Issue November 2011 | presented by jaxenter www.jaxenter.com

Java™ TECH JOURNAL

In today's world of software development, we hear plenty about Agile, but no-one really talks about eXtreme Programming anymore. In this issue of Java Tech Journal, we first introduce you to the key concepts of eXtreme Programming, before deep diving into some of the technical aspects of the approach, before finally asking the big question: is eXtreme Programming still relevant in 2011?

#13

eXtreme Programming

eXtreme Programming: In The Zone
An Introduction to eXtreme Programming

Key Process Patterns
Effective patterns for software development from around the world!

Succeeding With and Sustaining TDD
Getting the most out of TDD

Software Design: the Endangered Lessons of eXtreme Programming
Is XP Still Relevant?

eXtreme Programming: An Obituary
Announcement of Death Made by Top Agilist Gurus

An article on Test-driven Design

Succeeding With and Sustaining TDD

The rewards of TDD can be significant: dramatically reduced size, tests that improve developer understanding of system behaviors, more decoupled/cohesive designs, cleaner code, and the ability to change the system safely and rapidly to meet the demands of an iterative/incremental development process. Additionally, the number of defects can be orders of magnitude smaller than on a typical, comparably sized product. Individuals on teams that have achieved these results become test-infected, and will not willingly give up the practice.

by Jeff Langr

Until a team sees these benefits first-hand, however, the practice of TDD remains at risk. Various factors can trigger a gradual degradation away from TDD, to the point where developers abandon the practice and existing tests are stale and useless. Some of the main factors include the departure of skilled and influential team members, a shift in management focus, a particularly steep legacy software hurdle to overcome, and a slow build-and-test cycle. Further, the approach to TDD can unfortunately be a factor. A team without a solid grounding in TDD can quickly generate a poor codebase that makes the practice itself seem the culprit. Perhaps the most significant challenge to TDD is impatience. Turning around an existing, problematic source base for the better is not something achieved overnight, or even in a few weeks. Also, taking a team filled with TDD novices to a proficient level can take months. Teams successful with TDD have the following characteristics:

- dependable management support
- a team with solid peer support for the practice
- a review process
- a proper educational foundation
- either coaching support or sufficient prior experience with TDD
- developers who have approached TDD as a discipline
- continually evolving standards
- an ardent approach to refactoring
- high attentiveness to design and code quality
- high-quality tests that double as documentation
- a defined continuous integration process
- the use of coverage and other metrics as guidelines and education tools, not goals
- a comparable attentiveness to other forms of tests
- a unit test suite that runs rapidly

Management support

TDD is purely a developer practice [1], employed only by those crafting the code product. It produces, as a side effect, a suite of unit tests, each one verifying a small piece of behavior in the system. These tests are wholly developed, reviewed, and maintained by developers themselves. For many non-technical managers, the technical nature of TDD suggests a loss of control and requires a leap of faith: How do I ensure the developers practice TDD? Are the tests providing an appropriate return on investment? How do I verify that the developers are practicing it properly?

Customer-facing metrics can potentially answer the last two questions: Is the defect rate low? Is the rate of development remaining low over time, or is it increasing? Negative answers could indicate that TDD is not being properly applied, but it could also indicate other factors.

Management should insist that the team perform a root cause analysis for each defect uncovered. Was the cause a specification misunderstanding? Incomplete analysis? Process issue? Completely unexpected event? Configuration issue? Or just bad code? The key question regarding TDD practice: Is it reasonable to expect that the defect would have been prevented by a unit test introduced in a test-driven manner? If so, the team must learn from the oversight. Coaching or re-education may be required. However, managers must avoid chastising the team, and intervene only if such preventable defects become a continuing problem. Managers must show patience and fully support the team during the learning curve. They must provide training and coaching as needed, and can also help by promoting mechanisms for peer support and education. It is key for managers to remember that they are supporting a transition to a new technique that completely changes how developers approach building software. It's also important to understand the goal of TDD being a built-in piece of how programmers code, and that it will become an inseparable part of building code. Management should expect

that developers increase in speed after becoming proficient at TDD. James Grenning of Renaissance Software: "People tell me 'TDD is too slow, show me the fast way,' to which I say, 'Get good at being careful, and then you can go fast.'"

Peer support

An individual practicing TDD alone within a team cannot succeed, nor can two or three members of a larger team. An investment in unit testing of any form requires that all of the team members support the tests that are created. The use of TDD is a standard that must be agreed upon by the entire team.

Quorum on a team is needed: Over time, a minority of developers practicing TDD will eventually lose out if the remainder of the team is apathetic or hostile toward TDD. Even once quorum is attained, a few dissenting voices can significantly diminish the potential for success. A team that does not fully support TDD is a team that will continue to lose time with debate and rework. Continued dissension will eventually dissuade team members from practicing TDD.

Members of a true team support each other in their endeavors. Milo Todorovich, independent consultant, has instilled in his team a protocol for initiating a new pair session. He indicates that he expects and asks his teammates to "call him out" when he's straying from protocol.

Initially, management must support a fluid organization, in turn, allowing people to move to teams that work in the same manner that they prefer. Over time, however, as TDD becomes the predominant culture, it can become a key differentiator in hiring practices.

Review

Code produced without the review of third parties represents significant risk. As witnessed in the vast majority of codebases, open source or proprietary, the reality is that the majority of code produced is low quality, containing numerous defects. It is unfathomable, yet typical practice, that most product created by software professionals is not properly reviewed before it is shipped. The software industry has the uncommon characteristic that developers must continually build upon existing product; this environment further compounds the problem, as unreviewed, poor quality code generates much higher-than-necessary maintenance costs.

Not only must the team review production code, they must review the tests themselves, to support another key benefit. Tests produced by developers serve more than a single purpose. If crafted well, tests can act as the primary document for existing system classes. In TDD, each unit test defines a small bit of specification built into the system. The complete set of tests for a test-driven class describes all of the capabilities supported by that class.

Before tests are committed, some form of review is required to ensure that they provide this documentation capability. Other developers must be able to read and comprehend the tests. An ad hoc review process can work: Simply ask another developer to spend a few minutes reading each of the newly created tests. If they are unable to comprehend the test name, its steps, or its goals, then the tests fail review and must be improved before checking them in. Pairing is another form of review that can help ensure tests are eminently readable, but even if you're pairing, it's still best to get a third opinion from someone who wasn't intimately involved with test creation.

Improved design quality and living documentation are two of the most significant achievable benefits of TDD. As such, it is essential that the team achieve these goals, lest their lacking serve as a deterrent. Some form of software review is required. The practice of pair programming can represent a large portion of the review process.

Education

As indicated at the outset of this article, lack of proper training can lead to poor TDD practice, which can destroy any possibility for achieving its potential benefits. A primary focus of training must be on these benefits, otherwise students will have little clue that they can achieve them. Training must point out pitfalls for developers to avoid, such as the high future cost of inadequate refactoring.

It is possible for a team to become proficient at TDD without formal training. Pairing with experienced practitioners is one avenue. Another is to seek highly regarded print or online materials, including books, articles, tutorials, and how-to videos. Choosing these avenues demands that your team has follow-on support in the form of a skilled coach or other individuals on the team with prior, trustworthy experience in TDD.

Sustaining TDD (or for that matter, any sophisticated practice) over time requires a culture that embraces continual learning. Team members must be willing to seek more knowledge, communicate frequently, and to socialize code and tests, in order to combat the unending challenges associated with software development.

Coaching or prior experience

To do anything well, you must know what "well" looks like. With respect to TDD, there are many possible shapes that the resultant unit tests and code can look like. There are several nuances about TDD not necessarily obvious to beginning practitioners. While it's possible for teams to self-organize and succeed by determining their own practices, principles, and destiny, success is exceptionally more likely with the benefit of experience – either a coach dedicated to the team, or existing team members who can guide the rest of the team. An uncoached team may eventually figure things out, but more likely will abandon TDD in frustration without a guiding hand to keep them from completely unnecessary struggles and pitfalls.

Over time, teams need to seek the unbiased opinion of external folks. Any stable, small group of individuals tend to get into a rut that can be difficult for themselves to recognize. An external observer can often spot the challenges that team members cannot see from within the rut.

Discipline

TDD is a skill that requires dedication to learn and master. Students and practitioners must view TDD as a discipline. Regular practice becomes important, as is collaborative retrospection. The concept of *shu-ha-ri* [2] can help developers understand where they are at in the learning process, which in turn can help them understand what actions and decisions

are appropriate. Even at *ri*, the mastery level, students must continue to learn and practice. Most developers struggle with TDD initially. A significant first hurdle for each developer to jump over is a "light bulb moment," where the meaning and value of TDD suddenly becomes clear. This hurdle is different for each developer, and might take anywhere from a couple of days to a few months for an individual to overcome.

Standards

While the definition of TDD may seem clear, its practice in the wild suggests that many developers misinterpret its general purpose and technique. Fundamental discussions about TDD's goals and techniques can bring development to a standstill if debates are allowed to continue endlessly. The existence of many legitimate variant TDD techniques complicates the discussions even further.

The simple things, too, must be resolved, lest the team waste infinite numbers of small or large seconds each day: How do we name tests? What testing tool are we using? Where do the tests go? How are they reviewed? How do we name helper methods? Where do we put common object creation? What mock tools are we using? When do we mock? And so on.

Ensure that everyone has received proper education, and then ensure everyone is on the same page. You'll survive a small degree of variance, but significant disagreement will lead to rework, disillusionment, and sometimes to abandonment of the TDD effort.

As with all standards, revisit the TDD standards your team derives on a regular basis and discuss them ad hoc as needed. Standards become yet another part of the agile process: increment, reflect, and iterate. An initial increment of your TDD standards should take no longer than an hour to derive.

Documentation and test quality

Without developers seeking to understand and navigate the system by reading the tests produced by TDD, the earlier-recommended review of the unit tests themselves is of little value. With each newly encountered class in an object-oriented system, developers should first review the existing tests. Programmers should be able to read the list of test names and have a good understanding of the behaviors supoprted by a given class. Each step in the test should be clear, allowing readers to readily understand the goal verified by the test as well as understand the steps taken to accomplish that goal. A focus on test abstraction – the emphasis of the essential and the suppression of the irrelevant in each test – can help bring the test suite up to these high but reasonable standards [3].

In one Fortune 500 organization, developers quickly grew their coverage numbers by copying and pasting tests that were lengthy and obscure to begin with. The meaning of any given test was rarely very clear, and understanding one fully often took extensive time. Subsequent significant changes to a few key data structures rippled throughout a large number of these difficult tests. Some tests no longer compiled, and some tests now failed. Rather than improve the quality of these problematic tests, the developers began commenting out the tests that no longer compiled, and began ignoring the negative *results* of the test that no longer passed.

The problem with ignoring tests that no longer *compile* is that the return value on the investment to produce them is now zero. The problem with ignoring the results of tests that now *fail* is worse: the return value now becomes negative. A team might remember that one or two failing tests are "supposed to fail," because "they're currently broken." But as the number of failing tests swells to ten, a few dozen, or over a hundred (the amount in the Fortune 500 company), developers have no quick and simple way to determine why a test is failing: Is it being ignored, is it another problematic tests, or is it a new, real problem with the system itself?

Tests are monitors of your system health, but only if they are maintained diligently as living documents that accurately describe aspects of system behavior.

Design and refactoring

A key claim by practitioners is that employing TDD can assist in producing a high-quality design. There are two main reasons for the improved design. First, the interest in test-driving an application leads to a design that is easily testable. Michael Feathers asserts that there is "deep synergy between testability and design." [4] Testability demands high cohesion and low coupling, two primary indicators of a good design.

Second, the extensive code coverage generated by TDD allows continual factoring of the code to sustain a clean design. Developers are urged to ensure that with the introduction of each new small bit of functionality, the system retains the simplest and cleanest possible design. This continual incremental attentiveness to design quality is impractical without the rapid feedback of tests created via TDD.

The rapid cycles in iterative-incremental development exacerbate the design concern in software. Business needs can change dramatically each iteration, with new functionality that was never before considered. Teams have found themselves going from zero code to a sizable mess within a matter of weeks because they did not refactor continually and sufficiently. The design must be kept clean in order to continue introducing new features at a reasonable cost.

From the standpoint of sustaining TDD, a design that degrades increases the difficulty of writing tests, presenting yet another barrier to successful adoption.

Continuous Integration

A continuous integration server and agreed-on process for checking are essential for every software development team. The CI system is far more valuable, however, if it also runs a test suite for continual system verification. Developers must treat build failures reported by the CI server with a "stop-the-line" mentality: If the tests fail, the health of the system is in question. Before developers introduce any more code into a questionable environment, the team must investigate and fix the production system, tests, or build system itself before proceeding. In the absence of a CI system, the visibility of the TDD effort is significantly diminished, potentially to the point where only a few individuals care about the results they demonstrate. Sustaining TDD becomes far easier if the team embraces the tests and depends on them to indicate a key indicator of system health.

Coverage metrics

It would be ideal if a single, automatable metric could indicate whether or not a team was practicing TDD properly. Code coverage metrics comes closest, indicating whether or not lines of code are exercised when tests execute. Coverage metrics can definitively tell you that a team is *not* practicing TDD – low coverage means that there is far more code than tests that "drive in" that code. High coverage numbers, however, might simply mean that lines of code are getting exercised, not verified.

Teams doing TDD find that their resulting coverage numbers are usually in the 90% to 99% range. Coverage of 100% is unrealistic for a few reasons, some relating to the language and frameworks used, some relating to use of mechanisms to allow unit testing against code depending on external resources.

In any case, code coverage metrics are better viewed as an educational tool for use by developers only. Specifying a target goal for code coverage can have an extremely damaging side effect: developers will do whatever it takes to meet the goal, to the point of creating completely unmaintainable and useless tests. This situation was observed at one Fortune 500 company, where a VP mandated a certain code coverage percentage increase per iteration. Developers hastily crafted tests by copy-and-pasting long, unwieldy tests that verified little.

Outside of looking for improving trends for customer-facing metrics (satisfaction, delivery rate, and defects), the best answer as to whether or not developers are properly practicing TDD can come only from developers themselves. Experienced practitioners can quickly determine problems by examining the tests and production code. Metrics such as code coverage and cyclomatic complexity can help point the team to trouble spots, but they alone can not indicate that TDD is being done well. Only regular review of tests by the team can verify their quality.

Other tests

Unit tests alone are insufficient. By definition, they attempt to test code as isolated *units*. While it's important that you can verify the individual units of code that you create in your system, it's even more important that the units work together to meet the needs of the business.

In order to sustain TDD, you *must* invest also in such high-level tests. While you might survive on unit tests alone, it's more than likely that at some point you'll begin delivering defects because you haven't taken the time to ensure that the code integrates properly to fulfill customer needs.

Erik G. H. Meade, of EGHM Inc., reports that you can get away with unit tests only – i.e. no functional tests – for about six months. "After six months without functional tests, functionality begins to start breaking," says Meade. Catching up and adding tests after the fact is always a challenge, and moreso a challenge because the system hasn't been designed to be "functional test friendly," as Meade puts it.

If your team gets to that "catch-up" point, your efforts at TDD are at risk: Managers will perceive that your investment in TDD was insufficient to prevent the defects that functional tests might have caught. They will sometimes insist that your team spend their time elsewhere.

Fast tests

Unit tests produced by TDD must provide feedback in a reasonable time. It is possible to run the thousands of tests that cover a moderately-sized system in a few seconds, but only if they are not dependent on slower elements in the system (the primary culprit: database calls). Minimally, developers must run these tests prior to check-in, and optimally, developers should run these tests with every small change. If the unit tests run slowly, however, most developers will not wait the excessive time it takes to run all of them, which reduces their value in terms of providing rapid feedback. It becomes more difficult to practice TDD in such an environment: the sweet spot for TDD is when a developer can take a few seconds for every tiny change to the system. An extraordinarily slow test suite will delay check-ins to the continuous integration environment, further slowing the project: in general, the longer the feedback from introduction of an integration problem to its discovery, the more effort required to decipher the problem.

Conclusions

All of these challenges to success with TDD beg questions: Is TDD more effort than it's worth? Is it too difficult to expect developers to do? These questions can only be answered definitely by your team. Many teams have been wildly successful using TDD, many have failed. To answer the second question, developers with the proper aptitude for learning and a positive aptitude about wanting to learn can easily learn and ingrain TDD.

Is it more effort than it's worth? Here's a recap of the benefits touted in the first paragraph of this article: significantly fewer defects, dramatically reduced size, tests that improve developer understanding of system behaviors, more decoupled/cohesive designs, cleaner code, and the ability to change the system safely and rapidly. Experienced TDD practitioners, including this article's author, will claim that TDD helps them go faster than they would without tests.

Proper application of TDD will increase the return on your investment over time. Following the guidelines in this article will help you achieve and sustain success with TDD. Is your team up for the challenge?

Jeff Langr is a veteran software developer with nearly three decades of passion and experience. He's the author of three books, including Agile in a Flash (with Tim Ottinger, published January 2011) and Agile Java. Langr has written over 100 articles on software development. He runs Langr Software Solutions (http://langrsoft.com), a source for agile coaching and training, from Colorado Springs.

References

[1] The phrase "test-driven development" can be used in the context of acceptance tests, a practice sometimes referred to as acceptance test-driven development, or ATDD. References to TDD in this article regard only the developer practice.

[2] http://pragprog.com/magazines/2010-11/shu-ha-ri

[3] http://pragprog.com/magazines/2011-04/test-abstraction

[4] http://michaelfeathers.typepad.com/michael_feathers_blog/2007/09/the-deep-synerg.html

PDF edition

The free PDF magazine powered by JAXenter!

www.jaxenter.com

iPad edition

Get the FREE App in the iTunes Store!

Java Tech Journal is the digital-only magazine covering all the hot topics for Java and Eclipse developers. Each issue deep dives into one topic, be it the Scala programming language or Agile development. The magazine provides an insight into emerging trends in the Java universe, and offers best practices for established technologies on a monthly basis.