A Case Study:

14 Lessons in Building Quality Software

AT*SQA
ASSOCIATION FOR TESTING &
SOFTWARE QUALITY ASSURANCE
Global Certification Body

A Case Study:
14 Lessons in Building Quality Software

It's true, quality in software development projects doesn't just happen on its own. Quality usually doesn't happen when the project depends on a small group of heroes to ride in on their white horses and wave their shiny swords to vanquish the problems. Quality happens when careful planning is done, when the entire project team maintains a quality-conscious approach every step, when problems don't escape from the phase in which they were introduced. A quality product is a team effort. It's planned and predictable. It's without heroes, and it's faster and cheaper than a low quality effort. How can this be? Let's look at some sample projects. The first is a normal, low quality, late project. We'll call it project "Hurry Up" (HU for short).

"Hurry Up" got a bit of a late start due to the on-going maintenance issues of its predecessor project "Just Ship It" (JSI). JSI was handled by a Project Manager (PM) who felt it was more important to ship on time than to ship a quality project. So he did. This PM was rewarded for his ability to "pull it together", "get it out the door" and "meet that schedule." The JSI PM was given a bonus for meeting his schedule and is now vacationing in Tahiti while the team deals with the fallout of the numerous bugs and unhappy customers.

Lesson #1

Don't reward for shipping on schedule. Anyone can ship garbage. Base rewards on quality measures.

The JSI developers worked 80 hour weeks for the last month of the project. One heroic fellow was recognized for working 120 hours in one week, stopping only for brief rests. He heroically repaired multiple interfaces between applications. Those interfaces had not been properly specified (no design documents), no integration testing had been done (no time) and the QA team had fought with the issues throughout system test.

Lesson #2

Don't reward heroes for their Herculean effort late in the project to fix problems that could have and should have been fixed by the same people much earlier in the lifecycle.

The entire JSI team is down with the flu now due to lack of sleep.

Lesson #3

If you expect to work your people inordinate hours, you might want to consider corporate-sponsored flu shots!

HU was supposed to start three weeks ago, but the lingering effects of the flu, the nagging JSI maintenance problems and general team discord have slowed down the start. The analysts who are responsible for writing the requirements are in a rush. They got started late, the customer can't make up his mind and the PM is pressuring for completion. They write what they can in an MS Word document and ask for a review. The PM tells development to start coding and schedules a "quick" requirements review between the analysts and the developers.

Lesson #4

Always include QA and other project team members in all reviews to get the most well-rounded input possible.

The requirements specification is sent out via email and questions/responses are requested. Development has already started coding – they don't want any changes to be made. No one responds to the email, so the requirements are signed off as is.

Lesson #5

It's easy to ignore documents that are sent in email for approval. No response does not equal approval; no response equals "I didn't have time to read it."

Development is busy coding. They are hitting some problems because the interfaces between functions

aren't well-defined. This is necessitating recoding and substantially slowing down the schedule. When they ask the analyst for clarification, they're given a new user interface and two new items of functionality. They decide not to ask any more questions

Lesson #6

Don't start coding until the requirements are stable and understood or else budget time for rework.

As the development team nears the end of their scheduled time, it's apparent they won't make it. They begin to concentrate on the harder work, leaving the easier user interface and reporting tasks for last. While they had hoped to do unit testing, only a few developers are doing it and the effort is spotty at best.

Lesson #7

Code isn't "complete" until it works. Good unit testing is part of the development effort, not an optional item to be jettisoned when the schedule is tight.

The QA group is summoned by the PM. Having just completed yet another maintenance release for JSI, they are frazzled and grumpy. This is the first they've heard of HU and have little domain expertise. They are told that there is a requirements document but it may be somewhat out of date. No use cases were written. They'll have to create their own test data. They are now even grumpier!

Lesson #8

Is your test team always grumpy? Maybe they have good reasons!

They start writing test cases but are interrupted from that effort with the arrival of code to test. They hurry to create test data, guided by the developers and begin testing. It's soon obvious that they can't make much effective progress because they have only a partial UI and no reporting capabilities. All data verification will have to be done directly in the database.

Lesson #9

To maximize team efficiency, the project plan needs to consider testing efficiency as well. This may determine feature implementation order.

The software is buggy. The test team tests around the areas that aren't implemented or aren't working, but they are finding a number of blocking issues. Worse, when they get a bug fix from development, 30% of the time it doesn't fix the problem. In this state of code churning, the project hurtles past the deadline. The PM is pressured to ship (and he wants his trip to Tahiti too!). The developers and testers are told to increase their efforts, work together to achieve the goal, do whatever it takes….

Lesson #10

Buggy software takes longer to ship.

The product ships in an unknown state. Last minute functionality was added and received only cursory testing. A large number of identified bugs are still open, although all known critical problems have either been addressed or reclassified as "serious". The maintenance release is already being planned. The team is exhausted. They've worked heroic hours, again, and have produced a barely supportable product, again. The customer is unhappy, again. The product has features the customer doesn't want or understand and it's missing several major items they were expecting. Accolades come down from above for another "on-time" delivery.

What went wrong?

- Management doesn't recognize that "on time" doesn't equal "satisfied customers."
- The entire project team is driven by schedule. Every decision shows schedule, not quality, consciousness.
- The shortcuts taken to improve schedule time (unfinished requirements, insufficient system design, no unit test) actually made the project take longer.
- The maintenance release is in reality still the primary release, but now the unhappy customer is involved too.

What should have been done differently? What if they had applied the lessons?

No problem should have escaped from the phase in which it was introduced. Requirements problems went all the way through to the customer. Coding issues when through to system testing. People are exhausted, burned out and not utilized effectively. The rewards system is messed up! Six months after this project shipped (and eight maintenance releases later) an analysis was done to determine the origin of all the bugs. The analysis showed the following:

50% of the bugs were introduced in the requirements. These were due to unclear and vague requirements as well as functionality that was not defined and had to be introduced in a maintenance release. This also includes data issues and equipment issues where the test team didn't have the right data or equipment to reflect the customer's environment. Additionally all bugs associated with the unwanted features are counted here since those bugs wouldn't have occurred if the features hadn't been implemented.

15% of the bugs were due to design issues, particularly interfaces between code modules and the database.

25% of the problems were coding errors, both in new code and regressions introduced in the fixes.

10% of the problems were system integration issues that were only visible in the fully integrated environment.

A new PM was brought on board to lead project "Smarter Now" (SN). He listened carefully to the problems encountered by the development manager, QA manager and analysts and vowed that his project would not suffer the same consequences. To start with, he looked at the cost of quality numbers. Assigning costs for each bug, depending on the phase in which it was introduced versus the phase in which it was caught, he found the following:

50% of the total bugs were found by the test group in the system test phase.

The other 50% were found by the unhappy customers.

Employing widely used cost numbers, he assigned the following values:

- $1 for each bug found in the requirements review
- $5 for each bug found in the design review
- $10 for each bug found in unit test
- $100 for each bug found in system test
- $1000 for each bug found by the customer

Doing the math, he determined the 1000 bugs found in the product cost as follows:

- Found in requirements: 0
- Found in design: 0
- Found in unit test: 0
- Found in system test: 500 x $100 = $ 50,000
- Found by the customer: 500 x $1000 = $500,000
- Total cost of quality: $550,000

According to the bug distribution of the last project, if testing had been done throughout the lifecycle, the cost of quality should have been:

- Found in requirements: 500 x $1 = $ 500
- Found in design: 150 x $5 = $ 750
- Found in unit test: 250 x $10 = $ 2,500
- Found in system test: 100 x $100 = $ 10,000
- Found by the customer: 0
- Total cost of quality: $ 13,750

By doing the proper quality assurance, building quality in and verifying at each phase that there were no escapes, the cost of quality could have been reduced by more than $500,000. An ideal model, of course, but could it be done in practicality? The QA manager stepped forward to be the quality champion. The development group and analysts agreed to put quality first. The knowledgeable PM agreed to put quality first because he knew that building in quality from the start would keep the project on schedule and eliminate the 11th hour heroism.

As with the other projects, this one started with the requirements and started late. The PM told the analysts to write the most complete requirements possible and gave them the time they had requested at the beginning of the project, even though this now crossed into the allotted development time. Having never been given a reasonable timeframe before, the analysts enthusiastically launched the project with extensive customer meetings. The QA manager invited himself to these meetings, knowing that the more he understood of the customer's environment and needs, the better job he could do in monitoring quality throughout the lifecycle. He also knew that getting his team involved early would allow them to ferret out design issues, help the developers create good unit tests and allow the test team to build solid test cases that accurately reflected the customer's usage. The QA manager knew he had the team for the job. He had recruited them, hired them, trained them and motivated them.

Lesson #11

Hire the right people … build a strong QA team to build and maintain a strong quality consciousness in your organization.

During the requirements process, the QA team became more and more active as they worked to understand the customer's needs. They helped the analysts ensure that each requirement was testable.

Lesson #12

If requirements are testable, they provide enough details for the developers to accurately implement the functionality.

The requirements shaped up into exact statements of not just what the system had to do (functional), but also a description of how it had to do it (non-functional). The QA team knew that projects had suffered from usability and performance issues and worked with the analysts to define exact usability and performance requirements. The requirements effort took twice as long as it had on prior projects – almost one third of the total project time. Management fidgeted. The PM stayed calm.

A formal requirements review meeting was held with all the project team members and the customer. Each person had prepared by reading the document. The meeting lasted for three hours, finding 100 bugs. All agreed these were the best requirements ever produced by this organization. The 100 bugs were fixed and the requirements were approved at the next meeting. It's important to note that QA identified the majority of these problems in the review by continually asking, how will I test this? Vague or inaccurate requirements were identified. The customer clarified several points where the words written by the analyst didn't accurately convey the customer's needs.

Lesson #13

A cross-functional requirements review will ALWAYS save more money by preventing bugs than it costs in time and manpower.

The design phase took another month with multiple project team reviews as each component was documented. QA actively participated in this phase as well, concerned with developing their test cases and test data. The best way to verify design specifications is to use them as the basis for test case creation. If they aren't clear enough to make a test case, they aren't clear enough for coding. 50 bugs were found in this phase. Half the scheduled time was gone. The PM was still calm.

Also done at this phase in the project was the quality risk analysis. The QA team did a high level risk analysis and took each of the identified feature implementation items and assigned two numerical risk ratings: technical risk and business risk. Technical risk was used to rate the risk that was inherent in the code or implementation due to complexity, traditional instability, difficulty in creating test data, and any other technical risk factor. Business risk was used to assign a value to the impact to the customer if this item didn't work correctly. Development was asked to

review and provide input into the technical risk rating. The analysts and the customer were asked to do the same for the business risk rating. QA then multiplied the risk items together to get one risk number for each testable component of the software. Testing was prioritized based on the risk factor which allowed the team to mitigate the highest risk items first. In the event that there wasn't sufficient time at the end of the schedule, the QA team could talk about risk mitigation achieved versus risk still known to exist in the product as input into the business decision of releasing the software.

Development was excited to start coding because they had a clear path ahead. Having been working with the more technical QA members, they also had a clear understanding of what was expected of their code quality and how important unit testing would be. They developed their code and unit tests simultaneously and kept a rough count of the bugs they found. 50 bugs were found by the unit testing. Development felt it was the best and strongest code they had ever produced and they had done it in only two months where previous efforts of similar size had taken six months.

System testing began with a daily automated smoke test. This automation had been prepared while development was still writing the code and was used to verify that no issues had crept in to the daily build (remember that 30% regression rate on the previous project?). The QA team had a strong skill mix including technical testers who could verify unit tests and build automation as well as pure black box testers who specialized in the GUI and reporting aspects. The project was apportioned between the testers according to their strengths and the final test phase began. All test cases had been assigned to the features identified above and were prioritized within the features. Feature testing was ordered based on the risk factors assigned to them so that the highest risk items would be tested first. This allowed the QA team to be prepared to discuss a ship/no ship decision at any point in the testing in terms of risk mitigated.

The QA manager, based on past experience, knew that each bug found during testing would cost his testers four hours of bug investigation time, on average. Fewer bugs means faster testing. He had conservatively estimated that his team would find half the bugs they found on the previous project or 250. Since they had 1000 test cases, that meant, on average, one bug would be found for every four test cases. Given the average bug investigation time of four hours, one hour of bug investigation time was assigned to each test case in order to determine a reasonable schedule. Using this number, he determine that he would need 500 hours of test case execution time (average of 30 minutes per test case) plus an hour of bug investigation time per test case, or a total of 1500 hours which is about 38 manweeks of testing time. Reaching the 80% risk mitigation goal would require 30 weeks of test time. Spread across the five testers who had been involved in the project from the beginning, six weeks of testing was expected. While not ideal, this still met the project requirements. As testing commenced, the metrics proved the bug estimate was high and testing actually completed in a month and achieved 90% risk mitigation. 30 bugs did escape to the customer, but these were all lower risk issues.

Lesson #14

Cost of Quality metrics are easy to gather and help to focus a team on the high return activities

So what was the cost of quality on this release? Let's look at the numbers:

- Found in requirements: 100 x $1 = $ 100
- Found in design: 50 x $5 = $ 250
- Found in unit test: 50 x $10 = $ 500
- Found in system test: 150 x $100 = $ 15,000
- Found by the customer: 30 x $1000 = $ 30,000
- Total cost of quality: $ 45,850

Still not perfect, but a vast improvement over the previous effort. And it shows the areas that still need to improve. Too many problems are still getting to system test. We'd like to see no problems get to the

customer. But, even with these acknowledged areas to improve, we still saved half a million dollars on cost of quality and shipped on time.

Summary

Fiction? No. These cases were created from a combination of real projects with real humans. So what made the last project successful? More time? No, it actually took less time than previous projects. Fewer features? No, the customer received the functionality they needed. Heroics? No heroes needed. It was the people. A quality-conscious team guided by a smart PM and driven by an active, involved and capable QA team. A quality-focused team will produce a better project in a shorter amount of time, every time, but you have to have the right people to make it happen. We don't need the heroes to ride in at the end and save the project if it's never in distress. A planned project with a quality focus won't be in crisis. There may still be tradeoff decisions, which is why we still used risk-based testing to be sure we mitigated the highest risk first, but these can be informed decisions with measurable consequences.

People make our projects happen, regular people who are doing their jobs, not heroes (although some might argue that the PM was heroic to stand up and defend quality practices when the schedule lengthened at the beginning of the project). The QA team must have the skills, personalities and capabilities to perform a quality function throughout the lifecycle. Get that team together, give them responsibilities and integrate them into the project from the beginning. Building a quality team, like building a quality product, takes effort. Quality doesn't just happen, the right people make it happen.

ASSOCIATION FOR TESTING &
SOFTWARE QUALITY ASSURANCE
Global Certification Body

---

www.atsqa.org
