Sabotaging Quality

Sacrificing Long-Term for Short-Term Goals

Joy Shafer
joyous1@live.com

Abstract

From the very beginning of the project your team has had discussions about quality. You have unanimously agreed it's a top priority. Everyone wants to deliver a product of which they can be proud. Six months later, you find yourself neck-deep in bugs and fifty-percent behind schedule. The project team decides to defer fixing half of the bugs to a future release. What happened to making quality a priority?

One of your partners is discontinuing support for a product version on which your online service is dependent. You have known this was coming for years; you are actually four releases behind your partner's current version. The upgrade of your online service has been put off repeatedly. Now the team is scrambling to get the needed changes done before your service is brought down by the drop in support. You are implementing the minimum number of features required to support a newer version. In fact, you're not even moving to the most current version—it was deemed too difficult and time-consuming to tackle at this point. You are still going to be a release behind. Are you ever going to catch up? Is minimal implementation always going to be the norm? Where is your focus on quality?

Do these scenarios sound familiar? Why is it often so difficult to efficiently deliver a high-quality product? What circumstances sabotage our best intentions for quality? And, more importantly, how can we deliver quality products in spite of these saboteurs?

One of the most common and insidious culprits is the habit of sacrificing long-term goals for short-term goals. This can lead to myriad, long standing issues on a project. It is also one of the most difficult problems to eradicate. There are other saboteurs: competing priorities, resource deprivation, dysfunctional team dynamics, and misplaced reward systems, to name a few. In this paper I'll focus on the quality saboteur of sacrificing long-term goals for short-term goals.

I will discuss the benefits that can be gained when you make the right long-term investments and the types of problems you'll see if your team is solely pursuing short-term goals. I will show you practical strategies for slaying this saboteur or at least mitigating its effects.

Biography

Joy Shafer is currently a Consulting Test Lead at Quardev on assignment at Washington Dental Service. She has been a software test professional for almost twenty years and has managed testing and testers at diverse companies, including Microsoft, NetManage and STLabs. She has also consulted and provided training in the area of software testing methodology for many years. Joy is an active participant in community QA groups. She holds an MBA in International Business from Stern Graduate School of Business (NYU). For fun she participates in King County Search and Rescue efforts and writes Sci-Fi and Fantasy.

Copyright Joy Shafer 2011

1. Introduction

After twenty years as a tester or a test manager, I've seen team after team struggle with the complex problem of delivering high-quality software on time. I've been on teams that have excelled. I've also been on teams that have failed miserably. I've learned a lot from both experiences. There are reasons behind the failures. Sometimes they are even reasonable reasons.

In theory, fixing problems is a straightforward process: (1) understand the problem, (2) brainstorm a solution, (3) implement the solution, and (4) examine your results. If the results are not as desired, go back to step one. Otherwise, celebrate your success and look for the next problem to tackle.

In practice, fixing problems can be very challenging. What seems like the obvious reason for a problem may not be the reason at all. There are usually layers of issues, not the least of which are roadblocks thrown up by people. Individuals often have their own reasons for fighting change, which usually have nothing to do with purposefully sabotaging success.

In this paper I'll examine these saboteurs—not the people, but the reasons they may have for undermining quality—and discuss strategies for mitigating them.

2. Sacrificing Long-Term for Short-Term Goals

Sacrificing long-term goals for short-term goals is probably the most common problem and also the most difficult to eradicate. There are usually compelling arguments for investing in short-term goals. Some of them are quite justifiable. However, if your team consistently sacrifices long-term goals for short-term goals, you'll find yourselves unable to deliver efficiently even on those short-term goals.

The types of things that tend to be sacrificed are:

1. Investment in test, particularly test automation
2. Software quality and maintainability
3. Investment in infrastructure
4. Employee morale and vitality

2.1 Investment in Test Automation

It has been my experience that investment in test, particularly in test automation, pays off handsomely in the long run. In order to have an efficiently running development engine you'll need to have the following:

1. Automated Build Verification Tests (BVTs)
2. Continuous Integration
3. Unit testing
4. Automated regression tests
5. The ability to emulate external components

2.1.1 Automated Build Verification Tests (BVTs)

Build verification tests are those mainstream tests that you run every time you get a new code drop in order to verify that your code is functional and stable enough to test.

It is imperative that you have a robust set of BVTs. By robust, I don't mean extensive. You can start with a minimum suite and add on as desired. What I mean by robust is reliably functional. Not brittle. Not dependent on anything other than the code to make them pass. The test cases should be definitive: if a

Copies may not be made or distributed for commercial use

test fails it's because of a bug in the code, not because of a timing issue, a setup issue, a data issue, or some other non-code problem.

One of the teams I joined had BVTs that ran daily. Great! Except the pass rate for those BVTs was less than sixty percent—every day—for years! What was going on? Did no one care?

When I started asking questions, I was told, "A lot of those failures are because our service is dependent on other teams' services. They don't keep their integration environment stable, so when they are down, those tests will fail."

Other tests were failing because the person who used to maintain them had left the team and hadn't been replaced yet. (It's no surprise that this team was having trouble hiring and retaining good people.) Still other tests were failing due to hardware issues that no one had time to troubleshoot.

The clean-up process for this mess was excruciatingly slow, partly because, as you can imagine, failing BVTs was the tip of the iceberg for the deep-seated problems on this team. You'll hear a lot about them here, because they make a very good example of how a team can sabotage quality.

One of the things we did to mitigate these problems was to build emulators[1] for as many of the external dependencies as we could. The merits of building emulators warrant their own discussion later in this paper (section 2.1.5, Emulate External Components). We also made sure each BVT had a tester responsible for maintaining it, and we cleaned up the test lab environment—also discussed later (section 2.3.1, Maintaining your Physical Environment).

Creating BVTs that are not definitive is a common problem. I worked on a team where the data in their automation, including BVTs, was real customer data that was refreshed in the test environment on a regular basis. The problem was that testers could not depend on a particular piece of data being in the same state every time. This caused numerous failures in the automation with each data refresh.

When I pointed out that the automation should create its own data and then clean up after itself when it was done, I was told that that would be too difficult and time-consuming. The business rules were too complex. The team would never be able to recreate the exact conditions they needed.

I don't believe that. It would take an upfront investment in time, which considering how much time was spent by testers working around this issue with every database refresh, was easily justifiable. It would also take expert knowledge of the business rules, which I suspect may have been missing from the test team, but was certainly available somewhere in the company. That knowledge could and should be captured and transferred to the test team. These problems are solvable, once you get to the root of them and figure out how to get around the personalities involved.

2.1.2 Continuous Integration

I cannot overstress the importance of automated BVTs. Without automated BVTs, there can be no continuous integration.[2] Continuous integration is the process of checking in code and creating a new build on a continuous or near-continuous basis. It is a cornerstone of efficient software development. For continuous integration to work well, once the build is created, it should be automatically deployed to a test

---

[1] Two of the dependencies didn't get emulators, one of them because the service was extremely complex and regularly updated, so an emulator would have been difficult to build and maintain; that service also had a solid track record of keeping their integration environment stable, making an emulator less critical. The other service did not get an emulator because it was being deprecated.

[2] Martin Fowler, "Continuous Integration," *MartinFowler.com,* May 1, 2005, http://martinfowler.com/articles/continuousIntegration.html.

environment where the automated BVTs will kick-off. The developer will get very timely feedback on whether she has broken the build.

If your team does not integrate often—I recommend at least daily—you run the risk of building bad code on top of bad code. When you do finally integrate it will be more difficult to determine the causes of failures. Your development timeline will be much less predictable (read: you're going to slip, probably multiple times).

I worked with a team who only integrated their code once every few months. I was astounded that they ever shipped anything. By contrast I worked on a globally distributed team that integrated twice daily with an automated build/deployment/test system. It was the most efficient team on which I've ever worked.

2.1.3 Unit Testing

Unit testing is another activity that is often sacrificed due to 'not enough time.' In reality, you don't have enough time to forgo unit testing. Unit testing provides more than enough benefits to make it a justifiable activity.[3] Among these benefits are:

1. Cleaner code: Not only is the code less buggy, but it is often simpler and more elegant. The process of unit testing gives the developers a different perspective on their code. They may well come up with a better design in order to accommodate the required unit testing.
2. Tools: To facilitate their unit tests, developer will create test hooks, mock objects, and other devices very useful to the test team. Test and development should work closely together to share resources and knowledge.
3. Team Building: The developers acquire a better appreciation of the challenges testers face.

How do you start unit testing if your team has never done it before? The first thing you'll need, as is the case with many of these recommendations, is management support. Your managers need to communicate to the development team that unit testing is required—not optional. To facilitate the on-boarding process, you'll want to implement an appropriate measurement and reward system.

One of the teams I was on successfully added unit testing to its development efforts by insisting that code coverage be run on all code before it was checked in. The unit tests had to cover at least 70% of the code. This process was implemented for all new code as well as for every bug fix in the legacy code.

This worked very well. There were some holdouts on the development team, but statistics were made visible and soon there was a small competition among the developers for who could hit the highest coverage rate on new code check-ins.

Within a few months of implementing this process, our code had gone from zero unit tests to almost 16% of the code being covered by unit tests. Added to the other best practices this team was following, our development timeline became extremely predictable. We were held up as the 'poster child' for development efficiency.

2.1.4 Automated Regression Tests

Automated regression testing is a key component of an efficient and successful software development process. Without automated regression testing, teams of manual testers will need to comb through the software with every release, laboriously running the same tests they've run countless times before. In this case, it becomes easy to cut corners. You haven't seen a particular test case fail ever. Your manager is

---

[3] J. Timothy King, "Twelve Benefits of Writing Unit Tests First," *J. Timothy King's Blog,* July 11, 2006, http://blog.jtimothyking.com/2006/07/11/twelve-benefits-of-writing-unit-tests-first.

Copies may not be made or distributed for commercial use

breathing down your neck to sign-off. You skip the test this time, marking it ‘pass’. Oops! This time it would have failed.

Automation doesn’t lie. Well, sometimes it gives a false positive, but well-written automation will not give a false negative. Bugs that slip by the automation tend to be the ones for which a test case was never created.

Automation is also many times faster than manual testing—once it’s written. It becomes a no-brainer to run whatever automation you have on a regular basis, perhaps at night, to ensure that your continuous integration stays stable.

Sometimes it’s difficult to justify the expense of automation, especially if your team is starting from scratch. However, if you have an efficient test development team, equipped with proper knowledge and tools, you’ll find that the return on investment (ROI) for writing automated tests usually exceeds the ROI for manual tests, certainly within a few years and sometimes within six months.[4] Other benefits include being able to run tests that otherwise could not be run (better coverage), a more interesting and rewarding experience for the testers, and an increase in the agility of the development effort—test automation supports other best practices, such as continuous integration and code coverage measurement.

2.1.5 Emulate External Components

The concept of testing all the various parts of the system in isolation has been around for almost as long as software development. It used to be done with drivers and stubs. In vogue now are mock objects,[5] although, if done on a component level they are sometimes called emulators, simulators, or sinks.

It may seem like a daunting undertaking to build an entire component that mimics the functionality of one of your dependencies. However, you typically don’t need to build in all the functionality, or even build much at all. You just need to create a mechanism that takes input and gives appropriate output. There is no need to have it actually pull real data on the backend or anything close to the complex functionalities that the real software or service actually accomplishes.

If you’re working for a large company, I recommend asking around. Some other team may well have already built the emulator you need. Perhaps with a few modifications it will work for your purposes as well.

An added benefit of emulators is they are extremely handy—often essential—for performance testing. It is not uncommon for emulators to be built specifically for this purpose, typically near the end of the project when the team finally gets around to performance testing. However, you’ll find if you build the emulators up front, they are useful in every phase of the project, including unit testing and BVTs.

2.2 Software Quality and Maintainability

2.2.1 Maintaining Software Quality

Putting off bug fixes to the end of the project is another tactic that falls under the category of sacrificing long-term for short-term.

---

[4] Douglas Hoffman, “Cost Benefit Analysis of Test Automation,” paper presented at STAR99, 1999, http://www.softwarequalitymethods.com/html/papers.html#CostBenefit.

[5] Steve Freeman and Nat Pryce, *Growing Object-Oriented Software Guided by Tests*. (Boston: Pearson Education, 2010).

Copies may not be made or distributed for commercial use

When you are running up against your deadline and there is still a mountain of open bugs in your software, you will likely either slip or ship with bugs. It is common for teams to postpone bugs that normally would have been fixed because they don't want to miss their deadline. Keep in mind that for every known bug, there are also unknown bugs. If you ship with lots of known bugs, you run a higher risk that end-users will find even more bugs, some of which could be very costly.

When a team is close to shipping and behind schedule, it is very difficult to justify bug fixes, especially since they may destabilize a mostly-working software system. The solution is to fix bugs as you find them.[6]

There are a number of advantages to doing this:

1. You won't carry a backlog of bugs from release to release, causing the next project to start with a "bug debt." If your team lets the backlog get too high by pushing bug fixes to the next release, you'll eventually get to the point where the software is almost unmaintainable and the backlog is so large that it's daunting to even understand the extent of the work ahead.
2. Your development cycle will become more predictable. Some bugs are notoriously difficult to fix. If you put these off to the end and then discover a doozy—you're slipping.
3. Bug fixes may introduce new failures. If bugs are not fixed until the end of the project, then there is less time for the fallout bugs to be found. You will likely ship with some undiscovered, possibly serious, bugs.

Sometimes, with a particularly buggy area of the software, what makes the most sense is to refactor it instead of patching it up with individual bug fixes. Code refactoring means restructuring or rewriting an existing body of code without modifying its functionality. It's done to increase the maintainability of the software by making it more readable, extensible, modular and/or less complex.

Really buggy areas usually have high-cyclomatic complexity.[7] Cyclomatic complexity is a software metric that measures the number of linearly independent paths through a program's source code; some code coverage tools include the ability to measure cyclomatic complexity. Software with high cyclomatic complexity is typically also difficult to maintain, arduous to test well, and very brittle—poke it here, it bulges there.

Whether refactoring an area of the software is justifiable is a decision that needs to take into account how much time is spent dealing with the backlash of the bad software. Does it directly affect customers, and therefore customer support, marketing, the live site support team, and so on? Do the developers avoid making changes in that area because of the time and risk involved? Are lots of resources being spent because of the buggy code? Is it in a component that's going to be required for a long time yet to come? Is the technology obsolete? Will it not be supported with future versions of the operating system or database? If the answers are "yes," it may be worthwhile to refactor.

Even if refactoring doesn't make sense, keeping bug counts low on your software does.

For teams that already have a high backlog, I recommend doing a "quality" release—or maybe even several quality releases, depending on the size of your backlog. The focus of a quality release is on fixing bugs and/or refactoring areas of your software. New features should not be permitted into the quality release.

---

[6] James Shore, *The Art of Agile Development*, (Sebastopol: O'Reilly, 2007), http://jamesshore.com/Agile-Book/no_bugs.html.

[7] Julias Shaw, "Cyclomatic Complexity—What Is It and Why Should You Care?" *Java Metrics,* April 15, 2010, http://java-metrics.com/cyclomatic-complexity/cyclomatic-complexity-what-is-it-and-why-should-you-care.

Copies may not be made or distributed for commercial use

For teams that are mostly keeping up, or teams that are using an agile approach, one technique that works well for keeping bug counts low is a 'bug jail'. Any developer that has more than three or five or eight – you decide—bugs open against them isn't allowed to develop new code until they get their bug counts below the threshold. You might opt to move the threshold to a lower number the closer you get to release. Bugs should be fixed as they're found, not put off to the end of the project.

2.2.2 Software Maintainability

Maintainability is generally defined as the ability to make modifications to the software after delivery. I'd like to expand this definition a little to include not only software modifications, but also deployment—something you'll have to do if you make changes. How easily your software can be modified and deployed makes a huge difference in how much time you'll spend on maintenance after your software or service is on the market.

When building a version 1.0 software or service, it's easy to overlook maintainability as a goal for the product. Especially since new product development tends to be very time-sensitive—those products first on the market gain the largest market share—there will be lots of pressure from management to ship quickly. But if you keep maintenance in mind during the design phase of your product, you will save your team lots of pain and drudgery down the road.

As early as possible in the life of your product, start thinking about investing in the maintainability of your software or service. If not, you could find yourself in a situation similar to one of the teams I was on.

This team developed a service that supported cell-phone technology and, as you can imagine, our user-base grew exponentially. Unfortunately, the team had not spent any resources on maintainability for the service. As a result, when I joined the team, they were spending about eighty percent of their time trying to maintain the software and only about twenty-percent of their time was available to develop new features and meet partner requests.

Here is a litany of the issues facing this team caused by not investing in maintainability:

1. The live site was unstable

   This was the most insidious of the issues facing this team. Whenever there was a live site issue, which was almost daily, it was 'all hands on deck'. Whatever resources were required were yanked off of the project they were working on and told to fix the live site.

   The service had millions of users. Outages caught the eye of senior management and caused screams from the poor customer service technicians who had to deal with the irate customers. Not to mention the affect it had on our bottom line: fully forty percent of the people who tried the service abandoned it within the first few months. Once we stabilized the live site, this figure plummeted to eleven percent.

2. Deployment was a nightmare

   In addition to resources being poured in to maintaining the brittle live site, every time a fix for a live site issue was developed, it needed to be deployed. Not just deployed once to the live site, but also deployed to several different test environments for various purposes: BVT testing, functionality testing, integration testing, performance testing, and staging environment testing.

   Unfortunately, the deployment engine had been designed with a single deployment in mind. Every time a change was needed to deployment, not a rare occurrence, the change would need to be manually entered into a huge array of 'cluster' scripts. Of course, the changes never made it cleanly into all of the environments and countless hours were spent with each and every deployment troubleshooting what went wrong.

Copies may not be made or distributed for commercial use

Because the deployment system was so prone to problems, a lengthy process had developed around deployments to the live site. For each release a long detailed document was written to support the deployment. The deployment was tested numerous times in various environments, and the document was edited, updated, and approved. Even this arduous process didn't prevent deployment issues in production.

By the time I joined the team, the deployment mechanism was so huge and complicated, that to overhaul it into an efficient system would have been a large project on its own. This deployment project, I am sure, would have paid for itself within a few releases, but it was not supported by management. Thus, a huge amount of resources continued to be sucked into the maintenance and testing of the deployment engine.

3. The physical infrastructure was unreliable

   When I started on this team, fully fifty-percent of the machines in use in the test lab were past warranty. Upon investigation, I discovered some of the network routers were more than ten years old.

   Both the physical infrastructure and the software service on this team were poorly maintained. Every time there was an issue, the IT department called in the devs to look at it. Logging had not been added to the code for even the most basic troubleshooting. The devs would very quickly throw up their hands and push the problem back to the IT department. This friction added more stress to an already unhappy situation.

   A tremendous amount of resources were wasted on troubleshooting issues that would never have occurred in the first place in a well-run, properly-maintained test lab. Although not a software maintainability issue, the poorly maintained lab added significantly to the time the team spent on non-productive activities. This topic is covered further in section 2.3.1, Maintaining your Physical Environment.

Dealing with the above three issues probably took up about seventy percent of the team's available time. With proper investment in maintenance and infrastructure, teams should be efficient to the point where less than twenty percent of a team's time is taken up in maintenance activities.

2.3 Investment in Infrastructure

Surprisingly little has been written about investing in infrastructure. It has been my experience that investment in infrastructure almost always pays for itself, sometimes fairly quickly. By infrastructure, I mean the systems that support your core activity. The core activity is software development. In support of this are the physical infrastructure: servers, networks and software; logging and error handling infrastructures; the deployment system; development methodology and processes; and metrics. I'm going to discuss three of them here:

1. Physical Infrastructure
2. Logging
3. Metrics

2.3.1 Maintaining your Physical Environment

Maintaining your hardware and network system should not even be something there is a serious discussion about. Just do it!

Often when hardware goes bad, it does so spectacularly, with system beeps, blue screens, black screens or even smoke. Sometimes it simply won't power-up any more. These problems are generally easy to identify and remediate. However, sometimes when hardware goes bad, it does so discretely. It fails

intermittently. Or it doesn't fail completely, it just slows way down. Or it starts sending cryptic errors to other systems which cause them to fail but don't necessarily lead the troubleshooter back to the root cause of the problem. A ton of time can be wasted trying to troubleshoot hardware that should have already been scrapped.

I took over the test lab on the above referenced team shortly after I started there. One of the things I discovered very quickly was, in addition to the ancient hardware, patches were not applied on a regular basis. Many of the machines in the lab had not been patched in years. I was astounded that the team had been able to get away with this.

I was told the reason for the deplorable state was because the lab was always too busy. The IT people were never allowed the time that was needed to bring the lab down to get everything updated. With this team being in perpetual crisis mode, I could understand that mentality on some level. However, we needed to turn it around.

I coordinated with the appropriate people, over-communicated the changes we were about to make, and then implemented a process whereby once a month, the lab was unavailable from 6am to 10am for patching. This change went smoothly, and as it turned out, happened in the nick of time.

One of the network engineers, someone not on our team but on a different corporate IT team, accidentally created an open channel from our test lab to the Internet. This was discovered fairly quickly by corporate security and our lab was shut down. A security team came in and did an audit of our systems.

They were impressed that we were completely up to date with our patching, and as a result, released us to normal operations within 24 hours. If they had done the same audit two months earlier, there likely would have been some very negative consequences.

2.3.2 Logging

Some of the reasons to build logging into your software or service include providing the ability to troubleshoot issues, supplying a means to prove to partners that you're meeting your Service Level Agreement (SLA) requirements, and determining customer usage patterns.

Without appropriate logging, it may be almost impossible to troubleshoot an issue. Often logging is added by a team simply because they've run into a problem and cannot fix it until they understand it, and can't understand it without more information.

Logging can be instrumental in alerting you to brewing problems with enough advance notice to proactively prevent a crisis. A common example of this is impending performance issues. Logging and analyzing the appropriate statistics can let you know that, if the customer base continues to grow at the present rate, you have about six months before your live site will start to melt down.

Logging and analyzing customer usage patterns can help you design software to better meet their needs, market the software more effectively, and design better real-world test cases.

Ideally logging infrastructure is designed and built along with the initial development of the software. Logging itself can be added incrementally, but the structure and policies surrounding it should be in place from the beginning. Logging is a great example of a long-term investment that can provide tremendous benefit. Figure out how to squeeze it in.

2.3.3 Metrics

The main goal of collecting metrics is to understand the health of your system and processes so you know where to invest in improvements. A secondary goal of metrics, and one that many people overlook or underestimate, is to drive desirable behavior.

Copies may not be made or distributed for commercial use

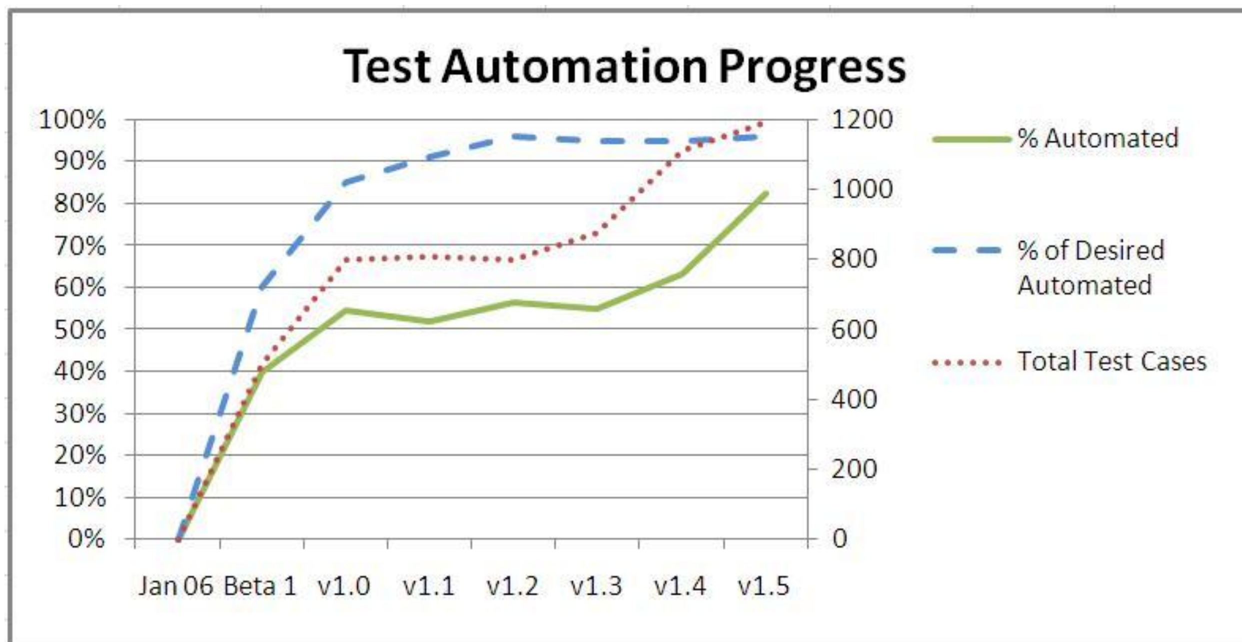
I'm not going to talk about which metrics you should track here, because that's not the focus of this paper. What I do want to mention is that some metrics take time and effort to collect. It is often difficult to make the argument to invest in metrics because they tend to be a long-term payoff activity. However, the teams I work with that have well conceived, well analyzed, and appropriately communicated metrics are also the teams who deliver the highest quality software on time.

You have to be careful with metrics, because you may end up driving undesirable behavior. For example, on one of the teams I was on, to help me determine which test metrics to track, I sent an inquiry to one of the test lead aliases asking which metrics others tracked, and specifically, how best to track test automation.

I got only a few positive responses. Mostly what I got was a barrage of advice from indignant test leads saying I should not even try to track such metrics because they would be used against me. I definitely thought it was worthwhile to track some type of metrics, but armed with all these dire warnings, I decided not to track the percentage of test cases that were automated, but rather to track 'of the cases that we want to automate, what percent is automated'?

At the time most of my test team was in India, and we were developing a couple of different brand new services. We went from zero to 95% automation, based on my statistics, within eight months. Thereafter we stayed about the same. I was pretty happy with the results until I looked at the total percent of test cases automated—it was running under 60%. This didn't feel right to me; I was sure that we should be automating a higher percentage of overall test cases.

I changed the reporting requirements to include the 'number of test cases' and 'percent of total test cases' as well as the previously collected 'percent of what we desire to automate.' Immediately thereafter both our total test cases and our automation percentage started to climb. Fairly quickly the team had automated 82% of the total test cases. The reporting change took place starting with v1.4 as shown below.



2.4 Employee Morale and Vitality

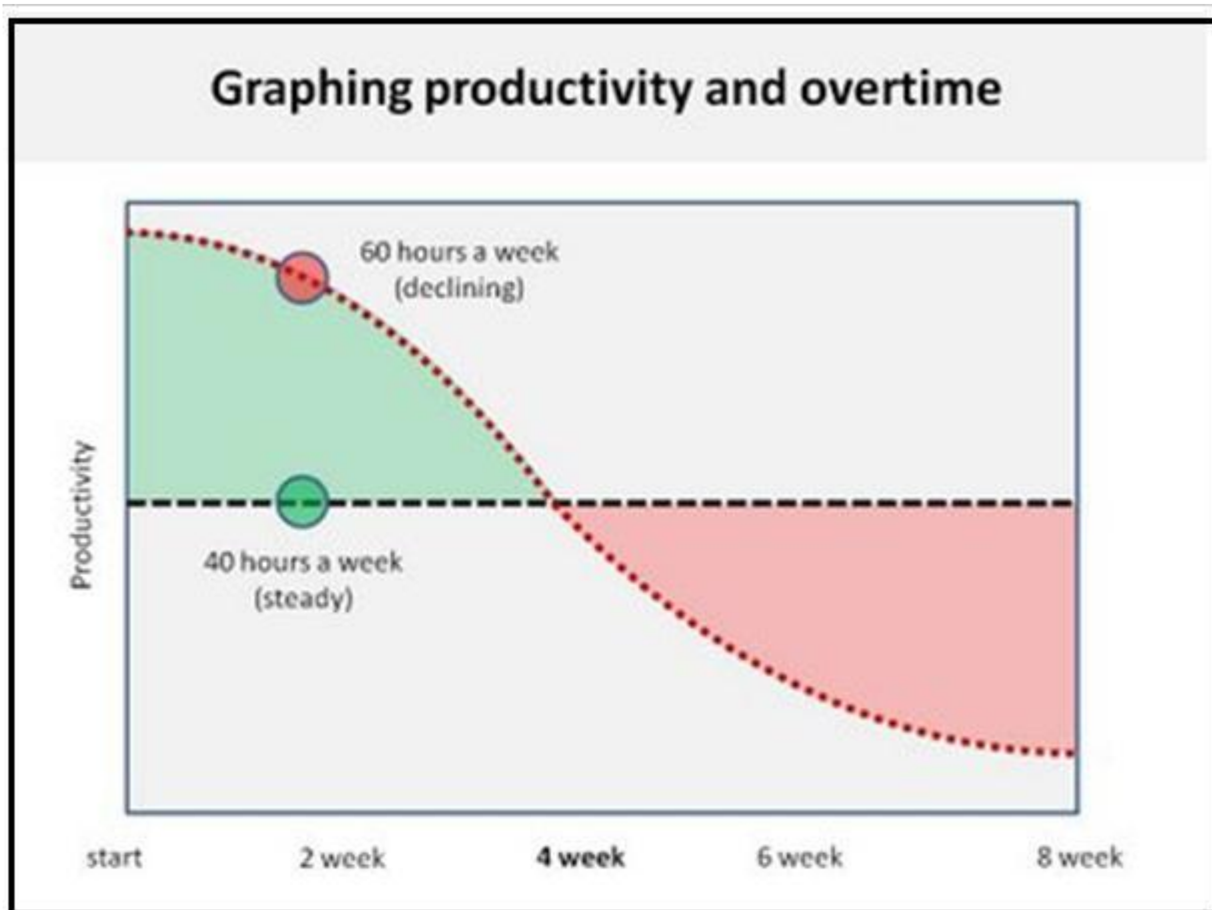
You're behind schedule, the ship date is looming, and it's too late to bring more people on board to help out. The obvious solution is to get the people you do have to work harder, smarter, and longer hours. On paper this looks like a viable option and, indeed, it usually works in the very short-term if you have a

Copies may not be made or distributed for commercial use

healthy team. Give your team a pep talk, get everyone to come in on the weekend, and viola, the work is done a few days earlier than it otherwise would have been done.

The problem is that working long hours can become a habit very quickly—a habit that is extremely destructive. For some teams, sixty-plus hour weeks are the norm. These teams tend to have myriad problems, not the least of which is productivity, in spite of the long hours. Numerous studies have shown that working fewer hours may increase productivity[8] and working longer hours actually decreases productivity.[9] I strongly suspect that an analysis of productivity by hours worked will show the classic bell curve.

Danc has a very interesting slide show on this topic on his blog where he depicts the following chart. He claims that the productivity loss during the recovery period, even for a relatively short time spent in crunch mode, is usually greater than the productivity gain during crunch mode. He also states that the teams who were working longer hours *reported feeling more productive* than those working a normal schedule, although when measured objectively they were less productive.



[8] Robert LaJeunesse, "Toward an Efficiency Week—Correlation between Workweek and Higher Productivity," *The CBS Interactive Business Network,* (Jan-Feb, 1999), http://findarticles.com/p/articles/mi_m1093/is_1_42/ai_53697784/pg_2/.
[9] Danc, "Rules of Productivity Presentation," *LostGarden*, September 28, 2008, http://www.lostgarden.com/2008/09/rules-of-productivity-presentation.html.

Copies may not be made or distributed for commercial use

In addition to defeating the purpose of working all those hours in the first place, there are other problems your team is likely to face if you don't keep working hours reasonable. These include high turnover, burn-out, demoralization, and lack of trust in management.

Most of you are probably familiar with the classic project resource triangle: time, money and quality. If you want to decrease the time to develop your software, you will need to increase the money (resources) and/or decrease the quality (fewer features, less testing). The part of this equation that is not always mentioned is that, at the end of the project, resources should not be depleted. You should be able to go right into the next project without recovery time. So, if you're in manufacturing, you should not be completely out of parts. In software development, your people resources should not all need to take a vacation.

I was on a project where, after literally working 60-80 hour weeks for over a year, five out of seven testers quit the team as soon as we shipped. (This became my first management job—I was the only one left except for the new guy.)

3. How to Get People to Follow Best Practices

I've gone on and on about best practices, and now you're thinking. "So, yeah, we know they work. That's why they're called 'best practices'. We're familiar with those best practices, on a conceptual level at least, because we're professionals and we've taken the time to understand what we should be doing to get it right. What we want to know is *how* to actually get to the point where we're doing those best practices instead of just talking about them. Something is always in the way."

What is usually in the way is people: people who don't know better; people who have competing priorities; people who have more power and influence than you. How do you get around that?

You need to understand the motivations people have for doing what they do and wanting what they want. Only then will you be able to effectively persuade them to change their minds.

3.1 Individual Goals may not Sync with Team Goals

Every single person on your team has their own personal goals, either stated or unstated. Sometimes these goals are in line with team and corporate goals, sometimes they are not. Sometimes hidden agendas of team members may wreak real havoc on your project.

A case in point is the project that I've used as an example throughout this paper (I knew going through that hell would give me some mileage somehow!). In this case, it took me a long time to figure out what was really going on.

Once I understood the magnitude of the problems facing this team and we started making progress toward fixing them, I found myself undermined at every turn—sometimes very subtly, sometimes more bluntly.

It turns out that the release manager thrived on crisis management. It was his moment to shine. He loved being pulled out of bed on Christmas morning because the live site was melting down. He had gotten kudos his entire career because he was always there when he was needed, willing to work tirelessly sixty-plus hours a week, sacrificing precious time with family and friends to put the corporate needs first.

But was he really putting corporate needs first? When we started to see change and, after the first few releases, the live site stabilized, he panicked. He was a mediocre manager in the best of times. He needed the adrenalin. He needed to be desperately needed. He needed a crisis.

Copies may not be made or distributed for commercial use

And he was entrenched. He had been on the team a long time. He had lots of friends and the support of his manager. In spite of the fact that the service was in horrible shape, he had always been told he was doing a good job.

Your first defense in a situation like this is to be aware of it. Any time you start making changes to the status quo, you are going to make some people unhappy for whatever personal reasons they may have. They will fight you.

Tread carefully, but figure out who you need to convince to make sure needed changes happen. Don't neglect to consider that even positive changes may be seen in a negative light by some team members.

3.2 Reward Teamwork

Unless the project is tiny, software development is very much a team effort. The best way, ultimately, to consistently deliver high quality software on time, is to have a reward structure in place that rewards teams for behavior that leads to successful releases.

This is common sense, however, I've seen very few companies who reward team effort rather than individual effort. Perhaps the fear is that if you reward the team, the individuals on the team won't work as hard because their personal effort is not being recognized. On most of the highly functioning teams on which I've worked, this would not have been the case. People will work even harder because of the team than they will on their own. The team is motivating; everyone works hard so that they won't let their teammates down.

If you can't get official corporate policy to support rewarding team effort, try to get your management to at least support team effort with recognition at meetings, extra days off, a team party or special function or something else the team desires.

Luckily, working on a successful team is a rewarding experience in itself. My best work experiences are those in which I am an integral part of a tight, high-performance team.

3.3 Tactics

There is no one way to accomplish your goals that will work for every situation, but I've found specific tactics very helpful:

1. Know best practices

   If you don't already know what the best practices are in software development, take some time to learn about them.[10] Best practices work.

2. Be polite but persistent

   Gently encourage team members toward best practices. You'll find if you're too heavy-handed you'll usually get beaten down. Take every opportunity to educate the team and management on best practices and why they should be followed. If you're persistent, eventually people will come around.

---

[10] Joel Spolsky, "The Joel Test: 12 Steps to Better Code," *Joel on Software,* August 9, 2000, http://www.joelonsoftware.com/articles/fog0000000043.html.

Copies may not be made or distributed for commercial use

3. Find your allies

   There are usually others who feel the same way you do. Enlist their help in fixing the problems on your team. They will often be delighted to find a like-minded person to help them with the frustrating problems with which they've been dealing, and if they've been on the team for a while, they may have some excellent insight into the reasons things have not yet been fixed.

4. Understand the real problem

   It's fairly easy to look at a team and say, "You're not doing this and that. Start doing this and that and your problems will be solved." There is a usually a reason they are not doing this and that already. You need to discover that reason, otherwise you will not be able to get around it.

5. Measure yourself

   I'm not just talking about bug metrics here. Measure your team on how well it does in meeting best practices. Take inventory. What are your current practices? Compare them to best practices. How far away are you and what steps do you need to take to get there?

6. Set clear goals

   This should be a natural result of measuring yourself. Figure out what is causing the most pain and fix that first. If the first solution you try doesn't work, try something else.

7. Celebrate successes

   Once you start seeing success, reward it. Brag about it at company meetings. Have a party. Give away special prizes. Tell the team how proud you are of their accomplishments. Usually success gets people motivated to go on to even more success.

Most importantly, don't give up!

4. Conclusion

Many things can sabotage quality. The most common is sacrificing long-term goals for short-term goals. Many best-practice activities require long-term investment. If your team is struggling with quality, you need to evaluate your situation and decide which best practices to implement first.

To convince the team to make the needed investments, you'll need to understand their motivations. Only if you know the real reasons behind resistance to change, will you be able to effectively persuade your coworkers, banish the saboteurs, and adopt more effective practices that will lead to higher quality, on-time releases.

Happy development!

Copies may not be made or distributed for commercial use

References

Bach, James. *Satisfice, Inc.*, last accessed August 14, 2011. http://www.satisfice.com.

Bird, Jim. "Zero Bug Tolerance." *DevOps Zone,* July 11, 2011. http://agile.dzone.com/news/zero-bug-tolerance-intolerance?mz=38541-devops.

Cohen, Bram. "How to Write Maintainable Code." *Advogato*, March 15, 2001. http://www.advogato.org/article/258.html.

Danc. "Rules of Productivity Presentation." *LostGarden*, September 28, 2008. http://www.lostgarden.com/2008/09/rules-of-productivity-presentation.html.

Fowler, Martin. "Continuous Integration." *MartinFowler.com,* May 1, 2005. http://martinfowler.com/articles/continuousIntegration.html.

Fowler, Martin. "Mocks aren't Stubs." *MartinFowler.com,* January 2, 2007. http://martinfowler.com/articles/mocksArentStubs.html.

Freeman, Steve, and Nat Pryce. *Growing Object-Oriented Software Guided by Tests*. Boston: Pearson Education, 2010. http://www.mockobjects.com/.

Hoffman, Douglas. "Cost Benefit Analysis of Test Automation." Paper presented at STAR99, 1999. http://www.softwarequalitymethods.com/html/papers.html#CostBenefit.

Kaner, Cem, James Bach, and Bret Pettichord, *Lessons Learned in Software Testing*. New York: Wiley, 2002.

King, J. Timothy. "Twelve Benefits of Writing Unit Tests First." *J. Timothy King's Blog,* July 11, 2006. http://blog.jtimothyking.com/2006/07/11/twelve-benefits-of-writing-unit-tests-first.

LaJeunesse, Robert. "Toward an Efficiency Week—Correlation between Workweek and Higher Productivity." *The CBS Interactive Business Network,* Jan-Feb, 1999. http://findarticles.com/p/articles/mi_m1093/is_1_42/ai_53697784/pg_2/.

McConnell, Steve. "Gauging Software Readiness with Defect Tracking." *Best Practices IEEE Software* 14, no. 3, (May/June 1997). http://www.stevemcconnell.com/ieeesoftware/bp09.htm.

Prasadrao, Saikalyan. "Cycolmatic Code Complexity Analysis for Microsoft .NET Applications." *The Code Project*, September 21, 2005. http://www.codeproject.com/KB/architecture/Cyclomatic_Complexity.aspx.

Rob. "How Can a Shorter Working Week Increase Productivity?" *Beyond Norms,* January 2, 2011. http://www.beyondnorms.com/index.php/2011/01/how-can-a-shorter-work-day-increase-productivity/.

Saktheee2008, "The Law of Diminishing Marginal Productivity of Labour, *HubPages*, accessed August 14, 2011, http://hubpages.com/hub/THE-LAW-OF-DIMINISHING-MARGINAL-PRODUCTIVITY-OF-LABOUR.

Seshadri, Shyam. "The Advantages of Unit Testing Early." *Google Testing Blog*, July 15, 2009. http://googletesting.blogspot.com/2009/07/by-shyam-seshadri-nowadays-when-i-talk.html.

Shaw, Julias. "Cyclomatic Complexity—What Is It and Why Should You Care?" *Java Metrics,* April 15, 2010. http://java-metrics.com/cyclomatic-complexity/cyclomatic-complexity-what-is-it-and-why-should-you-care.

Shore, James. *The Art of Agile Development*. Sebastopol: O'Reilly Media, 2007. http://jamesshore.com/Agile-Book/no_bugs.html.

Spolsky, Joel. "The Joel Test: 12 Steps to Better Code." *Joel on Software,* August 9, 2000. http://www.joelonsoftware.com/articles/fog0000000043.html.

Vaaraniemi, Sam. "The Benefits of Automated Unit Testing." *The Code Project,* Nov 8 2003. http://www.codeproject.com/KB/architecture/onunittesting.aspx.

Vandegriend, Basil. "The Importance of Maintainable Software." *Basil Vandegriend: Professional Software Development*, February 1, 2006. http://www.basilv.com/psd/blog/2006/the-importance-of-maintainable-software.

Wikipedia. s.v. "Zarro boogs." last modified August 6, 2011. http://en.wikipedia.org/wiki/Zarro_boogs.

Copies may not be made or distributed for commercial use