2009

PACIFIC NW

SOFTWARE QUALITY CONFERENCE

MOVING QUALITY FORWARD

OCTOBER 27-28, 2009

Conference Paper Excerpt

From the

CONFERENCE PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commecial use.

Test Faster

John Ruberto
Software Quality Leader, QuickBooks Online
Intuit, Inc.

John_Ruberto@Intuit.com

Abstract

Moving Quality Forward can be a daunting task. Making changes to the status quo is already difficult, and that is before you consider the complexity and interconnectedness of your existing practices. When pressed to make changes, it can seem like an overwhelming proposition.

This paper will show how our team responded to such a challenge: to cut the system test duration from 12 weeks down to 4 weeks. When initially presented with this challenge, we resisted. Our brainstorming sessions turned into justifications of the status quo and explanations of why it's an impossible task. We've always needed every minute of those 12 weeks, so of course it was an impossible task to remove 67% of the time.

The break through came when we built a model of the time we spent in the system test phase. That model allowed us to break the large, complex problem into a series of smaller, easier to implement solutions. The model we developed expressed the testing duration as a function of:

- Number of test cycles
- Number of test cases
- Rate which we could execute manual tests
- Number of defects we will find and handle
- Rate which we handle the defects
- Number of people on our test team

Now, instead of racking our brains on how to reduce the duration from 12 weeks, we could focus on a single variable at a time. For example, "how can we reduce the number of test cases executed, while maintaining the same quality levels?" Asking these questions were more productive, and lead to a number of changes that we implemented.

In the end, we succeeded in achieving a 4 week system test cycle, which enabled our company to release 5 versions of our product each year, instead of 1 release every 9 months. This improvement resulted in delivering more value to our customers, faster.

This paper will first provide some context about the industry, process, and business situation. Then, will show how we created the model based on our processes and practices, and how we used the model to create a series of smaller & easier problems to solve. The paper will then briefly describe the techniques utilized.

Biography

John has been developing software in a variety of roles for 23 years. He has held positions ranging from development and test engineer to project, development, and quality manager. He has experience in Aerospace, Telecommunications, and Consumer software industries. Currently, he is the Software Quality Leader for QuickBooks Online for Intuit, Inc. He received a B.S. in Computer and Electrical Engineering from Purdue University, an M.S. in Computer Science from Washington University, and an MBA from San Jose State University.

Introduction

This paper tells the story of a test team which was challenged to change due to changing business conditions. The challenge was to reduce the system test duration from 12 weeks down to 4 weeks. The complexity and difficulty of this change became apparent each time the question came up. Responses were overwhelmingly weighted towards explanations on why it was impossible, and defenses of the status quo.

The team did succeed in reducing the system test duration. The break though started happening when we started breaking down the difficult & complex (i.e. "impossible") problem into simpler problems. There was still a lot of hard work ahead, but the team was able to move forward to implement specific changes that allowed them to meet the goal.

This paper will first provide some context about the industry, process, and business situation. Then, will show how we created the model of test duration based on our processes, and how we used the model to create a series of smaller & easier problems to solve. Finally, the paper will briefly describe each of the techniques utilized.

Background

The company is a major supplier of telecommunications equipment in the United States, supplying to all of the major telecommunications service providers. The telecommunications equipment provides the "last mile" of connection, from the Telco central office to the home or office. The equipment provides both Plain Old Telephone Service (POTS) and high speed Internet through Asymmetric Digital Subscriber Line (ADSL).

Being a critical link in the telephone service, which provides dial tone and 911 emergency access, the industry practices are relatively strict for quality and validation of the quality. The processes used to build and verify the software had been reviewed and approved by standards setting bodies, and compliance is regularly audited.

The software for this project was a management system, providing features like service provisioning, maintenance, and system alerts. It's has a client/server architecture, built on a Unix platform, and written in C++. The typical project length was 9 months, roughly 6 months for development, and 3 months for System Verification Test.

The process used for verification was built upon 3 verification cycles after features were completed. The goal for the first cycle is to execute 100% of the available tests, in theory exposing all defects. Bug fixing is allowed during the first cycle. The second cycle is reduced in size, the goal to execute approximately 50% of the tests, with the main goal to verify the bug fixes and ensure no regressions occurred. The third cycle is a sanity test on the release build.

The product existed for several years with roughly the same 9 month delivery life cycle when a business change caused an acceleration of the development. Customers want to manage more equipment types with the same software, so this extra content was built in. Since this was a management platform, release was necessarily synchronized with the hardware delivery. Each equipment type had its own delivery schedule, resulting in many more releases for the management system. Over an 18-month period, the required life cycle went from 9 months, to 5, down to 3. Leaving 4 weeks for the verification test phase.

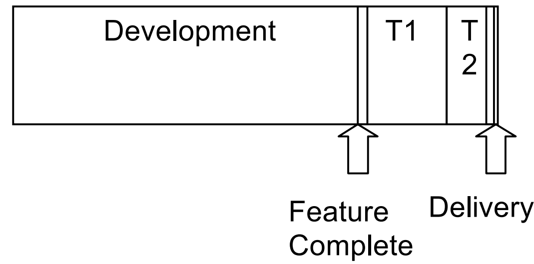

Figure 1. Development and Test Lifecycle Shows Three Test Cycles Following Feature Complete

Prior to the business change, the test duration for most projects was 12 weeks, and for all of the projects the team needed every minute of that 12 weeks, testing right up to the last minute. But, this was not a pure testing activity. Our customers were providing requirements during this time, which turned into Change Requests. Another development activity that occurred was the implementation of the database upgrade scripts and procedures, to optimize the schema migration, it was developed only after Code Complete and delivered to test 3-4 weeks afterwards.

The test team involved in this project was comprised of approximately 40 test engineers, with 10 engineers at the main development site and the balance located at an off-shore outsourcing partner. Most of these changes were driven by the managers & leaders of the test team, about 6 people in all. The skill sets and experience levels varied from Software Engineers working in quality, Quality Analysts with deep domain knowledge, and Quality Analysts with varied testing experiences. The entire journey was not an overnight success, it took approximately 2 years to reduce the test cycle duration from 12 weeks down to 4.

Model the Test Duration

Change Management Process

Initially, the desire to reduce the testing duration was presented as an aspirational goal: “Wouldn’t it be great to test in 10 weeks instead of 12?” To see if meeting the goal was feasible, we held several brainstorming sessions. These largely became gripe sessions over the impossibility of the task, it being someone else’s fault to fix, etc.

When the question was posed, “name 1 thing we spend our time on during the testing phase”. The answer was dealing with bugs. Asking, then, how can we reduce the number of bugs that we fine, then some specific ideas came forth.

Model the Time and Effort

We were more productive when trying to solve a specific problem (bug reduction), rather than the complex (pull 8 weeks out of a 12 week project). We broke the whole process down to a model, with several variables. To get our minds around this, we wrote it as a mathematical equation, though it was not that precise. Here is the resulting “equation”:

$$TestDuration \approx NumberofCycles \times \left[\frac{(TestCases \times TestExecutionRate + Defects \times DefectHandlingRate)}{NumberofTesters} \right]$$

The total test duration is a function of the number of test cycles, the number of test cases we execute, how fast we can execute those tests, the number of bugs we find along the way, how fast we can report those bugs and verify the fixes, and finally, greatly influenced by the number of testers we use to do all of this work.

Now, with this model, we can start asking more productive questions. Namely, to minimize the test duration, the following questions help guide more effective ideas:

- Do we really need to run 3 cycles?
- Can we start the first cycle early?
- How can we execute fewer test cases, while maintaining the same coverage?
- What can we do to increase the average test execution rate?
- We spend a lot of time dealing with defects? How can we reduce the defects that have to be handled?
- How can we handle each bug quicker?
- Where can we find more people to help? What is the best way to deploy new help?

These questions helped focus our thoughts and energies into productive changes.

Test Faster Tactics

This section describes many of the tactics that we ended up using to save time during the system testing phase. There wasn't a single "silver bullet", but taken together each of these tactics shaved some time off the duration.

Many of these tactics are described in greater detail in other papers and sources. This section of the paper will give a brief description of the tactic and how effective use of the tactic reduced our verification time. The references section of the paper will provide pointers to these other sources of information.

Tactic: Reduced Test Cycles

Start Regression Test Early

During one release, we ran an experiment by starting the regression test cycle early, before the Feature Complete milestone. The areas tested were focused on those that were largely complete already. We found a lot of bugs, but about the same as we would have if we waited until all the features were complete. The developers had completed approximately half of the features, and we planned the regressions around those areas.

The early start on regression testing allowed some of the developers to start fixing bugs earlier, and allowed us to deploy testers to other areas. There were a few cases where we had to completely retest areas, because they weren't fully baked when the tests were executed. But in our post-mortem, all agreed the benefit of earlier bug detection was worth the inefficiencies introduced.

Another issue to manage around this was the testers usually used this time, the few weeks before Feature Complete, to write the test plans for new features. We managed around this by having an outsourced test service provider execute these tests, while the new feature test plans were completed.

Tactic: Reduced Test Cases to Execute

Historical Analysis & Test Case Pruning

We had been building this project for a few years when presented with this challenge. While thinking about reducing the number of test cases to execute, we did a historical analysis of the test results over the various cycles. One interesting pattern, about 15% of the test cases never failed. In the history of the product, certain test cases never failed during system test. This lead to the question, why execute them again?

We used this information to de-prioritize these tests, using 2 tactics. We would make sure to execute the "golden test cases" only once for the entire test duration, making sure that we didn't use time executing them in cycles 2 and 3. The other tactic was to apply sampling for the tests in the same area. Instead of executing 150 test cases, we chose 20 at random and executed those. If all passed, then we didn't test the rest. If any failed, we planned to execute every "golden test", but never had to take that path.

Risk Based Testing

Related to using historical analysis, we prioritized all of the test areas based on risk, calling this Risk Based Testing. For each test area, we rated the tests in two criteria, impact to our customer if broken and likelihood of the area having problems. The ratings were high, medium, and low. For test areas that were rated with high customer impact and high likelihood of problem, we ran these first, with our best testers, and spend the most time. For the low/low tests, we followed similar procedures for the golden tests described above.

The customer impact was determined using several sources of information:

- We asked the customers what was most important to them
- The marketing & product management teams provided input
- The support team provided input
- We examined the user activity logs in the live servers.

Next, we rated the test areas on the likelihood of errors. Several of the techniques used were:

- Polling the test team showed several areas that always fail, and some that never fail.
- We mined the bug tracking system to confirm the opinions
- The code review database showed a high correlation between effective reviews and lower risk of defects
- We asked the developers, what areas were they nervous about.

Looking at this as a matrix, we would be sure to test thoroughly those areas that had the highest risk of impacting our customers. Those areas with low risk received less attention from the test team.

Customer Impact	Low	Medium	High
High	**Test Thoroughly**	**Test Thoroughly**	**Test Thoroughly**
Medium	**Test Moderately**	**Test Thoroughly**	**Test Thoroughly**
Low	**Test Lightly**	**Test Moderately**	**Test Moderately**

Likelihood of Serious Issues

The user activity log exercise, where we examined usage in live servers, was particularly illuminating. We found that 90% of the product usage was in 3 commands: Provisioning service, logging in, and logging out. This information guided our automation efforts.

Pairwise Analysis

One driving force behind the total number of test that had to be executed was driven by a variety of possible configurations that we had to validate. We calculated that one suite, which had to vary by platform (CPU speed), memory, and a few other variables would take 4.6 billion tests to exhaustively test the complete matrix.

Another test had required 72 different permutations, between 3 OS versions, several binary state variables, and 3 different product versions. We had to reduce the set to a more manageable number of test cases.

Pairwise analysis is a method for reducing a large, exhaustive, matrix down to a much smaller set of relevant tests. [1] The theory behind this method is that most issues arise from interactions between 2 variables, not more complex combinations.

The following table shows a simple example. Suppose you had 3 binary variables, this will lead to 8 permutations for an exhaustive test. If, instead, you only executed the 4 tests selected in the table, every permutation of the 3 pairs (AB, BC, and AC) will have been tested (00, 01, 10, and 11). This cuts your test matrix in half.

Variables and Possible States			Pairs and Selected Values		
A	B	C	AB	BC	AC
0	0	0	00	00	00
0	0	1			
0	1	0			
0	1	1	01	11	01
1	0	0			
1	0	1	10	01	11
1	1	0	11	10	10
1	1	1			

In the case encountered by our team, the full matrix would have taken 72 tests to fully cover. Each test took about 4 hours to execute. Instead, by using the pairwise analysis, we were able to pare the testing down to 12 test configurations, theoretically saving 6 staff weeks of effort.

Exploratory Testing

Our team was lucky; we had a couple of senior testers that had been around the industry and this product for years. They knew the product inside and out, and had deep customer and domain knowledge. These people are priceless. They have an ability to quickly assess a feature or area, without having to be guided by written test cases. They follow their nose, using exploratory testing.

Exploratory testing is the practice of using the product with an open mind, and looking for goodness or badness. It is testing guided by the experience and intuition of testers, with exact actions determined on the fly (i.e. not following a script). The exploratory tester will be guided by knowledge of what’s important for

the product, their experience, and by the observations they make while running the tests. They are exploring the product, looking for bugs.

For instance, we may ask a tester to check out a new interface for controlling the provisioned data rates for DSL service. Instead of consulting the specification or written test scripts, the tester will start with hands on activities. They may open the window, check out all the buttons and controls, visit each menu command, and start filling in forms. If the system behaves as expected, they will start looking elsewhere, or deeper into the interface. The mindset is that there are bugs in there, waiting to be discovered.

If, instead, they started by reading the specification or test plan, this may have biased their mental model on how the system should work – and they may have inadvertently skipped some promising areas or operations. Following a script constrains thinking.

Normally, these are the people we put on the difficult areas, and those rated the highest in risk. They can also reduce the overall test duration by giving an area the quick once over, instead of following a script. Frequently, 1 hour of an expert gives more valuable information than 2 days of a beginner following a script.

Another use for Exploratory Testing was when junior testers reported results that were too good to be true. A whole test suite ran perfectly, with zero failures. This sounded great, but just in case we had a senior tester give those features a quick once over. Turned out the quality was that good, and the second look increase our confidence.

The value of including exploratory testing was illustrated to me when presenting our test progress metrics to our senior leadership team. I had presented the test status, number of tests executed, number pass, and number failed. That week, we executed several hundred test cases, and had less than 5 test cases fail. Next, I presented the bug metrics, including a chart that shows the number of bugs found and number fixed that week. This chart showed 30 bugs were found.

One of the directors asked how we could have failed only 5 tests, but entered 30 bugs. I explained that the bugs were found in activities other than executing test cases. This answer solicited a follow up question about the quality of our test plans, if so many bugs “escaped” our test plans.

Thinking about this later, I came to the realization that we wrote our test cases based on the material in the specifications. The developers created their designs and code from the same set of specifications. The prepared test scripts were less “productive” in finding defects because the same thinking went into code generation as test generation. The exploratory tests found a lot of new issues because different thinking created those tests.

Tactic: Increasing the Test Execution Rate

Trimming and tuning the number of test cases had some benefit, but not nearly enough to dramatically reduce the overall duration of the test cycles. For the remaining tests, we want to increase the execution rate, which means test automation.

Test Automation

Test automation has several advantages for increasing the test execution rate.

- The obvious, computers move faster than humans, so the test steps are generally executed faster than a person can work a keyboard & verify information on a screen.

- Automated tests can also be run in parallel, running several test suites at once on different targets or clients.

- Automated tests can be triggered to run at night or on weekends. This is especially useful for build verification tests, either in a nightly build environment or for continuous integration systems.

- Automation can also be used for tasks other than actually running a test, to help the testers be more efficient. Test setup is a task that can often be very time consuming. Also, assisting testers in results analysis can save time & reduce errors.

- Automated tests can be executed more frequently than manual tests; the machines don't get bored/tired/burned out. This helps to find problems sooner, closer to the time when the problem was injected. Finding problems soon after they are introduced almost always allows the developers to fix them quicker.

A full discussion of test automation is beyond the scope of this paper. The references section provides some good starting points. Here are a few lessons we learned while instituting our test automation program:

- Start with a dedicated automation team. Balancing manual testing with starting an automation program is more prone to failure than starting with a dedicated team.

- Identify all the exploitable interfaces in the system first, before starting to automate tests.

- Exploit the machine to machine interfaces before starting UI automation. Interfaces like APIs, database access, etc. It usually easier to create tests for these interfaces, and the interfaces are less prone to changes than the UI. The resulting tests are likely to be more durable and run faster.

- Enlist help from the development team to build in testability. The development team should also be building unit and white-box tests.

- Develop automated tests using sound software development practices (design, frameworks, code reviews, test the tests, source code management, etc.). The automated tests are software too!

Test automation often suffers from the silver bullet syndrome, where its presented as a cure for all that ills software testing. Use of test automation has been a net positive in this project, but it does have its limitations. Some of the issues encountered were:

- Automated tests run very predictably, which fails to expose issues that may exist that are caused by interactions between test suites or variations of user actions.

- Inaccuracy of test results can cause your tests to be worse than worthless. False failures and missed defects will erode the confidence in the automated tests. We frequently ran the automated tests, then had to rerun them manually to confirm the results. A concerted effort to refactor the tests to eliminate these false failures fixed the problem, but the credibility of tests suffered for a while after the problems were fixed.

- Don't underestimate the effort required to maintain the tests in response to product changes.

Tactic: Fewer Defects to Manage

Out of all the topics covered in this paper, and techniques applied to reduce testing duration, simply having a better product with fewer bugs in itself is the most powerful. Doing an evaluation of all the tests that we executed, we could have executed the entire test program in 2 weeks or less, if it wasn't for the bugs. And that is without any test automation.

Because bugs exist, we have to run multiple cycles of test to retest bug fixes and verify that bug fixes did not introduce new bugs into the system. Also, multiple builds that are introduced during the test cycles, meaning downtime from testing while tests verified the build was OK, and installed on all the test servers.

Reducing the overall number of bugs is a huge reason why we were running tests anyway. Here are some of the practices that have proved effective in reducing the bugs, and increasing quality.

Effective Code Reviews

Code reviews are one of the best investments in quality. We measured the return on investment for reviews, by measuring the total effort it takes to conduct the review and comparing to the effort it would take to find the bugs by testing & fixing later in the test cycle. This return on investment ranged averaged 5:1, meaning 1 hour of effort reviewing the code saved 5 hours of finding and fixing the bugs during system test. This measurement really helped sell the idea of code reviews in the development team. We went from "there is no time to do code reviews" to "an hour of review now will save me half a days work next month".

Here are a few learning's for implementing our code review process:

- Start small. We initially tried IBM's Fagan Inspections, but found that the team became allergic to the formality and data recording. It was difficult to go from 0 to IBM in one month.

- We built a light-weight review tracking system. It was web based and easy to record the reviews. The system assigned a number to each review, so we could verify that each feature had the appropriate reviews. The tracking system had a record of the reviewers, how much time each person prepared for the review, and a list of defects found at the review. Each defect had an open/closed field to track closure.

- We created a report called the "ducks in a row" report, that listed each feature, and the tracking numbers for the design review & code review. This report was reviewed at the project reviews. The adage, "you get the behavior that you measure" helped here. An example report is shown here:

Feature	Feature #	Rqmt Review	Design Review	Code Review	Unit Test	Functional Test	System Test
8-way DSL Support	13482	#1234 3 pages, 2 defects	#1348 8 pages, 5 defects	#1532 LOC: 1354 4 defects	15 exec	12 exec 11 pass	9 exec 9 pass

- Start measuring simply: did the review happen or not? Then, as the team became used to the process, we started looking at the reviews for effectiveness. A couple of key measures for effectiveness was the review rate (LOC/hour), and the ROI (effort saved/effort expended).

Build Quality In

Many of the problems that we had to work with during the test phase were caused by mis-understandings during the definition & development phase. Sometimes the developers mis-understood the requirements and would build the feature incorrectly, or the tester mis-understood and failed to test properly.

Including the senior test engineers from the beginning of development helped ease this process. They would participate in the requirements, design, and sometimes the code reviews. Each time they participated up front, we noticed that they were able to prevent serious bugs before the bug was actually introduced.

Use Automation to Test Constantly

Automated tests can be used all the time, not just during the system verification phase. The tests can be executed with each nightly build during the development phase. This process has the benefit of finding the bugs within 24 hours of the offending check-in, making it much easier to isolate the cause and get a fix in place. Without this automation, many of these errors would be caught during the regression test, and slowing testing down.

Extending this concept even further is the idea of Continuous Integration (CI), where the automated tests are executed with each check-in. In CI systems, the code check-in typically triggers a build and executing a set of tests. Typically, these tests are the unit tests, written by the developers. Unit tests are ideal for CI, as they are usually very fast to execute, and don't require setup if the environment is mocked.

Using automated tests in this fashion requires the tests to be very durable. If the tests have a high false alarm rate, the effort to triage the results will soon overwhelm any benefit from executing the tests this often. Care needs to be taken to eliminate false alarms and contain the test suite to those that run reliably and fast enough.

It's also valuable to extend the build verification process, to before the check-in. At Intuit, we developed a capability for developers to execute the build verification tests prior to check-in. The developer can upload his/her private build to the test automation system, and execute the build verification, or any tests, prior to checking the code into source control. This gives the developer confidence that the change will not break the build, and not impact others.

Tactic: Handle Defects Faster

Complexity of these systems and the fact that humans are involved virtually guarantees that there will be bugs to find and fix during the test cycles. Here are a couple of practices that we developed to minimize the impact of these bugs.

Eliminate false alarms

One productive practice was holding "bug huddles". Instead of first writing a bug in the tracking system, the testers would have a short huddle with the team lead and developer, demonstrating the bug and having an actual talk. This eliminated many false alarms, improved the fix velocity, and the fix accuracy.

One metric that we tracked was the number of "false alarms", or reported bugs that turned out to be not bugs. These were filed as "works as designed", "duplicate", or "cannot duplicate". For the first few years, when we had the leisurely 12 weeks to verify the software, our false alarm rate reached up to 50% for some releases. Some focus on our bug handling process brought the false alarm rate down to 20%.

The first project where we started using the "bug huddles", we had 27 reported bugs, from which 26 were true bugs, and 1 false alarm. The bugs were fixed faster, as the developer and tester had a much richer communication than the bug description field in the tracking system.

Meet daily to expose blockers

In the old time, we had weekly project reviews where a lot of the issues that blocked test would be hashed out. Waiting up to a week to kick-start blocking bugs was too long. In the test faster world, we had to move to a daily focus. We held daily meetings where blocking issues were raised by the team and someone assigned to solve the blockers.

To make the daily process work, we had to have real-time access to data. We had to invest in a test management system, instead of relying on spreadsheets to track tests. The test management system gave everyone a real-time view of the blocked test cases, with minimal effort to collate the results. When using spreadsheets, it took 2-3 hours each week to compile the test reports.

For this project, we ended up creating our own test management system, called creatively the Test Case Management System (TCMS). The aspects of this tool which helped our productivity was easy importing and exporting of test cases and results to Excel (via a comma delimited file) and web access so that anyone in the world-wide testing team could enter results directly, and a single database to allow real-time access to results without a data-synchronization step.

We had attempted to run this overall project using the Scrum methodology. For various reasons, Scrum did not work for this organization, but the daily meeting was valuable.

Tactic: More People

We were using an outsourced testing partner for this project, to help supplement our staff, and reduce average costs. Being the manager of the test effort, and when pressed to reduce our duration, a question the senior leaders would ask "how many more contractors do you need to pull this in?" They felt like they were helping, but adding people to a project in the short term can be more of a hindrance than a help. Adding people to projects has benefits and drawbacks in both the short term and longer term.

The model described above has a very simplistic component when it comes to labor. It just assumes that adding any person increases the denominator, thus reduces the duration of the testing effort proportionally. That is obviously simplistic.

In the short term, adding people to a project can make an already late project even later. The new people will need to know where they can help, which will add to the test manager's workload coordinating the new tasks, and adjusting the priorities of the existing team, who are already pretty busy. The new people will likely need to be trained on how to access the test environments, where to find test cases, how to report defects, how to obtain licenses for test tools, and how to report progress. They will consume time and resources that were not in your original plan. The larger a team becomes, the communication paths become geometrically larger, which increases the management challenge.

That said, frequently an infusion of people in the short term can help the team in several ways. New people don't necessarily have to be "drop in replacements" for the existing test team. They can relieve the existing team of some tasks, helping the existing team focus on testing. The next section describes some of our experiences with including new people and some of the tasks they can undertake.

In the longer term, adding people can help to a certain degree. Once new people are trained and have some experience, they can constructively add value to the project. Implementing some of the tactics described above is actually optimal with new people. For example, executing the regression tests earlier is better accomplished with an addition of new people on the team, since new feature test generation and old feature test execution are done in parallel, instead of serial.

Another area to add people for the longer term is in automation. The existing test team may not have the skill set to be completely effective in test automation. People with a software engineering skill set should be added. Even if the existing team has the skills, adding people to take over the manual testing is very useful.

Keeping a larger team for the long term initially sounds promising, but remember that small teams are more effective than large teams. Also, a larger team increases the cost structure of the quality team.

What Can More People Do for You?

Sometimes, the gift of new people for a testing project comes in the short term. For instance, in trying to preserve the original release date for a project that is late. In this project, we frequently received the gift of people unskilled in the art of testing. This is when the team usually pushes back with statements like “we can’t bring the new people up to speed fast enough”, “training the new people will take us away from doing our jobs”, and my favorite, “have you read the Mythical Man Month?”

However, there are sources of additional people that can be put to good use in helping pull testing in. Many of these people already have product knowledge and can add value right away, without disrupting the existing team. Likely, this will end up being a long term positive, relationships are strengthened and knowledge shared. Here are a few examples:

- **Technical writers**. These folks operate the software just as much as testers and need to understand it well enough to explain how it works. Pulling the writers into the testing may impact the development of help content & product documentation, but when push comes to shove, often the documentation can be updated after the software has stabilized and been delivered.

- **Technical Support Engineers**. Enlisting the support team is a win-win. The support team knows the product, and can think like a customer. They benefit by getting hands on training for the product they will be supporting soon.

- **Product Managers**. They specified the requirements, they know how the system is supposed to work.

- **Developers**. Its good to walk a mile in other people’s shoes. You may get some pushback from using developers to run tests because of “independence”. If so, you can mix up the testing responsibilities, a developer can test features created by other developers, or execute regression tests. I don’t buy the arguments about developers needing to be independent because they don’t want to look bad, but a very real issue is “author blindness” where the creator fails to see flaws in his/her creation.

- **Customers**. The ultimate judge of correctness. Many customers are willing and happy to help, because in the end the software is built to benefit the customers. When inviting customers in to run tests, try to avoid it becoming a “dog and pony” show, run by marketing. The most valuable is gained in hands on interactions between customers and the product, and the interactions between customers and the test staff.

- **Others in the organization**. A few times we were offered time from engineers in other areas of the company, who didn’t have too much knowledge of our domain. Also, managers like to help and pitch in. These helpers can positively help in areas that should not be too much of a distraction for the existing test team. They can help verify bug fixes, where the bug description is pretty straight forward and the bug simple to verify. They can also help review and test the user documentation and training materials, which was created to help people not familiar with the product to learn.

Summary

This was an interesting challenge, to reduce the average test duration from 12 weeks down to 4 weeks, with a product that was growing in functionality and complexity with each release. As this paper shows, no single tactic “fixed” the problem, but instead the big problem was broken down into smaller challenges and each smaller challenge was chipped away with a larger variety of tactics. We succeeded in making this change, but it was not an overnight success. This change journey happened over a 2 year period.

References & Footnotes

Pairwise Testing

Czerwonka, Jacek, *Pairwise Testing in the Real World, Practical Extensions to Test Case Generators*. Proceedings of the Pacific Northwest Software Quality Conference, 2006, http://www.pairwise.org/docs/pnsqc2006/PNSQC%20140%20-%20Jacek%20Czerwonka%20-%20Pairwise%20Testing%20-%20BW.pdf

Test Automation

Fitch, Todd and Ruberto, John, *Building for a Better Automation Future: One Company's Journey*, Proceedings of the Pacific Northwest Software Quality Conference, 2007, http://www.pnsqc.org/proceedings/pnsqc2007.pdf

Dustin, Elfriede, Rashka, Jeff, and Paul, John, *Automated Software Testing – Introduction, Management, and Performance*, Addison-Wesley Professional, 1999

Kaner, Cem, Bach, James, and Pettichord, Bret, *Lessons Learned in Software Testing – A Context-Driven Approach*, Wiley, 2001

Fewster, Mark and Graham, Dorothy, *Software Test Automation – Effective use of Test Execution Tools*, Addison-Wesley Professional, 1999

Mosley, Daniel, and Posey, Bruce, *Just Enough Software Test Automation*, Prentice Hall PTR, 2002

Exploratory Testing

Bach, James, *Exploratory Testing Explained*, April 16, 2003, http://www.satisfice.com/articles/et-article.pdf

Risk Based Testing

Bach, James and Kaner, Cem, *Exploratory & Risk Based Testing*, 2004, http://www.testingeducation.org/a/nature.pdf

Besson, Stephane, *A Strategy for Risk-Based Testing*, http://www.stickyminds.com

Software Reviews

Fagan, Michael, *Design and Code Inspections to Reduce Errors in Program Development*, IBM Systems Journal, 1976