DZone

THE DZONE GUIDE TO

CODE QUALITY AND SOFTWARE AGILITY

2015 EDITION

BROUGHT TO YOU IN PARTNERSHIP WITH

AnswerHub · Sauce Labs · Tasktop · ThoughtWorks Mingle

Dear Reader,

"If houses were built like software projects, a single woodpecker could destroy civilization."
—Gerald Weinberg

Deep in our hearts we all know this is true. It's not that developers don't care about quality. It's just that we don't actually have infinite time to test nondeterministic systems (well, any physical system at runtime). And, *of course*, we all have to be agile, which infinite testing isn't.

On top of that there's the limited register space of the human brain. And the fact that some developer who no longer works here apparently preferred a lines-of-code to lines-of-documentation ratio of 3720:1.

Of course everybody knows that bugs cost some crazy figure—maybe $60 billion per year in the US alone. And in principle everybody understands that a bug caught earlier costs far less than the same bug caught later, in rough proportion to the difference in time between the two bug-catches.

The problem isn't that we don't *know* that it's important to keep software quality high. Rather, the problem is that it's very hard to know for certain how to do it. Uncertainty excuses future discounting. Technical debt tempts like a fee-free credit card. Up to a point, taking on debt is completely worth it. But where exactly does that point lie?

The Agile Manifesto teaches: *you probably don't know yet*. Fine, agreed, in principle. So when do you know? How many iterations before customers give up? And how immutable are those burndown lists anyway?

Software quality is a numbers game, software agility a fuzzy set of trade-offs and hand-offs at both code and organizational levels. In the real world, best practices and formal methodologies will only get you so far. The rest of the way is heuristic, and that's how we've approached our **2015 Guide to Code Quality and Software Agility**. Testing, refactoring, setting requirements, failing often with discipline—we've got it covered.

Check it out, let us know what you think—help us get it righter next time.

JOHN ESPOSITO
EDITOR-IN-CHIEF, DZONE RESEARCH
RESEARCH@DZONE.COM

Table of Contents

Credits

EDITORIAL
John Esposito
research@dzone.com
EDITOR-IN-CHIEF
G. Ryan Spain
DIRECTOR OF PUBLICATIONS
Jayashree Gopal
PROJECT MANAGER
Mitch Pronschinske
SR. RESEARCH ANALYST
Benjamin Ball
RESEARCH ANALYST
Matt Werner
MARKET RESEARCHER
Moe Long
MARKET RESEARCHER
John Walter
CONTENT CURATOR
Lauren Clapper
CONTENT CURATOR

BUSINESS
Rick Ross
CEO
Matt Schmidt
PRESIDENT & CTO
Kellet Atkinson
VP & PUBLISHER
Matt O'Brian
DIRECTOR OF BUSINESS DEVELOPMENT
Jane Foreman
VP OF MARKETING
Alex Crafts
sales@dzone.com
DIRECTOR OF MAJOR ACCOUNTS
Chelsea Bosworth
MARKETING ASSOCIATE
Chris Smith
PRODUCTION ADVISOR
Jillian Poore
SALES ASSOCIATE
Jim Howard
SALES ASSOCIATE
Chris Brumfield
CUSTOMER SUCCESS ASSOCIATE

ART
Ashley Slate
DESIGN DIRECTOR
Yassee Mohebbi
GRAPHIC DESIGNER

Special thanks to our topic experts Alexander Podelko, Alex Rosemblat, David Farley, Sergey Chernyshev, Colin Scott, and our trusted DZone Most Valuable Bloggers for all their help and feedback in making this report a great success.

WANT YOUR SOLUTION TO BE FEATURED IN COMING GUIDES?
Please contact research@dzone.com for submission information.

LIKE TO CONTRIBUTE CONTENT TO COMING GUIDES?
Please contact research@dzone.com for consideration.

INTERESTED IN BECOMING A DZONE RESEARCH PARTNER?
Please contact sales@dzone.com for information.

Executive Summary

DZone surveyed more than 600 IT professionals for our *Guide to Code Quality and Software Agility* to discover how organizations should prioritize various quality metrics as they mature, and to reveal how the types of software they produce influence their testing strategies. In this summary you will learn how the majority of organizations are managing software testing, and where their energy should be focused.

RESEARCH TAKEAWAYS

01. THE DEFINITION OF SOFTWARE QUALITY CHANGES DEPENDING ON CUSTOMER NEEDS

Data: Among companies with fewer than 100 employees, 52% of organizations that do software testing have a dedicated QA or testing team. For companies with 100 or more employees, this number increases to 79%.

Implications: Larger companies clearly have a greater focus on producing low-defect software. These companies generally serve more customers and are more mature. When companies have a larger customer base with several years worth of functionality expectations, the customers tend to be more conservative and ask that you solve their problems without introducing more. Smaller customer bases, like those of a startup, tend to be more forgiving of beta-quality software and care more about release speed and feature delivery. Startups themselves are more focused on the speed of innovation and often don't recruit testers in their early stages.

Recommendations: Software teams should follow the industry trend toward more testing for larger, more established products, and focus on speed of innovation for newer products. Refer to Johanna Rothman's article in this guide, "Why Your Managers Think Your Software Quality is Great—or Not", to see a graphic that lays out the best practices for software quality prioritization.

02. TECHNICAL DEBT WORSENS WHEN LEGACY CODE REFACTORING ISN'T A PRIORITY

Data: 61% of respondents say they have been limited in their ability to write automated tests because they needed to rewrite legacy code before they could write the tests.

Implications: Refactoring legacy code is a major challenge for software companies, and it's even more significant for larger corporations. Large enterprises will suffer greatly if they have sizable sections of code that can't be checked regularly with automated regression tests, therefore the technical debt incurred by not refactoring legacy code is greater for large organizations.

Recommendations: The longer organizations put off refactoring their legacy code, the more harmful the technical debt will become. Make refactoring legacy code and building automated tests for it a priority. Gil Zilberfeld's article in this guide, "Refactoring in a Legacy Code Jungle," is a great resource to guide organizations through this process.

03. DEVELOPERS CAN TEST, BUT MOST CAN'T REPLICATE THE VALUE OF PURE TESTERS

Data: Overall, developers handle most of the unit testing (90% of them handle it) while QA and testing teams focus on functional (70%) and user acceptance testing (UAT) (62%). In smaller organizations, developers do 11% more UAT and usability testing than they do in larger firms. 31% of organizations that do testing report they have no QA or testing teams. 92% of developers surveyed do some kind of testing, checking, and/or bug-finding at their organization.

Implications: It's clear that in many situations, developers are taking on the responsibilities of QA and testing teams, especially in smaller organizations, which are less likely to have those teams. Having dedicated, highly-skilled testers that are not merely recently-converted developers is an important step toward achieving low-defect systems [1].

Recommendations: Developers can manage testing in many low-maturity organizations, such as startups, but as a software product's customer base grows, the increased expectations for low-defect software necessitate having dedicated, skilled testers on the team. Read Andy Tinkham's article in this guide: "Testing: What It Is, What It Can Be" to discover what a modern testing team looks like and how such teams can provide immense value.

04. THE TYPE OF SOFTWARE PRODUCT CAN DETERMINE WHETHER MORE FOCUS SHOULD BE ON UP-FRONT TESTING OR MONITORING

Data: Respondents said they perform all types of tests more often when they are building high-risk and boxed software versus web applications and SaaS. For example, respondents building boxed and high-risk software were 7% more likely to always perform UAT and 9% more likely to always do functional tests than were developers who build SaaS and web apps.

Implications: More testing (especially exploratory testing) is needed before the first release of critical software that has a lower tolerance for failure. Less testing and more real-world environment monitoring is often better for SaaS and web applications, where it is possible to test changes and new features immediately and invisibly without degrading user experience.

Recommendations: Once again, teams should follow the industry trend and perform more testing up front for products that are "boxed" (not easily updatable) or high-risk (medical, financial) because low-defect software is essential right out of the gate in order for the product to retain customers. The majority of software, however, is web-based, and can still benefit from significant up-front testing, but it is more important to use heavy monitoring and instrumentation, not just to prevent negative user impact, but to recover quickly when problems inevitably arise. Emil Gustafsson's article in this guide, "Monitoring is Testing", gives a more granular view of the up front testing/monitoring mix you should be using.

[1] blog.codinghorror.com/making-developers-cry-since-1995

Key Research Findings

More than 600 IT professionals responded to DZone's 2015 Code Quality and Software Agility Survey. Here are the demographics for this survey:

01. Developers (36%) and Development Leads (21%) were the most common roles.

02. 63% of respondents come from large organizations (100 or more employees) and 37% come from small organizations (under 100 employees).

03. The majority of respondents are headquartered in Europe (41%) or the US (36%).

04. Over half of the respondents (73%) have over 10 years of experience as IT professionals.

05 . A large majority of respondents' organizations use Java (76%). C# is the next most popular language (36%).

ALMOST ALL DEVELOPMENT SHOPS TEST

Starting with the basics, we asked respondents whether their team does testing at all. 87% said they do, leaving 13% who don't. 95% of respondents said they believe that testing is necessary on all the software they develop, meaning 8% of respondents don't do testing, but believe they should. We told respondents to disregard sanity testing (basically to use the software as a customer would, to see if it works roughly the way you want it to) as a type of testing for the purposes of this question. This leads us to conclude that there is still a small contingent of developers that don't see the value in testing, but the overwhelming majority consider testing to be a vital part of software development.

SCRUM AND CUSTOM AGILE METHODS ARE FAIRLY POPULAR

53% of respondents said they use Scrum while 47% said they use a custom agile methodology. This was a *select many* question, so other common development practices included Test-Driven Development (40%), Waterfall (31%), and DevOps practices (28%). For those that employ Behavior-Driven Development (15%), which is a methodology built on top of TDD, if they answered BDD but not TDD, we added them to the TDD total. Kanban has 18% adoption among respondents, while Extreme Programming, even though most modern development practices are imbued with its principles, boasts only 8% who say they use the entire methodology.

OVER TWO-THIRDS OF ORGS HAVE A DEDICATED QA OR TESTING TEAM

Respondents were asked if they have a dedicated quality assurance (QA) and/or testing team or individual, and while 30% have neither, 49% have dedicated QA and 32% have dedicated testing. But since there is often disagreement about what QA includes, we asked all the respondents who had either of those teams whether the testing team is a part of the QA team for the next question. We found that most respondents (72%) have QA teams that include testers. 28% said QA and testing are separate in their organization.

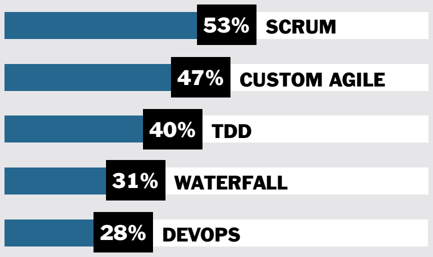
MOST DEVELOPERS DO SOME KIND OF TESTING

To most development shops, it makes sense that developers would write some of the tests since they're writing the code and can test their own contributions shortly after they're written. Unit testing is one type of test that's mainly in the domain of developers. We found that 92% of developers do some kind of testing, checking, and/or bug-finding at their organization. 76% of QA and/or dedicated testing teams do checking, but that question also included respondents who didn't have QA or testing teams. When you remove those respondents from this question, it's nearly 100% of QA or testing teams that participate in testing, as expected. Other major participants in testing among respondent's organizations were Product Owners (48%), the final customers (35%), Project Managers (28%), and beta testers (22%).

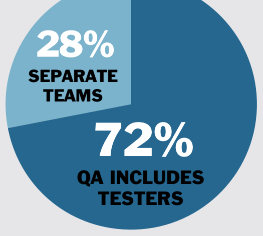
DEVELOPERS DO UNIT TESTING WHILE QA FOCUSES ON FUNCTIONAL AND UAT

As we mentioned in the previous section, unit testing is mainly handled by developers in most organizations, and that assertion is supported by our results. When asked what types of testing they handle, developers overwhelmingly said unit testing (90%). The next most common quality enhancing practices that developers perform are functional tests (68%), code reviews (68%), and integration testing (66%). QA and testing teams, in contrast to development, don't perform much unit testing (12%, 17%), but

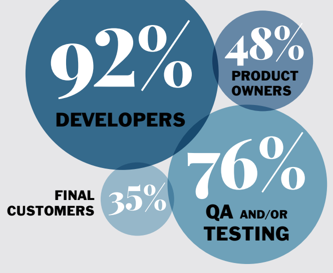
01. **METHODOLOGIES EMPLOYED**



02. **DOES YOUR ORGANIZATION DISTINGUISH BETWEEN QA AND TESTING?**



03. **WHO PARTICIPATES IN TESTING?**



they do perform a fair amount of functional tests (70%, 64%) and integration tests (56%, 54%). QA and testing teams tend to perform more usability tests (47%, 53%) and UAT (61%, 62%) than development (usability 17%, UAT 21%). Only 8% of organizations that test their software said that they use outsourced teams. The types of tests they do are similar to testing and QA teams with functional (62%), integration (45%), and UAT (43%) as their top three. The survey also found that developers are more likely than QA or testing to use static code analysis tools, and that security testing was the least common test type employed when development, testing, and QA were combined. When asked to rank their testing priorities, security was also in last place. The rankings were: 1. Functionality, 2. Usability, 3. Efficiency / Performance, 4. Security.

THE DEFINITION OF DONE (DOD) VARIES SIGNIFICANTLY AMONG ORGS

The most basic definition of done for software products is also the most common for respondents: 79% say its when all code compiles and builds for all platforms. Other common answers were: *Features reviewed and accepted by Product Owner* (65%), *Unit tests are implemented for new functionality and are all green with known failures noted* (63%), *New features are system-tested* (59%), and *Acceptance/story tests are written and passing* (52%). We also asked whether respondents had ever had deadlines in their current product team that caused them to release with less testing than they thought was necessary. 72% said yes.

MANY SHOPS RUN TESTS AS THEY CODE

Even though only 40% of respondents practice TDD, 61% run unit and functional tests as they code, which is one of the first steps to enacting TDD. 54% run those same tests before code is pushed to source control, and 49% run them when the code is deployed to an integration environment. Many respondents also have the groundwork in place for BDD, with 35% saying they have a language structured around their domain model that allows the average stakeholder to understand the business logic of the application and participate in its design. Still, only 15% said they practice BDD.

70% / 30% SPLIT BETWEEN PRODUCTION CODE AND TESTING CODE

While some philosophies consider testing code a form of production code, for the purposes of this question, we separated them by the code's purpose. 30% is the average amount of time respondents think they spend building code for tests, and they say that they're building production code the other 70% of the time. This is a fairly good split considering research from Delft University that shows CS students only spent 4% of their time writing test code, even though they self-reported much higher numbers [1]. If we're measuring this split against Fred Brooks' seminal work, *The Mythical Man Month*, then the split should try to reach 50/50.

IT ORGS TEND TO HAVE MORE DEVELOPERS AND FEWER TESTERS

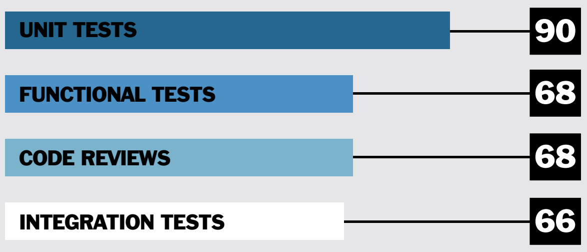
When asked about the ratio of testers to developers, the largest percentage (23%) made no distinction between testers and developers. The second largest percentage (20%), said the ratio was one tester for every five developers, or more. These results clearly point to more developers and fewer testers being the norm in IT organizations.

UNIT, FUNCTIONAL, AND INTEGRATION ARE THE MOST IMPORTANT TESTS FOR ORGS

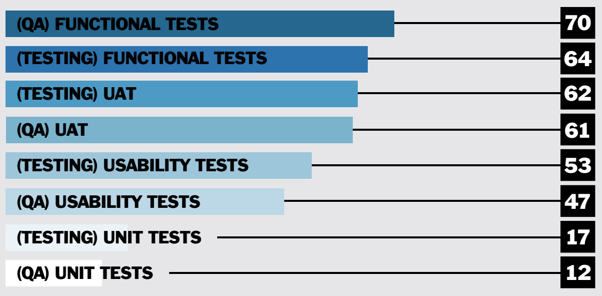
Taking their entire immediate product team (Devs, QA, Ops) into account, respondents were asked to give a *never, rarely, sometimes, often, always* ranking to various kinds of testing. Unit, functional, and integration ratings were heaviest at the 'always' end of the spectrum, indicating the importance of those types of testing. UAT and code reviews were also fairly important with an even distribution between 'sometimes,' 'often,' and 'never.' Usability testing, performance monitoring, and exploratory testing were firmly averaging in the middle of the rankings around 'sometimes,' while static code analysis and security testing seemed to be the least important, with most rankings around 'rarely' and 'sometimes.'

[1]: gousios.gr/pub/test-time-nier.pdf

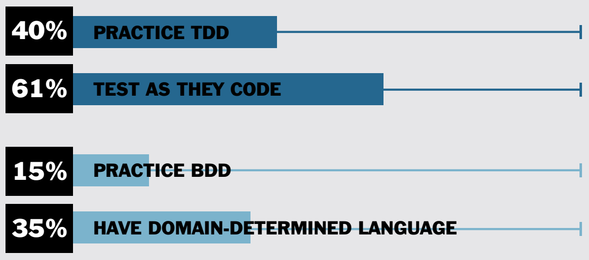
04. TESTING TYPES DONE BY EACH DEPARTMENT: DEVELOPERS



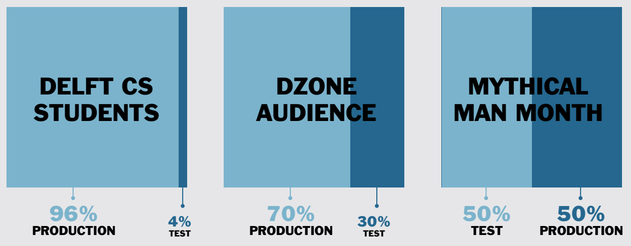
05. TESTING TYPES DONE BY EACH DEPARTMENT: TESTERS / QA



06. GROUNDWORK FOR TDD, BDD



07. TIME SPENT ON TEST CODE VS. PRODUCTION CODE



Why Your Managers Think Your Software Quality is Great—or Not

BY JOHANNA ROTHMAN

QUICK VIEW

01
Software quality depends on both an organization's goals and its customers' needs. The definition of quality for each product is ideally determined by managers based on these goals and needs.

02
Quality is more than just ensuring a lack of software defects. It can also include project cost, feature release time, and even team skill development.

03
Determine your primary quality metrics by deciding which things you would prioritize three weeks before launch.

04
The quality metrics that teams focus on should shift with the maturity of a product and the growth of its user base.

Have you ever worked on a project where you thought your software was great and your managers didn't? When that occurs, the team feels demoralized. "What did we do wrong? We thought we delivered what our customers (or managers) wanted. Why don't they like the software?"

We often think of software quality in terms such as "fit for use," "exceeds expectations," or even "we'll know it when we see it." However, I find it more interesting to think about Gerald Weinberg's definition of quality:

Quality is value to some person.

You have many *someones* investested in a software development project: customers, managers, the project team, and possibly other people across your organization. For example, if you have a help desk or a support department, those people might judge your product quality differently than you do as a developer or manager. Let's consider the different aspects of quality and the different things stakeholders care about.

CONSIDER ALL ASPECTS OF QUALITY

Quality is more than limiting or eliminating defects in your software. Quality also includes the software's feature set, when the software feature set will be available to customers (release date), and project cost. For some organizations quality includes increasing the knowledge and skills of team members as they complete the project.

As developers, you might focus on the feature set as the most important aspect of quality assessment. You might think an incomplete feature deserves top priority. But often, conversations with customers or managers (e.g., product owners, project managers) on this issue indicate a different set of priorities:

Product Owner: "We need the product to release in two weeks."
Developer: "Feature XYZ isn't done."
Product Owner: "I don't care. We have to meet the release date."

The release date is a big part of the quality definition for product owners and their customers.

I like to think about constraints, drivers, and floats when I think about what quality means to my project.The organization sets the constraints: release cost, project team, and project environment. We have all worked on projects where the original "constraints" were not, in fact, constraints at all. Depending on what's important to management, the release cost can increase or the project team can grow or change.

The customers care about the final product's release date (when they will receive their software), the set of features (what is included in the software), and defect levels (how well the software features operate upon product delivery). Your priorities should match theirs, and if you run out of time, sometimes you may descope the feature set or increase the expected defect levels. The point is, you need to know what quality really means for your project.

I like to provide project sponsors with options and clarify expectations at the beginning of a project. "If we're three weeks before the release date, and we're still finishing feature development, and we have more defects than we planned on having, where should we focus our attention?" I lay out the options clearly:

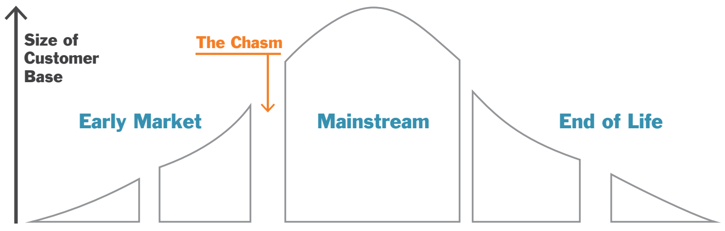
A. Finish the features
B. Fix the defects
C. Release on time, as-is
D. Stay within the budget, regardless of what you do

I tell the sponsors, "you can only choose one of these." This way, I force the sponsor to decide what is truly driving the project. If you know what your sponsor wants, in order of preference, you can make informed decisions throughout the project to deliver the quality the sponsor wants.

Some sponsors say, "I want it all. It's all #1." But they can't have all of these aspects driving the project. While a choice has to be made, let them know that all of their criteria will be recognized in the order that they are ranked. Each aspect of the project, and it's priority, defines this project's quality.

WHAT DO YOUR CUSTOMERS REQUIRE FOR PRODUCT QUALITY?

If you constantly work under time constraints, you might think that providing a good product on time is the best definition of quality. Not everyone wants something fast. One useful perspective on quality is to think about your product's life span and when certain customers may adopt of your product.



Technology Enthusiasts	Visionaries	Pragmatists	Conservatives	Skeptics
1. Feature Set	1. Time to Release	1. Low Defects	1. Low Defects	1. Low Defects
2. Time to Release	2. Feature Set	2. Time to Release	2. Feature Set	2. Feature Set
3. Low Defects	3. Low Defects	3. Feature Set	3. Time to Release	3. Time to Release

Quality Perspectives Across a Product's Lifecycle (from *Manage It! Your Guide to Modern, Pragmatic Project Management*)

If you have a product that solves a specific problem or small set of problems for the early market, you will have just a few customers, labelled in the Quality Perspectives chart as Technology Enthusiasts, who will expect a fast release. As you progress in the early market to the Visionaries, you will have more customers. The Visionaries want you to solve their problems, and their problems are all over the place. If you've ever played "feature leapfrog" with a competitor, you know this problem. This group still wants specific feature sets, but they care more about the speed of release than the Technology Enthusiast.

To hit the mainstream, you have to cross the "chasm" between Visionaries and Pragmatists. Many small companies never reach the mainstream because they don't create enough features to engage a mainstream market, they release products with too many defects, and/or they release at the wrong time, so they don't capture their potential customers.

Once your product hits the mainstream, things change. Your customers don't necessarily want more features. They want the features you produce to work. Release time is still important, but not as important as making sure the features work.

"Quality is not one size fits all."

The later in the market timeline you are, the less your customers care about the speed of release. Now the priority is on low defects—quality is not assumed here. You have to prove yourself each and every release. These later customers also care that you solve their problems without introducing more.

Although this product timeline can be helpful, you still can't assume it will tell you exactly what quality means to your customers. You may end up having all five types of customers in the chart, regardless of where your product is in its life span, and you'll have to choose which customers to satisfy first, second, third, etc. (and you may never get around to satisfying some customers).

I have worked on projects for which the release date was the sole priority (time to release). I've worked on projects that required we fix outstanding problems while avoiding the introduction of new problems (low defects). I've also been on projects where we had to make sure a few new scenarios worked well in order to meet a specific release date (low defects followed by time to release).

Each project is unique. If you know what your customers find valuable, you will be more successful.

WHAT YOUR MANAGERS FIND VALUABLE

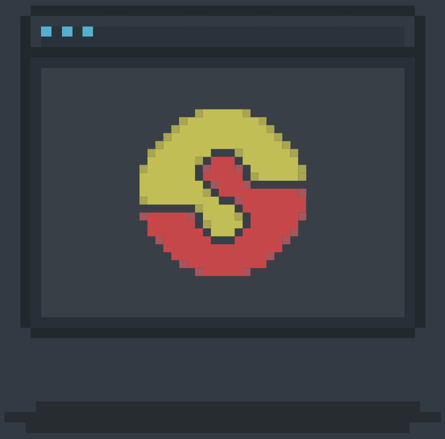
Managers care about revenue, customer retention/acquisition, and user experience. Even if you are an internal IT organization, managers care about whether your products allow your coworkers to do their jobs better or faster (which affects revenue and customer acquisition). If your coworkers (your customers in this case) don't enjoy working with the system, and if you don't make installation and upgrades/rollbacks easy, the system won't be used.

If you need to talk management about the quality of your software, plan to frame the conversation around revenue and user experience, as well as any other requirements.

QUALITY IS UNIQUE TO YOUR PROJECT

Quality is not one size fits all. Sure, you can work to reduce technical debt, or not create more, as you work; agile approaches can help you do this. However, you should also identify what's driving your project to create a quality experience. Although you may want to add more features and push back the release date, that might not be the primary quality metric for your customers. Once you know what's driving your project, you can decide how to organize the project and decide which technical practices will best enhance product quality.

JOHANNA ROTHMAN is the head of Rothman Consulting Group and the author of ten books on managing software development, hiring developers, and the job hunt. She writes two blogs at jrothman.com and createadaptablelife.com.



MARTIAL ARTS
HAS BRUCE LEE.

AUTOMATED TESTING
HAS SAUCE LABS.

Maybe you can't do a one-fingered push-up, but you can master speed and scale with Sauce Labs. Optimized for the continuous integration and delivery workflows of today and tomorrow, our reliable, secure cloud enables you to run your builds in parallel, so you can get to market faster without sacrificing coverage.

Try it for free at saucelabs.com and see why these companies trust Sauce Labs.

Dropbox salesforce YAHOO! PayPal intuit VISA

SAUCE LABS

Continuous Delivery for Better Software, Faster

Faster releases with better quality. If you follow this mantra, you are doing DevOps, whether you call it by that name or not. So why are so many companies neglecting quality?

Until recently, QA was something that happened after the code was written, and not something that needed to happen at each stage of development. However, new tools make it possible for any organization to include quality on day one of development. This "built in" approach not only keeps customers happy, but also makes the development team more efficient. While this implies major changes to processes, new automation tools and services can substantially ease the transition.

Modern QA tools allow organizations to build quality into the product from the start, versus finding problems late in the development process. They include whole testing grids that are on-demand and allow end-to-end functional testing to occur multiple times per day. Tools that increase test coverage immediately without new test cases or scripts. And tools that offer the same level of testing for both web and native mobile applications.

As organizations move to Continuous Delivery (CD), testing starts with automated unit tests on developer workstations, continues with automated functional testing in shared staging environments, and finishes with automated delivery or deployment. Everyone is instantly aware of defects that break builds, and because builds and testing occur continually throughout the development process, defects are found earlier and are easier to fix. The QA team morphs into facilitators. Building automation, and strategies to reduce the average number of bugs per release.

When organizations build quality into the product, they begin a virtuous cycle of better and faster releases

The more bugs you have, the more you will create. Software issues compound over time. However, when organizations build quality into the product from the start, they begin a virtuous cycle of better and faster releases. After all, if your competitors have better products because of awesome test automation, and you don't, you will lose.

WRITTEN BY LUBOS PAROBEK
VP PRODUCT, **SAUCE LABS**

Sauce Labs Automated Testing Platform by Sauce Labs

Our cloud-based platform helps developers securely test mobile and web applications across 600+ browser/OS platforms and mobile devices.

CATEGORY	API OR SDK?	OPEN SOURCE?
Test Platform	API	Yes

CASE STUDY

The Yahoo Mail team managed 20 VMs manually, which meant only 20 tests could be run in parallel. Needing to scale its testing environment to ensure quality, the team bumped its VM count to 100, but soon found that managing the internal infrastructure was too time-consuming. After moving to Sauce Labs, they increased the VMs they were using in order to speed up their builds and run more tests in parallel. Today, 30 Mail engineers run more than 10,000 integration and functional tests per day in parallel in 100 builds across Chrome, Firefox, and IE 10 against the server. The screenshots are especially helpful, because some of the test cases run great in a local environment, but when testing outside the Yahoo! network, there are challenges.

STRENGTHS

- Instant access to our automated testing platform with 600+ browser/OS/device configs
- Reduce testing time from hours to minutes by running tests in parallel
- Eliminate false positives due to unupdated browsers, operating systems, or residual data
- Optimized for popular CI systems, testing frameworks, tools, and services

NOTABLE CUSTOMERS

- Yahoo
- Yelp
- Salesforce.com
- Twitter

BLOG sauceio.com | TWITTER @saucelabs | WEBSITE saucelabs.com

Refactoring In A Legacy Code Jungle

BY GIL ZILBERFELD

QUICK VIEW

01
The first steps toward refactoring your legacy code for testability include a thorough read-through, re-organization of the file structure, and renaming of classes, functions, and variables.

02
Next, change the code to introduce access points for new tests using proven refactoring patterns.

03
Finally, separate and remove dependencies from the code.

Refactoring is a safe action when you have existing tests in place to make sure the working code isn't broken in the process. However, many organizations accumulate legacy code without building or maintaining corresponding tests, and you can't write proper tests until you've refactored the code. DZone's 2015 Code Quality and Software Agility survey results report that 61% of respondents were limited in their ability to write automated tests because they had legacy code that needed to be rewritten. In this situation, there are two choices: forgo the adventure altogether or do the brave deed and modify the code.

By leaving the code as-is, you incur costs not just in terms of ongoing maintenance, but you also have to add future maintenance costs to the equation. The code slowly rots away and future change costs rise.

When the project can no longer afford to take on more technical debt, modifying the code is the only choice. The problem is that writing tests for legacy code is hard. Depending on your language of choice (or maybe the current language wasn't even your choice), here are some of problems you might encounter when trying to write new tests:

- The tested object can't be created.
- The object can't be separated from its dependencies.
- There are singletons that are created once and impact different test scenarios.
- There are algorithms that are not exposed through a public interface.
- There are dependencies inherited by the tested code.

Yes, it's challenging to write new tests on legacy code—but this doesn't change the fact that legacy code is often the area of a product most in need of testing.

In other words, you're going into the jungle.

But before you do, you better tool up with enough refactoring techniques so that when you bump into trouble, you'll know what to do. Some of these techniques are automatic, which can cut out tedium and human error. Others are manual, and therefore carry an unknown amount of risk. You need to match the tool to the task at hand.

GETTING ACQUAINTED WITH THE HOSTILE ENVIRONMENT

Before you start moving code around, become familiar with your surroundings. The first step is to read the code, jump around from file to file, and think about how you might be able to organize the project a little better. Then start by reorganizing the source structure. Move co-located classes to separate files, move types into areas where you would expect to find them, and fix typos to increase readability and maneuverability working in the codebase. The structure is a model of the code, and if it's hard to understand the model, it will be difficult to refactor confidently. If we feel comfortable, we'll be more confident making further changes.

After things are in place, start renaming. Classes, functions, variables, files—anything that can improve readability. If programmers understand the code, the tests will be more effective (and there's not much point in testing the code if it can't be understood).

Be sure to modify names that don't fit their true purpose/describe their functions. For example, we have a function called "`getValidCustomer`" that returns a success code if a Customer object is updated from the database. It makes sense to rename it to "`PopulateValidCustomer`." (While we're at it, change it to a void method.) Now the names describe the function.

YES, IT'S CHALLENGING TO WRITE NEW TESTS ON LEGACY CODE—BUT THIS DOESN'T CHANGE THE FACT THAT **LEGACY CODE IS OFTEN THE AREA OF A PRODUCT MOST IN NEED OF TESTING.**

Choosing good names is not as simple as it seems. There's an art to it, especially with giant catch-all classes. If we use more accurate names, we can mentally (and structurally) refine our model. On the other hand, using generic names hides functionality, which causes inappropriate functionality to gravitate into these classes like a giant black hole. (If you've ever written a "-Utils" class, you know what I mean. If you bump into these classes, try to separate them and rename them properly.)

Renaming is low-risk, as it's mostly done automatically by the IDE. Usually, when doing pre-test renaming, it's recommended that you concentrate more on method names and variables in the code. These are usually small enough to modify without making any larger, potentially damaging changes.

PENETRATING THE FOLIAGE

In order to test code, you need to access it in different ways: probe the code and check the results; set up data and see how the code reacts; and replace dependencies with mock objects in order to control the tests. The more access points available, the easier writing tests will become. The setup and validation code will be shorter, less prone to error, and able to get better coverage.

We can change the code to introduce access points using these refactoring patterns:

- **Change accessibility:** Change method signature from private to public.
- **Introduce field:** In a long method that does many things, store tested data in accessible fields.
- **Add accessors:** If data is too hidden, use "getters" and "setters" to probe and modify that data.
- **Introduce interfaces:** If we want to mock a dependency, split its functionality into separate interfaces and mock the specific one you need. Then it can be used correctly once testing begins.
- **Virtualize:** Enable overriding and redefining functions by making them virtual.

Like renaming, these changes are also low-risk. However, you might encounter some resistance from peers who will say, "we shouldn't expose that, it's not proper design." Reassure them these exposures and modifications are temporary for the purposes of test design, and that a more sophisticated approach can be taken once the code is refactored and the new tests are built.

REMOVING OBSTACLES FOR MAKING A PATHWAY

Once you can access the code and its dependencies, it's time to actually move code around—this time it's not just for readability. By separating and removing dependencies, writing the tests becomes a simpler process.

There are many patterns available to remove dependencies from the code:

- **Move methods:** Especially in large classes, there are private methods that clearly don't belong in that class. When these methods also use dependencies, they can be moved to separate classes. I usually identify extractable bits of code in the complex methods, then extract to private methods in that class. You can also explore if these methods can be moved into other classes. From this, we get two benefits: the large class is reduced in size, and you can mock the new method instead of the direct dependency.
- **Extract classes:** The methods mentioned above can also be extracted to entirely new classes. An additional benefit is that you can specialize the new class, give it a proper name, and make sure that it won't become a black hole.
- **Introduce parameter:** When methods use a dependency directly (and probably more than one dependency), this process should be modified so that dependencies are sent to methods from the calling code. This way you can set up the dependency from the test. For example, if a function calls a static method, you can introduce a parameter that will contain the result of that static method call. Not only does this weed out the dependencies and make the code testable, it moves the dependency call up the chain. By doing this, you can use Extract Class for the tested code and benefit from having a separation of concerns.

Moving code around in these steps does increase the risk of affecting system function. Slow, thoughtful modifications, executed in pairs, will help to avoid breakage. Luckily, some of these modifications can be done automatically by the IDE, reducing that risk.

THE ADVENTURE BEGINS

Our journey began in the jungle with the prospect of modifying legacy code for the benefits of testability. The truth is, these techniques also apply to general refactoring—modifying the code to be simpler, modular, and more readable.

Teams usually don't stop here, though. After clearing a path, the real fun begins. You can separate more classes, extract logic from loops, invert conditionals, and make other higher-risk modifications. As the code is simplified, tests will become easier and more effective.

But the sun is setting, and you need to set up camp. You'll have to continue this journey on your own. There is a wealth of resources out there dedicated to this subject, so don't stop here.

GIL ZILBERFELD has over 20 years of experience developing, testing, managing, and designing software. He is a speaker, blogger, consultant, trainer, and practitioner of agile practices, both technical and procedural. He is also the author of *Everyday Unit Testing*.

Tasktop Integrates the Tools that Software Development and Delivery Teams Use to Build Great Software

Tasktop connects software delivery tools, disciplines, and professionals to create one cohesive team—turning practitioners into innovators and software into a competitive advantage.

Information Flow
from tool to tool and practitioner to practitioner

Collaboration
on defects, user stories, requirements, trouble tickets and more

Visibility
across the entire software development and delivery cycle

To learn more about syncing your artifacts: **www.tasktop.com/sync**

To learn more about collecting metrics from your tools: **www.tasktop.com/data**

WHITE PAPER

From Controlled Chaos to Differentiation: Why Companies Need to Integrate the Software Lifecycle
http://tasktop.com/why-SLI

TASKTOP

TASKTOP SYNC

TASKTOP DATA

Using Tool Integration to Take a Lifecycle View of Software Quality

One of the most hotly debated issues in software development and delivery is the definition of "software quality." One thing that's agreed is that software quality isn't simply the job of a QA department. It requires collaboration across the entire software development and delivery lifecycle.

For example, if the PMO, business analysts, production owners, developers, testers and others do not regularly collaborate on the requirements and defects, there is little chance that the team will produce a satisfactory application. Similarly, if project teams can't fully understand the nature of their defects, how can they know if the application is "done?"

But there are many impediments in the way, one of which results from the disparate tools used to develop and manage these artifacts. Each of the disciplines in the lifecycle has their own specialized tool, with their own view of these common artifacts. And, unfortunately, these tools aren't integrated with one another. So the only way to share and collaborate on these artifacts is through email, meetings and spreadsheets. Additionally, there is no straightforward way to get cross-project metrics and visibility. This introduces delays and errors, and ultimately reduces the ability of the team to produce a high-quality product.

"Making certain that software is of high quality requires every discipline in the software development and delivery lifecycle"

But if these tools were integrated, colleagues would be able to work more fluidly together. They would work on these artifacts in the tool they normally use, but with the benefit of updates from other colleagues working on the same artifact in their own tools. Additionally, reports could easily be generated that would support an organization's definition of "software quality." As an example, by automating traceability among requirements, tests and test results organizations can concentrate on defects in the most important areas of the application, enhancing the user's perception of quality and providing applications that delight their users.

WRITTEN BY BETTY ZAKHEIM
VP, TASKTOP

Tasktop Sync by Tasktop

TASKTOP

Tasktop Sync integrates software delivery tools, reducing the friction between stakeholders and increasing their capacity to do great work.

CATEGORY	API OR SDK?	OPEN SOURCE?
SDLC Tool Integration	SDK	No

CASE STUDY

When organizations buy new tools, they often forget to consider how that tool integrates with existing tools. After all, there are certain artifacts that are created by one discipline, that must be shared with other disciplines, or the value of that artifact is diminished. Defect reports logged by testers in a defect management tool, should automatically appear in the issue trackers the developers use. And user stories defined in a requirement management tool should automatically appear in the tools testers use to define their test cases. Tasktop synchronizes artifacts across all these tools to allow each practitioner to work in their tool of choice while getting continual updates from their colleagues.

STRENGTHS

- Synchronizes defects, requirements, tests, help desk tickets, issues, and much more
- Increases collaboration, visibility, and traceability
- Reduces errors, traceability gaps, and wasted time
- Enables non-developers to integrate tools across the entire development lifecycle
- Enterprise-grade performance and robustness; integrates complex tools and workflows

NOTABLE CUSTOMERS

27 of the Fortune 100; 7 of the top 25 world banks; 4 of the top 7 US insurers; 3 of the top 6 health plans—use Tasktop products

BLOG tasktop.com/blog | TWITTER @tasktop | WEBSITE tasktop.com

Monitoring *Is* Testing

BY EMIL GUSTAFSSON

QUICK VIEW

01
Boxed software, or software with high costs for each defect, needs more up front testing to ensure quality standards are met, while web-based service software needs more monitoring and instrumentation.

02
Focusing 100% on stable unit and functional tests will not help you find new bugs in your system. You need manual tests and techniques like fuzz testing.

03
Your application needs to have the proper instrumentation for exposing health properties such as requests per second, request completion time, failure rate, etc. This instrumentation itself also requires monitoring and maintenance.

Software quality has traditionally focused only on the number of defects in a system. Testing was the main technique for decreasing those defects. Today, we know that quality is not something you can test into a product—it has to be part of the product from the start. And while testing can still be used to ensure quality, there are other strategies we need to consider. The definition of quality is not the same for every organization, because the type of product being developed changes with an organization's needs and priorities.

TESTING DEPENDS ON THE DEPLOYMENT MODEL

First, let's consider **boxed software**: software that is packaged and sold in a "box" (virtual or physical). This type of software is difficult to update and feedback from users is traditionally scarce. Since updates take a long time and are typically expensive, extensive testing is needed before releasing boxed software. Because of cost, automated testing rather than manual testing would seem to be the preferred choice, but a balance of methods is necessary to effectively to monitor these programs.

A major problem with automated tests is that they are typically an ineffective method for identifying new bugs unless they are designed for that purpose. Unit and functional tests should be stable, consistent, and reproduce the same result every time. This results in an unfortunate blind spot with regards to new bugs, as these tests are only able to identify regressions—something that programming teams often forget.

In my experience, focusing 100% on stable unit and functional tests, without any other test component, will not give you the software quality you want. However, integration of other techniques can help fill the gaps. Manual testing finds new bugs and so do tools that are designed to find new tests. Fuzz testing, where inputs are mutated randomly, finds one category of bugs. There are also variants of fuzz testing tools that are smarter than the basic random-input versions. These will analyze the code to figure out what values to use. IntelliTest in Visual Studio 2015, formerly known as Pex, is such a tool.

If boxed software is at one end of the software model spectrum, **services** are at the other end, in particular, services in a cloud environment. Cloud services offer the potential to have multiple versions of your service running at the same time during deployments. In a service, the need for thorough up-front testing is not as important as it is in boxed software. Instead, the ability to detect defects in live applications is far more important. In fact, it's more effective to focus your software refinement efforts on detection in a services environment, because there is always the option to roll back to the last stable version at the push of a button.

Clearly, a different set of tools is required to ensure quality in services than what is required for boxed software. For example, in a service you want the ability to try a new version on only a single machine or even on a small subset of users before the new version is released to all users. You also need the ability to roll back to an older version and limit user impact if something goes wrong.

However, this does not mean that you stop all up-front testing for services. It only means that you might focus some (if not the majority) of your efforts on using different tools and techniques from boxed software testing tools. You also want to run tests continuously in production (TIP-testing). That is, you essentially want to run a certain set of tests all the time to make sure your service is healthy.

ALL SERVICES NEED INSTRUMENTATION AND MONITORING

How do you know your service is healthy and behaves the way it should? The answer is instrumentation and monitoring. Your application needs to have the proper instrumentation exposing health properties, such as requests per second, request completion time, failure rate, etc. This instrumentation, in turn, requires monitoring and maintenance, meaning everything from notifications that cause somebody to be woken up in the middle of the night, to dashboards with pretty graphs, and automatic actions such as automatically scaling by adding and removing instances to a cloud service.

Good monitoring is relative. For example, using the absolute number of a certain failure per second means that depending on how popular the service is, the monitor is more or less likely to trip. I've experienced this many times: an alert happens that has never happened before and when the problem is investigated, it turns out that the threshold for the alert is some absolute number that represented a failure rate of a few percent a few years ago, but with recent load, the absolute number only represents a fraction of a percent.

"Achieving software quality is about measuring user impact. It's not just about preventing bugs, but responding quickly once you find them."

My advice is to use relative monitors instead. When working on a service, failures (or other anomalies) should almost always be compared to the total amount of requests to your service. With relative monitors, the system triggers alerts based on percentages of that total load rather than absolute values. The only real exception to this rule is latency monitoring, since latency typically requires a different approach. For example, many shops monitor when the 95th percentile reaches a value higher than some absolute value. So if the slowest 5% of requests take longer than 200ms then you want to act. While the threshold here is an absolute value, the use of percentiles still gives you a relative property that you want in your arsenal of statistics to monitor. You want your monitors to trigger on real problems and not have too many false alarms because we all know what happens to somebody who cries wolf all the time – they get ignored!

While I recommend gravitating towards instrumentation and monitoring for services, this should be balanced with testing for optimum results. As for boxed software, a fair amount of instrumentation is necessary for an awareness of what problems the users have and how your product is being used. However, because the cost of updating boxed software is so high, you have to hedge your bets with more testing.

HYBRID SOFTWARE MODELS

There's a good deal of software that doesn't fall solely under the services or boxed categories. For example, there are a lot of apps being developed today—small applications typically installed on a phone or tablet. Apps are interesting to consider as a hybrid model. They are very close to services in how they behave, but they are like boxed software because new versions need to be downloaded and installed by the user. Apps are typically easy to install, but there is no guarantee that they will be updated. Also, different platforms (Android, iOS, Windows) take different lengths of time to review and deploy updates, so even within this category, you need to consider how much testing is needed versus relying on the ability to provide quick updates for your app. Ultimately, because of the similarities between apps and services, instrumentation of apps' behavior is very important in order to create a high quality app.

BACK TO DEFINING SOFTWARE QUALITY

There are several variables that affect how your organization should define software quality. User base size is a large part of the equation. If your software has a single user, you probably want less up-front testing than if you have millions of users.

Another consideration is the cost of a defect. If your software deals with trading stock on behalf of other users, an outage of just a few seconds could cost you a lot of money even with just a few users. So, in this case, a little more up-front testing is probably necessary, even though your software is a service. On the other hand, a service that provides daily stock quotes to millions of users probably has little need for significant up-front testing.

Life-critical software, like the code found in medical devices, is another example of code that needs more up-front testing, both because of the rules and regulations around their reliability, and because of the significant human cost of any defect in that software. If you're not working with embedded software, most modern boxed software can now be updated relatively quickly, so here are some basic principles you can follow to create high quality software in both boxed and service software settings.

- Make sure you have the capability to update your software quickly.
- Make sure you know how your software behaves in the hands of your users with instrumentation and monitoring, Preferably through limited release of your product to only a fraction of your total user base.
- Use automated tests to protect against regressions.

All in all, it comes down to observing how your software behaves in real life rather than in an artificial environment. That is how you achieve software quality: by measuring user impact, and not just preventing bugs, but responding quickly once you find them.

EMIL GUSTAFSSON is an Engineering Manager at Ericsson in Silicon Valley. He has spent the last 15 years developing distributed systems for government agencies and large companies. He frequently writes about software on his blog, Being Cellfish. Emil is also a certified Scrum master that never cared to pay for renewal.

Collaboration efforts derailed?

Get back on track.

Improve developer productivity with AnswerHub by:

Identifying topic experts

Retaining your team's knowledge

Reducing duplicate questions

TOP DEVELOPMENT TEAMS ARE SUCCEEDING WITH ANSWERHUB:

XBOX Linkedin ebay SAP

Learn how to get back on track with AnswerHub.

GET A DEMO TODAY

dzonesoftware.com AnswerHub

Faster Project Delivery Through Expert Identification

Building software is hard. Building great software is even harder. Building great software, delivered on time and under budget is nearly impossible—even under the best conditions. One of the key tenets of the agile process is that you iterate quickly as you build new features. Unfortunately, this can often leave inadequate time for proper specifications and clarification as you endlessly move from sprint to sprint. Worse, organizations very often have brilliant topic experts who take their know-how with them when they leave the building. This knowledge gap can lead to lost productivity and lost profit.

One of the key things that we get to do here at DZone Software is help developers and other tech teams share their knowledge better. Most organizations try to use tools like email or chat to help get their questions answered during a sprint, leading to the knowledge being captured in fundamentally transient systems that lack proper discovery through search. It becomes impossible to find new experts inside the organization, because everyone always goes to the same people, leaving them overwhelmed and under delivering on their own projects.

Building great software, delivered on time & under budget is nearly impossible – even under the best conditions.

With our TeamHub platform and the AnswerHub Q&A product, previously unknown experts inside the organization surface through their participation in a knowledge-sharing process that doesn't involve filling out a blank wiki page or chatting with someone. Organizational knowledge is captured so that as new things are learned, they surface easily in future development - leading to increased productivity and shortened training time. Sprint close rates increase because team members are able to find answers to their questions even after the experts have left the building or even left the company.

WRITTEN BY MATT SCHMIDT
PRESIDENT AND CTO, **DZONE**

AnswerHub by DZone Software

AnswerHub

Build Q&A communities like Quora and Stack Overflow with AnswerHub's enterprise application platform for collaboration and knowledge management.

CATEGORY	API OR SDK?	OPEN SOURCE?
Project Management, Ideation, Knowledge Management, Collaboration	API, SDK	No

STRENGTHS

- Q&A module allows users to post questions and crowdsource answers
- Identify topic experts to get questions answered by those who know best
- Document processes easily and create wikis to share knowledge within your team
- Earn badges and reputation points in our built-in gamification engine
- Integrates with tools like email, Parature, Confluence, Hip Chat, and more

CASE STUDY

Unity is a flexible development platform for creating multiplatform games and interactive experiences used by 1.7 million developers. Unity tried using forums to provide support, but as the community grew, the forums became saturated with repeat questions and endless threads, making it difficult for users to find relevant information in a timely manner. They turned to AnswerHub because the rich tagging system and advanced search bar minimizes the number of duplicate questions and accelerates the speed with which users can find answers. Unity's community traffic grew by over 20,000 visits (nearly 50%) per month in a year. Unity's community now supports more than 650,000 users, with 140,000+ questions and an average of 4.5 million page views per month.

NOTABLE CUSTOMERS

- Microsoft Xbox
- eBay
- GE
- Thomson Reuters
- IBM
- LinkedIn
- Epic Games
- Pixar
- Unity
- Oculus

BLOG dzonesoftware.com/blog

TWITTER @answerhub

WEBSITE dzonesoftware.com

DZONE.COM PRESENTS — THE 15TH ANNIVERSARY

THE AGILE MANIFESTO

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the **right**, we value the items on the **left** more.

SIGNATORIES

Kent Beck	Ron Jefferies
Mike Beedle	Jon Kern
Arie van Bennekum	Brian Marick
Alistair Cockburn	Robert C. Martin
Ward Cunningham	Steve Mellor
Martin Fowler	Ken Schwaber
James Grenning	Jeff Sutherland
Jim Highsmith	Dave Thomas
Andrew Hunt	

12 PRINCIPLES OF AGILE SOFTWARE

OUR HIGHEST PRIORITY IS TO SATISFY THE CUSTOMER THROUGH **EARLY AND CONTINUOUS DELIVERY** OF VALUABLE SOFTWARE.

WELCOME CHANGING REQUIREMENTS, EVEN LATE IN DEVELOPMENT. AGILE PROCESSES **HARNESS CHANGE** FOR THE CUSTOMER'S COMPETITIVE ADVANTAGE.

DELIVER WORKING SOFTWARE **FREQUENTLY**, FROM A COUPLE OF WEEKS TO A COUPLE OF MONTHS, WITH A PREFERENCE TO THE SHORTER TIMESCALE.

BUSINESS PEOPLE AND DEVELOPERS MUST **WORK TOGETHER DAILY** THROUGHOUT THE PROJECT.

BUILD PROJECTS AROUND MOTIVATED INDIVIDUALS. GIVE THEM THE **ENVIRONMENT AND SUPPORT** THEY NEED, AND TRUST THEM TO GET THE JOB DONE.

AGILE PROCESSES PROMOTE SUSTAINABLE DEVELOPMENT. THE SPONSORS, DEVELOPERS, AND USERS SHOULD BE ABLE TO MAINTAIN A **CONSTANT PACE INDEFINITELY**.

WORKING SOFTWARE IS THE PRIMARY MEASURE OF PROGRESS.

THE MOST EFFICIENT AND EFFECTIVE METHOD OF CONVEYING INFORMATION TO AND WITHIN A DEVELOPMENT TEAM IS **FACE-TO-FACE CONVERSATION**.

CONTINUOUS ATTENTION TO **TECHNICAL EXCELLENCE** AND **GOOD DESIGN** ENHANCES AGILITY.

SIMPLICITY—THE ART OF MAXIMIZING THE AMOUNT OF WORK NOT DONE—IS ESSENTIAL.

THE BEST ARCHITECTURES, REQUIREMENTS, AND DESIGNS EMERGE FROM **SELF-ORGANIZING TEAMS**.

AT REGULAR INTERVALS, THE TEAM REFLECTS ON HOW TO BECOME MORE EFFECTIVE, THEN TUNES AND **ADJUSTS ITS BEHAVIOR ACCORDINGLY**.

Your Code Is Like a Crime Scene:

Find Problem Spots With Forensic Methods

BY ADAM TORNHILL

QUICK VIEW

01
The same principles behind geographical offender profiling, a technique used by forensic psychologists, can also be applied to a codebase.

02
Your version control system contains much of the data you need to identify where to focus your code quality improvement efforts.

03
Sometimes, only a small percentage of the codebase is responsible for the majority of historic defects. By narrowing down their efforts, developers can often solve a large number of defects by working on a very small section of code.

We'll never be able to understand large-scale systems from a single snapshot of the code. Instead, we need to understand how the code evolves and how the people who work on it are organized. That is, we have to unlock the history of our system to predict its future. Follow along and see how your version control data can provide just the information you need to prioritize and improve the parts of your codebase that matter the most.

THE CHALLENGE OF LEGACY CODE

If you've spent some years in the software industry, you've probably encountered your fair share of legacy code. The real problem with legacy code isn't necessarily the lack of comprehensive unit tests or even excess complexity. The problem is that no one truly understands why the code looks as it does.

Legacy code is full of mysteries. For example, that strange if-statement that seems to do nothing was once introduced as a workaround for a nasty compiler bug that has now been fixed. The re-use of the 'userId' parameter to represent a time stamp was a quick and dirty fix that saved a deadline.

All these things are part of a system's story; a history that's often lost with the passage of time. What you're left with is a mess, and now you need to maintain it, add new features, and improve the existing code. Where do you start?

MOVE BEYOND CODE

Now, let's pretend for a moment that you are handed a complete map of the system. You immediately notice that it isn't your typical software diagram of boxes and cylinders. Instead, this map looks more topographical. It shows the distribution of complexity in your codebase along with information on the relative importance of each part. You're told that the map is generated based on how the team interacted with the codebase, so you know it's closer to reality than most of the documentation. This is good news! Now you know which components you need to grasp first and you know where the most difficult spots are located. All of this drives your learning.

Wouldn't it be great to have access to that information on your own projects organized so you can concentrate on the parts that require the most attention?. The good news is that you already have the data you need—just not presented in the manner you need. We'll uncover this information by taking inspiration from an unexpected field: forensic psychology.

LEARN FROM FORENSIC PSYCHOLOGY

Time and money are always important in commercial software projects, so you need to find a way to gradually improve the code while you maintain it. You also need to ensure that the improvements you choose to make do the most good. Even if some modules suffer from excess complexity, that doesn't mean you should focus on them. If the team hasn't worked on a particular module for a long time, there are probably other modules with more urgent matters where your efforts will have a greater effect on overall quality (that's why complexity metrics alone won't do the trick). To identify the most problematic modules, you need to prioritize all the design issues and technical debt you have in the codebase, hardly an easy task

Interestingly enough, crime investigators face similarly open-ended, large-scale problems. Modern forensic psychologists attack these problems with methods such as **geographical offender profiling**. Believe it or not, this method works for software developers too.

A geographical offender profile uses the spatial movement of criminals to calculate a probability surface for the location of the criminal's home base. This probability surface is projected onto a real-world map and the high probability areas are called *hotspots*.

Crime investigators use these probability distributions to focus their investigations. Instead of searching and supervising a vast area, law enforcement can now focus their efforts where they are most likely to apprehend their targets.

HOTSPOTS ARE BASED ON SPATIAL PATTERNS

Geographical offender profiling works because crimes, at least their geographic locations, are never random; the distribution of crimes follows a set of known principles. For a forensic psychologist, once they have a series of recorded crimes, they can detect patterns in the spatial behavior of the offender. They then use that information to predict where the criminal is located.

Software development is similar because code modifications aren't random either. Code changes for a reason: users want new features, bugs appear and are fixed, and code improves as we learn new ways to simplify it.

If you look into the evolution of a large system, you'll see that these modifications follow an uneven distribution. Some modules stabilize early during development while others remain in a state of flux. The latter is likely to be a problem; code that changes often does so either because the problem domain is poorly understood, or because the code suffers from quality problems.

THE GEOGRAPHICAL PROFILING TECHNIQUE PROVIDES AN ATTRACTIVE SOLUTION TO THE LEGACY CODE PUZZLE.

The geographical profiling technique provides an attractive solution to the legacy code puzzle. Every time we make a change to our code we give away a piece of information. A code change is like a vote for the importance of a module. What we need to do is to aggregate all those votes cast by the programmers who work on the system. Code changes are our equivalent to spatial movement—and all of those changes are recorded by our version control systems.

ANALYZE HOTSPOTS IN CODE

Version control systems are a gold mine, full of valuable information on change patterns in legacy code. To identify hotspots, we just need to traverse the source code repository and calculate the change frequency of each module. That gives us a prioritized picture of the most frequently modified code.

But there's more to a hotspot than pure change frequencies. To qualify as a hotspot, the code area also has to have a high likelihood of overall quality problems. We don't have a good metric for that within software. What we do have is a decent approximation based on complexity metrics from the source code. Everything from simple heuristics, like lines of code, to more elaborate metrics, like cyclomatic complexity, can potentially serve this purpose since the differences in predictive value are usually small enough to ignore.

If we combine change frequency with code complexity, we get an operational definition for hotspots. A hotspot is complicated code that programmers also have to work with often. Such code is often a maintenance nightmare. There's empirical research to support this claim: change frequency is one of the best predictors of software defects [1].

I've listed some key tools and resources for finding and visualizing hotspots in your own code.

Code Maat: A command line tool to mine and analyze data from version control systems.

Code Maat Gallery: A gallery of the best examples from various version control data visualizations.

Your Code as a Crime Scene: My book on forensic techniques in code quality management.

Now that we know how hotspots are found, here are some tips on how to best use that information.

USE HOTSPOTS IN PRACTICE

Hotspots in code, like their counterparts in crime investigations, aren't precise. Instead they suggest a probability of where most of the problems are located. A hotspot analysis can guide your team to the most beneficial areas to focus on for codebase improvement. Some obvious uses of hotspots are to identify code that's expensive to maintain, and to prioritize which sections of code need to be reviewed. You can also use hotspots to communicate with testers, who use the information to focus their testing around hotspot-dense feature areas.

Hotspots are a simple metric. That simplicity is a strength that translates to practice surprisingly well. In a recent analysis of one project I worked on, I found that system's hotspots made up just 4% of the code—but were responsible for 72% of all historic defects! In other words, if our team were to improve just 4% of that codebase, we would get rid of the majority of all defects. Similar situations have been found in empirical research on software defects [2].

JUST A BEGINNING

In this article we learned about hotspots as a way to direct our software quality improvement efforts. Hotspots let you narrow down a large system to specific, critical areas that need your attention.

Hotspot analysis is a powerful technique, but there's so much more we can do once we learn to analyze how our code evolves. Over the past years I've used version-control data to predict bugs, detect architectural decay, find organizational problems that show up in the code, evaluate Conway's Law, and more. Visit the links in the Analyze Hotspots in Code section of this article to download the tool that I use for finding hotspots and see examples of how it's done and how they're visualized.

[1]: http://research.microsoft.com/pubs/69126/icse05churn.pdf
[2]: http://www.research.att.com/techdocs/TD_100504.pdf

ADAM TORNHILL is a programmer that combines degrees in engineering and psychology. He's the author of Your Code as a Crime Scene, has written the popular Lisp for the Web tutorial and self-published a book on Patterns in C. Adam also writes open-source software in a variety of programming languages. His other interests include modern history, music and martial arts.

PROJECT MANAGEMENT SHOULDN'T SUCK
ThoughtWorks®
mingle®
• Work the way you want
• In-app communication
• Reports at your fingertips
ThoughtWorks.com/**mingle**

Is Your Process QA-friendly?

If you have a very vocal QA on your team, you're lucky. Quality Assurance is a unique department and can be an important catalyst to improve your processes. But what if your QA isn't speaking up? Here are some red flags that your current process isn't working for them.

- QAs have an overwhelming inventory of "to do" items
- QAs only speak at the end of the delivery process
- QA interaction with devs is strictly transactional or handover-related

FIXING QA COLLABORATION

If you recognized some of those red flags, here are some things you can do to help.

Involve QAs in Planning Meetings

Encourage QAs to voice their opinions during planning meetings: they can help the team oversee quality risks in the whole delivery process. Acting as a quality consultant, a QA is like a street lamp—he or she sheds light on the risks so the team can plan accordingly.

Pair Devs and QAs

When devs start coding user stories, they can pair with a QA to communicate testing expectations as acceptance tests. This isn't just ATDD (Acceptance Test Driven Development): it's also adding more communication opportunities for the team to stay on the same page.

Connect QA With the Deployment Process

In a continuous delivery environment, let QA deploy to production. Deployment can be a better use of their skills (and potentially more fulfilling) than spending time on searching for unimportant defects.

A QA should be like a street lamp: shedding light on the risks. Evolve your process to support your QA to act as a quality champion.

MOVING TOWARD A BETTER MODEL

Many teams represent their workflow on a physical or digital board. I recommend this because it encourages collaboration and visualizes your daily process. When you're ready to improve your QA process, it's easy to start by changing your board: for example, adding stages like "QA story review " or "deployed by QA." A flexible agile project management tool like Mingle can easily accommodate new stages like these, and will adapt as your team refines its new workflow. As your team and processes evolve, Mingle will support you instead of limiting you.

WRITTEN BY HUIMIN LI
PRODUCT MARKETING MANAGER, **THOUGHTWORKS**

Mingle by ThoughtWorks

ThoughtWorks mingle

Offers tailored workflows, customized reporting, and integrates with more than 50 tools.

CATEGORY	API OR SDK?	OPEN SOURCE?
Project Management	API	No

CASE STUDY

SunGard used ThoughtWorks' Mingle to manage a critical project to replace a legacy financial product for a large government agency. Mingle empowered the geographically distributed team to actualize high returns for their very first Agile project delivery, within the constraints of a fixed budget. Mingle's real-time visibility, ease of use, and inherent adaptability enabled the team to get the most out of Agile, while improving team productivity by 15% and yielding a highly profitable ROI (return on investment) of four times.

STRENGTHS

- Get going quickly with prebuilt templates. Change it anytime, easily
- Through tags, customized properties and people, you can track anything
- Integrate with GitHub and 50+ dev and QA tools
- Actionable team analytics, including burn-up charts and cycle time analytics
- Create custom reports via MQL and macros

NOTABLE CUSTOMERS

- Cisco
- SunGard
- Siemens
- Dillard's
- The trainline
- WestJet
- NHS

BLOG thoughtworks.com/mingle/blog | TWITTER @thatsmingle | WEBSITE thoughtworks.com/mingle

Testing: What It Is, What It Can Be

BY ANDY TINKHAM

QUICK VIEW

01

Some organizations are still stuck in old modes of testing, in which test cycles are often measured in months. The majority of orgs have modern tools and well-defined testing strategies, but they aren't as focused on being information providers.

02

Orgs with highly-effective testers have frequent collaboration between testers, developers, and other stakeholders. Testers use knowledge from several non-IT disciplines to look at problems from multiple angles and employ a good deal of statistical analysis of monitoring data.

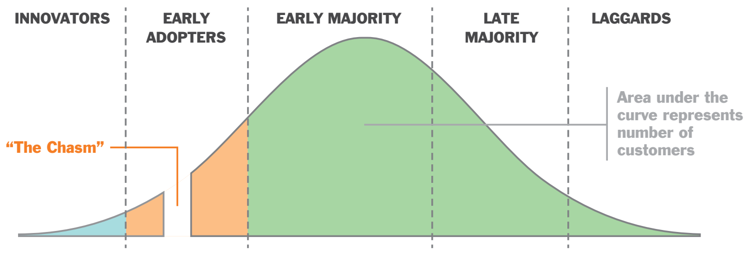
What do you think of when you hear the term "software testing"? Many people I've encountered say it's a group of people refining the software just before release, doing their darndest to break the beautiful software that was handed over to them, cackling with glee each time a new defect is found. Other people have an answer that involves someone as a gatekeeper, akin to the bridge keeper in Monty Python and the Holy Grail, *subjecting each release to questions for which a wrong answer can mean death (at least for the predetermined release date chosen back at the beginning of the iteration).*

While there certainly are testers who fit these stereotypes, the reality of testing is undergoing a bit of a renaissance. Test teams are in the process of adapting their work from the slow, documentation-heavy methods of the past into more flexible and rapid approaches, enabling them to better keep pace with software development. At the core of this adaptation is a realization that testing isn't about finding bugs. If it were, test teams would "fail" more and more as the overall team improved since there would be fewer bugs to find. Instead, modern testing focuses more on providing information at the time it's needed. Bugs are just one piece of that information.

TESTING AS IT IS

Laggards

In a progression similar to the Technology Adoption Lifecycle from Geoffrey Moore's *Crossing the Chasm* (Figure 1), different organizations are at various points in their responses to testing



Technology Adoption Lifecycle

changes in the software industry. At the back of the pack, some organizations function much as test organizations have for decades. Their test cycles are often measured in months. These organizations are likely feeling pressure to reduce their delivery cycle while still finding as many bugs as possible.

The Majority

Moving left along the curve, we find the majority of testing organizations. These teams have begun to adapt, at least in part, to increased time pressure from consumers. Some have brought exploratory methods into their testing, leaving behind rigid scripting and allowing the tester autonomy to immediately incorporate information gathered during testing. Some have also built automation into their efforts for well-defined, repeatable testing work, leveraging tools to perform those tasks rather than humans. Some may have even brought testing work forward in their development process, mobilizing the entire team to catch problems earlier and collaborate on fixes. Teams may not explicitly focus on being information providers and, as such, may be missing opportunities to reach their full potential. However, the testing team is probably achieving some success in keeping up with the rest of their organization, depending on how they interact with the developers.

Early Adopters

The early adopters portion of the curve is where we begin to really see testing innovations, achieving results beyond the old norms. Testers in these organizations routinely analyze systems they are testing from multiple viewpoints or "lenses" [1]. They go beyond tests that confirm software functionality and use risk-based test design to think about specific failure cases. At this level, testers move away from vague failure scenarios and instead draw on skills in experimental design and risk analysis, as they craft tests to reveal if certain imagined failures can actually be triggered. Here we see testing becoming a vibrant and exciting career path.

Testing jobs at the most forward-thinking organizations may not look anything like traditional testing jobs. They may involve skills like data analysis.

Testers in early adopter organizations often draw on concepts from psychology, both to recognize their own limitations (such as confirmation bias and inattentional blindness [2]) and to better understand how others might use their software. These testers are embedded in the overall team, working closely with analysts, developers, and designers to build software the entire team can be proud of releasing.

In teams within these organizations, testers can serve as headlights, illuminating things to come so that the team can react accordingly [3]. They can investigate areas of unanticipated feature interaction or operating conditions such as input data, limited resources (i.e., CPU or memory), and user behavior. These might be things that the larger team hasn't considered.

Innovators

Finally, there are innovators: the vanguards who are defining the new vision of software testing for the rest of the industry. Testing jobs in these organizations may not look anything like traditional testing jobs. They may involve skills like data analysis—digging into massive sets of data to provide their team with detailed insights around their system and users. These testers invent new ways of visualizing the information they gather, and communicate it to their teams effectively and efficiently.

Some teams on the cutting edge of testing work closely with their operations team, exploring the space of DevOps. These test teams may be better-equipped to leverage automation by increasing its sophistication, using it as a tool to support testing work, rather than replacing it. These organizations are continually refining their workflows, discarding those that no longer provide valuable information, tweaking others to keep them relevant, and introducing new tasks to answer questions the team didn't know they had. Testing in this type of organization is a rewarding challenge, a critical role to keep the team moving forward, and a far step from the perceived drudgery of more traditional testing.

THAT'S GREAT! HOW DO I GET THERE?

Most organizations fall into the middle portion of the curve. Moving towards the front of the curve takes effort, but it is achievable. To start, you can:

- **Analyze your current practices.** Effective testing provides information that the team needs. If a task only results in information the team and stakeholders don't value, it may be time to stop the task or change it to make it more useful.
- **Analyze "release day" emotions.** If the team has an uncomfortable mood on release day, it can be a red flag indicating that the team doesn't have all the information needed to have confidence in the release.
- **Pick one unanswered question that the team has at release.** Don't try to make changes all at once. It can be difficult to formulate clear questions, so the team needs time to gain experience. By identifying just one question to answer, you can begin building experience within your culture while making just a small number of changes.
- **Break down walls.** Teams build software best when they function as one team. Break down the walls isolating developers, testers, and analysts. Foster communication between groups that doesn't go through the bug tracker.
- **Look to the cutting edge.** The practices described here are only a small subset of what test teams can do, but they are good starting points. If you are uncertain where to start, seek help—either online, from the broader testing community, or by bringing in someone external with the expertise to meet your needs.

Effective testing provides information that the team needs.

Testing is a critical function in software development, and when utilized effectively, it provides a team with a steady flow of information. This allows the team to make quick, confident decisions, and to avoid repetitious and ineffective work while ensuring you deliver high quality software.

[1] testingbias.com/episodes/10
[2] youtube.com/watch?v=z-Dg-06nrnc
[3] bit.ly/dz-testinglesson

ANDY TINKHAM is the QA Practice Lead at C2 IT Solutions in Minneapolis, MN (c2its.com). He has worked in testing for 20 years, focusing on automation in testing, performance, and testing strategy. He is a founding member of the Association for Software Testing and the Twin Cities Test Automation Group and a frequent speaker. Recently, he co-hosted the Testing Bias podcast with Ian Bannerman and is planning to launch new podcasts in the near future. Follow him on Twitter (@andytinkham) or through his blog at testerthoughts.com.

What Elon Musk Can Teach Us About Agile Software Development

BY GERRY CLAPS

QUICK VIEW

01
The cross-functional, co-located teams of SpaceX and Tesla have shown solid adherence to the agile principle, "Individuals and interactions over processes and tools."

02
Tesla's regular firmware updates show that Elon Musk's company understands having software that works is more important than having comprehensive documentation.

03
Musk understands agile's principle of responding to change over following a plan. There wasn't enough world battery production for his initial plan, so he built his own factory.

After reading an insanely long Wait but Why *series looking into* ***Elon Musk*** *and* ***Tesla****, I realized that the entrepreneur extraordinaire has a link to agile software development that many seem to miss. And more importantly, it's something we can all learn from.*

The man has blown a personal $180m+ to try to change the world with electric cars (Tesla Motors), solar energy (SolarCity), and space rockets (SpaceX). Impressive, right? So it's not very surprising to hear people compare him to Tony Stark, a.k.a. "Iron Man."

But how does this relate to *Agile Software Development*? Let's break down the Agile Manifesto, line by line, to see where Musk puts us to shame (we'll use electric cars from Tesla Motors as ongoing examples).

"INDIVIDUALS AND INTERACTIONS OVER PROCESSES AND TOOLS"

An office where both design and engineering sit side by side? A place where equal weight is placed on both design *and* engineering? That's what SpaceX and Tesla have. That makes for some *real* cross-functional teams. Dependencies are easy to fix when the person with the solution is in the same room as you.

Musk knows that the old way of thinking doesn't quite cut it when you're trying to change the game. **He knows that there needs to be an intense focus on two things: the product and the people building the product**.

Tesla Motors has a $0 marketing plan. They sell directly to the public and only hire very passionate people.

"WORKING SOFTWARE OVER COMPREHENSIVE DOCUMENTATION"

Anyone a fan of Continuous Deployment? Tesla cars receive regular firmware updates, automatically. How is that possible you say? It's a bit like updating Google Chrome. When you connect to the internet, there's a quick check done to see if you have the latest version, and if you don't, it downloads in the background, and then is installed with your next browser open.

That means there's no need for thick user manuals or encyclopedia-like requirements documents for you to create. Simply sketch, prototype, and develop new features; adequately test them (automation helps, see Continuous Integration); and provide an initial prompt to the user when there's something new. **Your customers can continue using your updated software product, without having to do a thing**.

> It's very important to have a feedback loop, where you're constantly thinking about what you've done and how you could be doing it better.
>
> - ELON MUSK

"CUSTOMER COLLABORATION OVER CONTRACT NEGOTIATION"

If Tesla Motors were to negotiate a contract, they would have never developed an electric car to begin with. The 1900s came and went with a multitude of failed attempts at commercializing electric cars (*unfortunately for us*). In other words, electric cars are a proven way to destroy your business.

In spite of this, Elon Musk saw that electric cars were the future. He may not have directly spoken to customers (*initially*), but he did speak to the world. A zero-footprint car was the aim. And Tesla Motors was the best solution.

Had Musk opted for a better contract, we would not see the innovation that Tesla Motors has achieved to date. **To succeed, innovation must transcend contract negotiations.**

"RESPONDING TO CHANGE OVER FOLLOWING A PLAN"

Imagine you find out the perfect *commercial* design for an electric car battery requires you to use all the lithium ion batteries currently being produced in the world, *as they are being made*. Bummer, next idea I guess.

> Failure *is* an option here. If things are not failing, you are not innovating enough.
>
> - ELON MUSK

Not for Musk—he decided to build a Gigafactory that will produce more lithium ion batteries than the entire world was producing in 2013, by 2020. And, at a fraction of the cost (*by approximately one third*).

So I guess the lesson here is, if life throws you lemons, figure out a way to draw electricity from them.

It's not hard, it just requires effort. The above principles and practices are all simple things to execute. Yet so many large (and even small) organizations fail to do so. Part of it is a lack of self-awareness. Inefficiencies can be hard to spot with a workforce in the 1000s. Another part isn't though. When there's a sea of red tape and goals aligned solely to departmental revenue increases, there's probably a need for you to inject some of the practices Musk uses to get Tesla Motors innovating.

I couldn't end this any better than by providing a quote by the inspiration of Tesla Motors himself, Nikola Tesla.

> If your hate could be turned into electricity, it would light up the whole world.
>
> - NIKOLA TESLA

Start bringing positive change to the agile software development team you're a part of.

GERRY CLAPS is the VP of Customer Success at Blossom.io. He's passionate about all things Agile, Lean, Product, and Growth. He has worked as a Business Analyst, Scrum Master, and Product Owner for both the enterprise and consultancy. Follow him on twitter (@gclaps)

Code Review Checklist

This checklist includes basic things to look for in your code reviews, but you should also allow new styles and patterns specific to your own team to emerge and evolve. When they do, make your own code review checklist.

Remember: *Review your own code* before submitting it for a review.

Architecture/Design

- ☐ **Single Responsibility Principle:** The idea that a class should have one and only one responsibility. You might want to apply this idea to methods as well.
- ☐ **Open/Closed Principle:** If the language is object-oriented, are the objects open for extension but closed for modification? What happens if we need to add another one of x?
- ☐ **Code Duplication (DRY):** Don't Repeat Yourself is a common practice. One duplication is usually okay, but two are not.
- ☐ **Squint-Test Offenses:** If you squint your eyes, does the shape of this code look identical to other shapes? Are there patterns that might indicate other problems in the code's structure?
- ☐ **The Boy Scout Rule:** If you find code that's messy, don't just add a few lines and leave. Leave the code cleaner than you found it.
- ☐ **Potential Bugs:** Are there off-by-one errors? Will the loops terminate in the way we expect? Will they terminate at all?
- ☐ **Error Handling:** Are errors handled gracefully and explicitly where necessary? Have custom errors been added? If so, are they useful?
- ☐ **Efficiency:** If there's an algorithm in the code, is it using an efficient implementation? (e.g. iterating over a list of keys in a dictionary is an inefficient way to locate a desired value.)

Style/Readability

- ☐ **Method Names:** Methods should have names that reveal the intent of the API while fitting into the idioms of your language and not using more text than is necessary (e.g. it's not "send_http_data" it's "post_twitter_status").
- ☐ **Variable Names:** foo, bar, e: these names are not useful for data structures. Be as verbose as you need (depending on the language). Expressive variable names make code easier to understand.
- ☐ **Function Length:** When a function is around 50 lines, you should consider cutting it into smaller pieces.
- ☐ **Class Length:** 300 lines is a reasonable maximum for class sizes, but under 100 lines is ideal.
- ☐ **File Length:** As the size of a file goes up, discoverability goes down. You might consider splitting any files over 1000 lines of code into smaller, more focused files.
- ☐ **Docstrings:** For complex methods or those with longer lists of arguments, is there a docstring explaining what each of the arguments does if it's not obvious?
- ☐ **Commented Code:** Sometimes you'll want to remove any commented out lines.
- ☐ **Number of method arguments:** Consider grouping methods and functions with three or more arguments in a different way.
- ☐ **Readability:** Is the code easy to understand? Do I have to pause frequently during the review to decipher it?

Communicating Your Review

- ☐ **Ask questions:** How does this method work? If this requirement changes, what else would have to change? How could we make this more maintainable?
- ☐ **Compliment / reinforce good practices:** One of the most important parts of the code review is to reward developers for growth and effort. Few things feel better than getting praise from a peer. Try to offer as many positive comments as possible.
- ☐ **Discuss in person for more detailed points:** On occasion, a recommended architectural change might be large enough that it's easier to discuss it in person rather than in the comments. Similarly, if discussing a point and it goes back and forth, try to pick it up in person and finish out the discussion.
- ☐ **Explain reasoning:** It's often best both to ask if there's a better alternative and justify why a problem is worth fixing. Sometimes it can feel like the changes suggested can seem nit-picky without context or explanation.
- ☐ **Make it about the code, not the person:** It's easy to take feedback from code reviews personally, especially if we take pride in our work. It's best to make discussions about the code rather than about the developer. It lowers resistance, and it's not about the developer anyway, it's about improving the quality of the code.
- ☐ **Suggest importance of fixes:** Try to offer many suggestions, not all of which need to be acted upon. Clarifying if an item is important to fix before it can be considered done is useful both for the reviewer and the reviewee. It makes the results of a review clear and actionable.

Testing

- ☐ **Test Coverage:** New features should have tests. Are the tests thoughtful? Do they cover the failure conditions? Are they easy to read? How fragile are they? How big are the tests? Are they slow?
- ☐ **Testing at the Right Level:** Are the tests as low level as they need to be in order to check the expected functionality? Testing at a high level by accident can create a slow test suite, so it's important to be vigilant.
- ☐ **Number of Mocks:** If a test has more than three mocks in it, you should check if it is testing too broadly or the function is too large. Maybe it doesn't need to be tested at a unit test level and would suffice as an integration test.
- ☐ **Meets Requirements:** Review the requirements of the story, task, or bug which the work was filed against. If it doesn't meet one of the criteria, it's better to bounce it back before it goes to QA.

WRITTEN BY:

KEVIN LONDON is a software developer at Wiredrive. This list was distilled from his real-world code review best practices. He's also pursuing a Masters in Comp Sci at Georgia Tech and enjoys coding in Python and Django.

Solutions Directory

This directory contains solutions for source control, static code analysis, issue tracking, project management, code review, and test management. It provides feature data and product category information gathered from vendor websites and project pages. Solutions are selected for inclusion based on several impartial criteria, including solution maturity, technical innovativeness, relevance, and data availability.

SOFTWARE QUALITY

PRODUCT	CATEGORIES	HOSTING	FREE TRIAL	WEBSITE
Accurev by Borland	Source Control	On-Premise	30 Day Free Trial	borland.com
Acunote	Project Management	SaaS	30 Day Free Trial	acunote.com
Aha!	Project Management	SaaS	30 Day Free Trial	aha.io
ALM by HP	Project Management	On-Premise or SaaS	On-Premise: 60 Day Free Trial, SaaS: 30 Day Free Trial	hp.com
AnswerHub by DZone Software	Project Management, Ideation, Knowledge Management, Collaboration	SaaS, On-Premise	Free trial is 15 days, with access to all features.	dzonesoftware.com
Application Quality and Testing Tools by CA	Test Management	On-Premise	Available Upon Request	ca.com
Application Quality Management (AQM) Solution by Original Software	Test Management, Project Management	On-Premise	Available Upon Request	origsoft.com
Application Quality Management by Oracle	Test Management	On-Premise	Free	oracle.com
Appvance	Test Management	On-Premise or SaaS	N/A	appvance.com
Asana	Project Management	SaaS	Available Upon Request	asana.com
Assembla	Project Management	Saas	15 Day Free Trial	assembla.com
Basecamp	Project Management	SaaS	60 Day Free Trial	basecamp.com
Blazemeter	Test Management	On-Premise or SaaS	Free Tier	blazemeter.com
Blossom	Project Management	SaaS	14 Day Free Trial	blossom.co
Bugzilla by Mozilla	Issue Tracking	On-Premise	Open Source	bugzilla.org
Confluence by Atlassian	Project Management	On-Premise or SaaS	Available Upon Request	atlassian.com
DataMaker by Grid-Tools	Test Management	On-Premise	15 Day Free Trial	grid-tools.com

SOFTWARE QUALITY

PRODUCT	CATEGORIES	HOSTING	FREE TRIAL	WEBSITE
Development Testing Platform by Parasoft	Test Management	On-Premise	Available Upon Request	parasoft.com
EggPlant by TestPlant	Test Management	On-Premise and SaaS	Available Upon Request	testplant.com
Endevor by CA	Source Control	On-Premise	Available Upon Request	ca.com
Fitnesse	Test Management, Project Management	On-Premise	Open Source	fitnesse.org
Flow	Project Management	SaaS	30 Day Free Trial	getflow.com
FogBugz	Issue Tracking, Project Management	On-Premise or SaaS	Available Upon Request	fogcreek.com
Gauge by ThoughtWorks	Test Management	On-Premise	Open Source	getgauge.io
GitHub	Source Control, Issue Tracking, Code Review	On-Premise or SaaS	45 Day Free Trial	github.com
IntelliJ IDEA by JetBrains	Source Control, Static Code Analysis	On-Premise	Free Community Edition	jetbrains.com
JIRA by Atlassian	Issue Tracking	On-Premise or SaaS	7 Day Free Trial	atlassian.com
Kanban Tool	Project Management	On-Premise and or SaaS	14 Day Free Trial	kanbantool.com
LeanKit	Project Management	SaaS	30 Day Free Trial	leankit.com
Mingle by ThoughtWorks	Project Management	SaaS	30 Day Free Trial	thoughtworks.com
NeoLoad by Neotys	Test Management	On-Premise	25 Day Free Trial	neotys.com
Pivotal Tracker	Project Management	SaaS	Available Upon Request	pivotaltracker.com
Podio	Project Management	SaaS	Available Upon Request	podio.com
PPM by CA	Project Management	On-Premise	Available Upon Request	ca.com
ProductPlan	Project Management	SaaS	30 Day Free Trial	productplan.com
Quality Center by HP	Issue Tracking, Test Management	On-Premise	Available Upon Request	hp.com
Rally by CA	Project Management, Test Management	SaaS, On-Premise available	Free Community Edition	rallydev.com
Rational product line by IBM	Source Control, Issue Tracking, Project Management,	On-premise	90 Day Free Trial	ibm.com
Redmine	Project Management	On-Premise	Open Source	redmine.org

SOFTWARE QUALITY

PRODUCT	CATEGORIES	HOSTING	FREE TRIAL	WEBSITE
SauceLabs	Test Management	SaaS	Free for Open Source Projects	saucelabs.com
ScrumWorks by Collabnet	Project Management	On-Premise	30 Day Free Trial	collab.net
Silk Portfolio by Microfocus	Issue Tracking, Test Management	On-Premise	45 Day Free Trial	borland.com
SoapUI	Test Management	On-Premise	Available Upon Request	soapui.org
Sprint.ly	Project Management	SaaS	30 Day Free Trial	sprint.ly
Stash by Atlassian	Source Control, Code Review	On-Premise	Available Upon Request	atlassian.com
TargetProcess	Project Management	On-Premise or SaaS	Free, Standard, and On-Site packages, 30 Day Free Trial	targetprocess.com
Tasktop	Code Review, Project Management	On-Premise	N/A	tasktop.com
Team Foundation Server (TFS) by Microsoft	Project Management, Test Management, Source Control, Issue Tracking	On-Premise	Available Upon Request	visualstudio.com
TeamForge by Collabnet	Source Control, Project Management	On-Premise or SaaS	30 Day Free Trial	collab.net
Test Cloud by Xamarin	Test Management	SaaS	Available Upon Request	xamarin.com
Test Studio by Telerik	Test Management	On-Premise	Available Upon Request	telerik.com
TestComplete Suite by SmartBear	Test Management	On-Premise	30 Day Free Trial	smartbear.com
TouchTest	Test Management	SaaS and On-Premise	30 Day Free Trial	soasta.com
Trac by Edgewall Software	Issue Tracking	On-Premise	Open Source	trac.edgewall.org
Trello	Project Management	SaaS	Free, Gold Edition Available	trello.com
Tricentis Tosca Testsuite	Test Management	On-Premise	14 Day Free Trial	tricentis.com
VersionOne	Project Management	On-Premise or SaaS	30 Day Free Trial	versionone.com
Visual Studio by Microsoft	Source Control, Issue Tracking, Static Code Analysis	On-Premise or Saas	Free Trial Available	visualstudio.com
XL TestView by Xebia Labs	Test Management	On-Premise	Available Upon Request	xebialabs.com
YouTrack by JetBrains	Project Management, Issue Tracker, Change Management	On-Premise or SaaS	30 Day Free Trial	jetbrains.com
Zephyr	Test Management	On-Premise or SaaS	Free Community Edition	getzephyr.com

STATIC CODE ANALYSIS

PRODUCT	CATEGORIES	LANGUAGE SUPPORT	FREE TRIAL	WEBSITE
Bithound.io	Code Review	JavaScript	90 Day Free Trial	bithound.io
Black Duck	Open Source Auditing	All major languages	14 Day Free Trial	blackducksoftware.com
Checkstyle	Static Code Analysis	Java	Open Source	checkstyle.sourceforge.net
Clover by Atlassian	Code Coverage	Java, Groovy	30 Day Free Trial	atlassian.com
Code Central by Ncover	Code Coverage	All major languages	21 Day Free Trial	ncover.com
Code Climate	Static Code Analysis	PHP, Ruby, JavaScript, Python	14 Day Free Trial	codeclimate.com
CodeNarc	Static Code Analysis	Groovy	Open Source	codenarc.sourceforge.net
CodeRush	Static Code Analysis	C#, VB10, ASP .NET, HTML, JavaScript, XAML, C++,	30 Day Free Trial	devexpress.com
ConQAT by CQSE	Static Code Analysis	Java, C#, C++, JavaScript, ABAP, Ada and many other languages	Free	cqse.eu
Findbugs	Static Code Analysis	Java	Open Source	findbugs.sourceforge.net
Gitcolony	Code Review	All major languages	30 Day Free Trial	gitcolony.com
Infer by Facebook	Static Code Analysis	Objective-C, Java, or C	Open Source	fbinfer.com
Klocwork by Rogue Wave	Static Code Analysis	C, C++, C# and Java	Available upon request	roguewave.com
Open Logic Enterprise Rogue Wave	Open Source Auditing	All major languages	Free Edition	openlogic.com
Parasoft	Static Code Analysis	C, C++, Java, .NET (C#, VB.NET, etc.), JSP, JavaScript, XML, and other languages	Available upon request	parasoft.com
SAVE by Coverity	Static Code Analysis	C, C++, C# and Java source code	30 Day Free Trial	coverity.com
SonarQube	Static Code Analysis	All major languages	Open Source	sonarqube.org
Sonograph by Hello2Morrow	Static Code Analysis	Java, C#, C/C++	Free for non-commercial use	hello2morrow.com
Squale	Static Code Analysis	Java, C/C++, .NET, PHP, Cobol, ...	Open Source	squale.org
Upsource by JetBrains	Code Review	JavaScript	Free 10 User plan	jetbrains.com

diving deeper

INTO FEATURED SOFTWARE QUALITY PRODUCTS

Looking for more information on individual code quality and software agility solutions providers? Nine of our partners have shared additional details about their offerings, and we've summarized this data below.

If you'd like to share data about these or other related solutions, please email us at research@dzone.com.

AnswerHub

BY DZONE SOFTWARE

TEST MANAGEMENT	–
PROJECT MANAGEMENT	✓
ISSUE TRACKING	–
STATIC CODE ANALYSIS	–
SOURCE CONTROL	–

FEATURES

- Built in chat with user tagging notifications (e.g. @john)
- Team performance metrics/ visualizations
- Project stage tracking, documentation features

BlazeMeter

BY BLAZEMETER

TEST MANAGEMENT	–
PROJECT MANAGEMENT	–
ISSUE TRACKING	–
STATIC CODE ANALYSIS	–
SOURCE CONTROL	–

FEATURES

- Scalable load testing with JMeter or WebDriver
- Test from multiple geographic locations or on premise
- Real-time load profile shaping while test in flight
- Deep integration with Continuous Delivery pipelines and APM platforms.

Code Climate

BY CODE CLIMATE

TEST MANAGEMENT	–
PROJECT MANAGEMENT	–
ISSUE TRACKING	–
STATIC CODE ANALYSIS	✓
SOURCE CONTROL	–

FEATURES

- Cycle/Dependency visualizations
- Issue tracking integration
- Custom metrics and queries
- Technical Debt metrics

FogBugz

BY FOG CREEK SOFTWARE

TEST MANAGEMENT	✓
PROJECT MANAGEMENT	–
ISSUE TRACKING	✓
STATIC CODE ANALYSIS	–
SOURCE CONTROL	–

FEATURES

- Focus on executable documentation
- Built-in project management
- API available

Mingle

BY THOUGHTWORKS

TEST MANAGEMENT	–
PROJECT MANAGEMENT	✓
ISSUE TRACKING	–
STATIC CODE ANALYSIS	–
SOURCE CONTROL	–

FEATURES

- Scrum project features
- Kanban project features
- Built in chat with user tagging notifications (e.g. @john)
- Team performance metrics/ visualizations
- Risk management visualzations

Sauce Labs Automated Testing Platform

BY SAUCE LABS

TEST MANAGEMENT	✓
PROJECT MANAGEMENT	–
ISSUE TRACKING	–
STATIC CODE ANALYSIS	–
SOURCE CONTROL	–

FEATURES

- Mobile testing suite of tools
- Unit testing
- Functional testing
- Cloud-parallel VM testing

Tasktop Sync

BY TASKTOP

SDLC	✓
PROJECT MANAGEMENT	–
ISSUE TRACKING	–
STATIC CODE ANALYSIS	–
SOURCE CONTROL	–

FEATURES

- Syncronizes defects, requirements, tests, help desk tickets, issues and much more
- Increases collaboration, visibility and traceability
- Reduces errors, traceability gaps and wasted time
- Enables non-developers to integrate tools across the entire development lifecycle
- SDLC tool integration

Telerik Platform

BY TELERIK

TEST MANAGEMENT	–
PROJECT MANAGEMENT	–
ISSUE TRACKING	✓
STATIC CODE ANALYSIS	–
SOURCE CONTROL	✓

FEATURES

- Based on Git
- Supports Git server
- Built-in chat and user tagging
- Integration SDK and API available

YouTrack

BY JETBRAINS

TEST MANAGEMENT	–
PROJECT MANAGEMENT	✓
ISSUE TRACKING	✓
STATIC CODE ANALYSIS	–
SOURCE CONTROL	–

FEATURES

- Built-in chat with user tagging notifactions
- Custom metrics and visualizations
- Built-in project management
- Agile mgmt tools
- Large Org performance

diving deeper

INTO CODE QUALITY AND SOFTWARE AGILITY

Top 10 #Testing and #Agile Twitter Feeds

@LISACRISPIN	@JAMESMARCUSBACH	@MICHAELBOLTON	@S_COLSON	@S_2K
@JIMRBIRD	@WAKALEO	@JOHANNAROTHMAN	@GIL_ZILBERFELD	@MATTHEWMCCULL

Code Quality & Software Agility Zones

DevOps Zone

dzone.com/devops

DevOps is a cultural movement, supported by exciting new tools, that is aimed at encouraging close cooperation within cross-disciplinary teams of developers and IT operations/system admins. The DevOps Zone is your hot spot for news and resources about Continuous Delivery, Puppet, Chef, Jenkins, and much more.

Agile Zone

dzone.com/agile

In the software development world, Agile methodology has overthrown older styles of workflow in almost every sector. Although there are a wide variety of interpretations and specific techniques, the core principles of the Agile Manifesto can help any organization in any industry to improve their productivity and overall success. Agile Zone is your essential hub for Scrum, XP, Kanban, Lean Startup and more.

Performance Zone

dzone.com/performance

Scalability and optimization are constant concerns for the developer and operations manager. The Performance Zone focuses on all things performance, covering everything from database optimization to garbage collection, tool and technique comparisons, and tweaks to keep your code as efficient as possible.

Top Code Quality Refcardz

Getting Started with Git
bit.ly/dz-git

Mobile Web Application Testing
bit.ly/dz-mobileweb

Agile Adoption: Improving Software Quality
bit.ly/agileadopt

Getting Started with Domain-Driven Design
bit.ly/domaindriven

Top Code Quality Websites

Ministry of Testing
ministryoftesting.com

DevelopSense
developsense.com/blog

CodeBetter.com
codebetter.com

Top Code Quality Podcasts

Testing Podcast
testingpodcast.com

The Agile Life
thisagilelife.com

Git Minutes
episodes.gitminutes.com

glossary

AGILE A group of software development methods that involves fairly short development cycles with flexible requirements that evolve as the software is built. Self-organizing, cross-functional teams are often another key aspect of Agile.

AUTOMATED TESTING A form of verification that activates the software being tested and looks for a predicted outcome. It asserts that the test passed or failed based on whether the actual outcome matches the predicted outcome.

BEHAVIOR-DRIVEN DEVELOPMENT (BDD) A development methodology that builds on TDD by adding tools and a ubiquitous language that allow business managers to write tests themselves in the form of user stories.

CODE COVERAGE A measure of what percentage of the total lines or blocks of code is executed by your automated tests.

CODE REVIEW A systematic review of source code performed by a developer that did not write the code. The resulting discussion from that review is intended to identify mistakes overlooked in the initial development phase and to help improve the original developer's skills.

CONTEXT-DRIVEN TESTING A testing philosophy that asserts that there are no best practices for testing in every context, and that testing methods must be flexible enough to evolve with projects that often change in unpredictable ways.

EXPLORATORY TESTING A form of test design, test execution, and constant learning that involves a skilled tester flexibly using their experience and creativity to predict issues and experiment with no pre-determined methodology in an effort to more effectively test the software.

EXTREME PROGRAMMING (XP) An agile development methodology created by Kent Beck that includes frequent releases, unit testing all code, extensive code review, and pair programing. All of these practices have heavily influenced the software industry.

FUNCTIONAL TESTING A testing method that includes any type of test that checks a complete section of functionality within the whole system either through basic manual testing of the product or through automated scripts that run expected user actions.

INTEGRATION TESTING A testing stage that occurs after unit testing where software modules are tested as a group to ensure that they work together to complete more complex tasks.

ISSUE TRACKING SYSTEM A tool that stores, organizes, and presents visualizations of recorded feature requests and software bugs with various contextual information to help those who are tasked with fixing the bugs.

KANBAN A work management technique where the development process is illustrated through single tasks displayed in cards on a board for the team to see. On the board, each task is pulled from a queue by team members that are responsible for that task, and each task is tracked from definition to completion.

MANUAL TESTING Any test where a person attempts to complete a task with the software from an end-user's perspective, sometimes with additional tools or monitoring, and decides whether the test passes or fails by seeing if the actual outcome matches the desired outcome.

NEGATIVE TESTING A test strategy that explores how unexpected inputs will affect a system.

PAIR PROGRAMMING A development strategy that involves two people coding together at a single computer, each giving frequent feedback and working together as equals, even if skill levels differ significantly. Some definitions include scenarios where one person writes code while the other watches and gives feedback.

POSITIVE TESTING A test strategy that checks to see if specific inputs yield expected results.

QUALITY ASSURANCE (QA) OR SOFTWARE QUALITY ASSURANCE (SQA) A process, often owned by a separate department, that examines an organization's software engineering practices to ensure that products are meeting specified requirements. The department often includes all software testers.

REGRESSION TESTING Any form of software verification that checks to ensure that no functionality gets broken and no new bugs were created in the process of adding code to a program.

SANITY TESTING A simple, ad-hoc type of test that is often manual and used to check that certain software functionality works roughly as expected.

SCRUM The most well-known agile methodology. It involves short iterations of focused effort called "sprints" and encourages tight collaboration by small, self-organized teams that focus on quick delivery and fast responses to changing requirements.

SOURCE CONTROL A form of revision control (also called version control) that manages changes to a software project by allowing multiple programmers to work on the same source code by creating timestamped copies that can be rolled back, compared with, or merged into the mainline source code.

STATIC CODE ANALYSIS A type of software analysis that measures code without running it. A variety of complexity, security, or business metrics can be gathered depending on the tool used.

TECHNICAL DEBT A metaphor coined by Ward Cunningham to express the future burdens a software project inherits when code is written or designed in a quick but messy way, as opposed to writing better code that is more time-consuming and challenging up front.

TEST-DRIVEN DEVELOPMENT (TDD) An approach to software development that uses tests as the design specifications. In TDD, developers write tests before writing the actual program code. They move on to writing new code only when the code they just wrote passes the original tests.

UNIT TESTING A testing method that checks the functionality of individual source code units, often single methods, classes, or interfaces.

USABILITY TESTING A testing method that gathers feedback from real-world users who try to execute a given set of tasks using the software product. Its purpose is not just to find bugs but also to ensure that the user experience is as streamlined as possible.

USER ACCEPTANCE TESTING (UAT) A testing method that verifies that the application satisfies the entire user story outlined in initial business requirements.

USER STORY A description of a single action that a hypothetical user wants the software product to perform. It describes the type of user and then explains what action they want to perform and why.