## The Testing Mindset

There is particular mindset that accompanies "good testing" – the assumption that product is already broken and it is your job to discover it. You assume the product or system is inherently flawed and it is your their job to 'illuminate' the flaws.

Designers and developers often approach software with an optimism based on the assumption that the changes they make are the correct solution.

How could they do anything else? But they are just that – assumptions. Someone with a testing mindset is aware of their assumptions, and the possible limitations.

Without being proved assumptions are no more correct than guesses. Individuals often overlook fundamental ambiguities in requirements in order to deliver a solution; or they fail to recognise them when they see them. Those ambiguities are then built into the code and represent a defect when compared to the end-user's needs.

By taking a sceptical approach, we offer a balance.

Take nothing at face value. Always asks the question "why is it that way?" Seek to drive out certainty where there is none. Illuminate the darker part of the projects with the light of inquiry!

Sometimes this attitude can bring conflict to a team, but it can be healthy conflict if done properly. If people recognise the need for a dissenting voice, for a disparity of opinions, then the tension is constructive and productive. But often when the pressure bites and the deadline is looming the dissenting voice is drowned out.

## Test Early, Test Often

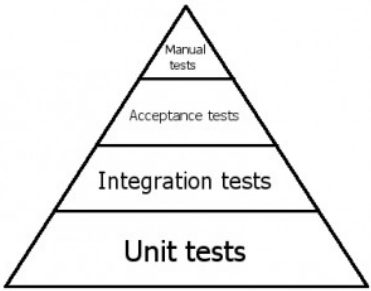
There is an oft-quoted truism of software engineering that states - "a bug found at design time costs ten times less to fix than one in coding and a hundred times less than one found after launch". Barry Boehm, the originator of this idea, quotes ratios of 1:6:10:1000 for the costs of fixing bugs in requirements, design, coding and implementation phases (waterfall).

If you want to find bugs, start as early as possible.

That means unit testing (qqv) for developers, integration testing during assembly and system testing. This is a well understood tenet of software development that is simply ignored by the majority of software development efforts.

This concept is epitomised by the (automated) testing triangle promoted in various circles. Like most aphorisms, there is truth in it but it oversimplifies things and it should be interpreted in your own context. Certainly you should have a well thought out structure to your testing.

In modern software development it is common to use continuous integration and delivery (qqv CI/CD). *Every* change a developer makes is tested (and possibly deployed) *every time* they commit it to the code repository.




Nor is a single pass of testing enough.

Your first past at testing simply identifies where the defects are. A second pass of (post-fix) testing is required to verify that defects have been resolved.

The more passes of testing you conduct the more confident you become and the more you should see your project converge on its delivery date. As a rule of thumb, anything less than three passes of testing is inadequate.

## Regression vs. Retesting

You must retest fixes to ensure that issues have been resolved before development can progress. So, *retesting* is the act of repeating the test that found a defect to verify that it has been correctly fixed.

*Regression testing* on the other hand is the act of repeating other tests in 'parallel' areas to ensure that the applied fix or a change of code has not introduced other errors or unexpected behaviour.

For example, if an error is detected in a particular file handling routine then it might be corrected by a simple change of code. If that code, however, is utilised in a number of different places throughout the software, the effects of such a change could be difficult to anticipate. What appears to be a minor detail could affect a separate module of code elsewhere in the program.

A bug fix could in fact be introducing bugs elsewhere.

You would be surprised to learn how common this actually is. In empirical studies it has been estimated that up to 50% of bug fixes actually introduce additional errors in the code. Given this, it's a wonder that any software project makes its delivery on time.

Better QA processes will reduce this ratio but will never eliminate it. Programmers risk introducing casual errors every time they place their hands on the keyboard. An inadvertent slip of a key that replaces a full stop with a comma might not be detected for weeks but could have serious repercussions.

Regression testing attempts to mitigate this problem by assessing the ‘area of impact’ affected by a change or a bug fix to see if it has unintended consequences. It verifies known good behaviour after a change.

Regression testing is particularly suited to automated testing.

## White-Box vs Black-Box testing

Testing of completed units of functional code is known as *black-box testing* because the object is treated asa black-box: input goes in; output comes out. These tests concern themselves with verifying specified input against expected output and not worrying about the mechanics of what goes in between.

User Acceptance Testing (UAT) is the classic example of black-box testing.

*White-box or glass-box testing* relies on analysing the code itself and the internal logic of the software. White-box testing is often, but not always, the purview of programmers. It uses techniques which range from highly technical or technology specific testing through to things like code inspections or pairing.

*Unit testing* is a classic example of white-box testing. In unit testing developers use the programming language they are working in (or an add-on framework like xUnit) to write short contracts for each unit of code.

What constitutes a unit is the subject of some debate but unit tests are typically very small, independent tests than run quickly and assert the behaviour of a particular function or object. For example a function that adds numbers together might be tested by checking that 1 plus 1 returns 2.

While this might sound trivial, unit tests are useful for debugging code, particularly complex code that changes frequently. A developer may make a change that they they think does not affect the output of a function, but a well written unit test will prove that the functions contract has not been broken.

Unit tests also help design and refactor complex code as they force programmers to break down the logic of their application in to manageable chunks.

## Verification and Validation

Sometime individuals lose sight of the end goal. They narrow their focus to the immediate phase of software development and lose sight of the bigger picture.

*Verification tasks* are designed to ensure that the product is internally consistent. They ensure that the product meets the specification, the specification meets the requirements... and so on. The majority of testing tasks are verification – with the final product being checked against some kind of reference to ensure the expected outcome.

For example, test plans can be written from requirements documents specifications. This verifies that the software delivers the requirements or meets the specifications. This however does not however address the 'correctness' of those documents!

> On a large scale project I worked on as a test manager, we complained to the development team that our documentation was out of date and we were having difficulty constructing valid tests. They grumbled but eventually assured us they would update the specification and provide us with a new version to plan our tests from.

When I came in the next day, I found two programmers sitting at a pair of computer terminals. While one of them ran the latest version of the software, the other would look over their shoulder and then write up the behaviour of the product as the latest version of the specification!

When we complained to the development manager she said "What do you want? The spec is up to date now, isn't it?" The client, however, was not amused; they now had no way of determining *what the program was supposed to do* as opposed to *what it actually did*.

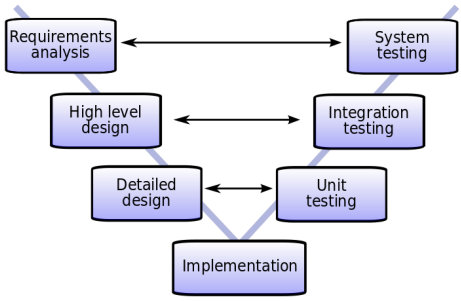
*Validation tasks* are just as important as verification, but less common.

Validation is the use of external sources of reference to ensure that the internal design is valid, i.e. it meets user's expectations. By using external references (such as the direct involvement of end-users) the test team can validate design decisions and ensure the project is heading in the correct direction (but some times all you have is validation, there is no 'internal' point of reference).

*Usability testing* is a prime example of a useful validation technique.

A note on the 'V-model'.

There exists a software testing model called the V-model, it illustrates different phases of testing in the SDLC, matching a phase of testing to a phase of waterfall development.




I don't like it at all.

It emphasises verification tasks over validation tasks.

Just like the waterfall model it relies on each phase being perfect and will ultimately only catch errors at the very end of the cycle. Errors will propagate from phase to phase, chewing up time and effort

But... the V-model does illustrate the importance of different levels of testing at different phases of the project.