



Driving Software Quality: How Test-Driven Development Impacts Software Quality

Lisa Crispin

Vol. 23, No. 6
November/December 2006

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.



© 2006 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

For more information, please see www.ieee.org/portal/pages/about/documentation/copyright/polilink.html.

Driving Software Quality:

How Test-Driven Development Impacts Software Quality

Lisa Crispin

Recently, software development teams using agile processes have started widely adopting test-driven development. Despite its name, “test driven” or “test first” development isn’t really a testing technique. Also known as test-driven design, TDD works like this: For each small bit of functionality the programmers code, they first write unit tests. Then they write the code that makes those unit tests pass. This forces the programmer to think about many aspects of the feature before coding it. It also provides a safety net of tests that the programmers can run with each update to the code, ensuring that refactored, updated, or new



code doesn’t break existing functionality.

TDD can also extend beyond the unit or “developer facing” test. Many teams, including my own, use “customer facing” or “story” tests to help drive coding. These tests and examples, written in a form understandable to both business and technical teams, illustrate requirements and business rules. Customer-facing tests might include functional, system, end-to-end, performance, security, and usability tests. Programmers write code to make these tests pass, which shows the product owners and stakeholders that the delivered code meets their expectations.

But does TDD really improve software quality?

Some statistics

If you like numbers, many studies have shown that TDD does improve quality. Accord-

ing to David Janzen’s paper, “Software Architecture Improvement through Test-Driven Development” (ACM Press, 2005), industry studies show that programmers using TDD produced code that passed between 18 and 50 percent more external test cases than code produced by control groups not using TDD. Additionally, the TDD programmers spent less time debugging. Janzen also cites academic studies that demonstrated significant improvements in external software quality and programmer productivity.

One such study, by Bobby George and Laurie Williams, found that although TDD might initially reduce productivity among programmers inexperienced with writing code test-first, the TDD developers wrote code that passed 18 percent more functional test cases than the control group (“An Initial Investigation of Test Driven Development in Industry,” SAC, 2003). The study also reported that the code TDD developers produced surpassed industry standards in code coverage.

Of course, statistics aren’t everything.

First-hand experience

I’ve worked on three different development teams using TDD and on several more teams that didn’t. I can tell you from first-hand experience that TDD produces code that has orders-of-magnitude fewer unit-level bugs, far fewer functional bugs, and an exponentially higher probability of meeting stakeholder expectations when compared to code produced by conventional programming techniques.

In 2000, I joined a team of eight programmers who had just adopted Extreme Programming. TDD is a central XP concept, so I was surprised when I, the sole tester, became overwhelmed with

defects in the delivered code. The problem? TDD is hard to learn. My programmer teammates couldn't figure out how to write effective unit tests, much less unit tests ahead of the code, so they weren't writing them. Either they had to master TDD or we needed to hire two more testers.

After many brown-bag discussions about TDD techniques and a lot of practice, I was testing nice clean code where I could actually focus on functional testing. Most bugs I found were simply due to a misunderstanding of a feature's requirements. I had time to automate the functional tests and do valuable exploratory testing to find any serious problems that might be lurking.

I repeated this experience with my next XP team at a different company. They had done TDD for about a year, but it was the only testing they were doing. They didn't have anyone in a tester role, and no one was doing functional testing. Also, the team wasn't colocated. Their customer was reasonably happy with the code; it was far more in line with their expectations, and far less buggy, than the code previous teams delivered (the project failed twice before this team took it on). Still, disconnects continued to arise between what the customer wanted and what the team delivered.

As the tester, I helped the customer design acceptance tests that illustrated expectations clearly. We wrote these tests either in advance or during the first couple days of each iteration. The programmers could use the tests to understand the big picture and code the features correctly. I can't give you hard numbers, but the customers were much more involved in the development process than they had been when the programmers were only testing at the unit level, and they were even happier with the results.

Again, on this team, because the unit tests caught the unit-level bugs, the team had time to automate customer-facing regression tests and perform effective exploratory testing. All of these activities helped produce clean code that provided the anticipated functionality.

My current team has been producing code test-first for two and a half years. As with my first team, it took much time

and learning to develop TDD competence. But once a team passes the painful learning curve, writing the unit tests becomes much easier—and indispensable. All the programmers I've worked with who have learned to code test-first say they would never go back to the old way.

We also use customer-test-driven development. During the first few days of each iteration, we discuss the stories with the product owners. The product owners give us business examples, either by simply drawing diagrams and flowcharts on a whiteboard and holding discussions or by creating spreadsheets. We ask questions to elicit requirements and flush out hidden assumptions. We use this information to write high-level test cases for each story. When a developer starts coding on a story, he or she looks at these test cases to ensure basic features are understood. As coding begins, we write a simple executable test for the story. The programmer writes code to make this test pass, and then we can add more test cases if necessary for adequate coverage.

Have we seen any results? We started practicing TDD at the beginning of 2004. The production defect rate in the second quarter of 2005 was down 62 percent from the same quarter in 2004. Between the second quarter of 2005 and the second quarter of 2006, the defect rate went down another 39 percent.

Satisfied stakeholders

Of course, defect rate is just one small measure of quality. If you could talk to our business stakeholders, you'd find they're satisfied, often even delighted, that we deliver just what they asked for each iteration. We've implemented features that competing companies thought were too complex to automate. Many factors come into play, including other agile practices such as short iterations, continuous integration, and refactoring. However, I think our ability to understand and capture business requirements, and then code to meet them, is a critical reason for our success.


How TDD helps

Why does driving development with developer and customer tests improve code quality? Business people and soft-

ware developers speak different languages. They have to find some common ground to work together on a project, and they develop what Brian Marick calls a *shared language* or *project creole* (see www.exampler.com/book/preface.html). As he points out,

Focusing conversation on written artifacts makes that joint language more concrete, more uniform, and makes it come together faster. In happy cases, a team's language and shared experience becomes a powerful tool of thought. On the business side, it makes it easier to envision useful new features. On the programming side, it helps produce the kind of well-tailored program that programmers find easiest to extend.

Writing tests first helps define a feature or story's scope, a key component of software development, according to Kent Beck (see his article "Aim, Fire," *IEEE Software*, Sept./Oct. 2001). Anything not covered in a test isn't going to be there. And as already stated here, test-first helps ensure good design. As Janzen reported in his paper, studies reveal that computational complexity is significantly lower in test-first projects while test volume and coverage are higher. Also, test-first not only helps business and technical people communicate but also helps programmers communicate with each other. This occurs while they're writing a piece of code and later on when they want to refactor it or add a feature. When you run regression tests covering all the code every time you check something in, you can make changes quickly and confidently. You don't have to worry about forgetting or misunderstanding an obscure requirement a year down the road. The tests will remind you.

As teams get better at TDD and discover new techniques for writing effective tests and capturing requirements and examples, software quality will only continue to improve. 

Lisa Crispin is coauthor of *Testing Extreme Programming* (Addison-Wesley, 2002). Contact her at lisa.crispin@gmail.com; <http://lisa.crispin.home.att.net>.