REDIRECTIONS

Editor: **Tim Menzies**
North Carolina State University
tim@menzies.us

# What Do We (Really) Know about Test-Driven Development?

Itir Karac and Burak Turhan

**Call for Submissions**

Do you have a surprising result or industrial experience? Something that challenges decades of conventional thinking in software engineering? If so, email a one-paragraph synopsis to timm@ieee.org (use the subject line "REDIRECTIONS: Idea: your idea"). If that looks interesting, I'll ask you to submit a 1,000- to 2,400-word article (in which each figure or table counts as 250 words) for review for the Redirections department. Please note: heresies are more than welcome (if supported by well-reasoned industrial experiences, case studies, or other empirical results).—*Tim Menzies*

**TEST-DRIVEN DEVELOPMENT** (TDD) is one of the most controversial agile practices in terms of its impact on software quality and programmer productivity. After more than a decade's research, the jury is still out on its effectiveness. TDD promised all: increased quality and productivity, along with an emerging, clean design supported by the safety net of a growing library of tests. What's more, the recipe sounded surprisingly simple: Don't write code without a failing test.

Here, we revisit the evidence of the promises of TDD.[1] But, before we go on, just pause and think of an answer to the following core question: What is TDD?

Let us guess: your response is most likely along the lines of, "TDD is a practice in which you write tests before code." This emphasis on its test-first dynamic, strongly implied by the name, is perhaps the root of most, if not all, of the controversy about TDD. Unfortunately, it's a common misconception to use "TDD" and "test-first" interchangeably. Test-first is only one part of TDD. There are many other cogs in the system that potentially make TDD tick.

How about working on small tasks, keeping the red–green–refactor cycles short and steady, writing only the code necessary to pass a failing test, and refactoring? What if we told you that some of these cogs contribute more toward fulfilling the promises of TDD than the order of test implementation? (Hint: you should ask for evidence.)

## 15 Years of (Contradictory) Evidence

Back in 2003, when the software development paradigm started to change irrevocably (for the better?), Kent Beck posed a claim based on anecdotal evidence and paved the way for software engineering researchers:

> *No studies have categorically demonstrated the difference between TDD and any of the many*

0740-7459/18/$33.00 © 2018 IEEE

Table 1. Systematic literature reviews on test-driven development (TDD).

| Study | Overall conclusion for quality with TDD | Overall conclusion for productivity with TDD | Inconsistent results in the study categories |
|---|---|---|---|
| Bissi et al.[3] | Improvement | Inconclusive | Productivity:<br>Academic vs. industrial setting |
| Munir et al.[4] | Improvement or no difference | Degradation or no difference | Quality:<br>• Low vs. high rigor<br>• Low vs. high relevance<br>Productivity:<br>• Low vs. high rigor<br>• Low vs. high relevance |
| Rafique and Mišić[5] | Improvement | Inconclusive | Quality:<br>Waterfall vs. iterative test-last<br>Productivity:<br>• Waterfall vs. iterative test-last<br>• Academic vs. industrial |
| Turhan et al.[6] and Shull et al.[1] | Improvement | Inconclusive | Quality:<br>• Among controlled experiments<br>• Among studies with high rigor<br>Productivity:<br>• Among pilot studies<br>• Controlled experiments vs. industrial case studies<br>• Among studies with high rigor |
| Kollanus[7] | Improvement | Degradation | Quality:<br>• Among academic studies<br>• Among semi-industrial studies |
| Siniaalto[8] | Improvement | Inconclusive | Productivity:<br>• Among academic studies<br>• Among semi-industrial studies |

> *alternatives in quality, productivity, or fun. However, the anecdotal evidence is overwhelming, and the secondary effects are unmistakable.*[2]

Since then, numerous studies—for example, experiments and case studies—have investigated TDD's effectiveness. These studies are periodically synthesized in secondary studies (see Table 1), only to reveal contradictory results across the primary studies. This research has also demonstrated no consistent overall benefit from TDD, particularly for overall productivity and within subgroups for quality.

Why the inconsistent results? Besides the differences in the study contexts listed in Table 1, other likely reasons are that

- TDD has too many cogs,
- its effectiveness is highly influenced by the context (for example, the tasks at hand or skills of individuals),
- the cogs highly interact with each other, and
- most studies have focused on only the test-first aspect.

Identifying the inconsistencies' sources is important for designing further studies that control for those sources.

Matjaž Pančur and Mojca Ciglarič speculated that the results of studies showing TDD's superiority over a test-last approach were due to the fact that most of the experiments employed a coarse-grained test-last process closer to the waterfall approach as a control group.[9] This created a large differential in granularity between the treatments, and sometimes even a complete lack of tests in the control, resulting in unfair, misleading comparisons. In the end, TDD might perform better only when compared to a coarse-grained development process.

## Industry Adoption (or Lack Thereof)

Discussions on TDD are common and usually heated. But how common is the use of TDD in practice? Not very—at least, that's what the evidence suggests.

For example, after monitoring the development activity of 416 developers over more than 24,000 hours, researchers reported that the developers followed TDD in only 12 percent of the projects that claimed to use it.[10] We've observed similar patterns in our work with professional developers. Indeed, if it were possible to reanalyze all existing evidence considering this facet only, the shape of things might change significantly (for better or worse). We'll be the devil's advocate and ask, what if the anecdotal evidence from TDD enthusiasts is based on misconceived personal experience from non-TDD activities?

Similarly, a recent study analyzed a September 2015 snapshot of all the (Java) projects in GitHub.[11] Using heuristics for identifying TDD-like repositories, the researchers found that only 0.8 percent of the projects adhered to TDD protocol. Furthermore, comparing those projects to a control set, the study reported no difference between the two groups in terms of

- the commit velocity as a measure of productivity,
- the number of bug-fixing commits as an indicator of the number of defects, and
- the number of issues reported for the project as a predictor of quality.

Additionally, a comparison of the number of pull requests and the distribution of commits per author didn't indicate any effect on developer collaboration.

Adnan Causevic and his colleagues identified seven factors limiting TDD's use in the industry:[12]

- increased development time (productivity hits),
- insufficient TDD experience or knowledge,
- insufficient design,
- insufficient developer testing skills,
- insufficient adherence to TDD protocol,
- domain- and tool-specific limitations, and
- legacy code.

It's not surprising that three of these factors are related to the developers' capacity to follow TDD and their rigor in following it.

## What Really Makes TDD Tick?

A more refined look into TDD is concerned with not only the order in which production code and test code are written but also the average duration of development cycles, that duration's uniformity, and the refactoring effort. A recent study of 39 professionals reported that a steady rhythm of short development cycles was the primary reason for improved quality and productivity.[13] Indeed, the effect of test-first completely diminished when the effects of short and steady cycles were considered. These findings are consistent with earlier research demonstrating that TDD experts had much shorter and less variable cycle lengths than novices did.[14] The significance of short development cycles extends beyond TDD; Alistair Cockburn, in explaining the Elephant Carpaccio concept, states that "agile developers apply micro-, even nano-incremental development in their work."[15]

Another claim of Elephant Carpaccio, related to the TDD concept of working on small tasks, is that agile developers can deliver fast "not because we're so fast we can [develop] 100 times as fast as other people, but rather, we have trained ourselves to ask for end-user-visible functionality 100 times smaller than most other people."[15] To test this, we conducted experiments in which we controlled for the framing of task descriptions (finer-grained user stories versus coarser-grained generic descriptions). We observed that the type of task description and the task itself are significant factors affecting software quality in the context of TDD.

In short, working on small, well-defined tasks in short, steady development cycles has a more positive impact on quality and productivity than the order of test implementation.

## Deviations from the Test-First Mantra

Even if we consider the studies that focus on only the test-first nature of TDD, there's still the problem of conformance to the TDD process. TDD isn't a dichotomy in which you either religiously write tests first every time or always test after the fact. TDD is a continuous spectrum between these extremes, and developers tend to dynamically span this spectrum, adjusting the TDD process as needed. In industrial settings, time pressure, lack of discipline, and insufficient realization of TDD's benefits have been reported to cause developers to deviate from the process.[12]

To gain more insight, in an ethnographically informed study, researchers monitored and documented the TDD development process more closely by means of artifacts including audio recordings and notes.[16] They concluded that developers perceived implementation as the most important phase and didn't strictly follow the TDD process. In particular, developers wrote more production code than necessary, often omitted refactoring, and didn't keep test cases up to date in accordance with the progression of the production code. Even when the developers followed the test-first principle, they thought about how the production code (not necessarily the design) should be before they wrote the test for the next feature. In other words, perhaps we should simply name this phenomenon "code-driven testing"?

TDD's internal and external dynamics are more complex than the order in which tests are written. There's no convincing evidence that TDD consistently fares better than any other development method, at least those methods that are iterative. And enough evidence exists to question whether TDD fulfils its promises.

How do you decide whether and when to use TDD, then? And what about TDD's secondary effects?

As always, context is the key, and any potential benefit of TDD is likely not due to whatever order of writing tests and code developers follow. It makes sense to have realistic expectations rather than worship or discard TDD. Focus on the rhythm of development; for example, tackle small tasks in short, steady development cycles, rather than bother with the test order. Also, keep in mind that some tasks are better (suited) than others with respect to "TDD-bility."

This doesn't mean you should avoid trying TDD or stop using it. For example, if you think that TDD offers you the self-discipline to write tests for each small functionality, following the test-first principle will certainly prevent you from taking shortcuts that skip tests. In this case, there's value in sticking with the rule that implies not to write any production code without a failing unit test. However, you should primarily consider those tests' quality (without obsessing over coverage),[17] instead of fixating on whether you wrote them before the code. Although TDD does result in more tests,[1,6] the lack of attention to testing quality,[12] including maintainability and coevolution with production code,[16] could be alarming.

As long as you're aware of and comfortable with the potential tradeoff between productivity and testability and quality (perhaps paying off in the long term?), using TDD is fine. If you're simply having fun and feeling good while performing TDD without any significant drawbacks, that's also fine. After all, the evidence shows that happy developers are more productive and produce better code![18]

**Acknowledgments**

Academy of Finland Project 278354 partly supports this research.

**References**

1. F. Shull et al., "What Do We Know about Test-Driven Development?," *IEEE Software*, vol. 27, no. 6, pp. 16–19, 2010.
2. K. Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2003.
3. W. Bissi et al., "The Effects of Test Driven Development on Internal Quality, External Quality and Productivity: A Systematic Review," *Information and Software Technology*, June 2016, pp. 45–54.
4. H. Munir, M. Moayyed, and K. Petersen, "Considering Rigor and Relevance When Evaluating Test Driven Development: A Systematic Review," *Information and Software Technology*, vol. 56, no. 4, 2014, pp. 375–394.
5. Y. Rafique and V.B. Mišić, "The Effects of Test-Driven Development on External Quality and Productivity: A Meta-analysis," *IEEE Trans. Software Eng.*, vol. 39, no. 6, 2013, pp. 835–856; http://dx.doi.org/10.1109/TSE.2012.28.
6. B. Turhan et al., "How Effective Is Test-Driven Development?," *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, eds., O'Reilly Media, 2010, pp. 207–219.
7. S. Kollanus, "Test-Driven Development—Still a Promising Approach?," *Proc. 7th Int'l Conf. Quality of Information and Communications Technology* (QUATIC 10), 2010, pp. 403–408; http://dx.doi.org/10.1109/QUATIC.2010.73.
8. M. Siniaalto, "Test Driven Development: Empirical Body of Evidence," tech. report, Information Technology for European Advancement, 3 Mar. 2006.
9. M. Pančur and M. Ciglarič, "Impact of Test-Driven Development on Productivity, Code and Tests: A Controlled Experiment," *Information and Software Technology*, vol. 53, no. 6, 2011, pp. 557–573.
10. M. Beller et al., "When, How, and Why Developers (Do Not) Test in Their IDEs," *Proc. 10th Joint Meeting Foundations of Software Eng.* (ESEC/FSE 15), 2015, pp. 179–190; http://doi.acm.org/10.1145/2786805.2786843.
11. N.C. Borle et al., "Analyzing the Effects of Test Driven Development

## ABOUT THE AUTHORS

**ITIR KARAC** is a project researcher in the M-Group research group and a doctoral student in the Department of Information Processing Science at the University of Oulu. Contact her at itir.karac@oulu.fi.

**BURAK TURHAN** is a senior lecturer in Brunel University's Department of Computer Science and a professor of software engineering at the University of Oulu. Contact him at turhanb@computer.org.

in GitHub," *Empirical Software Eng.*, Nov. 2017.

12. A. Causevic, D. Sundmark, and S. Punnekkat, "Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review," *Proc. 4th IEEE Int'l Conf. Software Testing, Verification and Validation*, 2011, pp. 337–346.
13. D. Fucci et al., "A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?," *IEEE Trans. Software Eng.*, vol. 43, no. 7, 2017, pp. 597–614.
14. M.M. Müller and A. Höfer, "The Effect of Experience on the Test-Driven Development Process," *Empirical Software Eng.*, vol. 12, no. 6, 2007, pp. 593–615; https://doi.org/10.1007/s10664-007-9048-2.
15. A. Cockburn, "Elephant Carpaccio," blog; http://alistair.cockburn.us/Elephant+carpaccio.
16. S. Romano et al., "Findings from a Multi-method Study on Test-Driven Development," *Information and Software Technology*, Sept. 2017, pp. 64–77.
17. D. Bowes et al., "How Good Are My Tests?," *Proc. IEEE/ACM 8th Workshop Emerging Trends in Software Metrics* (WETSoM 17), 2017, pp. 9–14.
18. D. Graziotin et al., "What Happens When Software Developers Are (Un)happy," *J. Systems and Software*, June 2018, pp. 32–47.