effect resulting from the order in which treatments are applied to participants. This study is vulnerable to an order bias resulting from the order in which inspections were performed and whether inspections were the first or second inspection on the day of inspection. We controlled for order bias in two ways. First, we performed the measurement inspections in random order within blocks of four, and we performed the method inspections (which involved only two groups) on code from one randomly selected participant from each group each day, alternating each day on which group's inspection was performed first. Second, we used inspection day and whether the inspection was performed first or second on the day of inspection as control variables during data analysis.

Implementation bias is an effect resulting from variability in the way a treatment condition is implemented or applied. Failure to write unit tests before program code may result in an implementation bias in the application of TDD. We do not have an objective measure to indicate whether the unit tests submitted by participants in this study were written before the program code, so we are unable to eliminate the possibility of an implementation bias affecting our results. However, we used the Eclipse Plug-in of the Clover[3] test coverage tool to provide an objective measure of the effectiveness of the submitted tests. Code coverage results showed an average of 82.58 percent coverage (including both statement and branch coverage) with a standard deviation of 8.92. We also conducted a postexperiment survey in which participants were asked to rate their effectiveness in implementing TDD on a 5-point Likert

3. http://www.atlassian.com/software/clover/.

scale, with 5 being high and 1 being low. To encourage honest answers, we gave the survey when the classroom instructor was not present, and told the students that their instructor would not see the surveys. The mean response to this survey item was 3.63. Of the 16 students in one of the two research groups who performed TDD, two rated their effectiveness in implementing TDD as a 5, nine rated it as a 4, two rated it as a 3, and three rated it as a 2.

4.5 Threats to External Validity

We have identified the following four threats to external validity that limit our ability to generalize results to the software development community:

1. The participants in the study were undergraduate students rather than professional programmers, and therefore did not have the same level of programming knowledge or experience as the average professional programmer. We attempted to minimize this effect by choosing participants from a class consisting mostly of juniors and seniors, and by including only the students with the highest pretest scores in the study. However, the participants still were not representative of the general population of programmers, and most likely represent novice programmers. This would have affected all of the research groups, but since the TDD method was most dependent on the ability of the programmers, it most likely biased the study in favor of code inspection.
2. Although they were not participants in the study, the inspectors were college students and did not have professional code inspection experience. Prior research has shown a positive correlation between inspector experience and the number of defects found [9], [10], [44]. However, our result—that inspection is more effective than TDD—is robust to this potential bias, which would have had the effect of reducing the likelihood of finding inspection to be more effective.
3. The nature of the experiment required changes to the code inspection process from what would normally be done in industry. First, authors were not invited to participate in the inspections, as described in Section 4.2.1, and second, we did not use an iterative cycle of Rework, Follow-Up, and Reinspection as described in Section 4.3.1 to ensure that all identified defects were corrected. Not inviting authors to participate would have resulted in understating the effectiveness of code inspection. Section 4.3.1 discusses the potential effect of not including an iterative cycle of Rework, Follow-Up, and Reinspection.
4. The inspectors performed multiple inspections in a short period of time, of code that performs the same function and is written to the same specification. This was a necessary part of the experiment, but would rarely, if ever, occur in practice. This could have resulted in the inspectors finding more defects in later inspections as they became more familiar with the specifications and with Java-based XML processing code. However, if this affected the results, we should have detected an order bias during data analysis. Use of inspection order as a control variable did not indicate an order bias in the results.

5 Results

This study included two dependent variables: the total number of major defects (Total_Majors) and the total number of hours spent applying the method (Total_Method_Hours). We used Bartlett's test of sphericity to determine whether the dependent variables were sufficiently correlated to justify use of a single MANOVA, instead of multiple ANOVAs, for hypothesis testing. A p-value less than 0.001 indicates correlation sufficiently strong to justify use of a single MANOVA [45]. Bartlett's test yielded a p-value of 0.602, indicating that a MANOVA was not justified, so we proceeded with a separate ANOVA for each dependent variable.

5.1 Tests of Statistical Assumptions

Tests of normality (including visual inspection of a histogram and z-testing for skewness and kurtosis as described below) indicated that Total_Method_Hours was not normally distributed, and Levene's test indicated Total_Method_Hours did not meet the homogeneity of variance assumption. These assumption violations were corrected by performing a square-root transformation. Throughout the remainder of this document, Total_Method_Hours refers to the transformed variable unless otherwise noted. Although hypothesis testing results are only presented for the transformed variable, the original variable produced the same hypothesis testing results. After performing the transformation, we used (3) and (4), as recommended by Hair et al. [46], to obtain Z values for testing the normality assumptions both within and across groups for both dependent variables, and found them to be normally distributed at the 0.05 level of alpha:

$$Z_{skewness} = \frac{skewness}{\sqrt{\frac{6}{N}}}, \tag{3}$$

$$Z_{kurtosis} = \frac{kurtosis}{\sqrt{\frac{24}{N}}}. \tag{4}$$

5.2 Defects Remaining

The Inspection+TDD and Inspection groups had the lowest mean number of defects remaining as shown in Table 4, whereas the TDD group had the highest.

Although the means reported in Table 4 provide useful insight into the relative effectiveness of each method on reducing defects, the numbers are biased as a result of both an unequal number of observations in the research groups, and the differing effects of programmer ability and inspection order on the groups. Searle et al. [47] introduced the concept of an "Estimated Marginal Mean" to correct for these biases by calculating a weighted average mean to account for different numbers of observations in the groups and by using the ANOVA model to adjust for the effect of covariates. The covariate adjustment is performed by inserting the average values of the covariates into the

TABLE 4
Descriptive Statistics of Defects Remaining by Group

Group	Mean	Std. Dev.	Min	Max
Inspection+TDD	9.71	5.99	3	19
Inspection	11.50	6.92	2	22
Control	14.43	8.52	2	23
TDD	15.11	5.13	7	25

(a)

Group	Mean	Std. Dev.	Min	Max
Inspection+TDD	12.29	7.70	4	24
Inspection	13.33	7.74	2	25
Control	14.43	8.52	2	23
TDD	15.11	5.13	7	25

(b)

(a) Adjusted for defects not fixed after method inspection, (b) Not adjusted for defects not fixed after method inspection.

model, thereby calculating a predicted value for the mean that would be expected if the covariates were equal at their mean values.

We used the pretest score as a covariate for programmer ability. We used two variables—inspection day order, and an indication of whether the inspection was performed first or second on the day of inspection—as covariates for inspection order. Table 5 shows the estimated marginal means of the number of defects remaining, in order from smallest to largest. The "Adjusted Estimated Marginal Mean" column includes the adjustment for defects not fixed after the method inspections. The main difference between the estimated marginal means and the simple means of Table 4 is that the estimated marginal mean of the TDD group is lower than the one for the control group. However, the ANOVA results described below indicate that the difference is not statistically significant.

We used ANOVA to test our hypotheses and as with the calculation for estimated marginal means, we used the pretest score to control for the effects of programmer ability, and measurement inspection order and whether the inspection was performed first or second on the day of inspection to control for the effects of inspection order. For hypothesis H1, which hypothesizes that code inspection is more effective than TDD at reducing software defects, we obtained different hypothesis testing results depending on whether we used the adjusted or the unadjusted defect counts as the dependent variable. The hypothesis is supported with the adjusted defect count variable, whereas it is not supported with the unadjusted variable.

TABLE 5
Estimated Marginal Means of Defects Remaining by Group

Group	Adjusted Estimated Marginal Mean	Unadjusted Estimated Marginal Mean
Inspection+TDD	8.21	10.65
Inspection	12.37	14.17
TDD	14.37	14.45
Control	16.15	16.20

For the adjusted defect count variable, we observed eta squared values of 0.190 for the effect of code inspection and 0.323 for the pretest score, indicating that code inspection and pretest score accounted for 19.0 and 32.3 percent, respectively, of the total variance in the number of defects remaining. Both the pretest score and whether Inspection was used as a defect reduction method were significant at the 0.05 level of alpha and both of these variables were negatively correlated with the number of defects. The use of TDD did not result in a statistically significant difference in the number of defects. However, this lack of significance may be the result of low observed statistical power for the effect of TDD, which was only 0.238. Table 6 summarizes the results of the ANOVA analysis on the adjusted defect count variable, with variables listed in order of significance.

For the unadjusted defect count variable, only the pretest score was found to be statistically significant, with a p-value of 0.003. The effects of inspection, TDD, and the interaction between inspection and TDD resulted in the following respective p-values: 0.119, 0.153, and 0.358. Therefore, based on the unadjusted defect counts, we would reject hypotheses H1 and H2.

To test hypothesis H2 (that the combined use of the two methods is more effective than either method alone) for the adjusted defect count variable, we performed one-tailed T-Tests with a Bonferroni adjustment for multiple

TABLE 6
ANOVA Summary for Adjusted Number of Defects Remaining

Source	Type III Sum of Squares	Degrees of Freedom	F	Significance (p-value)	Observed Power
Pre-Test Score	341.791	1	10.481	**0.002**	0.871
Inspection	167.930	1	5.150	**0.017**	0.583
TDD	55.327	1	1.697	0.103	0.238
TDD * Inspection	9.710	1	0.298	0.295	0.082
Measure Inspection Day Order	17.807	1	0.546	0.468	0.109
Measure Inspection Day	17.037	1	0.522	0.477	0.106

TABLE 7
Descriptive Statistics of Implementation Cost by Group

Group	Mean	Std. Dev.	Min	Max
TDD	11.01	5.96	6.00	24.25
Control	12.79	4.87	4.00	18.00
Inspection	20.88 [a]	5.56	13.50	30.00
Inspection+TDD	31.71 [a]	11.71	13.00	46.75

[a] includes 7 hours of inspection time—one hour of meeting preparation time for each of the three inspectors and one hour of inspection meeting time for the inspectors and the inspection moderator

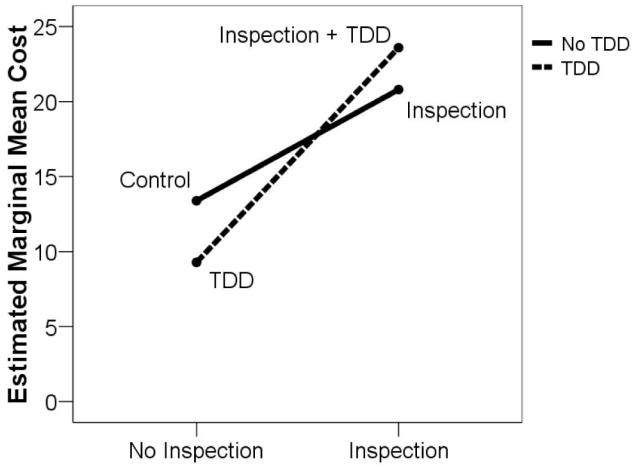


Fig. 4. Profile plot of square-root transformed estimated marginal mean implementation cost.

comparisons to compare the following two sets of means: 1) TDD versus Inspection+TDD and 2) Inspection versus Inspection+TDD. After the Bonferroni adjustment, a p-value of 0.025 was needed for both comparisons to support the hypothesis at the 0.05 level. The comparisons yielded p-values of 0.036 and 0.314, respectively, so we reject hypothesis H2.

A supplemental document, which can be found online, contains additional analysis of the number of defects remaining. This supplemental analysis includes descriptive statistics and ANOVA analysis of the number of defects found separately by the automated tests and by the measurement code inspection. The main result of the supplemental analysis is that hypothesis H1 is supported and H2 is not supported when using only automated testing to count defects. Neither hypothesis is supported when using only code inspection.

5.3 Implementation Cost

We performed an analysis of the implementation costs associated with TDD and code inspection but we did not explore the cost benefits of reducing software defects. Refer to Boehm [37] or Gilb and Graham [10] for in-depth treatment of cost savings associated with defect reduction. We measured cost in man-hours and found TDD to have the lowest mean cost and Inspection+TDD to have the highest, as shown in Table 7.

Fig. 4 presents a profile plot of the estimated marginal means. The solid line represents the two groups that did not use TDD, whereas the dashed line represents the two groups that did use TDD. The position of the points on the x-axis indicates whether the groups used code inspection. Following each line from left to right shows the cost effect of starting either with or without TDD and adding code inspection to the method used. Two important observations from the profile plot are that the use of TDD resulted in a cost savings, and that there is an interaction effect between the methods, as indicated by the fact that the lines cross. The interaction effect is supported by the ANOVA analysis at the 0.05 level of alpha as shown in Table 8, whereas the cost savings for TDD is not supported.

The finding of an interaction effect on cost between code inspection and TDD appears to be atheoretical. Further research is necessary to confirm, and if confirmed, to explain this effect. The lack of a finding of significance for the potential TDD cost savings is consistent with other TDD research. In our review of the TDD literature, we have not found reports of an initial development cost savings associated with TDD. We did, however, observe a low-statistical power of 0.107 for this effect, so if a cost savings is a real effect of TDD, we would not expect to have observed it in this study.

We used ANOVA to test hypothesis H3 (that implementation cost differs between the two methods). The ANOVA results are taken from the square-root transformed cost variable, although the original variable yielded the same hypothesis testing results. As with the test for the number of defects remaining, we used the pretest score as a control variable. We did not control for inspection order because the amount of time spent on inspections was held constant, leaving no opportunity for inspection order to affect cost. We obtained an eta squared of 0.517 for the effect

TABLE 8
ANOVA Summary for Implementation Cost

Source	Type III Sum of Squares	Degrees of Freedom	F	Significance (p-value)	Observed Power
Inspection	17.793	1	25.698	**0.000**	0.998
TDD * Inspection	2.988	1	4.315	**0.049**	0.514
Pre-Test Score	1.537	1	2.219	0.149	0.299
TDD	0.366	1	0.529	0.474	0.107

TABLE 9
Post Hoc Analysis for Implementation Cost by Group

Comparison	Mean Difference	Std. Error	Significance (p-value)
TDD vs. Inspection+TDD	-2.3098	0.42945	**0.000**
Control vs. Inspection+TDD	-2.0355	0.45550	**0.001**
TDD vs. Inspection	-1.3077	0.44913	**0.045**
Control vs. Inspection	-1.0334	0.47410	0.233
TDD vs. Control	-0.2742	0.42945	1.000

TABLE 10
Summary of Hypothesis Testing Results

Hypothesis	Description	Result
H1	Code inspection is more effective than TDD at reducing defects.	**Accepted** [a, b]
H2	The combined use of code inspection and TDD is more effective than either method alone.	**Not Accepted** [a]
H3	Code inspection and TDD differ in implementation cost.	**Accepted** [a, c]

[a] at the 0.05 level of alpha
[b] when defect counts are adjusted for defects found during method inspections but not corrected
[c] code inspection is more expensive

of code inspection—indicating that whether code inspection was used accounted for 51.7 percent of the total variance in implementation cost. The pretest score was not significant and had an eta squared of only 0.085. As stated above, we also found the use of TDD not to be significant. Table 8 presents a summary of these results, with variables listed in order of significance.

Post hoc analysis using Bonferroni's test indicates that the TDD group is significantly different from both the Inspection and Inspection+TDD groups, and that the control group is significantly different from the Inspection+TDD group—all at the 0.05 level of alpha. Table 9 summarizes these results, with comparisons listed in order of significance.

These results show that hypothesis H3 is supported, indicating that there is a cost difference between the use of code inspection and TDD, with the cost of code inspection being higher. Table 10 presents a summary of hypothesis testing results.

6 Discussion

The main result of this study is to provide support for the hypothesis that code inspection is more effective than TDD at reducing software defects, but that it is also more expensive. Resource constraints prevented us from implementing an iterative reinspection process, which resulted in some conflicting results in our finding that code inspection is more effective than TDD. As explained in Section 4.3.1, we presented two sets of results—one on a defect count variable that included an adjustment for uncorrected defects that should have been caught by an iterative reinspection process and one on a defect count variable that did not include the adjustment. Results from the adjusted defect count variable support the hypothesis that code inspection is more effective, whereas results from the unadjusted variable do not.

We believe that the adjusted defect count variable is more representative of what would be experienced in an industrial setting for two reasons. First, the programmers were students who, although motivated to do well on the programming assignment for a course grade, likely had a lower motivation to produce high-quality software than a professional programmer would have. We believe that few professional programmers would ignore defects found by inspection as some of our participants did, so the number of uncorrected defects would be lower than what we observed. Second, although it is impossible to be certain about which uncorrected defects would have been caught by an iterative reinspection process, we believe that most (if not all) of them would have been caught by an experienced inspection team with an experienced inspection moderator since verification of defect correction is the purpose of the reinspection step and software inspection has been found to be effective at reducing defects in numerous previously cited studies.

The supplemental analysis (which is available online) provides additional support for the hypothesis that code inspection is more effective than TDD at reducing defects. Here, we performed hypothesis testing on the defect counts obtained only by using the 58 JUnit tests described in Section 4.3.1 and found support for the hypothesis at the 0.01 level of alpha for the defect counts adjusted for uncorrected defects, and at the 0.1 level for the unadjusted defect counts.

This result is important because automated tests are less subjective than inspection-based defect counts. We performed the analysis in the main part of the study on the defect counts obtained from a combination of inspection and automated testing to be consistent with prior code inspection research and to avoid a potential bias in favor of TDD because of the relationship between TDD and automated testing. Therefore, the finding of code inspection being more effective when using only automated acceptance testing—in spite of this potential bias—provides strong support for the hypothesis. This support, however, is tempered by the fact that we did not find support for the hypothesis when using only the measurement inspections to count defects.

Another implication of this research is the finding that TDD did not significantly reduce the number of defects. Several possible explanations for this result exist. Low observed statistical power is one explanation. Another explanation, and the one that we believe accounts for the varying results summarized in Section 2.2 on the effectiveness of TDD as a defect reduction method, is that TDD is currently too loosely defined to produce reliable results that can confidently be compared with other

methods. A common definition of TDD is that it is a practice in which no program code is written until an automated unit test requires the code in order to succeed [38]. However, much variability is possible within this definition, and we believe it is this variability that accounts for the mixed results in the effectiveness of TDD as a defect reduction method. Additional research is necessary to add structure to TDD and to allow it to be reliably improved and compared to other methods.

7 Summary and Conclusions

We compared the software defect rates and implementation costs associated with two methods of software defect reduction: code inspection and test-driven development. Prior research has indicated that both methods are effective at reducing defects, but the methods had not previously been compared.

We found that code inspection is more effective than TDD at reducing defects, but that code inspection is also more expensive to implement. We also found some evidence to indicate that TDD may result in an implementation cost savings, although somewhat conflicting results require additional research to verify this result. Previous research has not shown a cost savings from TDD. The results did not show a statistically significant reduction in defects associated with the use of TDD, but results did show an interaction effect on cost between code inspection and TDD. We are currently unable to explain this effect. See Table 10 for a summary of hypothesis testing results.

These findings have the potential to significantly impact software development practice for both software developers and managers, but additional research is needed to validate these findings both inside and outside of a laboratory environment. Additional research is also needed to more clearly define TDD and to compare a more clearly defined version of TDD with code inspection.

Acknowledgments

The authors are grateful to Cenqua Pty. Ltd. for use of their Clover code coverage tool. They also thank the study participants and the inspectors for their time and effort, and Dr. Adam Porter for the use of materials from his software inspection experiments.

References

[1] G. Tassey, "The Economic Impact of Inadequate Infrastructure for Software Testing," technical report, Nat'l Inst. of Standards and Technology, 2002.

[2] B. George and L. Williams, "A Structured Experiment of Test-Driven Development," *Information and Software Technology,* vol. 46, no. 5, pp. 337-342, 2004.

[3] E.M. Maximilien and L. Williams, "Assessing Test-Driven Development at IBM," *Proc. 25th Int'l Conf. Software Eng.,* pp. 564-9, 2003.

[4] D.L. Parnas and M. Lawford, "The Role of Inspections in Software Quality Assurance," *IEEE Trans. Software Eng.,* vol. 29, no. 8, pp. 674-676, Aug. 2003.

[5] F. Shull, V.R. Basili, B.W. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What We Have Learned about Fighting Defects," *Proc. Eighth IEEE Symp. Software Metrics,* pp. 249-58, 2002.

[6] M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems J.,* vol. 15, no. 3, pp. 182-211, 1976.

[7] N. Nagappan, M.E. Maximilien, T. Bhat, and L. Williams, "Realizing Quality Improvement through Test Driven Development: Results and Experiences of Four Industrial Teams," *Empirical Software Eng.,* vol. 13, no. 3, pp. 289-302, 2008.

[8] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling, "What Do We Know about Defect Detection Methods?" *IEEE Software,* vol. 23, no. 3, pp. 82-90, May/June 2006.

[9] M.E. Fagan, "Advances in Software Inspections," *IEEE Trans. Software Eng.,* vol. 12, no. 7, pp. 744-51, July 1986.

[10] T. Gilb and D. Graham, *Software Inspection.* Addison-Wesley, 1993.

[11] O. Laitenberger and J.-M. DeBaud, "An Encompassing Life Cycle Centric Survey of Software Inspection," *J. Systems and Software,* vol. 50, no. 1, pp. 5-31, 2000.

[12] A. Aurum, H. Petersson, and C. Wohlin, "State-of-the-Art: Software Inspections after 25 Years," *Software Testing, Verification and Reliability,* vol. 12, no. 3, pp. 133-54, 2002.

[13] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski, "Software Inspections: An Effective Verification Process," *IEEE Software,* vol. 6, no. 3, pp. 31-36, May 1989.

[14] W.S. Humphrey, *A Discipline for Software Eng.,* ser. the SEI Series in Software Engineering. Addison-Wesley Publishing Company, 1995.

[15] R.C. Linger, "Cleanroom Software Engineering for Zero-Defect Software," *Proc. 15th Int'l Conf. Software Eng.,* pp. 2-13, 1993.

[16] T. Thelin, P. Runeson, and B. Regnell, "Usage-Based Reading—An Experiment to Guide Reviewers with Use Cases," *Information and Software Technology,* vol. 43, no. 15, pp. 925-38, 2001.

[17] T. Thelin, P. Runeson, and C. Wohlin, "An Experimental Comparison of Usage-Based and Checklist-Based Reading," *IEEE Trans. Software Eng.,* vol. 29, no. 8, pp. 687-704, Aug. 2003.

[18] T. Thelin, P. Runeson, C. Wohlin, T. Olsson, and C. Andersson, "Evaluation of Usage-Based Reading—Conclusions after Three Experiments," *Empirical Software Eng.,* vol. 9, nos. 1/2, pp. 77-110, 2004.

[19] V.R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M.V. Zelkowitz, "The Empirical Investigation of Perspective-Based Reading," *Empirical Software Eng.,* vol. 1, no. 2, pp. 133-64, 1996.

[20] C. Denger, M. Ciolkowski, and F. Lanubile, "Investigating the Active Guidance Factor in Reading Techniques for Defect Detection," *Proc. Third Int'l Symp. Empirical Software Eng.,* 2004.

[21] O. Laitenberger and J.-M. DeBaud, "Perspective-Based Reading of Code Documents at Robert Bosch GmbH," *Information and Software Technology,* vol. 39, no. 11, pp. 781-791, 1997.

[22] J. Miller, M. Wood, and M. Roper, "Further Experiences with Scenarios and Checklists," *Empirical Software Eng.,* vol. 3, no. 1, pp. 37-64, 1998.

[23] V.R. Basili, G. Caldiera, F. Lanubile, and F. Shull, "Studies on Reading Techniques," *Proc. 21st Ann. Software Eng. Workshop,* pp. 59-65, 1996.

[24] V.R. Basili and R.W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans. Software Eng.,* vol. 13, no. 12, pp. 1278-1296, Dec. 1987.

[25] A.A. Porter and L.G. Votta, "An Experiment to Assess Different Defect Detection Methods for Software Requirements Inspections," *Proc. 16th Int'l Conf. Software Eng.,* pp. 103-12, 1994.

[26] A.A. Porter, L.G. Votta Jr, and V.R. Basili, "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment," *IEEE Trans. Software Eng.,* vol. 21, no. 6, pp. 563-575, June 1995.

[27] F. Shull, F. Lanubile, and V.R. Basili, "Investigating Reading Techniques for Object-Oriented Framework Learning," *IEEE Trans. Software Eng.,* vol. 26, no. 11, pp. 1101-1118, Nov. 2000.

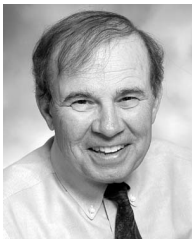
[28] J.F. Nunamaker Jr, R.O. Briggs, D.D. Mittleman, D.R. Vogel, and P.A. Balthazard, "Lessons from a Dozen Years of Group Support Systems Research: A Discussion of Lab and Field Findings," *J. Management Information Systems,* vol. 13, no. 3, pp. 163-207, 1997.

[29] J.F. Nunamaker Jr, A.R. Dennis, J.S. Valacich, D.R. Vogel, and J.F. George, "Electronic Meeting Systems to Support Group Work," *Comm. ACM,* vol. 34, no. 7, pp. 40-61, 1991.

[30] P.M. Johnson, "An Instrumented Approach to Improving Software Quality through Formal Technical Review," *Proc. 16th Int'l Conf. Software Eng.,* pp. 113-22, 1994.

[31] M. van Genuchten, C. van Dijk, H. Scholten, and D. Vogel, "Using Group Support Systems for Software Inspections," *IEEE Software,* vol. 18, no. 3, pp. 60-65, May/June 2001.
[32] S. Biffli, P. Grünbacher, and M. Halling, "A Family of Experiments to Investigate the Effects of Groupware for Software Inspection," *Automated Software Eng.,* vol. 13, no. 3, pp. 373-394, 2006.
[33] F. Lanubile, T. Mallardo, and F. Calefato, "Tool Support for Geographically Dispersed Inspection Teams," *Software Process: Improvement and Practice,* vol. 8, no. 4, pp. 217-231, 2003.
[34] C.K. Tyran and J.F. George, "Improving Software Inspections with Group Process Support," *Comm. ACM,* vol. 45, no. 9, pp. 87-92, 2002.
[35] M. van Genuchten, W. Cornelissen, and C. van Dijk, "Supporting Inspections with an Electronic Meeting System," *J. Management Information Systems,* vol. 14, no. 3, pp. 165-78, 1997.
[36] P. Vitharana and K. Ramamurthy, "Computer-Mediated Group Support, Anonymity, and the Software Inspection Process: An Empirical Investigation," *IEEE Trans. Software Eng.,* vol. 29, no. 2, pp. 167-80, Feb. 2003.
[37] B.W. Boehm, *Software Eng. Economics,* Prentice Hall, 1981.
[38] K. Beck, *Test Driven Development: By Example.* Addison-Wesley Professional, 2002.
[39] M.M. Müller and O. Hagner, "Experiment about Test-First Programming," *IEE Proc. Software,* vol. 149, no. 5, pp. 131-136, Oct. 2002.
[40] H. Erdogmus, M. Morisio, and M. Torchiano, "On the Effectiveness of the Test-First Approach to Programming," *IEEE Trans. Software Eng.,* vol. 31, no. 3, pp. 226-37, Mar. 2005.
[41] W.S. Humphrey, *Managing the Software Process,* ser. The SEI Series in Software Engineering. Addison-Wesley Publishing Company, 1989.
[42] T.L. Rodgers, D.L. Dean, and J.F. Nunamaker Jr, "Increasing Inspection Efficiency through Group Support Systems," *Proc. 37th Ann. Hawaii Int'l Conf. System Sciences,* 2004.
[43] C. Fox, "Java Inspection Checklist," 1999.
[44] R.G. Ebenau and S.H. Strauss, *Software Inspection Process,* ser. Systems Design and Implementation. McGraw Hill, 1994.
[45] L.S. Meyers, G. Gamst, and A.J. Guarino, *Applied Multivariate Research: Design and Interpretation.* Sage Publications, Inc., 2006.
[46] J.F. Hair, B. Black, B. Babin, R.E. Anderson, and R.L. Tatham, *Multivariate Data Analysis,* sixth ed. Prentice Hall, 2005.
[47] S.R. Searle, F.M. Speed, and G.A. Milliken, "Population Marginal Means in the Linear Model: An Alternative to Least Squares Means," *The Am. Statistician,* vol. 34, no. 4, pp. 216-221, 1980.

Jerod W. Wilkerson received the BS and MS degrees in accounting from Brigham Young University and the PhD degree in management information systems from the University of Arizona. Prior to receiving the PhD degree and joining the faculty at Pennsylvania State University, Erie, he spent several years in industry working as a software developer, a project and business manager, and a consultant. He founded and served as President of The Object Center—a consulting and training company focused on object technology and web development. His consulting and training clients have included the US Department of Defense, several state and local government agencies in Utah and Texas, and more than 20 business organizations—including Lockheed Martin, Raytheon Missile Systems, GMAC, J.P. Morgan Chase, and Iomega.



Jay F. Nunamaker Jr. received the BS and MS degrees in engineering from the University of Pittsburgh, the BS degree from Carnegie Mellon University and the PhD degree in operations research and systems engineering from Case Institute of Technology. He is the Regents and Soldwedel Professor of MIS, Computer Science, and Communication at the University of Arizona. He is a director of the Center for the Management of Information and the National Center for Border Security and Immigration at the University of Arizona. In a 2005 journal article in *Communications of the Association for Information Systems*, he was ranked as the fourth to the sixth most productive researcher for the period from 1991-2003. He was inducted into the Design Science Hall of Fame in May 2008. He received the LEO Award from the Association of Information Systems (AIS) at ICIS in Barcelona, Spain, in December 2002. This award is given for a lifetime of exceptional achievement in information systems. He was elected a fellow of the AIS in 2000. He was featured in the July 1997 Forbes Magazine issue on technology as one of eight key innovators in information technology. He received the professional engineer's license in 1965. He founded the MIS department at the University of Arizona in 1974 and served as its department head for 18 years.

Rick Mercer received the MS degree in computer science from the University of Idaho. He is currently a senior lecturer in the Department of Computer Science at the University of Arizona. He has served as an educator symposium chair for XP/Agile Universe 2004, OOPSLA 2006, and as cochair for ChiliPLoP 2005 through 2011. He is the author of six published textbooks targeted for the first year of the computer science degree and two free textbooks that integrate test-driven development into CS1 and CS2.