How to Develop High Quality Software

John J. Paliotta

john.paliotta@vectorcast.com

Abstract

When organizations design a physical product, a significant amount of time is spent designing the manufacturing process and on the quality assurance testing at each stage of production. If the product cannot be manufactured or tested efficiently, it will be redesigned.

In contrast to this, testing is often an afterthought for software development groups, with little thought given to the test and manufacturing process until late in the development cycle. During the design and development phases, the focus is almost always on functionality and performance. While these items are important, they are meaningless if the application is buggy and difficult to maintain.

This paper will define a software development process that spreads testing responsibilities across the entire organization and life-cycle, and improves quality by: ensuring the completeness and correctness of requirements, measuring the effectiveness of test activities, and implementing a continuous and repeatable test process.

Biography

John J. Paliotta co-founded Vector Software in 1990. In 1994, he and William McCaffrey built the first version of the VectorCAST products, VectorCAST/Ada. This product was initially sold to customers building Avionics, Military and Space applications. Currently Paliotta serves as the Chief Technology Officer and oversees all Engineering. He received his AB in Math from Boston College.

1 Overview

On a yearly basis, organizations commit themselves to key objectives, often achieved by improved processes and measured via metrics for performance, quality, revenue, and profitability. Measuring the performance involves testing and reporting. Savvy employees know to ask for a list of what they will be judged on before being reviewed. However when it comes to developing new software products, identifying goals for quality and testing is rarely considered.

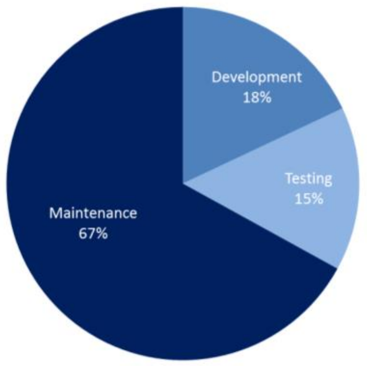
When organizations design a physical product, a significant amount of time is spent designing the manufacturing process and on the quality assurance testing at each stage of production. This allows manufacturing and test issues to be solved prior to the design being complete. If the design cannot be manufactured consistently and economically, it will be reworked.

Much has been written recently about the Internet of Things where nearly all electronic devices in the world will be web enabled. No one is sure of how the architecture will evolve, but it seems obvious to me that it must involve the migration of intelligence to these endpoints. Clearly it will not scale to have trillions of sensors transmitting raw data back to centralized data centers.

Based on this fact, it seems incumbent on the developers of this new wave of connected devices, to prove the robustness of the software that will control these endpoints. It might be a great feature if your refrigerator can be used to order groceries but not a good side effect if the compressor shuts down because the internet connection locks up.

2 The Current State of Quality

According to Cambridge University research[1], the global cost of debugging software has risen to $312 billion annually. Companies are clearly spending incredible amounts of money treating the symptom, but not nearly enough solving the root cause.



Multiple studies have shown that the largest component of software cost is not the original development, test, and manufacturing cost, but the post release maintenance cost.

As shown in this chart, the maintenance costs are generally double the original development costs. Is there any other industry that could support this cost structure? If it costs an auto company $10,000 to produce a car, do they expect the warranty costs for that car will be $20,000?

3 How did we get here?

Software is interesting for a variety of reasons, but one of its best and worst features is its malleability. It is so easy to change software that most applications live in a constant state of flux, with patches for bugs and new features constantly being bolted on, and often times breaking things in the process.

The malleability of software has led to a “we can fix things later” attitude among software developers. “During the next round of changes, we’ll make it better.” This attitude is so pervasive that a term was coined by Ward Cunningham for the cost associated with shortcuts taken; he calls this “Technical Debt[2]”.

---

[1] Financial Content: Cambridge University study states software bugs cost economy $312 billion per year

[2] http://en.wikipedia.org/wiki/Technical_debt

The fact is that most all applications have a pretty large technical debt, and as with financial debt, at some point the debt must be repaid.

In the past technical debt was often “paid off” via a complete application re-write, kind of a declaration of bankruptcy. As recently as 15 years ago, it was common to build an application, patch it for a number of years, and then do a complete rewrite once it got too slow and hard to maintain. In fact the first system I worked on out of college went through three implementations in six years, moving from 100% assembly language, to a high-level language, to new hardware and a second high-level language.

As software has become larger and more complicated, there is less green field development. Most new development relies on legacy code bases, and the code being built today will have a longer life-cycle than ever before.

The famous Heart Bleed bug from 2014 is a great example; the bug was introduced by an engineer in 2011 and existed un-detected in the OpenSSL code base for three years. By the time it was detected, the bug created an exploitable vulnerability in more than 17% of all secure web servers.[3]

4 There are no easy Answers

I recently bought a new piece of networking equipment, and found the following in the user guide:

“As with many electronic devices many errors can be resolved by turning the power off and back on to reset the device.”

So is that the solution? As more and more devices that we use every day contain software, will we be running around cycling power to everything in our houses? Clearly not. Consumer and market pressure will eventually force all vendors to improve quality or perish.

The debt analogy, as mentioned above, is perfect for software, because with Technical Debt, just like financial debt, there are no easy answers. Just as a lottery ticket is not a likely solution for financial debt, there are no “silver bullet” solutions to Technical Debt.

[3] http://en.wikipedia.org/wiki/Heartbleed#Affected_OpenSSL_installations

5 The Quality Improvement Process

Over the last 30 years, there have been many new technologies and development paradigm changes aimed at improving software quality (the lottery ticket approach). While all of these advances have provided significant productivity improvements, there has not been an associated improvement in quality.

Continuing with the financial analogy, the solution to financial debt requires you to: create a budget (set goals), lower spending (change behavior), pay down debt (reap benefits). Improving the quality of software requires the same measured approach.

When organizations think about improving software quality for legacy code bases, they are often overwhelmed by the scope of the problem. Technical Debt has been accrued over many years with many developers contributing to the problem, so it is unreasonable to hope for an immediate improvement. Quality Improvement must be seen as a process that is measured by incremental improvement, not a single activity that magically transforms a code base.

The first step is to develop a quality plan to minimize new technical debt as legacy applications are modified and new development is undertaken. The rest of this paper will focus on actionable ideas for developing such a plan.

5.1 Best Practices from Industry

Interestingly, many industries that build safety-critical software such as avionics, automotive, medical device, aerospace, railway, and industrial controls, have already formalized best practices to produce dependable software.

These best practices are formalized in development standards such as: DO-178B/C (Avionics), ISO 26262 (Automotive), IEC 615108 (Industrial Controls), FDA and IEC 62304 (Medical Devices) and the CENELEC standard (Rail Applications). While there standards do not ensure bug-free software, they certainly provide a framework for a repeatable process that improves quality.

If you look at these standards, they have three main objectives:

- Tests should prove that an application complies with its formal requirements
- Code coverage should be measured to ensure that testing is complete
- Each deployed version should undergo complete testing

5.2 Best Practices from Experience

From my 30 years building software and software tools, I would add the following objectives to the list:

- Requirements must be complete and correct
- Coding style and architecture should be easy-to-understand
- Testing must be part of everyone's responsibility
- Testing must be Automated and Continuous

6 A Practical Approach to Software Quality

6.1 Ensure Requirement Completeness

Many defects will be prevented by ensuring software requirements are complete. Take for example, writing the specifications for a square root function. While this is a simple function within a math library, it can be very poorly implemented if the requirements are not complete.

Example:

> *The square_root() function shall return the square root of its input for all valid values.*

This requirement seems simple, but what range of values should be supported? Will the inputs be 32 or 64 bit values? What do I do with out-of-range inputs? Should I log any errors? How do we log errors?

A better specification would be:

> *The square_root() function shall return the square root of its input for all valid inputs, and 0 for all invalid inputs. Valid inputs are positive 32-bit floating point numbers, zero and positive infinity. Invalid inputs are negative numbers, negative infinity and NaN. In the event of an error, the type of error and system time should be logged using the common error logging system.*

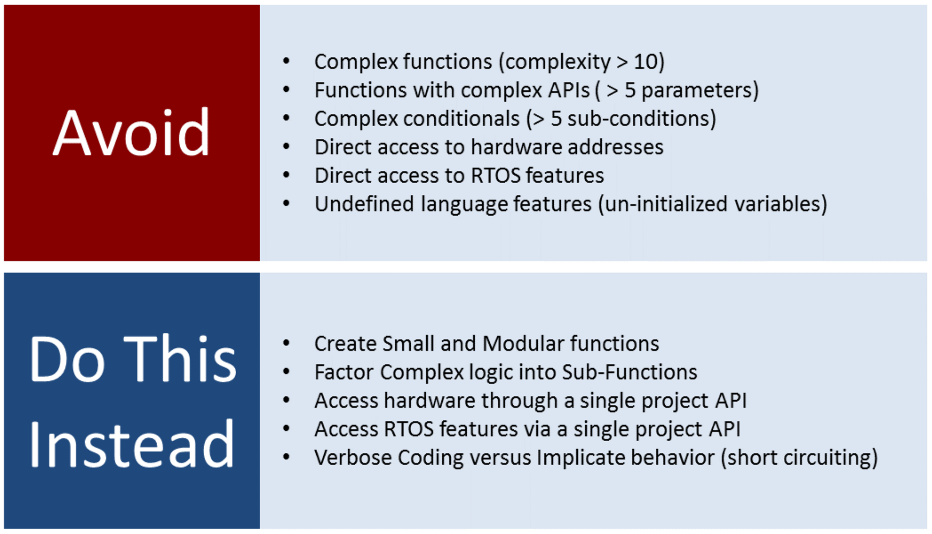
Based on this revised specification, an architect is able to build a set of low-level requirements and test cases that will validate correct performance, and the developer will have a clear understanding of the implementation required.

6.2 Write Simple Code

For code to have improved testability, it must be flexible and easy to understand. In the early days of software development, developers were held in high regard if they could make software small and fast. This was in a time when computer systems had limited CPU clock speed and limited memory. Engineers needed to be efficient to fit code onto the device. Programming tricks abounded, as did really hard to understand code!

As application size and lifecycles become larger and longer, there should be a premium on building code that is easy to understand and maintain. Fortunately, making code easy to understand dramatically improves testability.

Bob Gray of consulting firm Virtual Solutions said that "Writing in C or C++ is like running a chain saw with all the safety guards removed."[4] Some of these "safety guards" for coding style that will improve testability are:



Avoid

- Complex functions (complexity > 10)
- Functions with complex APIs (> 5 parameters)
- Complex conditionals (> 5 sub-conditions)
- Direct access to hardware addresses
- Direct access to RTOS features
- Undefined language features (un-initialized variables)

Do This Instead

- Create Small and Modular functions
- Factor Complex logic into Sub-Functions
- Access hardware through a single project API
- Access RTOS features via a single project API
- Verbose Coding versus Implicate behavior (short circuiting)

Figure 2: Follow these tips to help you write easily testable code

Every engineer will have their own idea of what simple means, so it is important that development groups implement their own set of rules, and it is critical that these rules be formalized with automated checks or at least peer review checklists.

[4] Byte (1998) Vol. 23 Nr 1-4. P. 70

6.3 Implement Quality Gates

Creating a culture of quality requires documented workflows for each team member that addresses his or her part of the quality challenge. For example, developers should have a set of prerequisites to meet prior to making commits to Configuration Management (CM). Without any restrictions, or “gates” on behavior, code bases will very quickly become buggy. On the other hand, if there are too many restrictions, a new feature might never get released.

The following diagram shows a reasonable set of quality gates to be met before code changes are committed to CM:

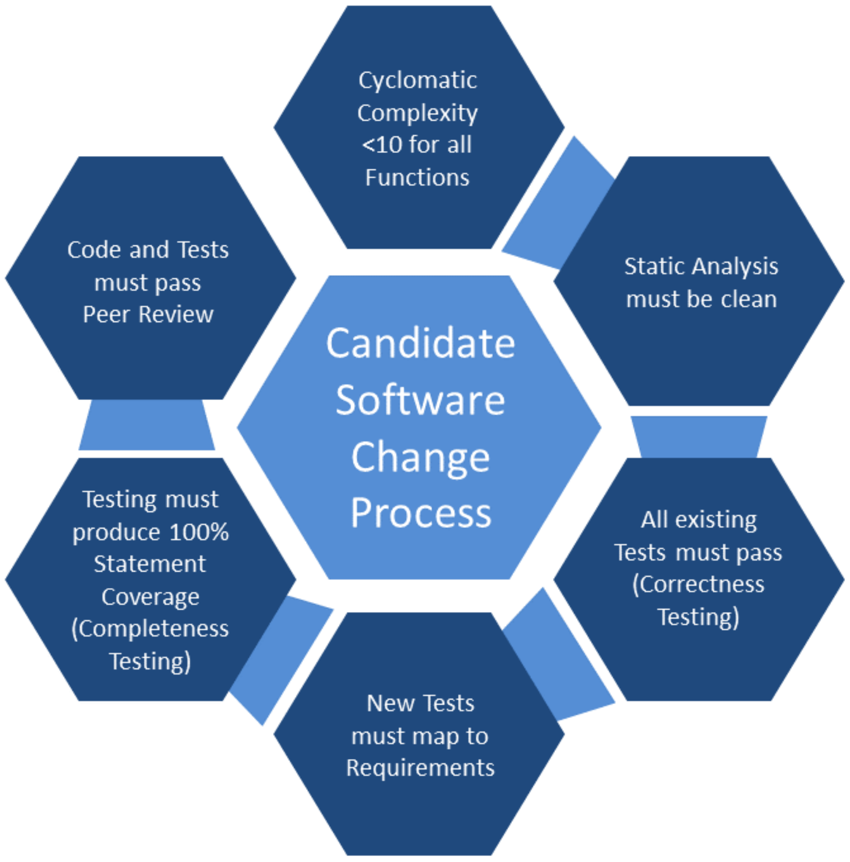

Cyclomatic Complexity <10 for all Functions
Static Analysis must be clean
All existing Tests must pass (Correctness Testing)
New Tests must map to Requirements
Testing must produce 100% Statement Coverage (Completeness Testing)
Code and Tests must pass Peer Review
Candidate Software Change Process

Figure 3: Candidate software change process

It is ideal if the quality gates can be automated with a tool, but it is not critical, and it is certainly not a reason to delay implementing any gates. As a first step, document a process, implement a checklist that reflects the process, and apply the checklist via peer review.

6.4 Develop Achievable Goals and Communicate the Rationale

Many people will look at Figure 3 and think that implementing this workflow is impossible. "We don't have tools." "We don't work that way." "We'll never get anything done." These excuses cannot be tolerated if you want to improve quality.

Whether you have a 20 year old code base, or all new code, you must create a workflow that makes sense for the organizations goals, and one that will be embraced by the team. One size does not fit all. For example, with a legacy application that has no formal requirements, limited test cases, and no test automation, it doesn't make sense to choose a goal of 100% code coverage. But a goal of 100% coverage for all "new" code does make sense.

Transitioning to a quality culture may be difficult for developers who consider the new "gates" to be more work and time consuming. As you start to introduce new processes, mentoring and training can help ease the transition. Create a group of senior team members to lead the transition, leverage training and tools, and validate the new workflow by comparing quality metrics for projects that adopt this new workflow against historical trends.

Hint: Start with team members who have an early adopter mentality.

Additionally it is critical to communicate the reasons for change with the whole team. A simple and honest message is best:

> *"We just lost a long term customer because they're unhappy with our buggy software. If we don't change we'll be out of business in a year."*

Well I hope things aren't that dire for your group, but the point is people need to understand the motivation for changes, and that improvement requires change.

6.5 Publish Metrics

The biggest thing that you can do to speed adoption is to publish data to the whole team that show tangible results. Focus on Good, Fast, and Cheap (from the project management triangle). The following are some metric ideas:

Good:	Bugs found by Customer, QA, and Developers
Fast:	Average time for a branch to get through QA
Cheap:	Percentage of staff time used for bug fixing versus new feature development

More important than the metrics that you choose to publish is the story that they tell, choose metrics that will move over short periods of time and publish trend data.

Consider an application that has been deployed to customers, but is still being enhanced with a three-month release cycle. Publishing the number of bugs found by customers compared to those found by developers or internal testing would provide actionable data for your team.

The following graph shows a reduction in customer reported bugs on each sequential release. The reduction in customer reported bugs tells your team that their quality efforts are working and will encourage them to continue.

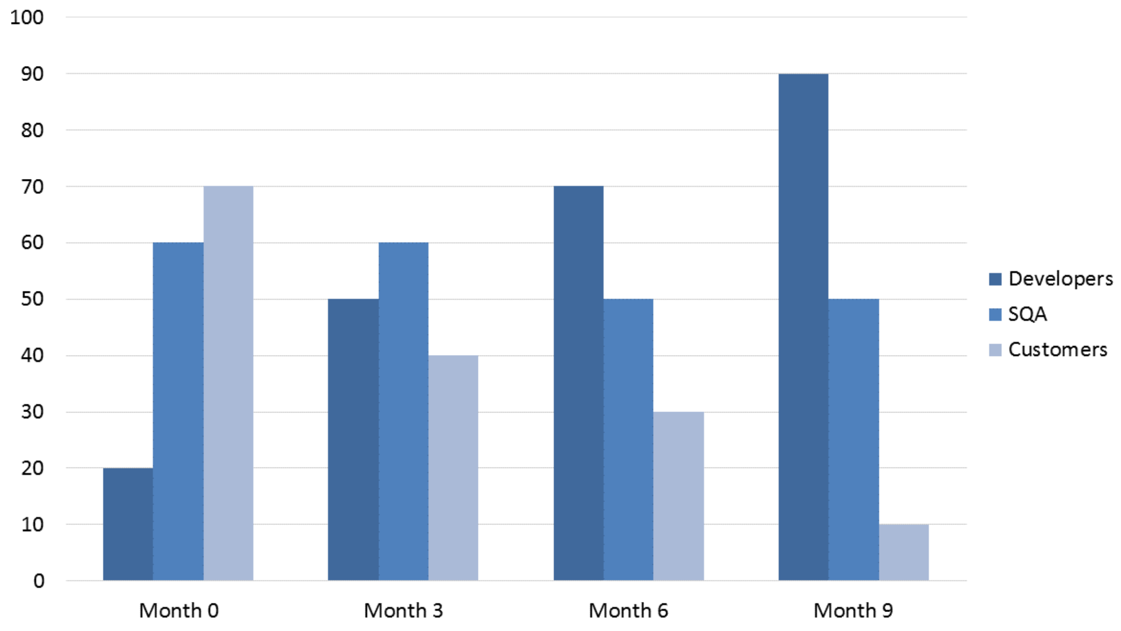


Figure 4: Fixing bugs before they are shipped is the most cost-effective way to deliver quality

If the data shows a negative trend this is also useful, as it helps the team understand the scope of the problem and accept that change is needed.

6.6 Test to Requirements

In the earlier section on requirements, I discussed the benefit of well written and complete requirements to ensure that a developer has all of the information to create a quality implementation. The second benefit of good requirements is that they provide a framework for testing. In fact the software development standards such as DO-178 (for avionics) and ISO 26262 (for automotive) specifically require that tests, and code, be mapped to requirements.

Mapping tests to requirements provides several benefits:

- **It ensures that tests validate what the software is supposed to do, not what it does do**. One of the big problems with developer-based testing is that the test values are often chosen by looking at the implementation rather than the requirements.
- **It ensures that there are not any forgotten cases**. If the requirement says that there are 100 different values for a message ID, shouldn't we have tests for all 100 values?
- **It promotes the testing of functional boundary cases**. Everyone agrees that software most always fails at boundary values, not at nominal values. In addition to testing at machine boundaries: (e.g. max Integer) it is important to test functional boundaries (e.g. max Speed)
- **It allows managers to understand which tests need to be re-run when requirements change**.

6.7 Measure Code Coverage

Code coverage analysis might have the highest return on investment of any development process change. Modern tools make it easy to implement code coverage as part of the build process, allowing coverage measurement with no changes to the test process.

One of the most misunderstood issues with code coverage is its relevance to software quality. Even within groups that must adhere to software development standards such as DO-178 (for avionics) and ISO 26262 (for automotive) there is confusion. So let me state it simply:

100% Code Coverage should not be the goal of software testing;
it should be the result of complete testing.

What this means is that Code Coverage Analysis goes hand-in-hand with the Requirements Based Testing described in the previous section. The only way to judge the sufficiency of your requirements based tests is by measuring the resultant code coverage.

Gaps in the code coverage might point to poorly implemented tests, which don't adequately test the requirements, but the gaps might also be the result of "extra" code (e.g., the requirement indicates there are 100 message IDs that should be handled, but the developer added 10 new ones and the requirement never got updated).

Code coverage data is also a great choice as an initial metric to publish to your team. As with everything I have mentioned so far, it is not important what the initial value is, but it is important to show coverage trending up to validate the "extra work" your team is doing. If you are not measuring code coverage, then you are testing "blind".

6.8 Speak the Language of Test

Everyone working on a software project has something to contribute to software quality. The system architects understand the big picture about how the application should function. The developers understand the implementation decisions made and how they will affect performance. The QA team understands the correctness of the application and the interdependencies between functions that might not be obvious to other team members.

It is important to involve all of these team members in developing the tests which will prove the correctness of the application.

Written language has been a boon to the development of the human race, but it is a really inefficient way to describe software bugs. One of the largest inefficiencies in the average development team is communicating efficiently. Read through any random bug report in your system, and it is likely that you'll often see the following timeline:

1.	Bug found:	QA sees something that is broken
2.	Bug documented:	A description of the bug is entered into the bug tracking system
3.	Developer gets bug:	Developer tries to duplicate the bug, can't, asks for more details
4.	More details added:	QA adds more details
5.	Bug duplicated:	Developer duplicates the bug
6.	Developer fixes bug:	Changes committed to fix the bug
7.	Fix sent to QA:	Fix provides a partial solution or has negative side effects.
8.	Bug sent to developer:	…

You might get dizzy following this ping pong match of bug assignment.

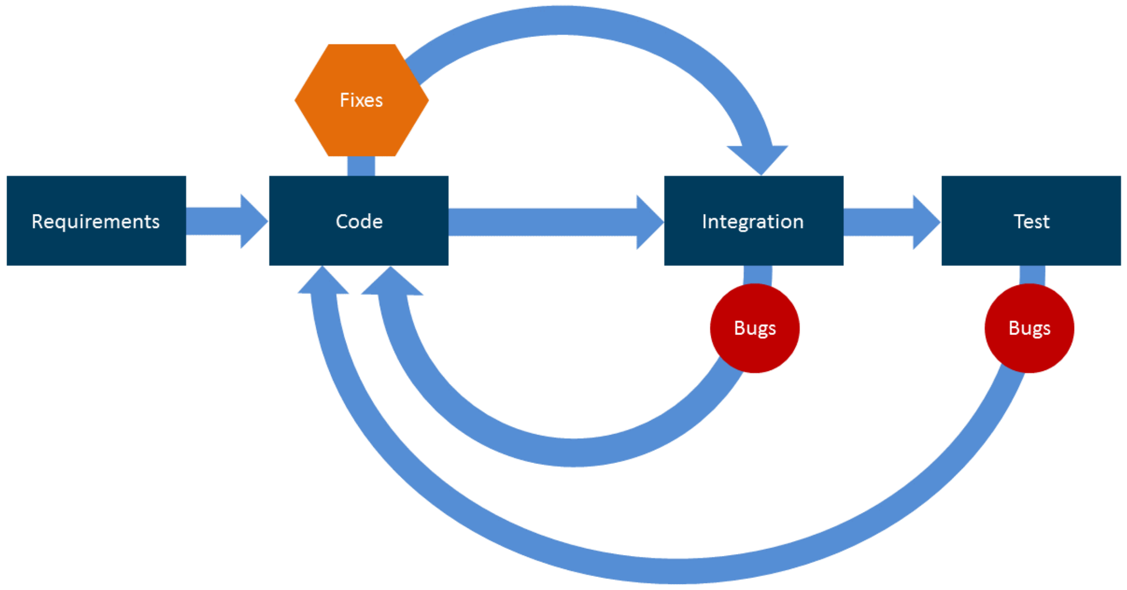


Figure 5: The back and forth game that many bugs experience.

It is much more efficient if test cases are used to document the initial bug rather than a text description. It's even better if the developer can easily run that same test. And it's perfect if the developer can run the test under control of the debugger to find the problem.

This workflow is what I call *Speaking the Language of Test*. It removes all of the inefficiencies that result from team members trying to describe a problem with words, and it gets rid of the back and forth Ping-Pong game that many bugs experience.

6.9 Implement Test Automation

In order for software testing to improve the quality of applications, it has to be thorough, easy and fast. This enables developers to have the ability to run any test, at any time, on any version of the code.

Every organization has developed a software build system allowing for unattended incremental application building, but most have not implemented a repeatable incremental testing infrastructure. Too often, testing is performed periodically with manual processes required, rather than constantly and incrementally with complete automation.

Each developer often implements their own testing methodology, and these methods are often nothing like what QA is using to test. If there is no organization-wide platform for testing, there is a significant lag between when bug is introduced and when it is found. The longer this lag time, the more difficult it is to find the cause, and the harder it is to fix. Ideally, each change to the software runs all tests that are affected by that change before the change is integrated.

6.10 Change Based Testing

Even with complete test automation it is likely that running "all tests" will take hours or days. Faced with this situation, it is reasonable that most groups run tests periodically rather than constantly. In fact, telling a developer that changed one line of code that they need to run a week of testing is a great way to ensure developers see testing as the enemy.

Change based testing solves this exact problem. It ensures that the quantity of testing is in proportion to the source changes by automatically choosing the sub-set of tests that are affected by a change and running only those tests. For small to mid-sized changes, test cycles take minutes rather than days.

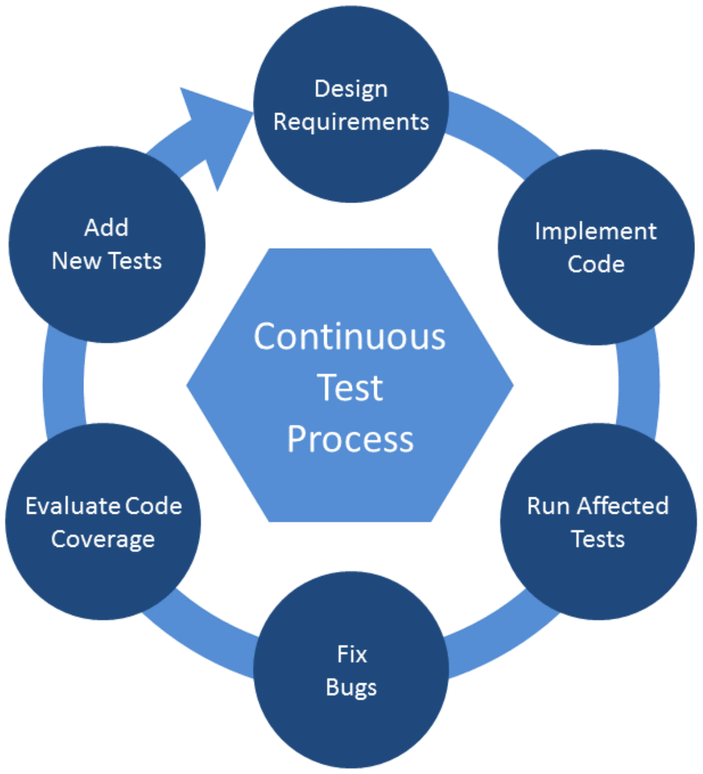


Figure 6: A continuous test process reduces development time

6.11 Invest in Development Tools

Over the last several years, there has been a steady flow of new tools for software testing. In order to create a quality culture, there are several of these tools that should exist within an organization's software toolbox.

The first is a **static analysis tool.** These tools are able to parse code to find problematic patterns within the source code that commonly result in bugs. In fact several coding standards have been developed by the industry to identify these patterns. For example, the MISRA standard[5] adopted by the automotive industry provides a set of rules that prohibit certain code constructs that are either ambiguous in the language standard or error prone in implementation.

The second is a **code coverage tool**. These tools add instrumentation bread crumbs to a code base during testing to document the statements and branch outcomes that have been tested. They make it easy to focus new test efforts on inadequately tested portions of an application.

The third is a **test automation platform**. If your team is going to *Speak the Language of Test* it is critical that they have a common platform to allow them to communicate. A test platform should support the easy creation and execution of black-box and white-box testing to ensure application correctness. These tools, among others, will help ease and ensure the process of better quality software.

6.12 Throw Away Your Source Code

Historically development teams produced source code as their work product. Emphasis has always been on the implementation details. While clearly the implementation is critically important, I would suggest that the emphasis should be placed on the Applications Programming Interface (API) and test cases. If a robust and well documented API is created and test cases are built to formalize the correct behavior of this API, then the implementation is really secondary.

In fact, I would argue to use your most senior developers for API and test design and utilize lower-cost, junior developers for the implementation. Changing the emphasis to the API will increase software reuse and decrease life cycle maintenance costs.

7 Business Advantages of Quality Improvement

There are many compelling business reasons for improving application quality. In our hyper-competitive global economy customer satisfaction is being driven like never before by software. Software is the primary controller of the human interface experience with the majority of electronic devices, and for many devices software is the brand.

Was the success of Apple's iPhone driven primarily by the hardware choices they made or the incredible ease-of-use provided by their ground breaking iOS software? Clearly the software played a huge role.

Improving software quality will result in the following business benefits:

- Fewer bugs go to integration, which shortens integration time
- Shorter integration time means faster release cycles
- Fewer bugs go to customers, leading to happier customers
- Happier customers lead to increased revenue and brand loyalty

[5] http://www.misra.org.uk/

8 Summary

If you are in the business of building software, you have a quality problem. It's that simple. There are no bug free applications. As with any problem, the first step to improvement is to recognize the problem, and then identify a plan for improvement, implement that plan and measure results.

In this paper, I have tried to provide a variety of process improvement initiatives that can be easily adopted by any development group, big or small, but in addition to a process change it is critical to invest in an attitude change. Quality requires a team effort, not just some magic dust from the QA department.

There are two key take-away items from this discussion:

- There are no silver bullets to improving quality
- If everyone on the team has a stake in testing and quality, a higher quality product will be the end result

Building high quality software is not an easy task; it requires a constant engagement in process improvement and metrics measurement. But creating a process that emphasizes testing throughout the development process will result in applications that are released faster with quality "baked in."

References

Byte 1998 Vol. 23 No 1-4: 70

Cambridge University Study States Software Bugs cost Economy $312 Billion per Year. Financial Content. January 8, 2013. http://www.jbs.cam.ac.uk/media/2013/financial-content-cambridge-university-study-states-software-bugs-cost-economy-312-billion-per-year/

Information is Beautiful, "Codebases," http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

McCormick, John. "Software Quality – By the Numbers." *eWeek. March 4, 2004.* http://www.eweek.com/c/a/Web-Services-Web-20-and-SOA/Software-QualitymdashBy-The-Numbers

Misra, "Misra Home," http://www.misra.org.uk/

Wikipedia, "Heartbleed," https://en.wikipedia.org/wiki/Heartbleed#Affected_OpenSSL_installations

Wikipedia, "Technical Debt," https://en.wikipedia.org/wiki/Technical_debt