Don't Test My Code
It'll Test itself.

Ananth Kasillingam, WID (eBeam)
Denin Sahovic, WID (eBeam)

Abstract – Current development model segregates software creation and verification into separate entities. And most of the verification is done via strenuous manual testing. This paper presents Layered Verification approach to software development made possible by an Integrated Team. The layered approach tests different aspects of a use case in different levels of automated testing. The layering accelerates software verification without compromising test coverage. Integrated team unifies developers and testers as one entity accelerating the feedback loop. Put together, Integrated Teams and Layered Verification enable continuous delivery of high quality software and faster turnaround time for features.

I. Introduction

A. *Striving for Excellence*

KLA-Tencor is a longtime leader in semiconductor equipment and yield management systems. We pride ourselves in creating high quality cutting edge products. To achieve high quality software we have a well-defined Software Development Lifecycle process.

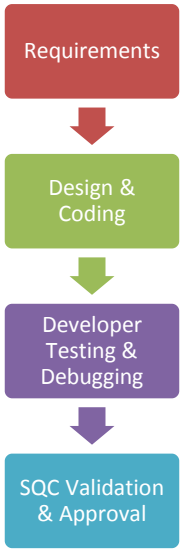

Figure 1 Simplified SDLC Process

The customer requirements are analyzed and documented, which influences our Design and the coding. Developers test, debug and fix issues before delivering the software to the watchful eyes of our SQCs.

SQCs verify all the use cases in the requirements document. Any issues found are logged as bugs and sent back to the developer. If everything passes, a seal of approval is made. This model has worked good enough to satisfy our customer expectations in the past.

B. *Changing Landscape*

Due to the better Software Development practices, improved tools and stability of software in general, customer expectations have increased. In today's world a customer expects:

1. Accelerated release of features.
2. Bug free software. (Nearly zero bugs)
3. Repeatable and high confidence upgrades.
4. Increased expectations on MTBI.
5. Zero downtime.

C. *Limitations*

The top complaints about us from our customers are:

1. Not nimble enough in delivering new features.
2. Upgrades give nightmares to Tool

Owners.

Reducing customer tolerance for bugs mandates increased testing of functionality. Increased complexity of software leads to:

1. Limited coverage of functionality with black box testing.
2. Increased testing time and need for resources.
3. Slower delivery of features.
4. Efforts spent in fixing hot customer bugs and verifications.
5. Self-feeding negative spiral.

Most of our tests are manual or fragile automated UI tests.

D. *Integrated approach*

Modern development tools and improved software architectures improve the chance of success. But they don't verify the quality of the software.

This paper discusses an integrated development and verification approach based on modern development thought processes in the software industry customized to KLA-Tencor's business model.

We are grateful to the Agile community as most of our ideas in this paper are influenced from their efforts.

II. Layered Verification

A. *Smarter not harder*

The first thought to improve software quality is to test harder. Experience shows this is inefficient and will not get us in the accelerated delivery path. Hence a more pragmatic approach would be to develop and test smarter.

The basic idea is to layer the tests into different levels and verify different aspects of a use case in each level. This approach presents a layered testing pyramid as shown in Figure 2.

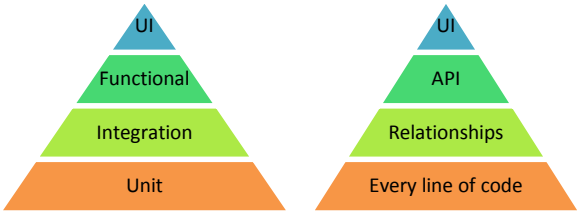

Figure 2 Smart Testing Pyramid

When you move up the pyramid:

1. Control on what can be manipulated decreases.
2. Fragility increases.
3. Test execution gets slower
4. Relationship availability improves.

Testing pyramid takes a smarter approach and uses stable and more controllable lower layers for:

1. Increased code coverage
2. Faster test execution
3. Reduced testing efforts

The next higher level tests only the untested relationships.

B. *Boolean Mantra for Automation Nirvana*

Layering model depends on the premise that all the tests are automated and answer with a definitive Yes or No about their success criteria. This enables an automated validation of the software without human intervention.

C. *Unit Tests and TDD*

Unit Testing is the process of testing individual units of source code. A unit is the smallest testable part of an application. In modern object oriented programs it would map to a class.

In a Unit test, configuration, current state, working environment and inputs of an object are controlled to generate an output that can be verified.

In real world applications the unit of interest or System Under Test (SUT) will depend on other units (dependencies). In this level of testing we isolate SUT from the dependencies. Substitutes such as dummy, stub and mock could be used to achieve isolation. In addition substitutes give a finer control in setting the test environment and verifying the results.

The characteristics of unit test would be:

1. Every line of code is covered with at least one unit test. (100% code coverage)
2. Unit tests are independent and they can be executed in any order.

3. Dependencies are mocked.
4. More control knobs available at lower levels.

Like most practitioners of Unit Testing we recommend Test Driven Development (TDD) where the Tests are written before the code. The model follows a cycle called Red → Green → Refactor

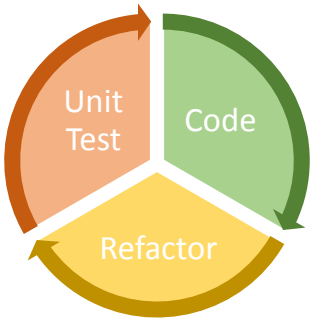

Figure 3 TDD Cycle

This cycle helps in creating a self-correcting loop, which starts with a failing test, followed by a simple code that satisfies the test and finally refactoring the code for maintainability. This structure gives freedom in improving the code and reducing technical debts with higher confidence due to the presence of a safety-net.

Unit tests are extremely fast and they are run on developer machines every time a piece of code is modified and with every build. Even UI behaviors can be Unit Tested to a large extent using modern UI patterns like MVVM, MVP, MVC, etc.

Unit tests cannot test the relationship between units.

D. Integration Tests

Integration tests arrange units into integrated aggregates and test the complex relationship aspects. They are defined such that:

1. Only aggregates of interest are tested.
2. Only relationship related use cases are tested.
3. Helps in validating parts of software system.

E. Functional Tests

Functional tests verify the system as a whole. They verify the specifications using externally visible APIs. This is a black box testing but uses function calls instead of UI. In our scenario we want to extend this idea to directly access CAO layer to run most of the tests circumventing the UI.

F. UI Tests

Still our UI is the portal to our tool. UI driven tests are written to validate the high level Customer Use cases. Since most of the functionality is verified using lower level tests, UI Tests just concentrate on Happy Path tests. These tests are automated using an automation framework like CUI, QTP, WinRunner, etc.

G. Manual Tests

Even though we don't recommend manual tests, in real world there will be few exceptions which require them. We expect nothing more than 5% of our use cases being Manual, and we should strive to achieve 0% Manual testing.

III. Integrated Teams and Continuous Delivery

Quality starts right from requirements where well defined Acceptance Criteria is defined in coordination with customers. TDD cycle converts them to lower level unit tests and code.

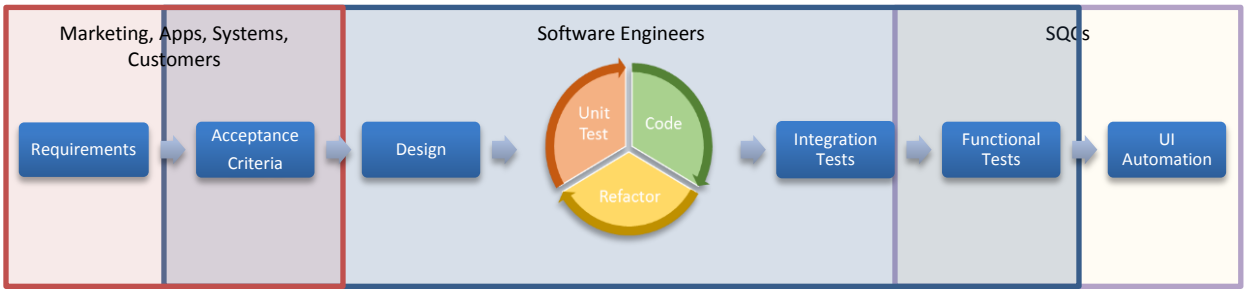

Figure 4 Integrated Teams

SQCs and Developers work together as a team to create high level Functional Tests. UI Automation is done by SQCs. They also work together to influence the design, code and test, blurring the line between Developers and SQCs.

Every check-in triggers an automated build, verifying the software using automated tests. A failing test fails the build and notifies the developer of last check-in to fix the issue. Every successful build is potentially shippable, enabling Continuous Delivery model. This nirvana leads us to a state of nimble, bug free delivery.

IV. Success Story

"eDX Smart Reference", a feature requiring accelerated delivery gave us a chance to try this model out. The feature had high customer visibility and was being added to the last official build with new features. Applying this model lead to delivery of the feature on time with zero bugs discovered by SQCs or Customers till date. Considering this is a heavily used complex feature, that's quite an achievement.

V. Where do we go from here

Currently eBeam has working frameworks for doing Unit Testing on eCougar, Functional Tests on Machine Control and IMC and Coded UI tests in association with WIN. There is a very rudimentary support for Integration Tests.

We need to improve on the existing frameworks and build a robust framework for Integration Testing. Putting these isolated models to work in unison for smarter testing, enabling the tools do the harder work. Finally, create a knowledge repository of what's the responsibility of each level of testing.

VI. References

[1] K. Beck, Test Driven Development By Example, Addison Wesley, November 18, 2002.

[2] G. Mesazaros, xUnit Tests Patterns: Refactoring Test Code, Addison Wesley, May 31, 2007.

[3] R. Osherove, The Art of Unit Testing: with examples in C#, December 07, 2013.

[4] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison Wesley, July 8, 1999.

[5] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison Wesley, July 27, 2010.