

A Managers Guide To Test Driven Development With Unit Testing Frameworks

Gary Brown
Salt Valley Software Services
Lincoln, NE
glbrown@inebraska.com

Overview

Test Driven Development (TDD) is a programming technique that has emerged from the *eXtreme Programming* (XP) community over the last few years. The goal of TDD is to produce “clean code, that works”¹, in other words, software that has a simple design, contains no duplication, and passes all of the tests. The result of using TDD is better code, in less time, with fewer defects.

A Unit Testing Framework (UTF) is a general-purpose test harness that allows developers to write and run tests against their code. There are UTFs for all of the popular object-oriented languages and web application frameworks². The tests are typically written in the same language as the code to be tested. The granularity of the tests ranges from verifying the behavior of individual functions, to whole objects, object collaborations, and complex components. Tests are used to verify that the code to be tested responds properly to valid inputs, invalid inputs, exceptions, events, errors, and boundary conditions. The frameworks are designed to run the tests quickly, providing rapid feedback. The tests are run in isolation, avoiding unwanted side effects in subsequent tests.

A few words about eXtreme Programming

XP is a software development process, created by Kent Beck³, which is based on the principles of simplicity, communication, feedback, and courage. While I have had very positive first hand experience with XP, I don't recommend it for most software development organizations. The name alone puts some people off. Industry rumors suggest that XP is undisciplined hacking (which couldn't be farther from the truth). If you are curious about XP, read the book. If you are interested in trying XP, do it by the book or leave it alone. It won't work for you if you don't follow all twelve of the practices.

TDD is one innovation from XP that can be adopted by non-XP teams. TDD is about programming. It doesn't matter how you arrive at your requirements and design specifications. TDD will help your developers write better code, in less time, with no defects.

TDD is not your father's testing practice

In most software development organizations, code is tested in-flight by programmers using debuggers and at end of cycle by quality assurance teams. This usually works fairly well, but it is slow, expensive, and error prone because it is done by repetitive manual testing. Even in organizations where automated testing tools are used, the scripts are tedious to write and difficult to maintain. The testing is usually done through the user

¹ Ron Jeffries, see <http://www.xprogramming.com>

² See <http://www.xprogramming.com/software.htm> for a comprehensive list. Also, see <http://www.junit.org>, <http://www.nunit.org>, and <http://www.vbunit.com> for more specific info.

³ See *eXtreme Programming Explained*, Kent Beck, Addison-Wesley, 2000, ISBN 0201616416

interface, which makes it harder to determine if the error is in the business logic or the presentation logic.

Most unit testing frameworks are designed to test the business logic. This strongly encourages the separation of business logic from presentation logic (which is widely recognized as a programming best practice). This does not eliminate the need to test the presentation logic. By all means, keep doing that. Over time though, the purpose of user interface testing will shift from rooting out defects, to validating the behavior of the application.

Do you have a bug list? Are you using a defect-tracking database? If so, the number of items documented in those tools can tell you a lot about your current testing practices. Look at the patterns of defect identification over time. Is there a spike near the end of each development cycle? Is there a spike immediately after each release?

If you are seeing those spikes, then your developers are inserting defects during development and some defects are missed during testing. This is not a condemnation; it is a statement of fact. The problems that we are trying to solve with software are always more complex than they appear. Software development is hard. TDD can help.

The Test Driven Development Process

Write a test, make it pass, remove duplication⁴, is the mantra of the test driven developer. TDD proceeds in baby steps. One design element is added at a time and verified one test at a time. This is not to say that each new feature requires only one test. Most new features will need a few tests. Some of them will need many tests. The goal is to verify that the new feature behaves correctly, and then to test everything that could possibly break.

Always start with a failing test

Write a test that declares the interface that you want for the new design element. This test will fail and in most cases won't even compile, because it doesn't exist yet. Next, write the simplest code that will pass the test. We may need to create a new object, or add a function to an existing object. Get the test to pass as quickly as possible. Add another test and make it pass. Continue until you have tested everything that could possibly break. Finally, review the new code and remove any duplication that was added to pass the tests.

Simple code is not stupid code

Simple here means, "Make everything as simple as possible, but not simpler"⁵. Account for every requirement. Cover all of the bases. Make the code do exactly what is needed, nothing more, nothing less. Test everything that could possibly break. Remove duplication. That is sufficient for today.

⁴ See *Test Driven Development By Example*, Kent Beck, Addison-Wesley, 2002, ISBN 0321146530

⁵ Albert Einstein, see <http://www.brainyquote.com/quotes/quotes/a/alberteins103652.html>

Every line of code that I write today should be focused on producing the maximum amount of current business value. Don't try to predict the future. That nice to have extra feature may never have enough business value to justify its development expense. Every data element that requires persistence doesn't need to live in a database. Every database update doesn't need to be wrapped in a transaction.

Test early and often

When I am using TDD, I write a few lines of code (usually about 5 – 10) and then run the tests. It is not unusual for me to run the tests once per minute, when I am in the *flow*. The test results tell me about the progress that I am making. They help me to learn about the problem and the solution. They give me confidence. They help me to build momentum. They tell me when I have made a mistake. They tell me when I am done.

Keep the code clean

All software becomes brittle and hard to maintain at some point in its lifecycle. TDD can help you defer that entropy, if you maintain the discipline of removing duplication as each new design element is added.

The code should speak for itself, clearly revealing its intention. Code that requires comments to explain *what* it is doing is a sure sign of trouble. Refactor⁶ that code until the intention is clear. Comments in code that explain *why* we are doing something are very useful. Keep those, but don't require them for every object or function. The code should clearly reveal its intention.

Working with a net

Making major changes to existing code is a high-risk activity. Even if you have the original developers and accurate documentation, a major design change strikes fear in the heart of the most battle-hardened veteran. This fear is well founded. The system is complex; it has hundreds of beautifully crafted features. We made it flexible, so many of those features are available from a variety of convenient places. They are afraid that they will irretrievably break it. They might be right.

Had that system been written using TDD, the fear factor would be dramatically reduced. Every design element has a set of tests that validate its behavior and tests everything that could possibly break. All duplication has been removed, so they don't have to make the same change in many places. The tests are a safety net that allows the team to go boldly where others fear to tread. They proceed in baby steps. The tests tell them when they make a mistake. They add tests where needed and remove obsolete tests. They get the job done quickly and confidently.

⁶ Refactoring is changing the internal workings of an object without changing its external interface. TDD helps here because the tests will tell you if you break the external interface.

TDD Tests are an asset

The tests document the code. The tests are concrete examples of how to use the code. The tests are always up to date. By reading the tests, I can see what inputs are required. What inputs are valid and those that are not. I can see what errors to expect and how to handle them. I know exactly what the results will be. The tests protect the code. As new design elements are added, the tests tell me immediately when I break an existing feature. I can easily correct that mistake.

TDD in legacy code

All existing code suffers from some amount of entropy. The older the code, the more fragile it will be. The problem is not linear. Even that brand new system you just rolled out has warts. We didn't know what we didn't know when we started, so we had to adjust in sub-optimal ways. Those new requirements in the middle of the project forced us to put in some work-arounds. We probably weren't as relentless at removing duplication, as we should have been. We had a deadline to meet!

TDD can be used to dramatically improve existing code. I would proceed on two fronts, defect removal and system enhancements. I would identify the most painful defect and write a test that demonstrates it. I would then write the simplest code needed to correct it. I may need to add more tests to demonstrate other failure scenarios and make them pass. Then, I would return to the code and remove any duplication associated with that design element. I will have removed that defect and improved the local area of the code. On to the next most painful defect ...

As I add new features, I will write tests that verify the current behavior and make sure to remove any related duplication in the existing code. I would refactor the existing code to make it easy to add the new design element. I would use TDD to add the new feature. Over time, I will create an ever more complete set of regressions tests. I will have dramatically reduced entropy and defects. Yes, it will probably slow me down for a while then, it will help me to go faster.

TDD is a competitive advantage

Your customer's needs are constantly evolving. They need new products and features. They need them now. If your development team can deliver them faster with no defects, you win. Play to win!

TDD is a skill

Like any worthwhile skill, some learning is required. The unit testing frameworks are very easy to use. Writing the tests can range from simple to sophisticated, depending on the complexity of the code under test. Writing simple, testable code requires a different approach. You need to know how to add the tests into your code base. You need to know how to integrate the tests into your build process.

Your team can learn these skills with no outside help. I did, and I am sure you have developers who can code circles around me. TDD isn't rocket science, but it is different than what you are doing now. Get some training and mentoring, it will help you get there a lot faster.