Editor: Forrest Shull ■ Fraunhofer Center for Experimental Software Engineering, Maryland ■ fshull@fc-md.umd.edu

What Do We Know about Test-Driven Development?

Forrest Shull, Grigori Melnik, Burak Turhan, Lucas Layman, Madeline Diep, and Hakan Erdogmus

What if someone argued that one of your basic conceptions about how to develop software was misguided? What would it take to change your mind?

That's essentially the dilemma faced by advocates of test-driven development (TDD). The TDD paradigm argues that the basic cycle of developing code and then testing it to make sure it does what it's supposed to do—something drilled into most of us from the time we began learning software development—isn't the most effective approach. TDD replaces the traditional "code then test" cycle. First, you develop test cases for a small increment of functionality; then you write code that makes those tests run correctly. After each increment, you refactor the code to maintain code quality.[1]

TDD proponents assert that frequent, incremental testing not only improves the delivered code's quality but also generates a cleaner design. If you haven't already tried TDD, what data might convince you to try radically changing your software development approach to get those benefits? Would the experience of a recognized expert help?

In this column, we offer both data regarding TDD's effectiveness and the critique of an expert based on applying it in the field.

Compiling the Evidence

Our data comes from a study conducted by five of us—namely, Burak Turhan, Lucas Layman, Madeline Diep, Forrest Shull, and Hakan Erdogmus.[2] The study was based on a systematic literature review to aggregate demonstrated evidence about TDD's effectiveness. The review searched the literature from 1999, looking for any study that provided some quantitative assessment of TDD's effectiveness compared to traditional software development. The search results were filtered for quality, which left 22 published articles that described 33 unique studies.

The review distinguished three types of studies:

- *Controlled experiments* compared TDD to traditional development under controlled conditions to minimize the effects of confounding factors, such as developer experience or the type of software being developed.
- *Pilot studies* reported comparisons under somewhat realistic conditions but tended to be of short duration or on small problems.
- *Industry studies* reported comparisons regarding TDD's effectiveness on real projects being developed for a customer under real commercial pressures.

Reasoning that more rigorous studies might be fewer in number but should be more trustworthy, the reviewers defined a category of "high rigor" studies that met the following conditions:

- The subjects included only graduate students or professionals—that is, people who are more experienced than the general population and who should behave the most like developers in industry or government organizations.
- The study used a TDD process description that matched the textbook definition and

 0740-7459/10/$26.00 © 2010 IEEE

ensured process conformance to some degree—that is, TDD was in fact the process being studied.
- The size of the development task and the number of developers working on it were significant—specifically, the study involved at least several hundred person-hours of effort.

We report the literature review results for three quality dimensions: delivered quality, internal code quality, and productivity.

For each dimension, we augment the study data with commentary from Grigori Melnik, senior program manager for Microsoft's Patterns and Practices group. His software engineering experience includes large e-business engineering projects for both corporations and government agencies, and he has adopted TDD in practice as a development project manager. Grigori is also a respected researcher, who has done his own prior analyses of TDD's effectiveness.[3]

Study type	TDD better (other studies)	TDD better (high-rigor studies)	No difference	TDD worse (high-rigor studies)	TDD worse (other studies)
Controlled experiment	√		√√	√√	
Pilot study	√	√√√√	√√		
Industrial use	√√√√√		√		
Totals	8	5	6	2	0

Figure 1. Summary of studies examining test-driven developments's effect on delivered software quality. The majority of studies (13) found TDD to be beneficial.

Evidence about Delivered Quality

Advocates of TDD argue that it delivers higher-quality software. The quality ensues from working on well-specified tasks, using frequent regression testing, and finding errors earlier in the rapid feedback cycle.

The Data

To examine evidence demonstrating this effect, we grouped all studies that reported results relating to the delivered software's perceived quality. Altogether, 21 studies reported metrics such as the percentage of the test set passed by the final product, the defect density or number of defects uncovered per test, or the quality assurance effort needed to deliver a satisfactory product.

Figure 1 summarizes the evidence reporting whether TDD did better or worse than the comparison approach or showed no substantial difference.

The results from pilot and industrial studies tended to support TDD's superior quality, with 12 studies showing better results for TDD and none showing worse. The evidence from controlled experiments was inconclusive: one study showed better results and two showed worse.

However, when we excluded less rigorous studies (results appearing in the darker area of Figure 1), the picture is muddier: five studies favored TDD's claims and two opposed it.

From these results, we concluded that moderate evidence exists for the argument that TDD tends to improve the code's external quality.

The Expert

Grigori agreed with our conclusion. In his experience, TDD leads to better software by helping developers think through the system design and so prevent bugs. As with unit testing, TDD doesn't replace skillful testers, but it does free them to find serious bugs in areas related to end-to-end scenarios and nonfunctional system characteristics.

Grigori also felt that TDD impacts software quality by helping fight sloppiness and encouraging coding discipline in the development team. He described working with a young, energetic programmer whose work unfortunately included many mistakes. Following TDD rigorously helped the programmer become more intentional in his work, thinking through the functionality he wanted to add. TDD doesn't just require skill and discipline; it also helps develop them.

Grigori saw an important metric missing from the literature—specifically, mean time to fix (MTTF). TDD's effects show up strongest in this metric. In his experience, TDD system problems are easier to diagnose and debug. The availability of the TDD regression test suite also helps immensely in this regard.

Evidence about Internal Quality

TDD advocates often cite its incremental nature and quick quality feedback as a driver of not only better code quality but also increased system modularity. Together with frequent refactoring, these development side effects should lead to more comprehensible, better organized, and more maintainable code.

The Data

Figure 2 shows our aggregation of 14 studies with results touching on code quality. It includes studies reporting object-oriented structure metrics such as coupling and cohesion, code-complexity measures, and code-density metrics that look at the size of modules or the LOCs required to implement a feature.

The overall picture is quite mixed. Across all studies, a small majority (6.5 studies) showed TDD performing better versus 3.5 that showed it performing worse. However, when we look only at the highest-rigor studies, the picture is exactly balanced, with 2.5 studies on each side of the issue.

At an aggregate level, then, we have to say that TDD shows no consistent effect on internal code quality.

The Expert

In Grigori's experience, however, TDD's effects on internal code quality are strongly positive and not at all mixed. He was skeptical of our results, saying that it would take a lot of high-quality studies to make him change his mind.

Teams experienced with TDD

Study type	TDD better (other studies)	TDD better (high-rigor studies)	No difference	TDD worse (high-rigor studies)	TDD worse (other studies)
Controlled experiment	√		√√		
Pilot study	~ ~	√ ~	√√	√ ~	~ ~
Industrial use	√√	√		√	
Totals	4	2.5	4	2.5	1

Figure 2. Summary of studies examining TDD's effects on internal code quality. The studies showed mixed results. A "~" represents a single study in which some metrics favored TDD and others favored the comparison approach. Such studies are counted as ½ when totaling columns.

Study type	TDD better (other studies)	TDD better (high-rigor studies)	No difference	TDD worse (high-rigor studies)	TDD worse (other studies)
Controlled experiment	√√	√	√		
Pilot study	√	√√√√	√√√	√√	√
Industrial use	√		√	√	√√√
Totals	4	6	6	4	5

Figure 3. Summary of studies examining the effect of TDD on productivity. The results are diverse across different types of study.

produce code that's cleaner overall, with less coupling, and thus easier to maintain. Testers can easily look into any part of the code base. Furthermore, TDD requires developers to expose the interfaces, which makes the system more easily extensible.

On Grigori's current project, his team now uses only TDD, but this wasn't always so. He reports that developing increments of the same system without TDD was significantly more painful. When the team started following TDD more rigorously, the internal quality improved—both in his subjective assessments of how clean the code is and objectively in the number of bug reports. Even the test code is cleaner because the team cares about making tests readable. Of course, Grigori added the caveat that team maturity might be a confounding factor; his team and system were both quite mature by the time the switch to TDD occurred.

Evidence about Productivity

Many potential adopters worry about a productivity hit when committing to TDD. Like any new practice, TDD will involve a learning curve. But beyond that, the proliferation of test cases associated with TDD must be managed and maintained and could therefore require more effort than a traditional approach.

The Data

Figure 3 summarizes results from the 25 studies we found that addressed productivity in some way—for example, by measuring development or maintenance effort, effort per LOC, or effort per feature.

The results differed drastically in different study types. Experiments tended to favor TDD, industrial studies tended to favor the traditional approach, and pilot studies were mixed. These broad results weren't affected by study rigor.

We're not sure what to make of the different messages coming from different study types. However, we can at least say that managing TDD's larger test suites hasn't shown a consistent negative effect on productivity.

The Expert

Grigori says the learning curve impacts productivity in the beginning. He recommends addressing the issue by pairing an inexperienced programmer with someone more experienced.

But taking the learning curve out of the picture and looking at longer-term effects, Grigori sees the productivity picture dependent on what you measure. (This may be one explanation for the muddled picture when we group results from multiple studies.) If the metric is "total lines of code written," TDD developers may perform better than many would expect. Grigori felt that his TDD teams wrote less code, even including the code to cover all the test cases. For one thing, the code required less rework. Furthermore, putting more upfront thought into what you're developing can help you see that what you're about to write isn't really needed after all. The result is less bloated systems.

Finally, productivity measures must also account for maintenance time to give a fair overview of the real effects. In Grigori's experience, code written in TDD style is much easier to maintain, leading to life-cycle cost savings compared to non-TDD development. Accounting for MTTF in the productivity measure would likely give TDD a distinct advantage. It's too easy to procrastinate—or forget—writing the test cases in test-last development, and the test cases make debugging the system a much easier task. By its nature, TDD helps you more quickly pinpoint code problems, and the regression test suite helps you make a good-quality fix much more quickly.

His experience has been that his TDD teams can find and fix bugs much faster than other teams.

The one major discrepancy we found between the data and Grigori's experience was related to internal code quality. This discrepancy might reflect variations across different software do-

mains and in specific business goals and team experience. Overall, however, it highlights the need for carefully monitored pilots of a technology to augment surveys of its potential risks and benefits.

Because we based this article on research findings, it seems only fair to give Grigori the last word. He and many TDD advocates find it useful to view TDD as a design rather than a development technique. This is because, in practice, TDD drives developers to think about how the system should be organized. The benefits related to better code testing are a nice side effect, but not the main point.

Grigori's stories come with the moral that TDD helps developers be more reflective and thoughtful about the tests that they do write, which is a good route to big returns on investment.

Erratum

The Voice of Evidence column ("Managing Variability in Software Product Lines" by Muhammad Ali Babar, Lianping Chen, and Forrest Shull) in *IEEE Software*'s May/June 2010 issue presented a categorization of difficulties and proposed solutions in dealing with variability in software product lines. The article should have noted that part of the work reported in the column was published in a previous article by Lianping Chen, Ali Babar, and Nour Ali, "Variability Management in Software Product Lines: A Systematic Review," in *Proceedings of the 13th International Software Product Line Conference*, ACM Press, Aug. 2009, pp. 81–90. Nour Ali of Lero, the Irish Software Engineering Research Centre, participated in the joint research that led to the development of the categorization reported in the prevoius article. That work was funded by Science Foundation Ireland grant 03/CE2/I303_1. We apologize for the omission.

References

1. K. Beck, *Test Driven Development: By Example*, Addison-Wesley, 2002.
2. 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. 399–412.
3. R. Jeffries and G. Melnik, "Guest Editors' Introduction: TDD—The Art of Fearless Programming," *IEEE Software*, vol. 24, no. 3, 2007, pp. 24–30.

Forrest Shull is a senior scientist at the University of Maryland's Fraunhofer Center for Experimental Software Engineering and director of its Measurement and Knowledge Management division. Contact him at fshull@fc-md.umd.edu.

Grigori Melnik is a senior program manager in the Patterns and Practices group at Microsoft. Contact him at http://blogs.msdn.com/agile.

Burak Turhan is a postdoctoral researcher in the University of Oulu's Department of Information Processing Science. Contact him at turhanb@computer.org.

Lucas Layman is a scientist at the University of Maryland's Fraunhofer Center for Experimental Software Engineering. Contact him at llayman@fc-md.umd.edu.

Madeline Diep is a PhD student at the University of Nebraska-Lincoln's Department of Computer Science and Engineering. Contact her at mhardojo@cse.unl.edu.

Hakan Erdogmus is an independent consultant, adjunct professor of computer science at the University of Calgary, and editor in chief of *IEEE Software*. Contact him at hakan.erdogmus@computer.org.