The Pragmatic Bookshelf

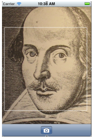
IN THIS ISSUE

* Bill Dudney on What's New in iOS5
* Venkat Subramaniam on Pure OO in Scala
* Jeff Langr on Test-Driven Development
* Jonathan Rasmusson on Self-Inflicted Scope Creep
* Brian Tarbox on Root Cause Analysis
* Dan Wohlbruck on the first computer virus

Issue #29
November 2011

Contents

FEATURES

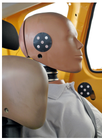


Inside iOS 5 6
by Bill Dudney

Bill takes us into the heart of iOS 5, introducing some powerful new APIs that have not received as much exposure as the APIs behind the new user features.

Scala for the Intrigued 12
by Venkat Subramaniam

In this third installment of his series on the Scala programming language, Venkat shows how Scala's OO purity leads to simple, elegant code.

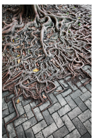


Test-Driven Development 17
by Jeff Langr

Jeff has written the case for TDD for managers. You may want to pass this article along to the manager you think needs to read it. Or study the article for when you need to make the case to a manager.

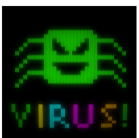
Self-Inflicted Scope Creep 22
by Jonathan Rasmusson

The client didn't ask for anything new. Why has the project grown?



Root Cause Analysis from Long Ago 26
by Brian Tarbox

Cleaning out his basement, Brian comes across a document that shows how little programming has changed in 23 years.



When Did That Happen? 29
by Dan Wohlbruck

One person predicted the coming of computer viruses. He also invented them.

Up Front

PragProWriMo and More

by Michael Swaine

If you ever wanted to write a book, this is the month to start, and we're ready to help. It's our third annual PragProWriMo writing month.

We have a lot going on this month: Six feature articles, a staff profile, calendar, John Shade, and our November writing blitz.

Bill Dudney shares some cool features of iOS 5 that you probably haven't head about in general tech press coverage, and that can help you develop cool new apps. Jonathan Rasmusson tells you how to avoid shooting yourself in the foot with self-inflicted scope creep. Dan Wohlbruck recounts the story of the first computer virus. Venkat Subramaniam is back with another in his series on the Scala language. Jeff Langr gives you all the arguments you need to convince managers of the need for test-driven development. John Shade opines that you have to be light on your feet to play in the clouds. And Brian Tarbox finds some poignant documentation while cleaning out his basement.

And then there's PragProWriMo.

A Month for Writers

This month, we're holding our third annual tech book writing month, piggybacking on the idea of National Novel Writing Month [U1] (NaNoWriMo). For you Pragmatic Bookshelf authors, this will be a great chance to make lots of headway on your books. If you're just starting your book, a sprint in November will get you heading toward critical mass. If you're almost done, November could be the final push you need to finish your book. If you aren't (yet) writing a book for us, we have advice for you, too. And a supportive community of other writers. And answers to your questions. And the motivation of weekly wordcount checkins, if you want that.

Your primary point of connnection should be the PragProWriMo 2011 Writing Forum [U2] on our site. I'd also recommend that you check out my blog [U3]; I'm going to try to post useful writing information there every day in November. And you should follow our PragProWriMo [U4] twitter account. It will be tweeting helpful and inspiring bits of writing wisdom.

We're looking forward to plenty of book writing action in November!

External resources referenced in this article:

[U1] http://www.nanowrimo.org

[U2] http://forums.pragprog.com/forums/235

[U3] http://www.swaine.com/wordpress

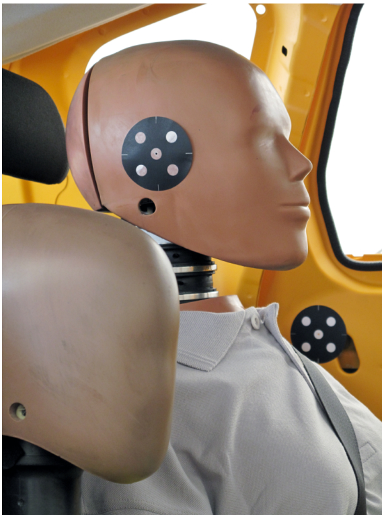
[U4] http://www.twitter.com/PragProWriMo

Test-Driven Development

A Guide for Non-Programmers

by Jeff Langr

Jeff presents a clear explanation of what Test Driven Development is all about, and what its potential benefits and risks are.



Test-driven development (TDD) is a programmer practice that's been employed by a growing number of software development teams for the past dozen years. Does TDD impact you personally? If you're a manager, what should you expect from teams using TDD? How do you know if they're doing a good job? Is there any advantage of TDD over sporadic after-the-fact unit testing?

First publicized widely with the advent of extreme programming in the late 1990s, TDD is a simple discipline that programmers use in their day-to-day, minute-to-minute building of software systems. The mechanism of TDD is simple:

1. Capture a small piece of system specification in the form of a coded software test.
2. After demonstrating test failure, build the smallest amount of production software that meets this specification—in other words, write the code that makes the test pass (and doesn't break any existing tests).
3. Review the new code in conjunction with the existing system, correcting any deficiencies in its design or the overall system design (this step is also known as the refactoring step).

A programmer repeats these three steps continually, with each cycle incrementally introducing a small piece of new behavior into the system.

Imagine growing a system in this manner from day one. By definition, all features in the system are documented in example form. Also by definition, all features in the system have been tested in some form. What sort of benefits can accrue from this practice? What costs does it carry?

Practicing TDD requires no high-end, high-cost tools (although value-adding, more sophisticated tools do exist). The tool of choice for TDD practitioners is a simple unit-testing framework that developers can download freely for virtually all modern programming languages. From a software management perspective, continuous integration (CI) tools such as Jenkins or CruiseControl can easily incorporate unit test execution into their build cycles. The builds are configured to fail if any of the tests fail—an indication that there may be a problem with the integrated software. Programmers can also track test execution statistics using such build environments.

The tests produced by TDD are of a different nature from tests produced by a QA or testing team. First, TDD tests are almost always written in the same language as the production system. Java test drivers code their tests in the Java programming language, C# programmers code their tests in C#, and so on. Second, since the tests are written in a programming language, TDD tests are

designed, implemented, and consumed only by programmers (although you might have visibility into the tests, sometimes through the CI server).

Primary Benefits of TDD

The development team seeks several benefits from employing TDD. The surface-level benefit is a higher-quality system: with every feature comprehensively unit tested, TDD can help produce software with significantly lower defects. For example, a TDD team at a large insurance company delivered a ~100,000-line Java application against which only 15 defects were reported in its first 11 months.

TDD also has the potential to provide "living documentation" (see Adzic, Gojko, *Specification By Example*, Manning Publications, 2011) for the classes in your systems. Programmers craft their TDD unit tests so that other developers can rapidly understand the code feature being tested. Such specifications by example [U1] stay in sync with the code, and provide a reliable way for developers to increase their understanding of system capabilities.

Design of a system is dramatically impacted by a team embracing TDD. The interest in unit testing immediately drives toward higher levels of cohesion and lower levels of coupling—the two primary indicators of a well-designed object-oriented system. This shift comes about primarily due to the simple fact that it is easier to test-drive a system with such a design.

More importantly, having the tests produced via TDD creates high levels of confidence to change the software. The significance of the ability to continually change code with low risk cannot be overstated. In an iterative/incremental development environment such as agile, you have no choice but to accept frequent changes to the code. TDD allows you to continually adapt the design to these changing needs. In other environments, the inability to safely change code leads to more rapid degradation of code quality, almost by definition.

The ability to change code safely also helps minimize the amount of unnecessary code in a system. One team was able to reduce their code base from 180,000 to 60,000 lines of code in 10 months by test-driving new features and incrementally adding tests to existing code. Casual analysis of most systems will suggest that most could be easily reduced in size by half, if only the developers had a safe means of doing so. This decrease in production code size can simplify maintenance and reduce the risk created by excessive code duplication.

Costs and Limitations of TDD

As with any new technique worth adopting, TDD incurs a learning curve overhead. You've already seen the three simple rules of TDD—that's really all there is to it. But the "how to" isn't the tough part. Instead, the primary challenge lies in developer habits: It's difficult for programmers with significant experience to re-learn their core approach to building software (and even more difficult for some to accept that their long-ingrained approaches hold room for improvement).

Success in transitioning to TDD will thus mostly depend on the enthusiasm of your team members, their ability to work together and support each other,

and their interest in continual introspection to recognize and fix approaches that need improvement. A team aligned with all of these conditions might have a solid foothold on TDD within a month or two. A group of individuals comprised of dissenters and apathetic programmers will likely not succeed without reorganization or intervention.

The actual labor cost of TDD is hard, if not impossible, to quantify. One study [U2] makes only the conclusion that quality increases but development slows with TDD. Other studies have demonstrated that initial development time increases from 15% to 35% but with increased quality.

The question becomes, what is the value of "increased quality?" Cost assessments for programming effort rarely include things like developer rework costs during the testing phase, the cost in delayed release, the cost of help desk/support time, the increased cost to introduce a new feature because the code is not clean, or the cost of losing a customer who leaves due to devastating defects.

TDD will not solve all development problems. It is only a portion of what is required to increase quality in your systems. TDD must be bolstered by additional testing, including some integration testing, acceptance testing, load/performance testing, and exploratory testing (to name a few). It still demands review of the software produced, and introduces a new need to review the tests as well. TDD is also not an end-all for design: You should still consider high-level up-front design, and you should still discuss design as a team on a frequent basis.

TDD is also not a substitute for thinking. While TDD derives code well for the vast majority of algorithms in your system, it will not necessarily help you with the insights needed for some complex algorithms (a rare need in most systems).

Metrics and Code Coverage

As a manager, how do you know anything about the quality of the unit tests produced by TDD? Metrics exist that purport to provide you with useful information, but take extreme caution in making decisions based on these numbers, or when using them as goals.

The most prevalent metric in use is something known as code coverage. For each run of tests, a code coverage tool can track which production lines of code are executed. If there are 100 lines of code in the production system, and the tests happen to execute, or exercise, 75 of those lines of code, the code coverage is 75%. Other coverage metric variants exist, including one known as branch coverage, but the general idea is the same: What percentage of the code (or code concepts) is coverered by tests?

Using TDD, the theoretical outcome should be 100% coverage: if a developer only writes production code in response to the existence of a test for that code, then all code should be covered. In practice, the coverage levels are typically in the high-90% range for various legitimate reasons (some involving the language and frameworks used, some involving the need to bypass interactions with external systems).

If TDD is capable of producing such high levels of coverage, then why not institute gating criteria that says that the developers are not done unless their coverage hits the high 90s? For starters, it's possible to attain high coverage metrics while producing poor tests that provide little value. The coverage number simply indicates whether or not code was executed, not whether it was proven to actually work. Trusting a coverage number is akin to trusting a programmer who says, "It compiled on my machine, let's ship it!"

Most programmers aren't nefarious, but in environments where they are expected to meet metric goals, many learn to make the least possible effort in order to meet the goal. In the worst cases, the quality of the tests is totally ignored. Over time, these tests become far more costly to maintain, which may ultimately lead to their abandonment [U3].

Technical metrics such as coverage are best consumed by the technical team only. Developers can learn how to improve their approach to TDD by exploring the areas in the code that are not properly covered. Low coverage numbers act as beacons for the team to uncover and correct trouble spots (not just in the code but in adherence to the practice).

So how do you know if the programmers are doing TDD well? Ultimately, the only metrics that matter and can't easily be "gamed" are customer or business facing metrics: customer satisfaction, number of defects, rate of delivery, and cost of development. The application of TDD should be able to impact these numbers (although other factors may negate these gains).

If your team has employed TDD for a while, and you haven't seen any customer-facing numbers improve, either your team is struggling with TDD, or other factors are in play. One possiiblity is that the programmers are challenged by a significant amount of existing code that remains untested. Too often, these legacy codebases are overly complex, and any changes can quickly generate numerous defects.

Beyond the customer-facing numbers, the tests themselves and design of the production system should be of high quality. Some facets of quality can be gleaned from metrics such as code complexity and code coupling, but the only way to truly know is to examine the test code and system design. That effort of course requires someone who knows what good tests, code, and design look like (and who is also someone you can trust).

TDD vs TAD

Many developers write unit tests after the fact—for code they've already built. A common term for this practice is POUT: Plain Old Unit Testing. I've also introduced the term Test-After Development, ("TAD") as a more direct contrast to TDD.

It's theoretically possible for a TAD developer to produce tests and code of quality and comprehensiveness comparable to TDD. In reality, it doesn't happen, for several reasons.

Developers writing unit tests after they've built the code typically have already tested the code manually. Developers view these tests as a double-check at best; more cynically, they view the tests only as fulfilling the wishes of some team or management mandate. They write no more tests than required: "I've

already proven my code works, why do I need to write tests for it?" For similar and other reasons, TAD tests are typically written only for a happy case and perhaps a small number of edge or exceptional cases.

Time is always a factor in software development—delivery always takes precedence over anything that happens between the code being written and being shipped. Testing gets short shrift, and even more so any development practice that falls in this time span, including reviews and TAD. In other words, these activities are a tad too late.

TAD programmers rarely go back and change the design of the system once unit tests are in place. Running out of time is again one reason, another is the lack of any immediate rationale to do so. So unlike TDD, where keeping the code base clean through refactoring is a built-in step, entropy will rapidly take over in a TAD system.

TAD proponents claim a typical coverage level of around 70% and believe that this is good enough. However, that means at least a quarter of your system remains rigid, and cannot be as safely or rapidly changed as business needs change. Some of the most complex and risky logic in your system will lie in that 30% of code without unit tests. Even if you have an excellent suite of acceptance tests to catch problems, changes in this area will come more slowly over time.

Recommendations

TDD is a skill that takes effort and desire to learn. Your team must agree that it is an appropriate tool for their circumstance. You should also seek to seed your team with at least one person with appropriate experience. TDD does not come without cost, but the potential benefits of test-driven development are too significant to dismiss.

About the Author

Jeff Langr is a veteran software developer with almost 30 years of professional software development experience. He is the author of three books: Agile in a Flash [U4] (written with Tim Ottinger and published in February 2011 by The Pragmatic Bookshelf), Agile Java, and Essential Java Style, as well as over 100 articles on software development (see http://langrsoft.com). Jeff owns the software consulting and training consultancy, Langr Software Solutions.

Send the author your feedback [U5] or discuss the article in the magazine forum [U6].

External resources referenced in this article:

[U1] http://martinfowler.com/bliki/SpecificationByExample.html

[U2] http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf

[U3] http://agileinaflash.blogspot.com/2009/09/stopping-bad-test-death-spiral.html

[U4] http://www.pragprog.com/refer/pragpub29/titles/olag/Agile-in-a-flash

[U5] mailto:michael@pragprog.com?subject=TDD

[U6] http://forums.pragprog.com/forums/134