5 Basic Building Blocks for Engineering High Quality Software using Vector Tools

Experience has taught users to avoid the latest versions of software applications until the inevitable maintenance releases, and patches have been released. Even large enterprises are not immune to buggy launches — we just have to look at the upgrade cycle of software in mobile phones to see a major release followed by several quick "fix" updates. While everyone is aware of the software quality gap that exists between the initial release and the stable release, unfortunately not much progress is being made toward solving the problem.

This technical article discusses **5 actionable ideas** to help development groups close the quality gap.

1. Test Coverage

Use Code Coverage Analysis

Code coverage analysis reports on the portions of the application source code that have been executed by a set of test cases. Analyzing code coverage is the best way to measure the completeness of your test activities. Without measuring code coverage you are "testing in the dark".

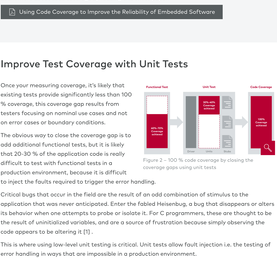

Figure 1 – Analysis of code coverage to measure test completeness

It is important to remember that while achieving "100 % code coverage" does not prove that an application is perfect, it is a critical component of engineering high quality software. In fact, all of the standards associated with safety critical software development mandate code coverage as part of the development process.

- Aerospace uses **DO-178B/C**,
- automotive has **ISO 26262**,
- **IEC 61508** for industrial controls,
- **FDA** and **IEC 62304** standards for medical devices and the
- **CENELEC** standard applies for rail applications.

Learn more about code coverage by reading this white paper:

Using Code Coverage to Improve the Reliability of Embedded Software

2. Unit Tests

Improve Test Coverage with Unit Tests

Once your measuring coverage, it's likely that existing tests provide significantly less than 100 % coverage, this coverage gap results from testers focusing on nominal use cases and not on error cases or boundary conditions.

Figure 2 – 100 % code coverage by closing the coverage gaps using unit tests

The obvious way to close the coverage gap is to add additional functional tests, but it is likely that 20-30 % of the application code is really difficult to test with functional tests in a production environment, because it is difficult to inject the faults required to trigger the error handling.

Critical bugs that occur in the field are the result of an odd combination of stimulus to the application that was never anticipated. Enter the fabled Heisenbug, a bug that disappears or alters its behavior when one attempts to probe or isolate it. For C programmers, these are thought to be the result of uninitialized variables, and are a source of frustration because simply observing the code appears to be altering it [1] .

This is where using low-level unit testing is critical. Unit tests allow fault injection i.e. the testing of error handling in ways that are impossible in a production environment.

3. Test Infrastructure

Make Tests Easy to Run, and Results Easy to Understand

In theory, it sounds like a simple plan: make your tests easy to run and the test results easy to understand. In practice, however, this can be a challenge. Historically, different flavors of tests are built and maintained by different engineers, often using different tools:

Figure 3 – The Test Pyramid includes different types of tests

- **Unit tests** are used to prove correctness of the low-level building blocks of an application
- **Service & API layer tests** are built to prove the correct functioning of complete sub-systems
- **Human-machine interface tests (HMI) / functional tests** are built to prove correctness from an end-user point of view

When tests are partitioned this way, each flavor of tests is owned and maintained by a different group of engineers rather than being shared across all members of the development team. In fact, in most organizations, it is probably impossible for a QA engineer to run a developer test or a developer to run a system test.

In order to improve quality, it should be possible for any member of the development team to run any test at any time on any version of the application.

The key to enabling this workflow is a common test collaboration platform, which captures all tests, along with their preconditions and expected results. Engineers should be able to run a single test, or all tests with the "click of a button". In addition, it is essential that engineers are able to quickly debug failing tests.

Additional information can be found in the following document:

Building a Flexible and Automated Testing Infrastructure

4. Test Efficiency

Implement Automated, Parallel, and Change-Based Testing

Once testing completeness is improved by code coverage analysis, and tests are deployed across the entire organization, the next step is to ensure that tests run quickly. One of the reasons tests are partitioned between multiple groups is that a complete system test might take hours or days to run. Obviously, if you ask a developer who has changed one line of code to run 10 hours of testing, you'll get some pushback. So how can we decrease test time, while still ensuring testing completeness?

The key is to build a testing infrastructure which is scalable, using parallel and change-based testing. Individual tests must be atomic, small, and fast. Too often test suites become tightly coupled over time with new tests simply being inserted into existing tests. This makes tests fragile and test maintenance time consuming. A simple thought to keep in mind when designing tests is, each test should define its own preconditions not rely on the output of other tests.

Beyond the benefits of test maintenance, re-architecting your tests to be atomic enables:

- **Change-based testing**, running only those tests affected by each software change
- **Parallel test execution**, running hundreds of individual tests simultaneously

While every organization has developed a software build system that allows for unattended incremental applications building, most have not implemented incremental testing. Too often, testing is performed periodically rather than constantly and incrementally with complete automation. Change-based testing (CBT) analyses each set of changes to the code base, and intelligently selects the sub-set of all tests affected by those changes. This results in complete testing in a fraction of the time of a full test run. In addition, change-based testing provides an accessible means for implementing a rigorous continuous integration (CI) development process; during the check-in phase of CI, CBT provides an efficient means to verify the build and detect problems early.

Figure 4 – Regression test of the test cases impacted by the code change

To improve speed even further, consider parallel testing. By integrating your test platform with a continuous integration server, and virtualized test machines, you can reduce total test times from hours to minutes, or minutes to seconds.

5. Refactoring

Refactor Code Bases to Improve Maintainability

Code refactoring is the process of restructuring application components without changing its external behavior (API).

Figure 5 – Code refactoring approach

Without refactoring, application code becomes overly complicated, and hard to maintain over time. As new features and bug fixes are bolted onto existing functionality, the original elegant design is often a casualty.

Code refactoring improves code readability and reduces complexity, hence maintenance cost. Code refactoring, executed well, offers the additional promise of resolving hidden, dormant, or undiscovered computer bugs or vulnerabilities in the system by simplifying the underlying logic and eliminating unnecessary levels of complexity.

Every application has fragile and buggy sections which developers are hesitant to change for fear of breaking existing functionality. The only way to confidently refactor these fragile modules is to ensure that you are building tests to formalize the expected behavior.

Conclusion

Over the last thirty years, there have been a steady flow of tools, design patterns, and development paradigm shifts. Many of these have promised improved quality without increased time or effort. It should be clear to everyone in the software industry by now, that there is not, and will never be, a silver bullet that provides improved quality at no "cost". The only sensible way to improve software quality is to improve the effectiveness of software testing.

Back to Testing Trends

References:

[1] Hristov, Ivan. Chasing Heisenbugs from an AKKA actor integration test with awaitility. September 16, 2012. honeysoft.wordpress.com/category/heisenbug/