Test Management: Measuring Quality in the Agile Enterprise

Measuring quality is not easy. Traditionally, software development teams have used defects found during test to help determine when a product is ready to deploy. But in Agile environments, quality is baked in from the start, and defect counts may not be the best or only way to measure quality. In this e-book, experts look at key metrics collected throughout the Agile lifecycle to help managers get the data they need and determine when the application is ready to go live.

<table>
<thead>
<tr>
<th>1</th>
<th>Beyond Burn-Downs: Metrics for Enterprise Agile</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Defect Tracking: Lean Principles for Getting the Right Data at the Right Time</td>
</tr>
<tr>
<td>3</td>
<td>Designing and Implementing ALM Quality Dashboards</td>
</tr>
<tr>
<td>4</td>
<td>Four Ways to Reduce Software Testing Cost Without Sacrificing Quality</td>
</tr>
</tbody>
</table>
Beyond Burn-Downs: Metrics for Enterprise Agile

BY HOWARD DEINER

MOVING YOUR enterprise into an Agile mindset is complicated. Decision makers want numbers. Understanding what to track and how to track it as you move your business from a plan-driven mentality into an Agile business-value-driven one can be challenging. Here are some ideas to get you started.

TRADITIONAL VERSUS AGILE DEVELOPMENT METHODS

Traditional Waterfall thinking can be summarized as this: Carefully plan each and every detail before you start your project, right down to each and every task’s what, when and who. Then, once you have the grand plan, execute on it while meticulously tracking and reporting against the plan. In other words, plan the work; then work the plan.

The Agile process takes exception to this mind-set, reminding us in the Agile Manifesto that we should favor “responding to change over following a plan.” That seems like sound advice for at least two reasons: (1) Business requirements are always trying to chase best value versus development expense; and (2) Dwight Eisenhower, the guy responsible for planning and executing the largest amphibious landing in history, would approve—he is quoted as saying, “In war, plans are nothing; planning is everything.”

WHAT’S WRONG WITH THE WATERFALL METHOD?

The trouble with tracking progress against a very detailed plan is twofold. One, it’s not possible for anyone to know all of the requirements and tasks up front, because the devil is in the details. Things change. People change their minds. Deal with it.
And two, traditional tracking against plan measures costs, not benefits. But what we really need to measure is value delivered and avoidance of

**WHAT ARE SOME AGILE TRACKING CATEGORIES?**

There are so many things Agile that can be tracked that it’s almost hard to start. But two areas that everyone knows of and are good are velocity and burn-down. These are predictability measurements and should not be used for measuring productivity, lest we fuel a gaming effect:

**Velocity** is the primary measure that we use to derive duration of a set of features in an Agile project.

**Release burn-down charts** should show both points completed and points added iteration by iteration.

**Other metrics to consider**

**Running Tested Features:** The desired software is broken down into named features (requirements, stories), which are part of what it means to deliver the desired system. For each named feature, there are one or more automated acceptance tests which, when they work, show that the feature in question is implemented. This metric shows how many features are passing all their acceptance tests.

**Business Value Burn-Up:** These are tracked just like a story point burn-up but are based on product-owner-assigned business value as delivered.

**Defect Count:** There are a few of types: post-sprint defect arrival, post-release defect arrival and defect resolution.

**Technical Debt:** This is “undone work” that usually occurs when the team is driven too hard to produce sprint point output. Technical debt stories are added to the product backlog and prioritized just like any other stories. This metric can track where technical debt is at and what the trending is.

**Work in Process:** This metric tracks number of items the team has in process at all times. Get this to trend to 1. If it gets too high, a Scrum master may want to foster better collaboration.

**Story Cycle Time:** This metric tracks how long a story goes from in work to done and helps keep the team focused on a story.

**Code Metrics:** These aid in making sure that you don’t accumulate technical debt over time. Here are a few of them:

- Cyclomatic complexity
- Coding standards violations
- Code duplication
- Code coverage
- Dead code
- Code dependencies
- Abstractness
future costs. The business needs features completed that make the best sense for right now, not just a plan well executed.

**WHAT ‘AGILE’ METRICS SHOULD I BE TRACKING?**

Organizations have to be careful about what they measure, because they get what they measure. Here are a few items that are fairly easy to measure, but have an extremely anti-Agile smell to them:

**KLOC (thousands of lines of code).**
The thought is, most productive developers get rewarded. But we end up rewarding the verbose coder who is gaming the system.

**Tasks completed.** Here we reward those who game the system by making up endless tiny nubs of tasks and then doing them. But did we produce anything of business value from all that work?

**Time worked on task.** If we want someone to sit quietly and stand at our sides while we work on a project, we should probably employ golden retrievers. They are loyal and don’t complain very much. But we need engagement, not just attendance. We think about tracking to that same end, too.

Instead of tracking such metrics, we should look for Agile metrics that have a certain set of characteristics:

**They affirm and reinforce Lean and Agile principles.** “Working software is the primary measure of progress.”

**They measure outcomes, not output.** Would you rather be 90% done on everything, but have nothing to show in working software, or 70% with delivered software?

**They follow trends, not numbers.** When you learn to fly an airplane, your first tendency is to watch the dials constantly. You end up never being on course and always over-correcting, something we need to avoid.

**They are part of a small set of metrics and diagnostics.** We need to focus on the production of software artifacts that really matter (the code) and have as little overhead as possible to help us stay on course.

**They are easy to collect.** A good friend once said to me “I can report status or I can change status, but I can’t do both at once.” We need to spend the majority of our time producing software, not reporting on its production.

**They reveal, not conceal, context.** Agile is all about honesty and transparency. If we collect data to satisfy hidden agendas, we are not being true to these values.
They provide fuel for meaningful conversation. Metrics are merely data. We need to find trends and patterns in the data to gather information of what we are observing. We then find wisdom when we can relate that back to causation. When we apply that wisdom and improve our software development process, we have completed the desired Agile “inspect and adapt” feedback loop. But if we collect data just for the sake of collecting data, we are wasting time and getting away from the task of creating software.

HOW SHOULD I PROCEED?
Here are some takeaways:

- Be smart about what you measure.
- Don’t measure just because you can.
- Make use of the metrics you gather, or don’t bother collecting them.
- Look for process metrics that reinforce Lean and Agile.
- Look for code metrics that are useful for keeping technical debt down.
- If we get what we measure, we will become an Agile and Lean software-making machine.
Defect Tracking: Lean Principles for Getting the Right Data at the Right Time

By Paul McMahon

SOME PEOPLE argue that defects must be tracked throughout the lifecycle in order to have high-quality software. Others argue against tracking defects early. To determine the optimal answer for your organization, you need to examine your current processes and determine where your problem areas are. This article tells you what questions you need to ask, why they are important and how you can use the answers to make the best decision for your organization with respect to defect tracking.

IS THE DATA YOU COLLECT ADDING VALUE?
Few would argue with the well-known software engineering principle that the more defects we find early, the higher the quality of software we can develop. But does this principle hold true in all organizations under all project conditions?

A few years ago I was working with a client that wanted to increase its software team’s agility but wanted to make sure that any changes did not jeopardize its CMMI level 3 compliance. To achieve this goal, we started by building flow diagrams of the processes the software team was using.

We intentionally did not look at the client’s documented CMMI level 3 processes, but rather built these flow diagrams based on what people told us they did to get their software done. We annotated each flow diagram with any process assets—procedures, templates, guidelines—that people told us they used. Any process asset that existed in their CMMI level 3 process repository that did not end up on a flow diagram became a candidate for elimination or streamlining.

Before eliminating or streamlining an asset, we asked a few questions: If
no one was using something in the repository, why was it there? Were we wasting time teaching people about certain process assets or steps in a process that added no value? Did we believe if people did follow a certain process step that they weren’t following that it would help them get their job done more effectively?

Let me give you a specific example. This organization had a defect tracking system that required a great deal of data to be collected about each defect entered into the system—for example, phase found, phase injected, type of defect, reason for defect, what phase the defect should have been found in. The process also required periodic analysis of the collected data. When we built the flow diagram for the defect tracking system that showed what people were actually doing, we found out that everyone was entering all the data because the defect tracking tool required it, but no one was going back and analyzing or using that data.

On further investigation, we determined that the reason all the data about each defect was being required was this: Someone wrongly thought that the CMMI required it. We also discovered that the process was so onerous it had actually discouraged people from entering real defects found. After we streamlined the process, eliminating much of the data required and even eliminating the need to report certain types of defects, we found that people started using the system more consistently. This activity led to improved software quality and ended up strengthening the organization’s CMMI compliance because people were following their defined process more consistently.

**WHEN TO TRACK DEFECTS: THREE QUESTIONS**

This story leads to a few questions. How did we ensure we were not jeopardizing the organization’s CMMI compliance when we streamlined the defect reporting system? Why would reducing the information that needed to be entered concerning a defect and reducing the types of defects reported improve quality? Doesn’t this fly in the face of the well-known software engineering principle that the more defects you find early, the higher the quality of your software?

To understand the answers to these questions, let me first share three questions each organization should ask about defects that will help you decide when to start tracking them.

1. What are the two or three most common types of defects that are causing the greatest trouble to your software quality?
2. When are these defects most often injected into the product?
3. For each piece of data you are considering to collect for each
defect ask this: Who is going to use this data, and how will what that person is going to do with the data help improve the quality of the software?

The Pareto principle tells us that 80% of the defects in a product are caused by 20% of the problems. I have found through years of consulting that most organizations know the two or three most common types of defects that are causing their biggest problems, and they have a very good idea when these defects are being injected into the product. I have also found that you don’t need to collect a great deal of information about a defect to be able to take effective action to counter future occurrences and improve the quality of your software.

**FOCUS ON THE MOST VALUABLE DATA**

How did we ensure we weren’t jeopardizing CMMI compliance? It is a common myth many hold that the CMMI requires certain data to be collected about defects. The CMMI does expect organizations to conduct peer reviews, and it expects organizations to analyze data from peer reviews, such as defect data. But the CMMI does not tell you what specific data about your defects you need to collect or the types of defects you need to collect.

Each organization should determine the answers to these questions based on what makes sense for the organization. Once you know what defects are most important to look for and when they most often get injected into your product, you can easily determine the optimal time in the lifecycle to focus on defect tracking.

For example, if an organization tends to have a lot of trouble with poorly defined or ambiguous requirements that get injected early, you need to track and take action early to eliminate those defects. But if your problems are primarily in detailed design or coding and requirements tend to be stable, then the payback may not be there for tracking defects early.

Reducing the amount of data you collect and focusing on the most valuable data can encourage your team to use your defect tracking system more consistently. Contrary to what many of us have been taught, it is less important how many defects you find early and more important that you find the most critical and potentially costly defects right when they are about to be injected into your software.

By first asking the three questions about defects, your organization can put in place a lean defect tracking system that will help focus on the most costly problems in order to get the optimum payback for your defect tracking investment.
Senior IT managers need to contend with an assortment of software development projects at the same time. They may be in different stages of application lifecycle management (ALM) using various development methodologies—with parts of ALM performed by geographically distributed teams. And there is no shortage of challenges: Identifying the projects and the measures that make sense across projects, implementing the collection of data from a variety of different sources, normalizing them across projects of different sizes, and selecting meaningful ways of presenting all this information and using it effectively.

Approaching the analysis, design and implementation systematically will make the ALM quality dashboard more straightforward, actionable and effective.

ANALYSIS FOR AN ALM QUALITY DASHBOARD

1. Metrics for the ALM quality dashboard. Metrics needed for the ALM dashboard must span evenly across all phases of the application lifecycle—requirements gathering, design, development, testing and maintenance.

2. Normalization needed across projects. Applications can be either small or large; mainframe-based, client-server or mobile. They may be multi-year, large-investment projects or short-run, uncomplicated ones.

The ALM quality dashboard needs to have more or less the same measures on all projects so that organizations are comparing them in a meaningful way. Weighting projects based on size and complexity is often needed for meaningful comparisons.
3. **Actionable metrics.** The metrics need to be actionable, especially if they will be a part of an ALM quality dashboard.

**DESIGN OF AN ALM QUALITY DASHBOARD**

1. **A handful of metrics.** ALM quality dashboards are useless if there are too many metrics. The whole idea of a dashboard is you don’t have time to wade through an ocean of metrics to quickly get a comprehensive picture of your quality. It is always possible to classify all the metrics you could measure as primary, secondary and tertiary depending on how much effect they have on overall ALM quality. Dashboards should focus on only the primary ones.

2. **Display design.** An ALM quality dashboard display should be restricted to one screen full of information. Increasingly, tablets and smartphones are becoming what are considered to be ideal vehicles for dashboards.

3. **Drill-down capabilities.** Dashboards are just springboards for further analysis. Further action may not be possible without deeper analysis of the metrics on a dashboard. Drilling down will enable developers to unearth these anomalies with statistics.

**IMPLEMENTATION, FOLLOW-UP AND ACTION**

**Data from multiple sources.** Data for ALM quality metrics may come from multiple test management solutions, some central and some distributed. Making all this information flow to a central system for analysis and action is not trivial. The dashboard system needs to accommodate direct-feed input from database queries, reports, spreadsheets or data in flat files exported and sent. Monitoring of such sources is needed to ensure that all data from all sources have been received before the dashboards are generated.

1. **Comparable timeline matching.** The timelines reflected in the data need to be comparable. This is of special importance if some of the data sources are networked to the ALM dashboard software but others are asynchronous and come through file transfers, spreadsheets or reports.

2. **Corrective actions.** Corrective actions and feedback are ideally fed directly from the dashboard to the people responsible for the various quality metrics. If comments or questions can be emailed directly from the dashboard to the people responsible for that metric, at that level of drill-down, it will improve course corrections.

3. **Follow-up and loop back.** Report-
ing systems should allow feedback and discussion on an ongoing basis as part of a feedback loop. Metrics that are out of range and actions that need to be taken and were taken should all be available on the dashboard. These capabilities are possible when dashboards are implemented as part of collaborative software like Microsoft SharePoint Portal.

ALM quality dashboards need to convey a large amount of critical information to IT management for observation, analysis and corrective action. In reality, projects could be in various stages of the ALM lifecycle, in-house or outsourced.

The development teams could be geographically dispersed, and the applications may be of varying size and complexity. Creating a single ALM quality dashboard in such cases is challenging, but if we focus on a few essential characteristics, it can be designed, implemented and used well.
SAY A COMPANY believes it is paying too much for testing. Now, if the bottleneck for general software development is testing, then cutting test staff will decrease your costs, but at the risk of introducing poor quality into your products. But the sheer number of testers compared with developers seems unrealistically high. What is going on here and how can you fix it?

Here is a four-step process to reduce test cost. I have used it. It works.

1. **Walk around and listen.** Take a stroll around the office and look at the testers. Are they at the keyboard, working, running the software, installing it, making plans and updating them, or are they doing something else?

   The something else is the killer. It might be supporting some other application, troubleshooting the current production version, waiting for a build, waiting for a server, waiting for a decision maker to decide what the right thing was for the software to do. This tends to manifest in lots of time in email or hanging out at the coffee pot, water cooler or someone else’s cubicle, or plenty of Web surfing.

   The point is that the tester is blocked. These blockages don’t just slow down the project; they turn the tester into a very expensive paper-weight. So I want to get into the ranks and find out what is really going on.

   Some of the time, the problem is systemic. The organization doesn’t even realize that many of its people are sitting around waiting for work. When it happens, people often feel they need to hide these moments. After all, they are embarrassing and, to the hatchet man, being not busy indicates redundancy, and that can lead to layoffs.

   As a manager, I want to find the sys-
temic blockage and fix it. I will work with the tester to figure out either how to knock down the obstacle or bring the problem to the organization’s attention so it is not mislabeled a test cost. Another option may be to redirect the tester to work on something more actionable.

One of the biggest conclusions I came to when researching the book was this idea: Three strikes and you are not out. There is always something a good tester can do to influence the outcome of the project. Even if system forces are keeping the testers down, it is usually possible to make some progress with a strong can-do culture.

The second aspect of culture is that some folks might not be blocked but just don’t want to work that hard. If you do management by walking around, say, twice around the office every day, and see the same people not working ever and find out they are not blocked, you’ve just identified your low performers. You can take any traditional corrective action, but if it were me, I’d start with the managers. How did this situation happen?

2 Remove barriers to high performance. Once it’s possible for our testers to actually make progress, we’ll take a look at how that time breaks down. Are they actually doing testing, running experiments to see if the software will work, or are they doing other things? Going to meetings, writing documentation, setting up test environments, loading databases with test data so the tests can execute—these things are incidental. They are not essential to the task at hand.

If possible, I’d like the testers to write down how they spend their time for a week or two—to break the work down into major categories. Examples include testing, setup, documentation, meetings and administrative tasks.

If the testers are spending a great deal of time on documentation, one thing they are doing is not testing. This is also true for setup and meetings.

It might not be a good idea to eliminate meetings, documentation and setup, but it is not uncommon to find a team spending 80% of its time doing other things besides testing. Get that down to 60%—just take a small chunk out of it—and you could double productivity without having to adopt a methodology, buy a tool or send anyone to training.

The trick is to see those things as waste and eliminate them. That may take some willpower, some decisiveness and some ingenuity.

I said you could do it. I never said it would be easy.

3 Speed the test process. Speaking of waste, let’s take a look at what that 20% of time testers are testing. The real question is what are they doing?

They may be trying to reproduce a
bug or exploring around the issue, trying to characterize it. Perhaps the tester is writing a bug report or attending a triage meeting.

Notice that none of those things is essential for software testing—they only exist because bugs exist. For that matter, because they take time, doing those activities subtracts from time spent actually testing.

Forget 20%. Some teams I’ve worked around have spent as little as 5% of their time making actual progress testing—the remainder of that time is spent on these incidental things that exist only because a bug was there.

We may not be able to eliminate bugs, but in many cases, the software could be a whole lot better before it gets to test, and that means testers can spend more time testing and less communicating about issues and waiting for builds.

At this point, I want to look at how that testing time is spent and ask if we can get builds of better quality before they get to test.

There are lots of ways to do this. One way is to simply present the low-quality initial builds to team members as an engineering problem and ask the members of the team to come up with the corrective actions to fix them.

**4 Eliminate excess WIP inventory.**

In physical manufacturing, people understand that excess work-in-progress (WIP) inventory is a sort of waste; it’s money put into the system that will not have value, and, in some cases, could lead to discounted sales and clearances. Once products hit the five-and-dime, the manufacturer is probably losing money on every part—because it made products in huge batches, beyond what the market could bear, instead of selling a dribble at a time.

It’s the same thing with software. Excess WIP can lead to waste. Once your team is performing, you may want to reduce work in progress and pursue a one-piece flow. It’s challenging and can change the way you think about software testing—and it’s totally worth it.

This method is not easy and it is not instant. One claim I can make in good faith: I have worked with companies that applied these principles and have moved from a test bottleneck to an abundance of test-quality assurance time and energy.

Teams I’ve worked with have hired more developers without hiring more testers, or they’ve moved some testers on to other teams or expanded the responsibilities of the test discipline to include more product quality issues.
ABOUT THE AUTHORS

HOWARD DEINER is an independent software consultant at Deinersoft Inc., specializing in Agile process and practices. He has a background spanning 36 years in the industry, with extensive domain knowledge in commercial software, aerospace and financial services.

PAUL E. MCMAHON, principal at PEM Systems, helps large and small organizations move toward increased agility and process maturity. He has taught software engineering at Binghamton University, conducted workshops on engineering process and management and published two books and more than 40 articles.

NARI KANNAN is the CEO of appsparq Inc., a mobile applications consulting company based in Louisville, Ky. Kannan has more than 20 years of experience in IT, having started out as a senior software engineer at Digital Equipment Corp. He has since served variously as vice president of engineering or chief technology officer of six Silicon Valley startup companies. Email him at nari@appsparq.com.

MATT HEUSSER is the principal consultant at Exelon Development, where he recruits, trains and does software testing and development. The initial lead organizer of the Great Lakes Software Excellence Conference and lead editor of How to Reduce the Cost of Software Testing, Heusser sits on the board of directors for the Association for Software Testing.

© 2012 TechTarget Inc. No part of this publication may be transmitted or reproduced in any form or by any means without written permission from the publisher. TechTarget reprints are available through The YGS Group.

ABOUT TECHTARGET: TechTarget publishes media for information technology professionals. More than 100 focused websites enable quick access to a deep store of news, advice and analysis about the technologies, products and processes crucial to your job. Our live and virtual events give you direct access to independent expert commentary and advice. At IT Knowledge Exchange, our social community, you can get advice and share solutions with peers and experts.