The Effect of Test-Driven Development on Code Quality and Programmer Productivity

Abstract.

Proponents of Test-Driven Development, introduced as part of the "Extreme Programming" movement, claim that it increases programmer productivity and code quality; however, the research community has failed to confirm these claims or even come to a consistent conclusion regarding the effect of Test-Driven Development on productivity and code quality. Differing experimental contexts may be shrouding the real conclusions of this research. Aggregating conclusions based on experimental context illuminates patterns that suggest that Test-Driven Development increases code quality at the expense of productivity for professional programmers in an industry context; however, further research utilizing appropriate statistical methods must occur to determine if these patterns are indicative of a real effect.

1. INTRODUCTION

Since its introduction with the advent of Extreme Programming, researchers have attempted to quantify the efficacy of Test-Driven Development (TDD), especially with respect to existing programming paradigms. They have especially focused on discovering how the use of TDD affects programmer productivity and code quality compared to test-last paradigms such as the waterfall methodology. Analyzing programming methodologies in this way enables managers of development teams to determine which methodology to employ to increase productivity, improve the quality and maintainability of code, and decrease the number of defects in order to decrease the cost of the project. Currently, the research community has failed to provide concrete conclusions to software managers regarding the efficacy of TDD compared to more traditional development methodologies, rendering them unable to make a decision based on an empirical analysis.

However, a careful analysis of a characteristic subset of research regarding TDD highlights how differences in experimental contexts may be shadowing real conclusions. By aggregating conclusions of research done in similar contexts, patterns emerge that support the hypothesis that TDD improves code quality at the expense of a moderate productivity decrease during the implementation of industry-level products. These suggestive patterns necessitate additional research that can provide statistically significant conclusions of this narrowed hypothesis.

The remainder of this article is organized as follows. Section 2 provides a background of TDD. Section 3 offers a short description of each study that will be examined. Section 4 analyzes the conclusions of those studies relating to productivity and code quality and attempts to reconcile the discrepancies discovered in the existing research. Finally, Section 5 summarizes the analysis and motivates additional research.

2. TEST-DRIVEN DEVELOPMENT

TDD is a programming paradigm that dictates that programmers should develop software using iterative cycles of feature development. The implementation of a single feature is called a TDD *cycle*, or just a *cycle*.

A cycle begins with developers writing a suite of failing automated unit-tests that serve as a specification for the feature's behavior. These unit-tests are frequently constructed using a platform such as JUnit to provide Integrated Development Environment (IDE) support. Next, developers write the minimal amount of code to ensure the newly constructed unit-tests pass. Finally, they refactor any code as necessary to remove duplication or improve the structure of the new code.

Over its lifetime, the project accumulates a large suite of unit-tests that can verify the correctness of the system. Because each cycle begins and ends with all unit-tests passing, TDD provides assurance that the addition of a new feature does not break any existing functionality. Proponents of TDD argue that this invariant results in higher quality code with a fewer number of defects and loosely coupled components. They also assert that TDD increases productivity because programmers spend less time fixing existing functionality and more time developing new features.

3. EXISTING RESEARCH

This section contains several subsections that briefly introduce a study, experiment, or analysis performed to examine the effects of TDD on productivity and code quality. Works are introduced in chronological order and will be referred to by the authors' names after this introduction.

3.1 George and Williams: "A Structured Experiment of Test-Driven Development"[1]

George and Williams hypothesized that utilizing TDD would result in an increase in code quality (measured in unit-tests passed) and productivity (measured in hours to complete development). They divided 24 professional programmers into two groups, one that utilized TDD and one that employed a "waterfall-like practice" [2] . Programmers in each group pair-programmed a small bowling game in Java. The project was minimally complex: the average solution contained roughly 200 Lines of Code (LOC), not including test code. George and Williams determined that TDD resulted in an 18% increase in code quality, but at the cost of a 16% decrease in productivity[3].

3.2 Erdogmus et al: "On the Effectiveness of the Test-First Approach to Programming"[4]

Erdogmus et al. recruited third-year computer science undergraduates to participate in a study. The students were split into a test-first group (TDD) and a test-last group and instructed to implement the same bowling game as used in George and Williams[5]. The experiment concluded that TDD does not result in higher quality code nor increased productivity. However, they were able to determine that writing more unit-tests results in higher quality code, regardless of whether the tests were written before the implementation[6]. Additionally, they showed that utilizing TDD results in more unit-tests written, suggesting that using TDD may improve code quality indirectly because more unit-tests are written[7].

3.3 Müeller and Höfer: "The Effect of Experience on the Test-Driven Development Process"[8]

Müeller and Höfer compared the use of TDD by novice and expert programmers. Both groups implemented an elevator-control system in Java using TDD, a task that took on average 3-4 hours. The participants were provided with a skeleton implementation (about 400 LOC) and test code that they used as a starting point. The average solution added 40 LOC and modified an additional 60[9]. Additionally, each participant programmed their solution in a modified IDE that provided the researchers with analytics regarding adherence to TDD principles. Using these metrics, Müeller and Höfer showed with statistical significance that the expert programmers wrote unit-tests with better coverage and better conformed to TDD principles than the novices[10].

3.4 Nagappan et al: "Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams"[11]

Nagappan et al. studied how TDD affected the quality of production systems developed by software teams at IBM and Microsoft. Each team consisted of 6-10 programmers and each product was a complex system, consisting of 6,000 to 155,000 LOC at initial deployment [12]. The researchers used defects per KLOC as the metric for code quality, where a defect was defined as a programming error that allows the product to enter an "erroneous state"[13]. The results for each team were compared against a team in the same organization developing a comparable product without TDD. The authors concluded that TDD had an astounding impact on code quality—they observed a 40-90% decrease in defects per KLOC at the expense of an estimated 15-35% decrease in productivity [14]. The productivity was not quantitatively established; rather, the researchers asked the team managers to estimate the increase in development time[15].

3.5 Dogša and Batič: "The Effectiveness of Test-Driven Development: An Industrial Case Study"[16]

Dogša and Batič studied three 12-person teams embarking on "comparable, medium-sized projects", only one of which employed TDD[17]. All three teams were tasked with implementing a simulator of a complex, but well specified component of a wireless communication system for the same customer. This component plays a critical role in the customer's development process, and any failures would have a direct impact on consumers utilizing the customer's system; therefore, the customer had an independent group perform acceptance testing[18].

The researchers measured code quality via several metrics: defect density (number of defects per KLOC), failed system and acceptance tests (%), and McCabe cyclomatic complexity. Defect density has been previously defined. System tests are black-box tests written and executed by the programmers to serve as a specification for the system's behavior. The independent group provided by the customer wrote and administered the acceptance tests, to which the teams did not have access. McCabe cyclomatic complexity is a count of all possible paths of execution of a program; a smaller McCabe complexity is an indication of simpler, and therefore better, code[19].

Dogša and Batič hypothesized that their findings would align with those of George and Williams—utilizing TDD would result in a higher code quality at the expense of programmer productivity. Informally, their findings supported their hypothesis. The team that utilized TDD had

the best code quality metrics in all three categories, but had a productivity loss of nearly 50% compared to the other teams[20]. However, these results cannot be generalized because of the small sample size and a lack of statistical analysis.

3.6 Pančur and Ciglarič: "Impact of Test-Driven Development on Productivity, Code, and Tests: A Controlled Experiment"[21]

Pančur and Ciglarič performed this experiment with the students of two consecutive iterations of a senior-level Distributed Systems undergraduate course. Students taking the class were taught the fundamentals of TDD and then divided into two groups. One group utilized TDD to implement course projects, and the other group used a test-last methodology. Both groups implemented the same set of projects, which were either completed individually or with a partner, depending on the project. For each project, the students were provided with a comprehensive suite of acceptance tests. The researchers examined code quality utilizing the McCabe cyclomatic complexity metric as well as the percentage of acceptance tests passing. They examined productivity by counting the number of features implemented per hour.

Pančur and Ciglarič concluded that there may be benefits of TDD relative to a test-last methodology, but if they exist, they are small and not very significant[22]. However, they were able to show with statistical significance that TDD does result in higher branch coverage of unit-tests, indicating TDD improves the quality of unit-tests[23].

4. AGGREGATION OF CONCLUSIONS

The presented research has come to two different conclusions regarding the benefits of TDD:

1. TDD provides no benefit to code quality or productivity relative to other programming methodologies.
2. TDD provides a clear, significant benefit to code quality at the expense of reduced productivity relative to other programming methodologies.

Therefore, at a minimum, one can safely conclude that TDD does not reduce the quality of code and may reduce productivity. To the manager of a software team, TDD does not look like a good option by this analysis. However, in some cases TDD appeared to provide huge quality improvements at the expense of slightly reduced productivity. In some contexts, this is an acceptable, or even desired, trade off. Therefore, more specific conclusions need to be made.

By recognizing that these studies vary with respect to the experience of the participants and the context in which the team worked, they can be divided into groups in an attempt to draw more meaningful conclusions.

4.1.1 Experience of Participants

George and Williams, Nagappan et al., and Dogša and Batič utilized professional or expert programmers in their research. Erdogmus et al. and Pančur and Ciglarič used undergraduate students as participants. The research by Müeller and Höfer represents an interesting third case, as they examined the differences in the utilization of TDD by both demographics.

The research using the professional demographic converges on a common conclusion—TDD results in higher quality of code with a reduction in productivity. However, their conclusions do not agree on the degree to which TDD impacts code quality or productivity. George and Williams reported an 18% increase in quality with a 16% decrease in productivity, Nagappan et al. discovered an improvement in quality of 40-

90% with a reduction of productivity of 15-35%, while Dogša and Batič merely asserted that a quality improvement and productivity reduction existed without attempting to quantify it meaningfully. Despite the fact that these studies agree with each other, their findings cannot be generalized as none of the studies have a sample size large enough to be statistically significant. However, their agreement does suggest that TDD employed by professional programmers *may* provide some benefit to code quality and *may* negatively impact productivity.

Interestingly, the experiments with the student demographic found no statistically significant difference in code quality or productivity when utilizing TDD, but they were able to draw other interesting conclusions regarding the tests themselves. Erdogmus et al. found a correlation between the number of unit-tests and code quality, regardless of the programming methodology employed. Pančur and Ciglarič found evidence that while TDD does not change code quality or productivity, it does improve the efficacy of the unit-tests themselves.

Müeller and Höfer's results can inform our interpretation of these two groups of research. Recall that their research concluded that professional programmers adhere to TDD principles better than student programmers. This suggests that one reason for the discrepancy between these two groups of research could be each demographics' use of TDD. It is possible student programmers' improper use of TDD erases its potential benefits. Anecdotal support for this hypothesis can be found by comparing the research by George and Williams and Erdogmus et al. Both these studies utilized the same, or at least a very similar, programming problem; however, the study that utilized professional programmers found a code quality benefit to TDD while the study that used student participants did not. This idea could be quantifiably supported if the research presented utilized the Müeller-Höfer method of collecting TDD adherence metrics. Without that data, however, this idea must remain a hypothesis.

4.1.2 Implementation Setting

Similar to the previous section, the research can be divided into groups depending on the setting in which the project was implemented. A *lab setting* is one in which the participants are given a problem to solve by the researchers. An *industry setting* is one in which the participants work on a team to solve a real-world problem for a customer. All of the research dealt with a lab setting except for Nagappan et al. and Dogša and Batič, which used an industry setting. Müeller and Höfer will be left out in this section, as they did not attempt to study code quality or productivity.

The research done in a lab setting disagreed in their conclusions. George and William concluded that TDD had a discernable effect on code quality and productivity; Erdogmus et al. found no differences attributed to TDD.

However, research done in an industry setting agreed that TDD provided a significant benefit to code quality with a reduction in productivity.

While it is hard to draw meaningful conclusions about the effect of TDD in a lab setting, TDD seems to have a large positive effect on code quality in an industry setting, although this observation is not supported by statistical analysis. One may conjecture that TDD may provide larger benefits to code quality in a team environment, where the project is too large for one single programmer to understand all aspects and components of the code base, but this cannot be known without future research.

5. CONCLUSIONS

Research to determine TDD's effect on code quality and programmer productivity has failed to come to a single conclusion. A representative sample of research in this area has shown two different conclusions: the utilization of TDD does not affect code quality or programmer productivity significantly, or it increases code quality at the expense of programmer productivity. However, a closer examination of the context in which research has been done allows us to divide the existing work into meaningful groups and analyze each group individually. This method discovered anecdotal evidence that professional programmers see significantly code quality benefits by the utilization of TDD at the expense of a moderate productivity decrease. Additionally, there is evidence, again anecdotal, to suggest that projects large enough to require a team of programmers may see larger effects of TDD. However, these thoughts will remain hypotheses until new research with a large enough sample size to be statistical significant can be performed. This will be a daunting task, as there are many confounding variables that will need to be somehow controlled. Nevertheless, this research will be necessary to provide the managers of software teams with a reliable analysis of the consequences of utilizing TDD.

---

[1] George, Boby, and Laurie Williams. "A Structured Experiment of Test-driven Development." *Information and Software Technology* 46.5 (2004): 337-42. Web.

[2] George, Boby, and Laurie Williams. p. 337.

[3] George, Boby, and Laurie Williams. p. 340.

[4] Erdogmus, H., M. Morisio, and M. Torchiano. "On the Effectiveness of the Test-first Approach to Programming." *IIEEE Trans. Software Eng. IEEE Transactions on Software Engineering* 31.3 (2005): 226-37. Web.

[5] Erdogmus et al. p. 228.

[6] Erdogmus et al. p. 233.

[7] Erdogmus et al. p. 236.

[8] Müller, Matthias M., and Andreas Höfer. "The Effect of Experience on the Test-driven Development Process." *Empirical Software Engineering Empir Software Eng* 12.6 (2007): 593-615. Web.

[9] Müller, Matthias M., and Andreas Höfer. p. 603.

[10] Müller, Matthias M., and Andreas Höfer. p. 606, 610.

[11] Nagappan, Nachiappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. "Realizing Quality Improvement through Test Driven Development: Results and Experiences of Four Industrial Teams." *Empirical Software Engineering Empir Software Eng* 13.3 (2008): 289-302. Web.

[12] Nagappan, Nachiappan et al. p. 295.

[13] Nagappan, Nachiappan et al. p. 297.

[14] Nagappan, Nachiappan et al. p. 297.

[15] Nagappan, Nachiappan et al. p. 298.

[16] Dogša, Tomaž, and David Batič. "The Effectiveness of Test-driven Development: An Industrial Case Study." *Software Qual J Software Quality Journal* 19.4 (2011): 643-61. Web.

[17] Dogša, Tomaž, and David Batič. p. 643.

[18] Dogša, Tomaž, and David Batič. p. 647.

[19] Dogša, Tomaž, and David Batič. p. 651.

[20] Dogša, Tomaž, and David Batič. p. 652-653.

[21] Pančur, Matjaž, and Mojca Ciglarič. "Impact of Test-driven Development on Productivity, Code

and Tests: A Controlled Experiment." *Information and Software Technology* 53.6 (2011): 557-73. Web.

[22] Pančur, Matjaž, and Mojca Ciglarič. p. 571.

[23] Pančur, Matjaž, and Mojca Ciglarič. p. 571.