18

June 2012

testing experience

The Magazine for Professional Testers

Test Center of Excellence

How can it be set up?

printed in Germany

print version 8,00 €

free digital version

www.testingexperience.com

ISSN 1866-5705

Managing Technical Test Debt

A Critical Success Factor in Agile Delivery

by Bob Galen

I've been programming and testing for over a quarter of a century. One of the things I like about the agile methods is that they gave a name to something that developers always had to deal with – Technical Debt. Everyone knew about it, but not having a representative term sort of lessened its impact or import. It was just so easy to ignore if we didn't know what to call it.

You know what I mean...

Technical debt is that sinking feeling that developers have when trying to change old, brittle, poorly maintained code. When they draw straws and the short straw gently prays that they can get over just one more duct-taped repair to the codebase without it falling apart. Where team members mumble softly to themselves during a build – please, please don't fail.

Fast forward to the agile methodologies and we've quantified it. We try to attack it with other techniques like Test Driven Development (TDD) and refactoring and speak about incremental development, working in end-to-end slices, where we can mitigate technical debt and properly clean up after ourselves.

In this article I want to shift the perspective from development or code-centric technical debt and draw a correlation between it and testing. You see, I think testers and software testing suffer from the same sort of problems, but nobody is really talking about it in the same way. I want to introduce the term Technical Test Debt (TTD) as the testing equivalent to technical debt – so let's explore that notion a bit further.

Technical Test Debt

Test debt is more broadly nuanced than its development counterpart. The most direct comparison is to test cases that fall out of repair. I've always felt that test cases are the testing equivalent to lines of code from a deliverable perspective. Additionally, they both have similar design and creation characteristics.

So, technical test debt can be created where testers have failed to keep their test case documentation (assuming manual in this case) up-to-date with the evolution of the product or project. But that's not the only contributor to TTD. Here's a list that defines some of the testing activities and/or artifacts that, if done poorly or deferred, inevitably begin to erode the testing integrity of a software project –

TTD in Artifacts

- Undefined or unkempt test cases – dated, not matching current software functionality
- Badly executed exploratory testing that kept poor documentation surrounding session results
- Lack of collaborative test design – strategies for testing challenging project details are lost or non-existent
- Lack of collaborative user story writing w/o acceptance tests
- Lack of acceptance tests or user story maintenance and archival

TTD in Automation

- Badly designed test automation – prone to break, not tolerant of application changes w/o large-scale maintenance; hard to extend for new functionality
- Not keeping up with test automation maintenance – either at a framework or test case perspective as the AUT evolves
- Automated application coverage actually declining over time

TTD in Collaboration and Support

- Developers not doing their part in keeping up with quality work (unit testing, design focus, inspection focus, and an overall quality focus) within the team
- Ratio issues (and the team not doing their part) – potentially agile teams without or with insufficient numbers of testers – and the team not able or willing to pick up the slack

TTD Commitment to Quality

- Tests that aren't run during each sprint or iteration due to a lack of time or because of planned risk-based testing
- Deferred major test intervals; more focused towards non-functional tests
- Testers aren't even assigned to agile teams
- Not taking a "whole team" view towards quality and investing in code reviews and inspections, root cause analysis, and continuous improvement

I've categorized them into areas just to focus the discussion. There's nothing special about the categorization, nor do I consider the list to be exhaustive – meaning, I suspect there are "other" kinds of test debt out there to worry about.

So, technical test debt is often the case where we've become lazy in our tidying up of our testing artifacts and activities of all kinds. That laziness can literally be just that, team members who ignore this work. The debt can also be driven by other factors, which I'll explore next.

Root Causes

TTD primarily happens because of several factors; probably lack of time being the most prevalent. Usually there are far more developers on agile projects than testers and this can create a natural skew in the teams' attention away from quality factors and mitigating TTD.

Another factor is driven from the business side. Quite often Product Owners and other stakeholders struggle with the true cost of software development. They like to consider only initial coding costs including minimal or initial testing in their views. Beyond this, they often try to ignore debt, whether software or TTD, as being irrelevant or a nuisance to the business – being purely something for the team to resolve on their own. So business funding and prioritization can be the inhibitor in this case.

There is another, much more subtle factor that can't be blamed on "them" or on leadership. One part of it is certainly skill based. Team members may simply not be trained in solid testing practices and the "why" behind managing TTD. Another part, however, is centered on the teams' overall commitment to using professional testing practices in their delivery of solid code. This is the sort of laziness I referred to earlier and it segues quite nicely into a similar problem in the agile development community.

Testing Craftsmanship

Along with others, 'Uncle' Bob Martin has recently been championing the notions of software craftsmanship and professionalism with respect to software development. I believe the same notions apply to our craft of testing.

At some fundamental level we, the testers within agile teams, **allow** TTD to creep into our projects. We don't make it visible to our teams and our stakeholders and we allow it to increase. While it frustrates us and reduces our overall productivity, we would rather choose to live with it as a normal course of events.

I guess an equivalent in software is developers hacking their code. Sure, it works now, but it incrementally makes everyone's job harder over time. And inevitably the team becomes less productive and creative; and from a business perspective – less competitive.

So the prime directive in battling TTD surrounds your **resolve to do things right the first time**. To increase your professionalism to not create it or, at the very least, make the compromises visible to your team and your agile stakeholders so that they can make a decision on how much is "good enough".

Exploring TTD Specifics

Now I want to go through the four core areas of technical test debt. Exploring each more fully in turn and looking more towards the specific drivers in each case while making recommendations how mature agile teams should defend against or battle against these debts.

If you recall, the four core TTD areas are:

1. Artifacts
2. Automation
3. Collaboration and Support
4. Commitment to Quality

Let's explore each in turn –

Artifacts

This is one of the more prevalent cases of TTD in many agile teams – primarily because the testers on the teams are usually viewed as being independently responsible for testing work – that work being clearly outside the whole teams' purview. There's an interesting anti-pattern I've seen where the teams are very focused on design and working code – even the quintessential sprint review being primarily focused on working code. Rarely does the team make their testing efforts transparent in the review. I think this is a huge mistake.

One place to turnaround this TTD problem area starts in sprint planning – asking teams to better represent test work in the same way that they represent designs and code. I'd even like to see the teams' Done-Ness requirements expanded to include TTD implications – ensuring it gets the right level of transparency.

Beyond this, the whole teams need to understand the value of their testing artifacts – of writing test cases, creating scripts, logging session results from exploratory testing sessions, etc. It's not a question of wasting time or being "un-agile", but more so focusing on creating appropriately leveled and sufficient documentation so that further testing can be easily quantified, repeated, and automated. Don't view this as waste or gold–plating. Instead, these are simply solid, context-based, and professional test practices.

And finally, show these results in your sprint reviews as they are part of your "working code" deliverables, and they should be exposed, highlighted, and appreciated!

Automation

Very few "entry level" agile teams have high degrees of test automation in place when they initially adopt the methodologies. Usually, they're dealing with legacy code of varying degrees of quality, a somewhat under-maintained repository of manual test cases, and some bits of UI-based automation that needs a lot of care and feeding. This represents their testing coverage for their application set.

Quite often this creates a tenuous situation where these teams and their customers or Product Owners minimize the importance of test automation. Instead the testers are simply consumed with mostly manually testing each sprint's results – so there is never the time for creating or maintaining automation.

A frequent mistake in these contexts is assuming that **only the testers** can write test automation. Nothing could be further from the truth! Establishing a team-based culture where literally anyone can write automation is a start to handling this debt.

Addressing automation debt is a simple exercise. It begins with the product backlog and the Product Owner's commitment to beating down their TTD. In this case, automating all new features is a great way to start – so that you don't continue to dig yourself deeper in debt.

Beyond new feature stabilization, you'll want to continue to increase your automation infrastructural investment and automation coverage. The truth is that it's your safety net for catching issues as you move more iteratively and quickly. So the team should look for high priority automation targets in their legacy codebase and place stories in the backlog for automating them.

Often organizations will reserve a percentage of their backlogs for this sort of activity – say 10-20% depending upon how much debt they're trying to clean up. The key point is to stay committed to automation until you hit your debt reduction targets.

Collaboration and Support

As I said above in the automation section, the business clearly needs to take a stand fighting TTD. It's as simple as that. If the business won't support the team's efforts controlling TTD, then it will continue to grow and incrementally undermine the effectiveness of the team. However, it goes beyond simple business support. The teams themselves need to play a strong role in helping stakeholders understand, quantify, estimate level of effort, and then effectively execute TTD reduction efforts.

How?

It begins with planning, both at the release and sprint levels, within the product backlog. Teams need to define backlog items or user stories that clearly target the debt. And you can't simply come to the business and ask to fix five years' worth of TTD all at once.

You need to collaborate with them; identifying critical areas and partnering with the business in prioritization. You also need to explain the impacts in terms they can understand – defect exposure, maintainability limitations, new feature development inhibition, and performance reductions are just a few relevant focus areas.

In addition, you need to articulate a clear strategy regarding TTD. It can't simply be statements like – "we're totally broken... and we need to fix everything now...". Instead you need to define a thoughtful strategy and make recommendations that connect to ongoing business strategies. For example, your strategy might align in this way –

- Initially stopping incremental TTD increases (stopping the insanity)
- Establishing baselines for improvement, quarterly targets that align with the existing product roadmap
- Incrementally, sprint over sprint and release over release, quantify improvement focus points via user stories with clear scope and clear acceptance tests
- Demonstrating the team's internal commitment to improvement at Sprint Reviews – showing results and improvement data trending wherever possible

This sort of holistic game planning, high degrees of transparency, and alignment to your specific challenges and business context goes a long way in gaining business level trust and buy-in.

Commitment to Quality

I should have placed this issue first in the list, as I fundamentally think that teams have forgotten their central agile commitment to quality in high TTD environments. Perhaps a strong contributor to this is the general lack of organizational understanding of the dynamics of software quality and software testing.

The commitment to quality doesn't magically appear within every agile team. It needs to be coached and supported within the team by leadership. In this case, I believe it's fostered and developed by a top-down focus. Functional leadership, Scrum Masters, and Product Owners need to understand that they have a responsibility for delivering high quality results. That it's not simply a slogan, but the real truth in the organization. That leadership "means what they say".

And this isn't a simple and narrow view towards quality. No! It's a broad view that focuses on inspections of design and code as a collaborative cornerstone of developing quality products. A mindset of testing as a broad and varied activity – executed by professionals who understand the nuance of functional and non-functional testing. And leadership that also understands the returns that an investment in test automation can deliver – both in test coverage but also in allowing for design and code change nimbleness.

I have a personal test for many organizations –

- Do your developers willingly test when their team needs it?
- Or develop extensive automation hooks as part of the design process?
- Or passionately construct unit and functional automation by pairing with their test colleagues?
- Essentially, do they view all quality practices as an active part of their getting their (coding) job done?

Not by direction or obligation, but because they've bought into the value of quality up-front. If your answer is yes, then you've fostered a whole-team view towards quality commitment. If not... then you've still got some work to do.

Wrapping Up

Chris Sharpe is an active agile coach out of Portland, Oregon. At the 2009 Pacific Northwest Software Quality Conference he presented a paper on Managing Software Debt. In his presentation he referred to TTD as Quality Debt, while more pervasively looking at the debt challenge beyond development-centric technical debt and TTD – instead viewing it systemically within software projects. I mention it because I think Chris has gleaned the true nature of the software debt problem. It's broadly nuanced across all software development activity and an anti-pattern standing between agile teams and their true potential.

Another aspect to this is the serious role that leadership plays in "debt management". I spent several years as a Director of Development in a fairly mature agile shop. I was also their agile evangelist and the sponsor for agile and lean focused continuous improvement.

I had a habit of emphasizing quality and doing things right in almost every conversation I held with the team. I felt that for every ten conversations I had, I needed to emphasize craftsmanship, refactoring, proper testing, and other quality practices in nine of them.

Why?

Because it was incredibly easy for my agile teams to get the wrong impression regarding an effective quality balance, and fall back into their pre-agile bad practices of giving quality lip service.

For example, thinking that time was a more important variable for a project. Or to forget that they were empowered to "stop the line" and fix foundational issues. They had been programmed for years in traditional management and development approaches that rarely served quality the way they should have. I felt that I had to break through these historic habits and patterns with a strong message – shared consistently, clearly, deliberately, and often. I encourage all of you to do the same in your renewed commitment towards recognizing and eradicating Technical Test Debt!

I hope you found these thoughts minimally useful, and I encourage you to "face your debt". Thanks for reading,

Bob.

> biography

Bob Galen
is a VP and Agile Coach at Deutsche Bank Global Technologies in Cary North Carolina. He's also President of RGCG, LLC a technical consulting company focused towards increasing agility and pragmatism within software projects and teams. He has over 25 years of experience as a software developer, tester, project manager and leader. Bob regularly consults, writes and is a popular speaker on a wide variety of software topics. He is also the author of the book Scrum Product Ownership – Balancing Value from the Inside Out. He can be reached at bob@rgalen.com