A Software Testing Primer

An Introduction to Software Testing

by Nick Jenkins

http://www.nickjenkins.net

Table of Contents

The Need for Testing

My favourite quote on software, from Bruce Sterling's "The Hacker Crackdown" –

> The stuff we call "software" is not like anything that human society is used to thinking about. Software is something like a machine, and something like mathematics, and something like language, and something like thought, and art, and information.... but software is not in fact any of those other things. The protean quality of software is one of the great sources of its fascination. It also makes software very powerful, very subtle, very unpredictable, and very risky.
>
> Some software is bad and buggy. Some is "robust," even "bulletproof." The best software is that which has been tested by thousands of users under thousands of different conditions, over years. It is then known as "stable." This does NOT mean that the software is now flawless, free of bugs. It generally means that there are plenty of bugs in it, but the bugs are well-identified and fairly well understood.
>
> There is simply no way to assure that software is free of flaws. Though software is mathematical in nature, it cannot by "proven" like a mathematical theorem; software is more like language, with inherent ambiguities, with different definitions, different assumptions, different levels of meaning that can conflict.

Software development involves ambiguity, assumptions and flawed human communication.

Each change made to a piece of software, each new piece of functionality, each attempt to fix a defect, introduces the possibility of error. With each error, the risk that the software will not fulfil its intended purpose increases.

Testing reduces that risk.

We can use QA processes to attempt to prevent defects from entering software but the only thing we can do to reduce the number of errors already present is to test it. By following a cycle of testing and rectification we can identify issues and resolve them.

Testing also helps us quantify the risk in an untried piece of software.

After modifications have been made, a piece of software can be run in a controlled environment and its behaviour observed. This provides evidence which informs the decision to move to the next phase of the project or to attempt to rectify the problem.

And finally in some (dare I say enlightened?) software development efforts, testing can actually be used to drive development. By following statistical models of software development and methods such as usability testing, software development can move from an unfocused artistic endeavour to a structured discipline.

> This primer will be unashamedly pro-testing. I am, and have been for ten years, a tester. As a tester I have suffered endless indignities at the hands of project managers and development teams that resulted in enormous amounts of stress for myself and my teams.
>
> There are plenty of books written about and for project managers and software developers.
>
> This is probably not one of them.

Different Models of Software Development

The Waterfall Model

Making something can be thought of as a linear sequence of events. You start at A, you do B and then go to C and eventually end up at Z. This is extremely simplistic but it does allow you to visualise the series of events in the simplest way and it emphasises the importance of delivery with steps being taken towards a conclusion.

Below is the "Waterfall Model" which shows typical development tasks flowing into each other. Early in the history of software development it was adapted from engineering models to be a blueprint for software development.

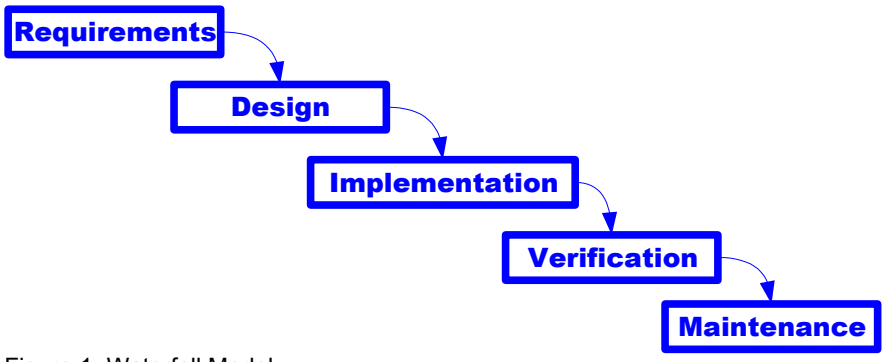


Figure 1: Waterfall Model

The five steps outlined are :

- *Analyse* the requirements of the project and decide what it is supposed to do
- *Design* a solution to meet these requirements
- *Implement* the design into a working product
- *Verify* the finished product against the design (and requirements)
- *Maintain* the project as necessary

The Waterfall Model was widely adopted in the early days of software development and a lot of blame has been laid at its door.

Critics argue that many problems in software development stem from this model. Early development projects that followed this model ran over budget and over schedule and the blame was attributed to the linear, step-wise nature of the model.

It is very rare that requirements analysis can be entirely completed before design and design before development and so on. It is far more likely that each phase will have interaction with each of the other phases.

In a small project this is not a problem since the span from "analyse" to "implement" may be a period of weeks or even days. For a large scale project which span months or even years the gap becomes significant. The more time that passes between analysis and implementation, the more a gap exists between the delivered project and the requirements of end-users.

Think about a finance system which is 'analysed' one year, designed the next year and developed and implemented the following year. That's three years between the point at which the requirements are captured and the system actually reaches its end users. In three years its likely that the business, if not the whole industry, will have moved considerably and the requirements will no longer be valid. The developers will be developing the wrong system! Software of this scale is not uncommon either.

A definition of requirements may be accurate at the time of capture but decays with frightening speed. In the modern business world, the chance of your requirements analysis being valid a couple of months after it has been conducted is very slim indeed.

Other versions of the waterfall model have been developed to alleviate this. One, the Iterative Waterfall Model, includes a loop as shown below.

This model attempts to overcome the limitations of the original model by adding an "iterative" loop to the end of the cycle. That is, in order to keep up with changing requirements the "analysis" phase is revisited at the end of the cycle and the process starts over again.

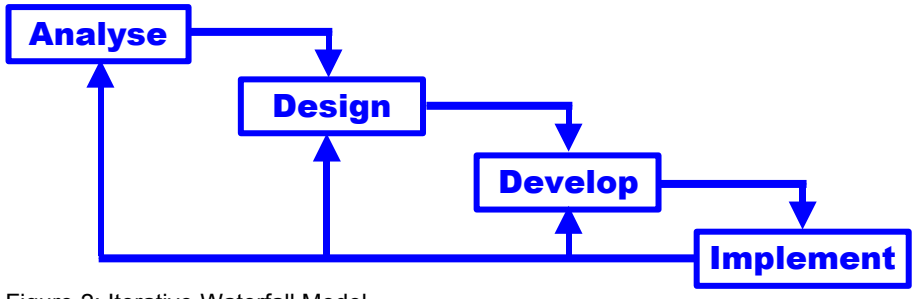


Figure 2: Iterative Waterfall Model

This alleviates the situation somewhat but still introduces a considerable lag between analysis and implementation. The waterfall model implies you have to complete all the steps before you start the process again. If requirements change during the life of the project the waterfall model requires the completion of a full cycle before they can be revisited.

Other Models of Software Development

In response to the waterfall model any number of new models have been developed and touted as alternatives. Some are interesting and many have promise but all have their pitfalls and none have delivered the quantum improvements they have promised.

Iterative or Rapid Development

In iterative development, the same waterfall process is used to deliver smaller chunks of functionality in a step-by-step manner. This reduces the management and overheads in delivering software and reduces the risk inherent in the project.

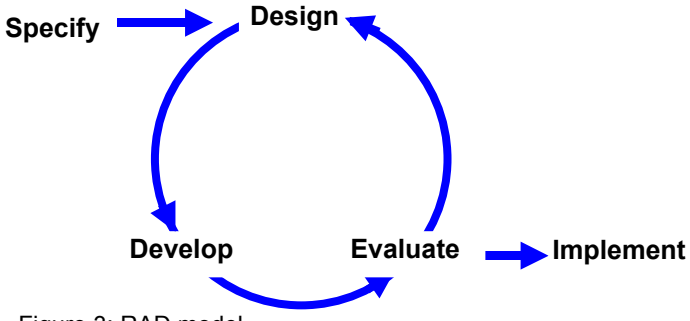


Figure 3: RAD model

One of the major reasons cited for software project failure (and a common sources of defects) is poor quality requirements. That is a failure to correctly specify *what* to build. By delivering small chunks and validating them, the project can self correct and hopefully converge on the desired outcome. This contrasts with the long lag times in the waterfall model.

A variation on this theme is "Rapid Applications Development" or RAD.

The phases are similar to waterfall but the 'chunks' are smaller. The emphasis in this model is on fast iterations through the cycle. Prototypes are designed, developed and evaluated with users, involving them in the process and correcting the design. The model is particularly suited to projects in rapidly changing environments where the team needs to adapt to different situations.

Incremental Development

Note that not all iterations need be complete, fully functional software. Nor should they necessarily focus on the same areas of functionality. It is better to deliver separate 'increments' of functionality and assemble the whole project only at the conclusion of the production phase. This way you can take partial steps towards your completed goal. Each unit can be individually developed, tested and then bolted together to form the overall product or system.

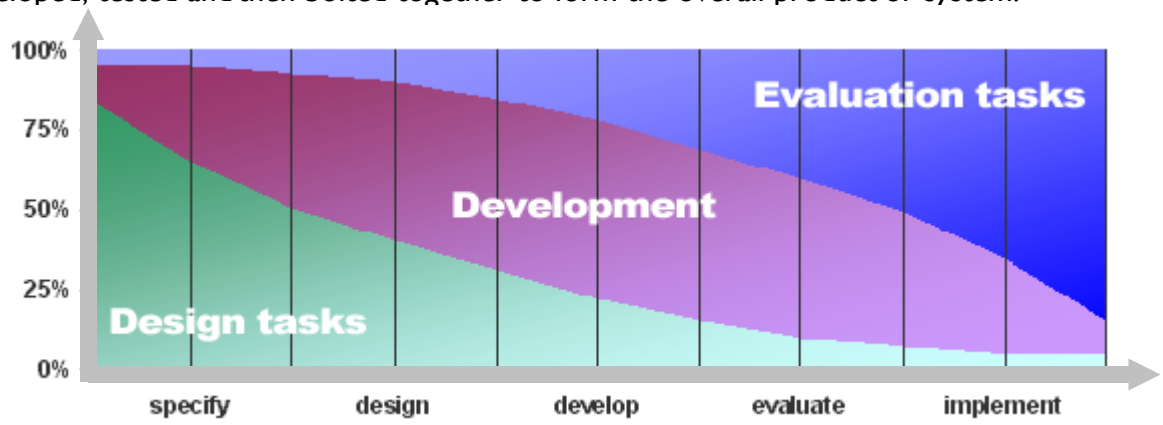


Figure 4: Incremental development

The diagram above indicates progress through the development life-cycle in an iterative / incremental development. Early on the iterations focus on 'design tasks' and the emphasis is on making design decisions and prototypes. As the project progresses tasks shift to development where the bulk of the coding is done. Finally, the emphasisis on testing or evaluation to ensure that what has been developed meets requirements and is solid, stable and bug free (ha!).

The New Software Development Methodologies

On the back of 'rapid' development methodologies have come a swag of 'process-lite' software development models that focus on harnessing the creativity of software developers. They include things like SCRUM, Extreme Programming and AUP.

The methods involved range from the harmless or practical to the outright bizarre.

The models have their advocates in the industry and some notable successes but they also have their crashing failures and the overall performance of these techniques can be best described as mediocre. A friend of mine summed it up best when he said, "They work really well if you have a good team, but then anything works really well if you have agood team."

This primer will focus on testing in traditional methodologies although some of the fundamental concepts mirror those from newer software development methodologies.

> I have to agree with Fred Books in his seminal paper, "No Silver Bullet", when he said "*There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.*"
>
> He was reacting to cries for a change in software development practices which would bring about order of magnitude improvements in the productivity of software development (which would match the order of magnitude increases being made in computer hardware).
>
> Brooks argued that while innovation was useful, no single process would revolutionise software development. What was needed instead was a disciplined, ordered approach that delivered step-wise improvements to the process.
>
> That was written in 1986 – and we're still looking for the holy grail.

Testing in the Software Development Life Cycle

The purpose of testing is to reduce risk.

The unknown factors within the development and design of new software can derail a project and minor risks can delay it. By using a cycle of testing and resolution you can identify the level of risk, make informed decisions and ultimately reduce uncertainty and eliminate errors.

Testing is the only tool in a development's arsenal which reduces defects. Planning, design and QA can reduce the number of defects which enter a product, but they can't eliminate any that are already there. And any kind of coding will introduce more errors since it involves changing something from a known good state to an unknown, unproved state.

Ideally testing should take place throughout the development life cycle. More often than not (as in the waterfall model) it is simply tacked onto the back end. If the purpose of testing is to reduce risk, this means piling up risk throughout the project to resolve at the end – in itself, a risky tactic.

It could be that this is a valid approach. By allowing developers to focus on building software components and then, at a later date, switching to rectifying issues it allows them to compartmentalise their effort and concentrate on one type of task at a time.

But as the lag between development and resolution increases so does the complexity of resolving the issues (see "Test Early, Test Often" in the next chapter). On any reasonably large software development project this lag is far too long. Better to spread different phases of testing throughout the life cycle, to catch errors as early as possible.

Traceability

Another function of testing is (bizarrely) to confirm what has been delivered.

Given a reasonably complex project with hundreds or perhaps thousands of stake-holder requirements, how do you know that you have implemented them all? How do your prove during testing or launch that a particular requirement has been satisfied? How do you track the progress of delivery on a particular requirement during development?

This is the problem of traceability.

How does a requirement map to an element of design (in the technical specification for example) and how does that map to an item of code which implements it and how does that map test to prove it has been implemented correctly ?

On a simple project it is enough to build a table which maps this out. On a large-scale project the sheer number of requirements overwhelm this kind of traceability. It is also possible that a single requirement may be fulfilled by multiple elements in the design or that a single element in the design satisfies multiple requirements. This make tracking by reference number difficult.

If you need a better solution I would suggest an integrated system to track requirements for you. There are off-the-shelf tools available which use databases to track requirements. These are then linked to similar specification and testing tools to provide traceability. A system such as this can automatically produce reports which highlight undelivered or untested requirements. Such systems can also be associated with SCM (Software Configuration Management) systems and can be very expensive, depending on the level of functionality.

See the section on "change management" for a discussion of these systems.

The Testing Mindset

There is particular philosophy that accompanies "good testing".

A professional tester approaches a product with the attitude that the product is already broken - it has defects and it is their job to discover them. They assume the product or system is inherently flawed and it is their job to 'illuminate' the flaws.

This approach is necessary in testing.

Designers and developers approach software with an optimism based on the assumption that the changes they make are the correct solution to a particular problem. But they are just that – assumptions.

Without being proved they are no more correct than guesses. Developers often overlook fundamental ambiguities in requirements in order to complete the project; 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, the tester offers a balance.

The tester takes nothing at face value. The tester always asks the question "why?" They seek to drive out certainty where there is none. They seek to illuminate the darker part of the projects with the light of inquiry.

Sometimes this attitude can bring conflict with developers and designers. But developers and designers can be testers too! If they can adopt this mindset for a certain portion of the project, they can offer that critical appraisal that is so vital to a balanced project. Recognising the need for this mindset is the first step towards a successful test approach.

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, actually quotes ratios of 1:6:10:1000 for the costs of fixing bugs in requirements,design, coding and implementation phases.

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

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

Nor is a single pass of testing enough.

Your first past at testing simply identifies where the defects are. At the very least, 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 in any phase 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 a test to verify that a found defect 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, isutilised 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.

> It is quite common for regression testing to cover ALL of the product or software under test.
>
> Why? Because programmers are notoriously bad at being able to track and control change in their software. When they fix a problem they will cause other problems. They generally have no idea of the impact a change will make, nor can they reliably back-out those changes.
>
> If developers could, with any certainty, specify the exact scope and effects of a change they made then testing could be confined to the area affected. Woe betide the tester that makes such an assumption however!

White-Box vs Black-Box testing

Testing of completed units of functional code is known as *black-box testing* because testers treat the object as a black-box. They concern themselves with verifying specified input against expected output and not worrying about the logic of what goes on in between.

User Acceptance Testing (UAT) and Systems Testingare 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. Ituses techniques which range from highly technical or technology specific testing through to things like code inspections.

Although white-box techniques can be used at any stage in a software product's life cycle they tend to be found in Unit testing activities.

Verification and Validation

Testers often make the mistake ofnot keeping their eye on 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 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 output is as expected.

For example, test plans are normally written from the requirements documents and from the specification. This verifies that the software delivers the requirements as laid out in the technical and requirement specifications.

It does not however address the 'correctness' of those requirements!

> 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 onscreen 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 users 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. *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, but I won't reproduce it here since I think it is a terrible model. It illustrates different phases of testing in the SDLC, matching a phase of testing to a phase of requirements/design.
>
> I don't like it because 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. And just like the waterfall model there is an updated version which attempts to rectify this but only serves to muddy the waters.
>
> But the V-model does illustrate the importance of different levels of testing at different phases of the project. I leave it as an exercise for the reader to uncover it.

If the aim of a software development project is to "deliver widget X to do task Y" then the aim of "functional testing" is to prove that widget X actually does task Y.

Simple ? Well, not really.

We are trapped by the same ambiguities that lead developers into error. Suppose the requirements specification says widget "X must do Y" but widget X actually does Y+Z? How do we evaluate Z? Is it necessary? Is it desirable? Does it have any other consequences the developer or the original stake-holder has not considered? Furthermore how well does Y match the Y that was specified by the original stake-holder?

> Here you can being to see the importance of specifying requirements accurately. If you can't specify them accurately then how can you expect someone to deliver them accurately or for that matter test them accurately?
>
> This sounds like common sense but it is much, much harder than anything else in the software development life cycle. See my Primer on Project Management for a discussion.

Alpha and Beta Testing

There are some commonly recognised milestones in the testing life cycle.

Typically these milestones are known as "alpha" and "beta". There is no precise definition for what constitutes alpha and beta test but the following are offered as common examples of what is meant by these terms :

> *Alpha* – enough functionality has been reasonably completed to enable the first round of (end-to-end) system testing to commence. At this point the interface might not be complete and the system may have many bugs.
>
> *Beta* – the bulk of functionality and the interface has been completed and remaining work is aimed at improving performance, eliminating defects and completing cosmetic work. At this point many defects still remain but they are generally well understood.

Beta testing is often associated with the first end-user tests.

The product is sent out to prospective customers who have registered their interest in participating in trials of the software. Beta testing, however, needs to be well organised and controlled otherwise feedback will be fragmentary and inconclusive. Care must also be taken to ensure that a properly prepared prototype is delivered to end-users, otherwise they will be disappointed and time will be wasted.

White Box testing

White-box or glass-box testing relies on analysing the code itself and the internal logic of the software and is usually, but not exclusively, a development task.

Static Analysis and Code Inspection

Static analysis techniques revolve around looking at the source code, or uncompiled form of software. They rely on examining the basic instruction set in its raw form, rather than as it runs. They are intended to trap semantic and logical errors.

Code inspection is a specific type of static analysis. It uses formal or informal reviews to examine the logic and structure of software source code and compare it with accepted best practices.

In large organisations or on mission-critical applications, a formal inspection board can be established to make sure that written software meets the minimum required standards. In less formal inspections a development manager can perform this task or even a peer.

Code inspection can also be automated. Many syntax and style checkers exist today which verify that a module of code meets certain pre-defined standards. By running an automated checker across code it is easy to check basic conformance to standards and highlight areas that need human attention.

A variant on code inspection is the use of peer programming as espoused in methodologies like Extreme Programming (XP). In XP's peer programming, modules of code are shared between two individuals. While one person writes a section of code the other is reviews and evaluates the quality of the code. The reviewer looks for flaws in logic, lapses of coding standards and bad practice. The roles are then swapped. Advocates assert this is a speedy way to achieve good quality code and critics retort that its a good way to waste a lot of people's time.

As far as I'm concerned the jury is still out.

Dynamic Analysis

While static analysis looks at source code in its raw format, dynamic analysis looks at the compiled/interpreted code *while it is running* in the appropriate environment. Normally this is an analysis of variable quantities such as memory usage, processor usage or overall performance.

One common form of dynamic analysis used is that of memory analysis. Given that memory and pointer errors form the bulk of defects encountered in software programs, memory analysis is extremely useful. A typical memory analyser reports on the current memory usage level of a program under test and of the disposition of that memory. The programmer can then 'tweak' or optimise the memory usage of the software to ensure the best performance and the most robust memory handling.

Often this is done by 'instrumenting' the code. A copy of the source code is passed to the dynamic analysis tool which inserts function calls to its external code libraries. These calls then export run time data on the source program to an analysis tool. The analysis tool can then profile the program while it is running. Often these tools are used in conjunction with other automated tools to simulate realistic conditions for the program under test. By ramping up loading on the program or by running typical input data, the program's use of memory and other resources can be accurately profiled under real-world conditions.

Unit, Integration and System testing

The first type of testing that can be conducted in any development phase is **unit testing**.

In this, discrete components of the final product are tested independently before being assembled into larger units. Units are typically tested through the use of 'test harnesses' which simulate the context into which the unit will be integrated. The test harness provides a number of known inputs and measures the outputs of the unit under test, which are then compared with expected values to determine if any issues exist.

In **integration testing** smaller units are integrated into larger units and larger units into the overall system. This differs from unit testing in that units are no longer tested independently but in groups, the focus shifting from the individual units to the interaction between them.

At this point "stubs" and "drivers" take over from test harnesses.

A stub is a simulation of a particular sub-unit which can be used to simulate that unit in a larger assembly. For example if units A, B and C constitute the major parts of unit D then the overall assembly could be tested by assembling units A and B and a simulation of C, if C were not complete. Similarly if unit D itself was not complete it could be represented by a "driver" or a simulation of the super-unit.

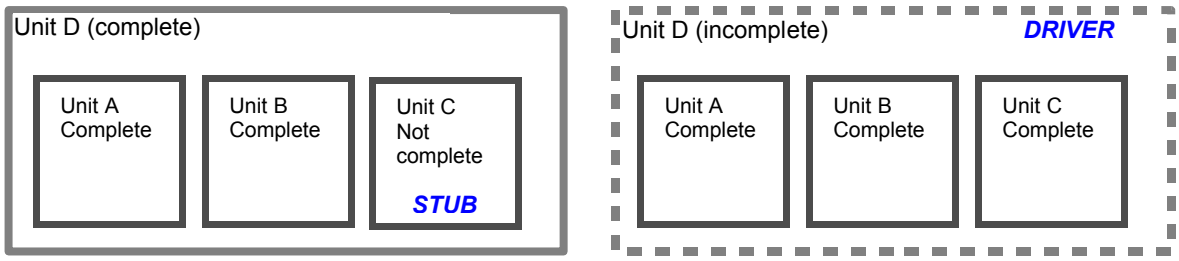


Figure 5: Integration Testing

As successive areas of functionality are completed they can be evaluated and integrated into the overall project. Without integration testing you are limited to testing a completely assembled product or system which is inefficient and error prone. Much better to test the building blocks as you go and build your project from the ground up in a series of controlled steps.

System testing represents the overall test on an assembled software product. Systems testing is particularly important because it is only at this stage that the full complexity of the product is present. The focus in systems testing is typically to ensure that the product responds correctly to all possible input conditions and (importantly) the product handles exceptions in a controlled and acceptable fashion. System testing is often the most formal stage of testing and more structured.

The SIT or Test Team

In large organisations it is common to find a "SIT" or independent test team. SIT usually stands for "Systems Integration Testing" or "Systems Implementation Testing" or possibly "Save It, Testing!"

And is the role of this team unit, system testing or integration testing?

Well, nobody really knows. The role of the SIT team usually is not unit, integration nor system testing but a combination of all three. They are expected to get involved in unit testing with developers, to carry through the integration of units into larger components and then to provide end-to-end testing of the systems.

Sometimes the expectation is that the SIT team will become the companies Quality Assurance team, even though they have no direct influence on the way software is developed. The assumption is that by increasing the length and rigour of testing it will improve the quality of *released* products – and so it does.

But it does nothing to increase the quality of *built* products – so it's not really QA.

In the best of worlds, this team can act as an agent of change. It can introduce measures and processes which prevent defects from being in written into the code in the first place; they can work with development teams to identify areas which need fixing; and they can highlight successful improvements to development processes.

In the worst of worlds the pressure on software development drives longer and longer projects with extended test cycles where huge amounts of defects are found and project schedules slip. The testing team attracts blame for finding defects and for long testing cycles and nobody knows how to solve the problem.

Acceptance Testing

Large scale software projects often have a final phase of testing called "Acceptance Testing".

Acceptance testing forms an important and distinctly separate phase from previous testing efforts and its purpose is to ensure that the product meets minimum defined standards of quality prior to it being accept by the client or customer.

This is where someone has to sign the cheque.

Often the client will have his end-users to conduct the testing to verify the software has been implemented to their satisfaction (this is called "User Acceptance Testing" or "UAT"). Often UAT tests processes outside of the software itself to make sure the whole solution works as advertised.

While other forms of testing can be more 'free form', the acceptance test phase should represent a planned series of tests and release procedures to ensure the output from the production phase reaches the end-user in an optimal state, as free of defects as is humanly possible.

In theory Acceptance Testing should also be fast and relatively painless. Previous phases of testing will have eliminated any issues and this should be a formality. In immature software development, the Acceptance Test becomes a last trap for issues, back-loading the project with risk.

Acceptance testing also typically focusses on artefacts outside the software itself. A solution often has many elements outside of the software itself. These might include : manuals and documentation; process changes; training material; operational procedures; operational performance measures (SLA's).

These are typically not tested in previous phases of testing which focus on functional aspects of the software itself. But the correct delivery of these other elements is important for the success of the solution as a whole. They are typically not evaluated until the software is complete because they require a fully functional piece of software, with its new workflows and new data requirements, to evaluate.

Test Automation

Organisations often seek to reduce the cost of testing. Most organisations aren't comfortable with reducing the amount of testing so instead they look at improving the efficiency of testing. Luckily, there are a number of software vendors who claim to be able to do just this! They offer automated tools which take a test case, automate it and run it against a software target repeatedly. Music to management ears!

However, there are some myths about automated test toolsthat need to be dispelled :

- *Automated testing does not find more bugs than manual testing* – an experienced manual tester who is familiar with the system will find more new defects than a suite of automated tests.
- *Automation does not fix the development process* – as harsh as it sounds, testers don't create defects, developers do. Automated testing does not improve the development process although it might highlight some of the issues.
- *Automated testing is not necessarily faster* – the upfront effort of automating a test is much higher than conducting a manual test, so it will take longer and cost more to test the first time around. Automation only pays off over time. It will also cost more to maintain.
- *Everything does not need to be automated* – some things don't lend themselves to automation, some systems change too fast for automation, some tests benefit from partial automation – you need to be selective about what you automate to reap the benefits.

But, in their place, automated test tools can be extremely successful.

> The funniest business case I have ever seen for automated test tools ran like this :
>
> – Using manual testing, we find X number of defects in our software
> – It costs $Y to find and fix these defects each year (developer + tester time)
> – We can buy an automated test tool for $Z/year
> – Since $Z < $Y we should buy the tool and, as a bonus, *it will find < X defects*
>
> So, not only are you comparing the one off cost for buying tools (without set-up or maintenance) with the cost of manual testing, but the tool will do away with the manual testers as well – the ones who find all those pesky defects!

The Hidden Cost

The hidden costs of test automation are in its maintenance.

An automated test asset which can be written once and run many times pays for itself much quicker than one which has to be continually rewritten to keep pace with the software.

And there's the rub.

Automation tools, like any other piece of software, talk to the software-under-test through an interface. If the interface is changing all the time then, no matter what the vendors say, your tests will have to change as well. On the other hand, if the interfaces remain constant but the underlying functional code changes then your tests will still run and (hopefully) still find bugs.

Many software projects don't have stable interfaces. The user-facing interface (or GUI) is in fact the area which has most change because its the bit the users see most. Trying toautomate the testing for a piece of software which has a rapidly changing interface is a bit like trying to pin a jellyfish to the wall.

What is Automated Testing Good For?

Automated testing is particularly good at :

- Load and performance testing – automated tests are a prerequisite of conducting load and performance testing. It is not feasible to have 300 users manually test a system simultaneously, it must be automated.
- Smoke testing – a quick and dirty test to confirm that the system 'basically' works. A system which fails a smoke test is automatically sent back to the previous stage before work is conducted, saving time and effort
- Regression testing – testing functionality that *should not have changed* in a current release of code. Existing automated tests can be run and they will highlight changes in the functionality they have been designed to test (in incremental development builds can be quickly tested and reworked if they have altered functionality delivered in previous increments)
- Setting up test data or pre-test conditions – an automated test can be used to set-up test data or test conditions which would otherwise be time consuming
- Repetitive testing which includes manual tasks that are tedious and prone to human error (e.g. checking account balances to 7 decimal places)

Pitfalls of Test Automation

Automating your testing should be treated like a software project in its own right.

It must clearly define its requirements. It must specify what is to be automated and what isn't. It must design a solution for testing and that solution must be validated against an external reference.

Consider the situation where a piece of software is written from an incorrect functional spec. Then a tester takes the same functional spec. and writes an automated test from it.

Will the code pass the test?

Of course. Every single time.

Will the software deliver the result the customer wants? Is the test a valid one?

Nope.

Of course, manual testing could fall into the same trap. But manual testing involves human testers who ask impertinent questions and provoke discussion. Automated tests only do what they're told. If you tell them to do something the wrong way, they will and they won't ask questions.

Further, automated tests should be designed with maintainability in mind. They should be built from modular units and should be designed to be flexible and parameter driven (no hard-coded constants). They should follow rigourous coding standards and there should be a review process to ensure the standards are implemented.

Failure to do this will result in the generation of a code base that is difficult to maintain, incorrect in its assumptions and that will decay faster than the code it is supposed to be testing.

Testing the design

Requirements, design and technical specifications can be tested in their own right.

The purpose of evaluating a specification is threefold:

- To make sure it is accurate, clear and internally consistent (verification)
- Evaluating how well it reflects reality and what the end-user expects (validation)
- Making sure it is consistent with all previous and subsequent phases of the project

The technical specification is an embodiment of the requirements which should then flow through to subsequent phases like production and testing. If the requirements are poorly specified then not only will the product be inadequate but will also be incredibly difficult to verify.

If the technical specification is out of synch with the requirements then it is likely that the development team will be well on its way to producing the wrong product. Because these documents are often produce in parallel (i.e. the steps in the waterfall model overlap) it is very common for discrepancies to creep in.

Each assertion in a specification should be reviewed against a list of desirable attributes:

- *Specific* – it is critical to eliminate as much uncertainty as possible, as early as possible. Words like "probably", "maybe" or"might" indicate indecision on the part of the author and hence ambiguity. Requirements including these words should be either eliminated or re-written to provide some kind of surety as to the desired outcome.
- *Measurable* – a requirement which uses comparative words like “better” or “faster” must specify a quantitative or qualitative improvement must do so with a specific value (100% faster or 20% more accurate).
- *Testable* – in line with the above, a requirement should be specified with some idea of how it should be evaluated. A requirement which is not testable is ultimately not ‘provable’ and cannot therefore be confirmed either positively or negatively.
- *Consistent*- if one requirement contradicts another, the contradiction must be resolved. Often splitting a requirement into component parts helps uncover inconsistent assumptions in each, which can then be clarified.
- *Clear* - requirements should be simple, clear and concise. Requirements composed of long-winded sentences or of sentences with multiple clauses imply multiple possible outcomes and so lack clarity. Split them up into single statements.
- Exclusive – specs should not only state what will be done, but explicitly what will *not* be done. Leaving something unspecified leads to assumptions.

Further, it is important to differentiate requirements from design documents. Requirements should not talk about “how” to do something and a design specs should not talk about “why” to do things.

Usability Testing

Usability testing is the process of observing users' reactions to a product and adjusting the design to suit their needs. Marketing knows usability testing as "focus groups" and while the two differ in intent many of the principles and processes are the same.

In usability testing a basic model or prototype of the product is put in front of evaluators who are representative of typical end-users. They are then set a number of standard tasks which they must complete using the product. Any difficulty or obstructions they encounter are then noted by a host or observers and design changes are made to the product to correct these. The process is then repeated with the new design to evaluate those changes.

There are some fairly important tenets of usability testing that must be understood :

- *Users are not testers, engineers or designers* – you are not asking the users to make design decisions about the software. Users will not have a sufficiently broad technical knowledge to make decisions which are right for everyone. However, by seeking their opinion the development team can select the best of several solutions.
- *You are testing the product and not the users* – all too often developers believe that it's a 'user' problem when there is trouble with an interface or design element. Users should be able to 'learn' how to use the software if they are taught properly! Maybe if the software is designed properly, they won't have to learn it at all ?
- *Selection of end-user evaluators is critical* –You must select evaluators who are directly representative of your end-users. Don't pick just anyone off the street, don't use management and don't use technical people unless they are your target audience.
- *Usability testing is a design tool* – Usability testing should be conducted early in the life-cycle when it is easy to implement changes that are suggested by the testing. Leaving it till later will mean changes will be difficult to implement.

One misconception about usability studies is that a large number of evaluators is required to undertake a study. Research has shown that no more than four or five evaluators might be required. Beyond that number the amount of new information discovered diminishes rapidly and each extra evaluator offers little or nothing new.

And five is often convincing enough.

If all five evaluators have the same problem with the software, is it likely the problem lies with them or with the software ? With one or two evaluators it could be put down to personal quirks. With five it is beyond a shadow of a doubt.

The proper way to select evaluators is to profile a typical end-user and then solicit the services of individuals who closely fit that profile. A profile should consist of factors such as age, experience, gender, education, prior training and technical expertise.

> I love watching developers who take part as observers in usability studies. As a former developer myself I know the hubris that goes along with designing software. In the throes of creation it is difficult for you to conceive that someone else, let alone a user (!), could offer better input to the design process than your highly paid, highly educated self.
>
> Typically developers sit through the performance of the first evaluator and quietly snigger to themselves, attributing the issues to 'finger trouble' or user ineptitude. After the second evaluator finds the same problems the comments become less frequent and when the third user stumbles in the same position they go quiet.
>
> By the fourth user they've got a very worried look on their faces and during the fifth pass they're scratching at the glass trying to get into to talk to the user to "find out how to fix the problem".

Other issues that must be considered when conducting a usability study include ethical considerations. Since your are dealing with human subjects in what is essentially a scientific study you need to consider carefully how they are treated. The host must take pains to put them at ease, both to help them remain objective and to eliminate any stress the artificial environment of a usability study might create. You might not realise how traumatic using your software can be for the average user!

Separating them from the observers is a good idea too since no one performs well with a crowd looking over their shoulder. This can be done with a one-way mirror or by putting the users in another room at the end of a video monitor. You should also consider their legal rights and make sure you have their permission to use any materials gathered during the study in further presentations or reports. Finally, confidentiality is usual important in these situations and it is common to ask individuals to sign a Non-Disclosure-Agreement (NDA).

Performance Testing

One important aspect of modern software systems is their performance in multi-user or multi-tier environments. To test the performance of the software you need to simulate its deployment environment and simulate the traffic that it will receive when it is in use – this can be difficult.

The most obvious way of accurately simulating the deployment environment is to simply use the live environment to test the system. This can be costly and potentially risky but it provides the best possible confidence in the system. It may be impossible in situations where the deployment system is constantly in use or is mission critical to other business applications.

If possible however, live system testing provides a level of confidence not possible in other approaches. Testing on the live system takes into account all of the idiosyncrasies of such a system without the need to attempt to replicate them on a test system.

Also common is the use of capture-and-playback tools (automated testing). A capture tool is used to record the actions of a typical user performing a typical task on the system. A playback tool is then used to reproduce the action of that user multiple times simultaneously. The multi-user playback provides an accurate simulation of the stress the real-world system will be placed under.

The use of capture and playback tools must be used with caution, however. Simply repeating the exact same series of actions on the system may not constitute a proper test. Significant amounts of randomisation and variation should be introduced to correctly simulate real-world use.

You also need to understand the technical architecture of the system. If you don't stress the weak points, the bottlenecks in performance, then your tests will prove nothing. You need to design targeted tests which find the performance issues.

Having a baseline is also important.

Without knowledge of the 'pre-change' performance of the software it is impossible to assess the impact of any changes on performance. “The system can only handle 100 transactions an hour!” comes the cry. But if it only needs to handle 50 transactions an hour, is this actually an issue?

Performance testing is a tricky, technical business. The issues that cause performance bottlenecks are often obscure and buried in the code of a database or network. Digging them out requires concerted effort and targeted, disciplined analysis of the software.

The Purpose of Test Planning

As part of a project, testing must be planned to ensure it delivers on its expected outcomes. It must be completed within a reasonable time and budget.

But test planning represents a special challenge.

The aim of test planning is to decide where the bugs in a product or system will be and then to design tests to locate them. The paradox is of course, that if we knew where the bugs were then we could fix them without having to test for them.

Testing is the art of uncovering the unknown and therefore can be difficult to plan.

The usual, naïve retort is that you should simply test "all" of the product. Even the simplest program however will defy all efforts to achieve 100% coverage (see the appendix).

Even the term coverage itself is misleading since this represents a plethora of possibilities. Do you mean code coverage, branch coverage, or input/output coverage ? Each one is different and each one has different implications for the development and testing effort. The ultimate truth is that complete coverage, of any sort, is simply not possible (nor desirable).

So how do you plan your testing?

At the start of testing there will be a (relatively) large number of issues and these can be uncovered with little effort. As testing progress more and more effort is required to uncover subsequent issues.

The law of diminishing returns applies and at some point the investment to uncover that last 1% of issues is outweighed by the high cost of finding them. The cost of letting the customer or client find them will actually be less than the cost of finding them in testing.

The purpose of test planning therefore is to put together a plan which will deliver the right tests, in the right order, to discover as many of the issues with the software as time and budget allow.

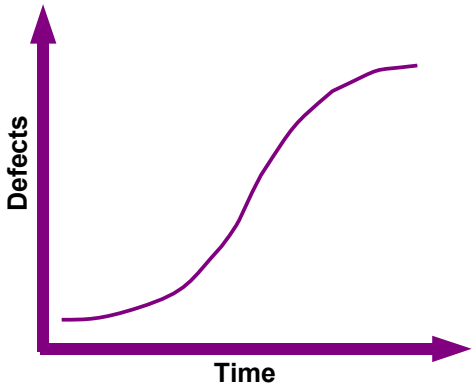


Figure 6: Typical defect discovery rate

Risk Based Testing

Risk is based on two factors – the likelihood of the problem occurring and the impact of the problem when it does occur. For example, if a particular piece of code is complex then it will introduce far more errors than a simple module of code. Or a particular module of code could be critical to the success of the project overall. Without it functioning perfectly, the product simply will not deliver its intended result.

Both of these areas should receive more attention and more testing than less 'risky' areas.

But how to identify those areas of risk?

Software in Many Dimensions

It is useful to think of software as a multi-dimensional entity with many different axes. For example one axis is the code of the program, which will be broken down into modules and units. Another axis will be the input data and all the possible combinations. Still a third axis might be the hardware that the system can run on, or the other software the system will interface with.

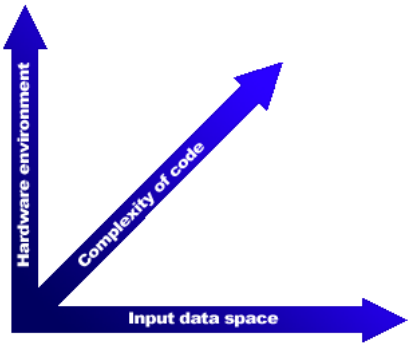


Figure 7 : Software dimensions

Testing can then be seen as an attempt to achieve "coverage" of as many of these axes as possible .

Remember we are no longer seeking the impossible 100% coverage but merely 'best' coverage, verifying the function of all the areas of risk.

Outlining

To start the process of test planning a simple process of 'outlining' can be used :

1. List all of the 'axes' or areas of the software on a piece of paper (a list of possible areas can be found below, but there are certainly more than this).
2. Take each axis and break it down into its component elements.

   For example, with the axis of "code complexity" you would break the program down into the 'physical' component parts of code that make it up. Taking the axis of "hardware environment" (or platform) it would be broken down into all possible hardware and software combinations that the product will be expected to run on.
3. Repeat the process until you are confident you have covered as much of each axis as you possibly can (you can always add more later).

This is a process of deconstructing the software into constituent parts based on different taxonomies. For each axis simply list all of the possible combinations you can think of. Your testing will seek to cover as many elements as possible on as many axes as possible.

The most common starting point for test planning is functional decomposition based on a technical specification. This is an excellent starting point but should not be the sole 'axis' which is addressed – otherwise testing will be limited to 'verification' but not 'validation'.

Axis / Category	Explanation
Functionality	As derived from the technical specification
Code Structure	The organisation and break down of the source or object code
User interface elements	User interface controls and elements of the program
Internal interfaces	Interface between modules of code (traditionally high risk)
External interfaces	Interfaces between this program and other programs
Input space	All possible inputs
Output space	All possible outputs
Physical components	Physical organisation of software (media, manuals, etc.)
Data	Data storage and elements
Platform and environment	Operating system, hardware platform
Configuration elements	Any modifiable configuration elements and their values
Use case scenarios	Each use case scenario should be considered as an element

Test Case Identification

The next step is to identify test cases which 'exercise' each of the elements in your outline.

This isn't a one-to-one relationship. Many tests might be required to validate a single element of your outline and a single test may validate more than one point on one axis. For example, a single test could simultaneously validate functionality, code structure, a UI element and error handling.

In the diagram shown here, two tests have been highlighted in red : "A" and "B".

Each represents a single test which has verified the behaviour of the software on the two particular axes of "input data space" and "hardware environment".

For each point in each axis of your outline decide upon a test which will exercise that functionality. Note if that test will validate a different point on a different axis and continue until you are satisfied you have all points on all axes covered.

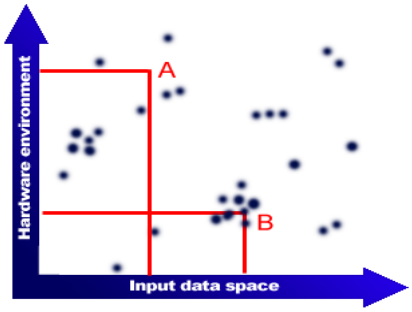


Figure 8 : Test case identification

For now, don't worry too much about the detail of how you might test each point, simply decide 'what' you will test. At the end of this exercise you should have a list of test cases which represents your near-to-perfect, almost 100% coverage of the product.

Test Case Selection

Given that we acknowledge we can't achieve a 100% coverage, we must now take a critical eye to our list of test cases. We must decide which ones are more important, which ones will exercise the areas of risk and which ones will discover the most bugs.

But how?

Have another look at the diagram above – we have two of our axes represented here, "input data space" and "hardware environment". We have two test "A" and "B" and we have dark splotches, which denote bugs in the software.

Bugs tend to cluster around one or more areas within one or more axes. These define the areas of risk in the product. Perhaps this section of the code was completed in a hurry or perhaps this section of the 'input data space' was particularly difficult to deal with.

Whatever the reason, these areas are inherently more risky, more likely to fail than others.

You can also note that test A has not uncovered a bug it has simply verified the behaviour of the program at that point. Test B on the other hand has identified a single bug in a cluster of similar bugs. Focussing testing efforts around area B will be more productive since there are more more bugs to be uncovered here. Focussing around A will probably not produce any significant results.

When selecting test cases you must try to achieve a balance.

Your aim should be to provide a broad coverage for the majority of your outline 'axes' and deep coverage for the most risky areas discovered. Broad coverage implies that an element in the outline is evaluated in a elementary fashion while deep coverage implies a number of repetitive, overlapping test cases which exercise every variation in the element under test.

The aim of broad coverage is to identify risky areas and focus the deeper coverage of those areas to eliminate the bulk of issues. It is a tricky balancing act between trying to cover everything and focusing your efforts on the areas that require most attention.

Test Estimation

Once you have prioritised the test cases you can estimate how long each case will take to execute.

Take each test case and make a rough estimate of how long you think it will take to set up the appropriate input conditions, execute the software and check the output. No two test cases are alike but you can approximate this step by assigning and average execution time to each case and multiplying by the number of cases.

Total up the hours involved and you have an estimate of testing for that piece of software.

You can then negotiate with the Project Manager or your product manager for the appropriate budget to execute the testing. The final cost you arrive at will depend on the answers to a number of questions, including: how deep are your organisation's pockets? how mission critical is the system? how important is quality to the company? how reliable is the development process?

The number will almost certainly be lower than you hoped for.

> In general – more iterations are better than more tests. Why? Developers don't often fix a bug at the first attempt. Bugs tend to cluster and finding one may uncover more. If you have a lot of bugs to fix you will have to have multiple iterations to retest the fixes. And finally, and honestly, in a mature software development effort, many tests will uncover no bugs!

Refining the plan

As you progress through each cycle of testing you can refine your test plan.

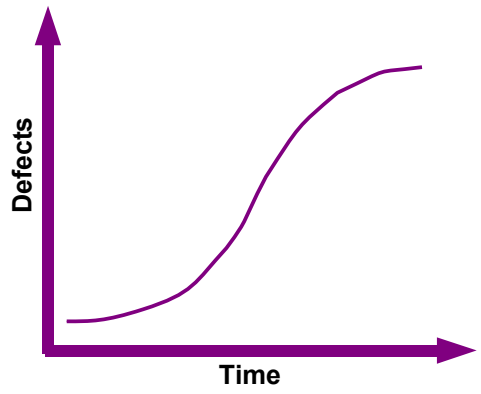


Figure 9: Defect discovery rate again

Typically, during the early stages of testing not many defects are found. Then, as testing hits it stride, defects start coming faster and faster until the development team gets on top of the problem and the curve begins to flatten out. As development moves ahead and as testing moves to *retesting* fixed defects, the number of new defects will decrease.

This is the point where your risk/reward ratio begins to bottom out and it may be that you have reached the limits of effectiveness with this particular form of testing. If you have more testing planned or more time available, now is the time to switch the focus of testing to a different point in your outline strategy.

Cem Kaner said it best, "The best test cases are the ones that find bugs." A test case which finds no issues is not worthless but it is obviously worth less than a test case which does find issues.

If your testing is not finding any bugs then perhaps you should be looking somewhere else. Conversely, if your testing is finding a lot of issues you should pay more attention to it – but not to the exclusion of everything else, there's no point in fixing just one area of the software!

Your cycle of refinement should be geared towards discarding pointless or inefficient tests and diverting attention to more fertile areas for evaluation.

Also, referring each time to your original outline will help you avoid losing sight of the wood for the trees. While finding issues is important you can never be sure where you'll find them so you can't assume the issues that you are finding are the only ones that exist. You must keep a continuous level of *broad coverage* testing active to give you an overview of the software while you focus the *deep coverage* testing on the trouble spots.

Summary

1. Decompose your software into a number of 'axes' representing different aspects of the system under test
2. Further decompose each axis of the software into sub-units or 'elements'
3. For each element in each axis, decide how you are going to test it
4. Prioritise your tests based on your best available knowledge
5. Estimate the effort required for each test and draw a line through your list of tests where you think it appropriate (based on your schedule and budget)
6. When you execute your test cases, refine your plan based on the results. Focus on the areas which show the most defects, while maintaining broad coverage of other areas.

Your intuition may be your best friend here!

Risky code can often be identified through 'symptoms' like unnecessary complexity, historical occurrences of defects, stress under load and reuse of code.

Use whatever historical data you have, the opinions of subject matter experts and end users and good old common sense. If people are nervous about a particular piece of code, you should be too. If people are being evasive about a particular function, test it twice. If people dismiss what you think is a valid concern, pursue it until you're clear.

And finally, *ask the developers*.

They often know exactly where the bugs lurk in their code.

Test Scripting

There are several schools of thought to test scripting.

In risk averse industries such as defence and finance there is a tendency to emphasise scripting tests before they are executed. These industries are more concerned with the potential loss from a software defect than the potential gain from introducing a new piece of software. As a consequence there is a heavy emphasis on verifiable test preparation (although observance of this verification might only be lip-service!) And in some industries, external compliance issues (legal compliance for example) mean that a script-heavy approach is mandated.

On the other hand, in Commercial-Off-The-Shelf (COTS) software development, a looser approach is normally taken. Since speed-to-market is more important than the risk of a single software defect, there is considerable latitude in the test approach. Specific test cases may not be documented or loosely documented and testers will be given a great deal of freedom in how they perform their testing.

The ultimate extension of this is exploratory or unscripted testing.

In this form of testing, there is a considerable amount of preparation done but test cases are not pre-scripted. The tester uses their experience and a structured method to 'explore' the software and uncover defects. They are free to pursue areas which they think are more risky than others.

Scripting, it is argued, is a waste of time. On a big project the amount of time spent on scripting can actually exceed the amount of time in execution. If you have an experienced, educated tester with the right set of tools and the right mindset, it would be more effective and more cost efficient to let them get at the software right away and find some defects.

This concept is almost heresy in some camps.

> I sit somewhere in the middle of this debate.
>
> Preparation is essential. Some scripting is good but too much is not. Exploratory testing relies on good testers and fails without them. But a lot of time can be 'wasted' in scripting methodologies by writing scripts that will never find a defect.
>
> I do think that 'exploratory testing' produces better (thinking) testers than script heavy methodologies. In script heavy methodologies there is a tendency to believe the hard work is over when the scripting is done. A monkey could then execute the scripts and find defects – this is a dubious conclusion at best.
>
> But sometimes all you have are monkeys and it is a more efficient use of resources to allow your experienced testers to script, and use the simians to execute.
>
> In the end, don't let adherence to a particular methodology blind you to the possibilities of other approaches. Trainyour testers in all the possibilities and let them use their judgement.

There is also an important legal aspect to this as Cem Kaner points out in his book "Testing Computer Software". If you are responsible for releasing a piece of software that causes financial loss you could be liable for damages. Further, if you cannot prove that you have conducted due diligence through adequate testing you may be guilty of professional negligence. One of the goals of test preparation therefore is to provide an audit trail which shows the efforts you have made to verify the correct behaviour of the software.

Test Cases

A test case documents a test, intended to prove a requirement.

The relationship is not always one-to-one, sometime many test case are required to prove one requirement. Sometimes the same test case must be extrapolated over many screens or many workflows to completely verify a requirement. There should be *at least* one test case per requirement however.

Some methodologies (like RUP) specify there should be two test cases per requirement – a positive test case and a negative test case. A positive test case is intended to prove that the function-under-test behaves as required with correct input and a negative test is intended to to prove that the function-under-test does not provoke an error with incorrect input (or responds gracefully to that error).

> This is where the debate about what to script, and what not to, really heats up. If you were to script separately for every possible negative case, you would be scripting till the cows come home.
>
> Consider a 'date-of-birth' field in a software application. It should only accept 'correctly formatted' dates. But what is a correct format? It is probably possible to generalise this from the requirements and come up with a single test case which specifies all the acceptable date formats.
>
> But what about the negative case? Can you possible extrapolate all the possible inputs and specify how the system should react? Possibly, but itwould take you forever. To generalise it you could simply say that the system should produce and error with 'unacceptable input'
>
> I tend to favour the approach that a positive test case implies a negative case.
>
> If your positive case also documents how the software is expected to handle exceptions then it covers both the positive cases and the negative cases. If your testers are well trained and educated then they will attempt all the possible input values in an attempt to provoke an exception.

In reality the number of cases depends on the latitude you allow your testers.

Storing Test Cases

There are a variety of ways to store test cases.

The simplest way is in a word-processing document of a spreadsheet.

One of the common form is a TSM or Test Script Matrix (also known as a Traceability matrix). In a TSM each line item represents a test case with the various elements of each case (see below) stored in the columns. These can be good for a small test effort since it is relatively easy to track scripting and execution in a spreadsheet but in larger projects they prove difficult to manage. The extent to which they actually aid traceability is also questionable since they don't enforce change control and aren't very good at one-to-many mappings.

In more complex software development efforts a database or specialist test case management tool can be used. This has the benefit of enforcing a standard format and validation rules on the contents of the test case. It can also be used to record execution on multiple test runs, produce reports and even assist with traceability by linking back to requirements in a separate database. It can also enforce change control and track the history of changes and execution.

Elements of a Test Case

The following table list the suggested items a test case should include:

ITEM	DESCRIPTION
Title	A unique and descriptive title for the test case
Priority	The relative importance of the test case (critical, nice-to-have, etc.)
Status	For live systems, an indicator of the state of the test case. Typical states could include : Design – test case is still being designed Ready – test case is complete, ready to run Running – test case is being executed Pass – test case passed successfully Failed – test case failed Error – test case is in error and needs to be rewritten
Initial configuration	The state of the program before the actions in the "steps" are to be followed. All too often this is omitted and the reader must guess or intuit the correct pre-requisites for conducting the test case.
Software Configuration	The software configuration for which this test is valid. It could include the version and release of software-under-test as well as any relevant hardware or software platform details (e.g. WinXP vs Win95)
Steps	An ordered series of steps to conduct during the test case, these must be detailed and specific. How detailed depends on the level of scripting required and the experience of the tester involved.
Expected behaviour	What was expected of the software, upon completion of the steps? What is expected of the software. Allows the test case to be validated with out recourse to the tester who wrote it.

Tracking Progress

Depending on your test approach, tracking your progress will either be difficult or easy.

If you use a script-heavy approach, tracking progress is easy. All you need to do is compare the number of scripts you have left to execute with the time available and you have a measure ofyour progress.

If you don't script, then tracking progress is more difficult. You can only measure the amount of time you have left and use that as a guide to progress.

If you use advanced metrics (see next chapter) you can compare the number of defects you've found with the number of defects you expect to find. This is an excellent way of tracking progress and works irrespective of your scripting approach.

Adjusting the plan

But tracking progress without adjusting your plan is wasting information.

Suppose you script for 100 test case, each taking one day to execute. The project manager has given you 100 days to execute your cases. You are 50 days into the project andare on schedule, having execute 50% of your test cases.

But you've found no defects.

The hopeless optimist will say, "Well! Maybe there aren't any!"and stick to their plan. The experienced tester will say something unprintable and change their plan.

The chance of being 50% of the way through test execution and not finding defects is extremely slim. It either means there is a problem with your test cases or there is a problem with the way in which they are being executed. Either way, you're looking in thewrong place.

Regardless of how you prepare for testing you should have some kind of plan. If that plan is broken down into different chunks you can then examine the plan and determine what is going wrong.

Maybe development haven't delivered the bulk of the functional changes yet? Maybe the test cases are our of date or aren't specific enough? Maybe you've underestimated the size of the test effort? Whatever the problem you need to jump on it quick.

The other time you'll need your plan is when it gets adjusted for you.

You've planned to test function A but the development manager informs you function B has been delivered instead, function A is not ready yet. Or you are halfway through your test execution when the project manager announces you have to finish two weeks earlier

If you have a plan, you can change it.

Coping with the Time Crunch

The single most common thing a tester has to deal with is being 'crunched' on time.

Because testing tends to be at the end of a development cycle it tends to get hit the worst by time pressures. All kinds of things can conspire to mean you have lest time than you need. Here's a list of the most common causes:

- Dates slip – things are delivered later than expected
- You find more defects than you expected
- Important people leave the company or go off sick
- Someone moves the end date, or changes the underlying requirements the software is supposed to fulfil

There are three basic ways to deal with this :

- Work harder – the least attractive and least intelligent alternative. Working weekends or overtime can increase your productivity but will lead to burn out in the team and probably compromise the effectiveness of their work.
- Get more people – also not particularly attractive. Throwing people at a problem rarely speeds things up. New people need to be trained and managed and cause additional communications complexity that gets worse the more you add (see "The Mythical Man Month" by Frederick Brooks).
- Prioritise – we've already decided we can't test everything so maybe we can make some intelligent decisions about what we test next? Test the riskiest things, the things you think will be buggy, the things the developers think will be buggy, the things with highest visibility or importance. Push secondary or 'safe' code to one side to test if you have time later, but make everyoneaware of what you're doing – otherwise you might end up being the only one held accountable when buggy code is released to the customer.

The fourth and sneakiest way is also one of the best – contingency.

At the start, when you estimate how long it is going to take you to test, add a little 'fat' to your numbers. Then, when things go wrong as they invariably do, you will have some time up your sleeve to claw things back.

Contingency can either be implicit (hidden) or explicit (planned in the schedule). This depends on the maturity of your project management process. Since teams have a tendency to use all the time available to them, it is often best to conceal it and roll it out only in an emergency.

Defect Management

Defects need to be handled in a methodical and systematic fashion.

There's no point in finding a defect if it's not going to be fixed. There's no point getting it fixed if it you don't know it has been fixed and there's no point in releasing software if you don't know which defects have been fixed and which remain.

How will you know?

The answer is to have a defect tracking system.

The simplest can be a database or a spreadsheet. A better alternative is a dedicated system which enforce the rules and process of defect handling and makes reporting easier. Some of these systems are costly but there are many freely available variants.

Importance of Good Defect Reporting

Cem Kaner said it best - "the purpose of reporting a defect is to get it fixed."

A badly written defect report wastes time and effort for many people. A concisely written, descriptive report results in the elimination of a bug in the easiest possible manner.

Also, for testers, defect reports represent the primary deliverable for their work. The quality of a tester's defect reports is a direct representation of the quality of their skills.

Defect Reports have a longevity that far exceeds their immediate use. They may be distributed beyond the immediate project team and passed on to various levels of management within different organisations. Developers and testers alike should be careful to always maintain a professional attitude when dealing with defect reports.

Characteristics of a Good Defect Report

- *Objective* – criticising someone else's work can be difficult. Care should be taken that defects are objective, non-judgemental and unemotional. e.g. don't say "your program crashed" say "the program crashed" and don't use words like "stupid" or "broken".
- *Specific* – one report should be logged per defect and only one defect per report.
- *Concise* – each defect report should be simple and to-the-point. Defects should be reviewed and edited after being written to reduce unnecessary complexity.
- *Reproducible* – the single biggest reason for developers rejecting defects is because they can't reproduce them. As a minimum, a defect report must contain enough information to allow anyone to easily reproduce the issue.
- *Explicit* – defect reports should state information clearly or they should refer to a specific source where the information can be found. e.g. "click the button to continue" implies the reader knows which button to click, whereas "click the 'Next' button" explicitly states what they should do.
- *Persuasive* –the pinnacle of good defect reporting is the ability to champion defects by presenting them in a way which makes developers want to fix them.

Isolation and Generalisation

Isolation is the process of examining the causes of a defect.

While the exact root cause might not be determined it is important to try and separate the symptoms of the problem from the cause. Isolating a defect is generally done by reproducing it multiple times in different situations to get an understanding of how and when it occurs.

Generalisation is the process of understanding the broader impact of a defect.

Because developers reuse code elements throughout a program a defect present in one element of code can manifest itself in other areas. A defect that is discovered as a minor issue in one area of code might be a major issue in another area. Individuals logging defects should attempt to extrapolate where else an issue might occur so that a developer will consider the full context of the defect, not just a single isolated incident.

A defect report that is written without *isolating* and *generalising* it, is a half reported defect.

Severity

The importance of a defect is usually denoted as its "severity".

There are many schemes for assigning defect severity – some complex, some simple.

Almost all feature "Severity-1" and "Severity-2" classifications which are commonly held to be defects serious enough to delay completion of the project. Normally a project cannot be completed with outstanding Severity-1 issues and only with limited Severity-2 issues.

Often problems occur with overly complex classification schemes. Developers and testers get into arguments about whether a defect is Sev-4 or Sev-5 and time is wasted.

I therefore tend to favour a simpler scheme.

Defects should be assessed in terms of impact and probability. Impact is a measure of the seriousness of the defect when it occurs and can be classed as "high" or "low" – high impact implies that the user cannot complete the task at hand, low impact implies there is a workaround or it is a cosmetic error.

Probability is a measure of how likely the defect is to occur and again is assigned either "Low" or "High".

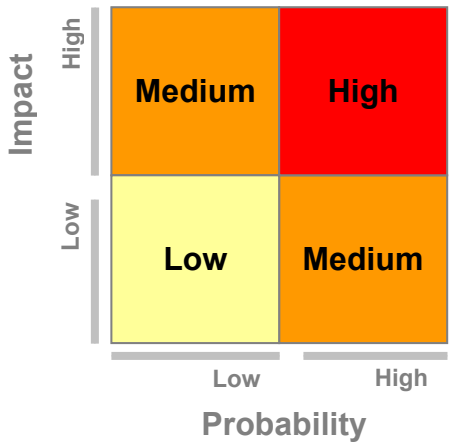


Figure 10: Relative severity in defects

Defects can then be assigned a severity based on :

Impact/Probability	= Severity
High / High	High
High / Low	Medium
Low / Low	Low

This removes the majority of debate in the assignment of severity.

Status

Status represents the current stage of a defect in its life cycle or workflow.

Commonly used status flags are :

- New – a new defect has been raised by testers and is awaiting assignment to a developer for resolution
- Assigned – the defect has been assigned to a developer for resolution
- Rejected – the developer was unable to reproduce the defect and has rejected the defect report, returning it to the tester that raised it
- Fixed – the developer has fixed the defect and checked in the appropriate code
- Ready for test – the release manager has built the corrected code into a release and has passed that release to the tester for retesting
- Failed retest – the defect is still present in the corrected code and the defect is passed back to the developer
- Closed – the defect has been correctly fixed and the defect report may be closed, subsequent to review by a test lead.

The status flags above define a life cycle whereby a defect will progress from "New" through "Assigned" to (hopefully) "Fixed" and "Closed. The following swim lane diagram depicts the roles and responsibilities in the defect management life cycle :

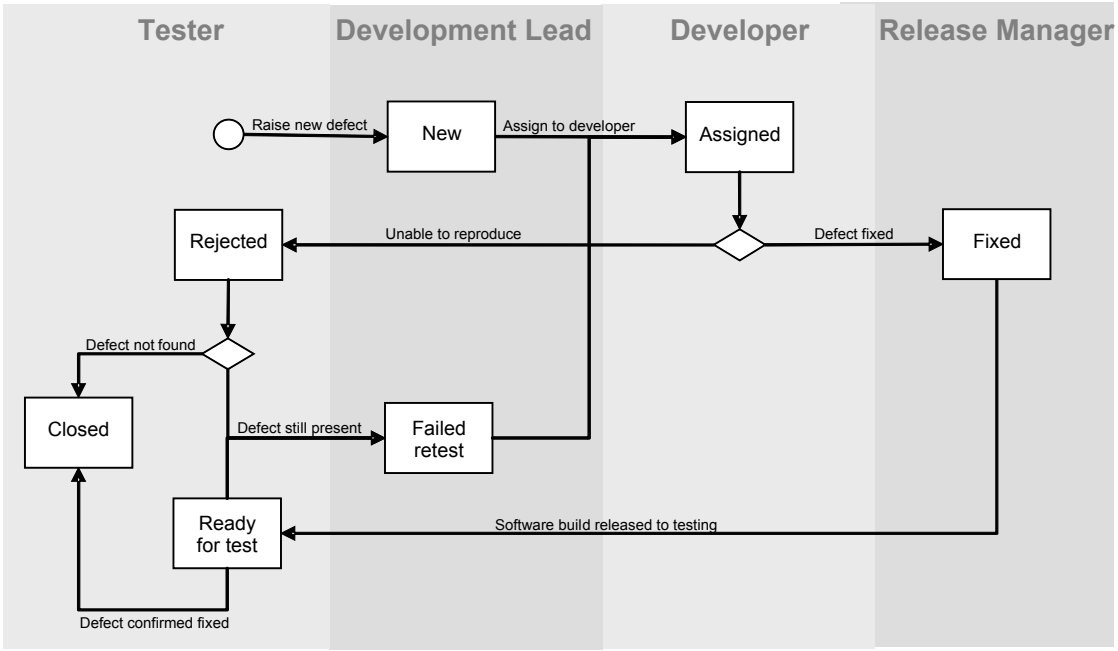


Figure 11: Defect life cycle swim lane diagram

Elements of a Defect Report

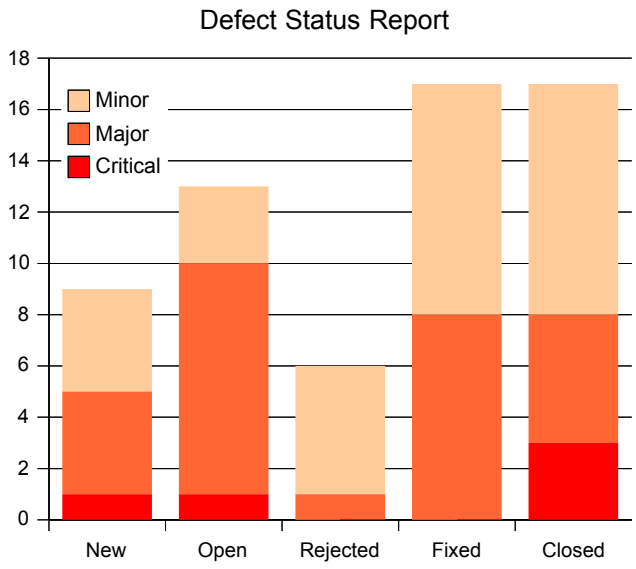
ITEM	DESCRIPTION
Title	A unique, concise and descriptive title for a defect is vital. It will allow the defect to be easily identified and discussed. *Good : "Closed" field in "Account Closure" screen accepts invalid date* *Bad : "Closed field busted"*
Severity	An assessment of the impact of the defect on the end user (see above).
Status	The current status of the defect (see above).
Initial configuration	The state of the program before the actions in the "steps to reproduce" are to be followed. All too often this is omitted and the reader must guess or intuit the correct pre-requisites for reproducing the defect.
Software Configuration	The version and release of software-under-test as well as any relevant hardware or software platform details (e.g. WinXP vs Win95)
Steps to Reproduce	An ordered series of steps to reproduce the defect *Good :* *1. Enter "Account Closure" screen* *2. Enter an invalid date such as "54/01/07" in the "Closed" field* *3. Click "Okay"* *Bad: If you enter an invalid date in the closed field it accepts it!*
Expected behaviour	What was expected of the software, upon completion of the steps to reproduce. *Good: The functional specification states that the "Closed" field should only accept valid dates in the format "dd/mm/yy"* *Bad: The field should accept proper dates.*
Actual behaviour	What the software actually does when the steps to reproduce are followed. *Good: Invalid dates (e.g. "54/01/07") and dates in alternate formats (e.g. "07/01/54") are accepted and no error message is generated.* *Bad: Instead it accepts any kind of date.*
Impact	An assessment of the impact of the defect on the software-under-test. It is important to include something beyond the simple "severity" to allow readers to understand the context of the defect report. *Good: An invalid date will cause the month-end "Account Closure" report to crash the mainframe and corrupt all affected customer records.* *Bad: This is serious dude!*
(Proposed solution)	An optional item of information testers can supply is a proposed solution. Testers often have unique and detailed information of the products they test and suggesting a solution can save designers and developers a lot of time.
Priority	An optional field to allow development managers to assign relative priorities to defects of the same severity
Root Cause	An optional field allowing developers to assign a root cause to the defect such as "inadequate requirements" or "coding error"

Software Defect Reports

At a basic level defect reports are very simple: number of defects by status and severity. Something like the diagram to the right.

This shows the number of defects and their status in testing. As the project progresses you would expect to see the columns marching across the graph from left to right – moving from New to Open to Fixed to Closed.

More complicated defect reports are of course possible. For example you might want to have a report on defect ageing – how long have defects been at a certain status. This allows you to target defects which have not progressed, which have not changed status over a period of time. By looking at the average age of defects in each status you can predict how long a given number of defects will take to fix.



	New	Open	Rejected	Fixed	Closed
Critical	1	1	0	0	3
Major	4	9	1	8	5
Minor	4	3	5	9	9

Figure 12: Defect status report

By far and away, the best defectreport is the defect status trend as a graph. This shows the total number of defects by status over time (see below).

The great advantage of this graph is that it allows you to predict the future.

If you take the graph at September and plot a line down the curve of the 'fixed' defects – it cuts the x-axis after the end of December. That means that from as early as September, it was possible to predict that not all of the defects would be fixed by December, and the project would not finish on time.

Similarly be following the curve of the 'new' defects you can intuit something about the progress of your project.

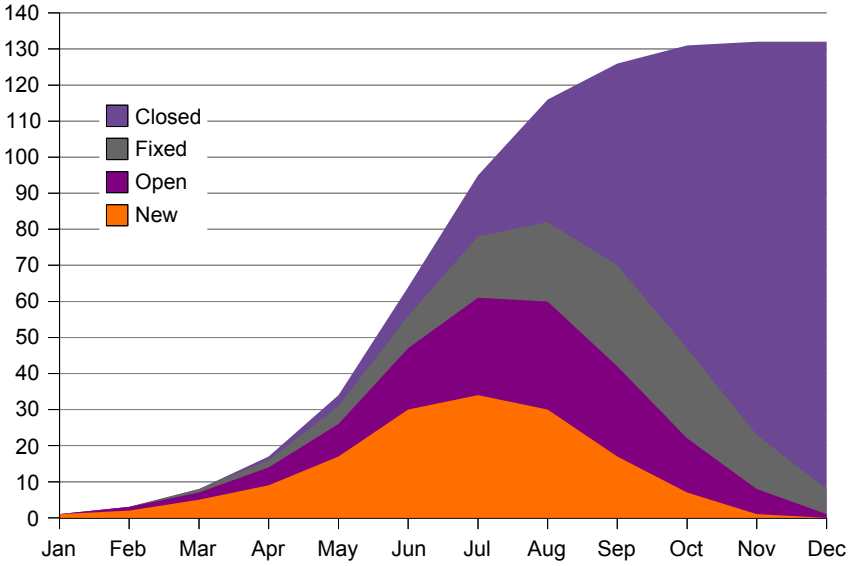


Figure 13: Defect trends over time

If the curve peaks and then is flat, then you have a problem in development – the bugs aren't staying fixed. Either the developers are reintroducing bugs when they attempt to fix them, your code control is poor or there is some other fundamental issue.

Time to get out the microscope.

Root Cause Analysis

Root cause analysis is the identification of the root cause of a defect.

Basic root cause analysis can often be extremely illuminating – if 50% of your software defects are directly attributable to poor requirements then you know you need to fix your requirements specification process. On the other hand if all you know is that your customer is unhappy with the quality of the product then you will have to do a lot of digging to uncover the answer.

To perform root cause analysis you need to be able capture the root cause of each defect. This is normally done by providing a field in the defect tracking system in which can be used to classify the cause of each defect (who decides what the root cause is could be a point of contention!).

A typical list might look like this :

Classification	Description
Requirements	The defect was caused by an incomplete or ambiguous requirement with the resulting assumption differing from the intended outcome
Design Error	The design differs from the stated requirements or is ambiguous or incomplete resulting in assumptions
Code Error	The code differs from the documented design or requirements or there was a syntactic or structural error introduced during coding.
Test Error	The test as designed was incorrect (deviating from stated requirements or design) or was executed incorrectly or the resultant output was incorrectly interpreted by the tester, resulting in a defect "logged in error".
Configuration	The defect was caused by incorrectly configured environment or data
Existing bug	The defect is existing behaviour in the current software (this does not determine whether or not it is fixed)

Sub-classifications are also possible, depending on the level of detail you wish to go to. For example, what kind of requirement errors are occurring? Are requirements changing during development? Are they incomplete? Are they incorrect?

Once you have this data you can quantify the proportion of defects attributable to each cause.

Classification	Count	%
Requirements	12	21%
Design Error	5	9%
Code Error	9	16%
Test Error	18	32%
Configuration	9	16%
Existing bug	3	5%
TOTAL	56	100%

Figure 14: Sample root cause analysis

In this table, 32% of defects are attributable to mistakes by the test team, a huge proportion.

While there are obviously issues with requirements, coding and configuration, the large number of test errors means there are major issues with the accuracy of testing.

While most of these defects will be rejected and closed, a substantial amount of time will be spent diagnosing and debating them.

This table can be further broken down by other defect attributes such as "status" and "severity". You might find for example that "high" severity defects are attributable to code errors but "low" severity defects are configuration related.

A more complete analysis can be conducted by identifying the root cause (as above) and how the defect was identified. This can either be a classification of the phase in which the defect is identified (design, unit test, system test etc.) or a more detailed analysis of the technique used to discover the defect (walkthrough, code inspection, automated test, etc.) This then gives you an overall picture of the cause of the defect and how it is detected which helps you determine which of your defect removal strategies are most effective.

Metrics

The ultimate extension of data capture and analysis is the use of comparative metrics.

Metrics (theoretically) allow the performance of the development cycle as a whole to be measured. They inform business and process decisions and allow development teams to implement process improvements or tailor their development strategies.

Metrics are notoriously contentious however.

Providing single figures for complex processes like software development can over-simplify things. There may be entirely valid reasons for one software development having more defects than another. Finding a comparative measure is often difficult and focussing on a single metric without understanding the underlying complexity risks making ill informed interpretations.

Most metrics are focussed on measuring the performance of a process or an organisation. There has been a strong realisation in recent years that metrics should be useful for individuals as well. The effectiveness of a process is directly dependent on the effectiveness of individuals within the process. The use of personal metrics at all levels of software development allow individuals to tune their habits towards more effective behaviours.

Testing Metrics for Developers

If the purpose of developers is to produce code, then a measure of their effectiveness is how well that code works. The inverse of this is how buggy a particular piece of code is – the more defects, the less effective the code.

One veteran quality metric that is often trotted out is "defects per thousand Lines Of Code" or "defects per KLOC" (also known as defect density). This is the total number of defects divided by the number of thousands of lines of code in the software under test.

The problem is that with each new programming paradigm, defects per KLOC becomes shaky. In older procedural languages the number of lines of code was reasonably proportional. With the introduction of object-oriented software development methodologies which reuse blocks of code, the measure becomes largely irrelevant. The number of lines of code in a procedural language like C or Pascal, bears no relationship to a new language like Java or .Net.

The replacement for "defects/KLOC" is "defects per developer hour" or "defect injection rate".

Larger or more complex software developments take more time for developers to code and build. The number of defects a developer injects into his code during the development is a direct measure of the quality of the code. The more defects, the poorer the quality. By dividing the number of defects by the total hours spent in development you get a comparative measure of the quality of different software developments.

$$\text{Defect Injection Rate} = \frac{\text{Number of defects created}}{\text{developer hours}}$$

Note that this is not a measure of efficiency, only of quality. A developer who takes longer and is more careful will introduce less defects than one who is slapdash and rushed. But how long is long enough? If a developer only turns out one bug free piece of software a year, is that too long? The use of one metric must be balanced by others to make sure a 'balanced scorecard' is used. Otherwise you might be adjusting for one dimension to the exclusion of all others.

Measures of development efficiency are beyond the scope of this text.

Test Metrics for Testers

An obvious measure of testing effectiveness is how many defects are found – the more the better.

But this is not a comparative measure.

You could measure the defects a particular phase of testing finds as a proportion of the total number of defects in the product. The higher the percentage the more effective the testing. But how many defects are in the product at any given time? If each phase introduces more defects, this is a moving target. And how long are you going to wait for your customers to find all the defects you missed in testing? A month? A year?

And suppose that the developers write software that has little or no defects? This means you will find little or no defects. Does that mean your testing is ineffective? Probably not, there are simply less defects to find than in a poor software product.

Instead, you *could* measure the performance of individual testers.

In a 'script-heavy' environment measures of test efficiency are easy to gather. The number of test cases or scripts a tester prepares in an hour could be considered a measure of his or her productivity during preparation. Similarly the total number executed during a day could be considered a measure of efficiency in test execution.

But is it really?

Consider a script-light or no-script environment. These testers don't script their cases so how can you measure their efficiency? Does this mean that they can't be efficient? I would argue they can. And what if the tests find no defects? Are they really efficient, no matter how many are written?

Let's return to the purpose of testing – to identify and remove defects in software.

This being the case, an efficient tester finds and removes defects *more quickly* than an inefficient one. The number of test cases is irrelevant. If they can remove as many defects without scripting, then the time spent scripting would be better used executing tests, not writing them.

So the time taken to find a defect is a direct measure of the effectiveness of testing.

Measuring this for individual defects can be difficult. The total time involved in finding a defect may not be readily apparent unless individuals keep very close track of the time they spend on testing particular functions. In script-heavy environments you also have to factor in the time taken scripting for a particular defect, which is another complication.

But to measure this for a particular test effort is easy – simply divide the total number of hours spent in testing by the total number of defects logged (remember to include preparation time).

$$\text{Defect Discovery Rate} = \frac{\text{Number of defects found}}{\text{tester hours}}$$

Note that you should only count defects that have been fixed in these equations.

Why?

New defects that haven't been validated are not defects. Defects which are rejected are also not defects. A defect which has been fixed, is definitely an error that need to be corrected. If you count the wrong numbers, you're going to draw the wrong conclusions.

Other Metrics for Testing and Development

Just as developers are responsible for errors in their code, so testers should be responsible for errors in their defect reports. A great deal of time can be wasted by both development and test teams in chasing down poorly specified defect reports or defects that are reported in error.

So a measure of testing effectiveness therefore becomes the number of defect reports rejected by development. Of course you should minimise this value, or use it as a proportion of defects logged and target each tester and the test team overall with a 0% goal – no errors.

Also you can target other metrics like response times to defect reports.

If a developer takes too long to respond to a defect report he can hold the whole process up. If they take too long to fix a defect the same can happen. Also testers can sit on defects, failing to retest them and holding up the project.

But be careful of using these measures too prescriptively.

Defects are funny beasts. They are inconsistent and erratic. They defy comparisons. One "minor" severity defect might seem like another but might take ten times as long to diagnose and resolve. The idiosyncrasies of various software products and programming languages might make one class of defects more difficult to fix than another.

And while these will probably average out over time, do you really want to penalise a developer or tester because they get lumped with all the difficult problems?

Food for thought...

You're measuring defect injection rate.

You're measuring defect detection rate.

If you do this for a long time you might get a feel for the 'average' defect injection rate, and the 'average' defect detection rate (per system, per team, per whatever). Then, when a new project comes along, you can try and predict what is going to happen.

If the average defect injection rate is 0.5 defects per developer hour, and the new project has 800 hours of development, you could reasonably expect 400 defects in the project. If your average defect detection rate is 0.2 defects per tester hour, it's probably going to take you 2000 tester hours to find them all. Have you got that much time? What are you going to do?

But be careful – use these metrics as 'dashboard' numbers to highlight potential issues but don't get too hung up on them. They are indicative at best.

Things change, people change, software changes.

So will your metrics.

Release Control

When you release a product 'into the wild' you will need to know what you released, to whom and when. This may seem trivial but in a large development environment it can be quite tricky.

Take for example a system where ten developers are working in parallel on a piece of code. When you go to build and release a version of the system how do you know what changes have been checked in ? When you release multiple versions of the same system to multiple customers and they start reporting bugs six months later, how doyou know which version the error occurs in ? How do you know which version to upgrade them to, to resolve the issue?

Hopefully your version control system will have labelled each component of your code with a unique identifier and when a customer contacts you, you can backtrack to that version of code by identifying which version of the system they have.

Further your change control system will let you identify the changes made to that version of the system so that you can identify what caused the problem and rectify it. Yourchange control system should also record the version of the product that contains the fix and therefore you can upgrade all customers to the correct version.

Typically you should track the following items for each release :

- The version number of the release
- The date of the release
- The purpose of the release (maintenance, bug fix, testing etc.)
- For each software component within the release:
  - the name and version number of the component
  - the date it was last modified
  - some kind of checksum which confirm the integrity of each module

Sample version control database :

Version	Date	Type	Components			
			Name	Version	Last Mod	Hash
3.0.5.28	27-05-2004	Internal	Kernel	2.3.0.25	25-05-2004	#12AF2363
			GUI	1.0.0.12	27-05-2004	#3BC32355
			Data Model	2.3.0.25	23-05-2004	#AB12001F
			I/O Driver	2.3.0.01	01-04-2004	#B321FFA
3.0.5.29	05-06-2004	Internal	Kernel	2.3.0.28	03-06-2004	#2C44C5A
			GUI	1.0.0.15	01-06-2004	#32464F4
			Data Model	2.3.0.25	23-05-2004	#AB12001F
			I/O Driver	2.3.0.01	01-04-2004	#B321FFA
3.0.5.30	etc.…	…	…	…	…	…

(NB: A hash simply a mathematical function that takes the file as input and produces a unique number to identify it. If even a tiny portion of the original changes the hash value will no longer be the same. The hashes in the table above are represented as MD5 hashes in hexadecimal.)

Release controls should apply within development as well as outside. When thedev team releases a product to the test team, it should follow a controlled process. This ensures that the teamsare working on the same builds of the product, that defects are being tracked and resolved against a particular build and that there is a minimal lag in defect resolution.

Verification and Smoke Testing

An important part of the release control process is the verification of the basic functionality of a particular release. Often errors in the build, assembly or compilation process can result in faults in the delivered release. Spending time identifying and isolating these faults is frustrating and pointless since they are simply artefacts of a bad build.

A release should be verified for basic functionality through the use of 'smoke tests'. That is, before it is passed to any other group the team responsible for building a release should run a series of simple tests to determine that the release is working properly. This catches gross functional errors quickly and prevents any lag in releases. These tests can be automated as part of the build/release process and the results simply checked before the release is shipped.

> The term 'smoke testing' comes from the electronics industry. If an engineer designs a circuit or an electrical system the first test to be conducted is to plug it into the power briefly and switch it on. If smoke comes out then you switch it off and it's back to the drawing board (or soldering iron). If no smoke appears then you it's basically okay and you can try some more tests on it to see if it is working properly. The sameprinciple can be applied in software development.

A release should be verified for completeness as well.

Often releases can be built without a particular data or configuration file which is required. Part of the release process should be a simple test which verifies that all of the expected components are present in a release. This can be done by maintaining a separate list of necessary components which can then be automatically compared to a particular build to determine if anything is missing.

Release Notes

One extremely valuable piece of documentation that is often neglected in projects is a set of release notes. Accompanying each release should be a brief set of notes that details (in plain English) the changes made to the system or product with this release.

The major reason for including release notes is to help set expectations with end users. By including a list of 'known issues' with a particular release you can focus attention on important areas of the release and avoid having the same issue reported over and over again.

If the release is a Beta release to tester/customers the notes may also include some notes on how end-users can be expected to log feedback and what kind of information they should provide. If the release is a normal production release then the notes should be more formal and contain legal and license information and information on how to contact support.

Release Notes : ABR SDK v3.0.5.29

Delivery Items

This release image contains three separate archive files:

- **Documentation archive** - this archive contains all of the necessary architectural and installation documentation for the SDK.
- **SDK Installation Archive** - this archive is intended to be installed by users who require a desktop version of the application.
- **SDK Require Binaries** - an archive containing only the 'executable' portions of the SDK This archive is intended for individuals who intend to integrate the SDK into an application.

Known Issues

1. Threadpool variation of pool sizes via registry is not enabled

An interim multi-threading model used a number of settings in a registry key that allowed changes to be made to the number of threads available to the SDK. These settings are no longer used and the configuration of the threading model is fixed within the SDK.

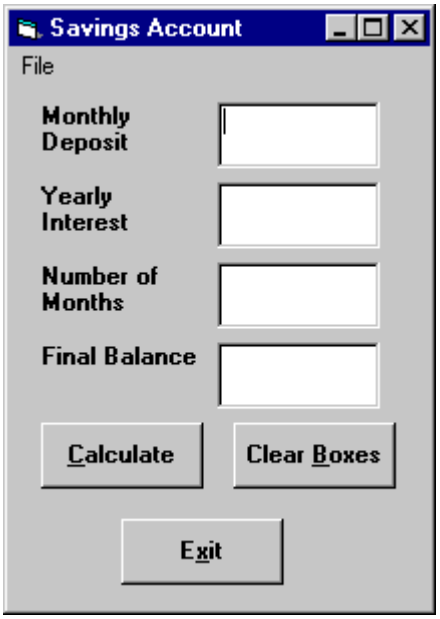
2. Removal of ABR.DLL

Part of the pre-processor functionality was contained within a separate DLL for historical reasons – the ABE DLL...

Complexity in Software

Many people underestimate the complexity of software. Software, they argue,is simply a set of instructions which a computer must follow to carry out a task. Software however can be unbelievably complex and as, Bruce Sterling puts it, 'protean'or changeable.

For example, the "Savings Account" application pictured at right is a simple application written in Visual Basic which performs simple calculations for an investment. Touse the application the user simply fills in three of four of the possible fields and clicks "Calculate". The program then works out the fourth variable.



If we look at the input space alone we can work out its size and hence the number of tests required to achieve "complete" coverage or confidence it works.

Each field can accept a ten digit number. Thismeans that for each field there are 10^{10} possible combinations of integer (without considering negative numbers). Since there are four input fields this means the total number of possible combinations is 10^{40}.

If we were to automate our testing such that we could execute 1000 tests every second, it would take approximately 3.17×10^{29} years to complete testing. That's about ten billion, billion times the life of the universe.

An alternative would be to seek 100% confidence through 100% coverage of the code. This would mean making sure we execute each branch or line of code during testing. While this gives a much higher confidence for much less investment, it does not and will never provide a 100% confidence. Exercising a single pass of all the lines of code is inadequate since the code will often fail only for certain values of input or output. Weneed to therefore cover all possible inputs and once again we are back to the total input space and a project schedule which spans billions of years.

There are, of course, alternatives.

When considering the input space we can use "equivalence partitioning". This is a logical process by which we can group "like" values and assume that by testing one we have tested them all. For example in a numerical input I could group all positive numbers together and all negative numbers separately and reasonably assume that the software would treat them the same way.I would probably extend my classes to include large numbers, small numbers, fractional numbers and so on but at least I am vastly reducing the input set.

Note however that these decisions are made on the basis of our assumed knowledge of the software, hardware and environment and can be just as flawed as the decision a programmer makes when implementing the code. Woe betide the testerwho assumes that 2^{64}-1 is the same as 2^{64} and only makes one test against them!

Glossary

Acceptance Test	Final functional testing used to evaluate the state of a product and determine its readiness for the end-user. A 'gateway' or 'milestone' which must be passed.
Acceptance Criteria	The criteria by which a product or system is judged at Acceptance . Usually derived from commercial or other requirements.
Alpha	The first version of product where all of the intended functionality has been implemented but interface has not been completed and bugs have not been fixed.
API	Application Program Interface – the elements of a software code library that interacts with other programs.
Beta	The first version of a product where all of the functionality has been implemented and the interface is complete but the product still has problems or defects.
Big-Bang	The implementation of a new system "all at once", differs from incremental in that the transition from old to new is (effectively) instantaneous
Black Box Testing	Testing a product without knowledge of its internal working. Performance is then compared to expected results to verify the operation of the product.
Bottom Up	Building or designing software from elementary building blocks, starting with the smaller elements and evolving into a lager structure. See "Top Down" for contrast.
Checksum	A mathematical function that can be used to determine the corruption of a particular datum. If the datum changes the checksum will be incorrect. Common checksums include odd/even parity and Cyclic Redundancy Check (CRC).
CLI	Command Line Interface – a type of User Interface characterised by the input of commands to a program via the keyboard. Contrast with GUI.
CMM	The Capability Maturity Model – a model for formal description of the five levels of maturity that an organisation can achieve.
Critical Path	The minimum number of tasks which must be completed to successfully conclude a phase or a project
Deliverable	A tangible, physical thing which must be "delivered" or completed at a milestone. The term is used to imply a tactile end-product amongst all the smoke and noise.
DSDM	Dynamic Systems Development Methodology – an agile development methodology developed by a consortium in the UK.
Dynamic Analysis	White box testing techniques which analyse the running, compiled code as it executes. Usually used for memory and performance analysis.
End-user	The poor sap that gets your product when you're finished with it! The people that will actually use your product once it has been developed and implemented.
Feature creep	The development or a product in a piece-by-piece fashion, allowing a gradual implementation of functionality without having the whole thing finished..
Glass Box Testing	Testing with a knowledge of the logic and structure of the code as opposed to "Black Box Testing". Also known as "White Box Testing".
Gold Master	The first version of the software to be considered complete and free of major bugs. Also known as "Release Candidate".
GUI	Graphical User Interface – a type of User Interface which features graphics and icons instead of a keyboard driven Command Line Interface (CLI qqv). Originally known as a WIMP (Windows-Icon-Mouse-Pointer) interface and invented at Xerox PARC / Apple / IBM etc. depending on who you believe.
Heuristic	A method of solving a problem which proceeds by trial and error. Usedin Usability Engineering to define problems to be attempted by the end-user.
HCI	Human Computer Interaction – the study of the human computer interface and how to make computers more "user friendly".

Incremental	The implementation of a system in a piece-by-piece fashion. Differs from a big-bang approach in that implementation is in parts allowing a transition from old to new.
Kernel	The part of software product that does the internal processing. Usually this part has no interaction with the outside world but relies on other 'parts' like the API and UI.
Milestone	A significant point in a project schedule which denotes the delivery of a significant portion of the project. Normally associated with a particular "deliverable".
MTTF	Mean Time To Failure – the mean time between errors. Used in engineering to measure the reliability of a product. Not useful for predicting individual failures.
Open Source	A development philosophy which promotes the distribution of source code to enhance functionality through the contributions of many independent developers.
Prototype	A model of software which is used to resolve a design decision in the project.
QA	Quality Assurance – the process of preventing defects from entering software through 'best practices'. Not be confused with testing!
Release Candidate	The first version of the software considered fit to be released (pre final testing).
Requirement	A statement of need from a stake-holder identifying a desire to be fulfilled
ROI	"Return On Investment" – a ratio which compares the monetary outlay for a project to the monetary benefit. Typically used to show success of a project.
RUP	Rational Unified Process – a software development methodology focussed on object-oriented development. Developed by the big three at IBM-Rational Corporation.
Scope creep	The relentless tendency of a project to self-inflate and take on more features or functionality than was originally intended. Also known as 'feature creep'.
Shrink wrapped	Software that is designed to be sold "off-the-shelf" and not customised for one user
SLA	Service Level Agreement – an agreement between two parties as to the minimum acceptable level of service a piece of software or a process must provide
Show Stopper	A defect that is so serious it literally stops everything. Normally given priority attention until it is resolved. Also known as "critical" issues.
Static analysis	White Box testing techniques which rely on analysing the uncompiled, static source code. Usually involves manual and automated code inspection.
Stakeholder	A representative from the client business or end-user base who has a vested interest in the success of the project and its design
Testing	The process of critically evaluating software to find flaws and fix them and to determine its current state of readiness for release
Top Down	Building or designing software by constructing a high level structure and then filling in gaps in that structure. See "Bottom Up" for contrast
UAT	User Acceptance Test(ing) – using typical end-users in Acceptance Testing (qv). Often a test as to whether a client will accept or reject a 'release candidate' and pay up.
Usability	The intrinsic quality of a piece of software which makes users like it. Often described as the quality present in software which *does not annoy* the user.
Usability Testing	User centric testing method used to evaluate design decisions in software by observing typical user reactions to prototype design elements
User Interface	The top 10% of the iceberg. The bit of software that a user actually sees. See CLI and GUI, different from the Kernel or API.
Verification	The process of checking that software does what it was intended to do as per its design. See "Validation". Sometimes posited as "are we making the product right?"
Validation	Checking that the design of a software system or product matches the expectations of users. See "Verification". Sometimes posited as : "are we making the right product?"
White Box Testing	Testing the program with knowledge and understanding of the source code. Usually performed by programmers, see Black Box Testing for contrast.
XP	eXtreme Programming – A form of agile programming methodology