Software Testing Primer v2

Table of Contents

PREFACE TO THE SECOND EDITION 3

INTRODUCTION 1
The Need for Testing
Models of Software Development

QUALITY IN SOFTWARE 5
Value, Waste and Quality
Built-In Quality
Test Driven Development (TDD)
Behaviour Driven Development (BDD)

CONCEPTS OF TESTING 9
The Testing Mindset
Test Early, Test Often
Regression vs. Retesting
White-Box vs Black-Box testing
Verification and Validation

FUNCTIONAL TESTING 12
The Role of Functional Testing
White Box testing
Unit, Integration and System testing
Acceptance Testing
Test Automation

NON-FUNCTIONAL TESTING 18
Testing the design
Usability Testing
A/B Testing & Testing in Production
Performance Testing

TEST PLANNING 22
The Purpose of Test Planning
Software in Many Dimensions
Test Identification

TEST PREPARATION 27
Test Scripting
Test Cases
Acceptance Criteria
Exploratory Testing

TEST EXECUTION 32
Tracking Progress
Adjusting the plan
Defect Management

REPORTING AND METRICS 38
Defect Reporting
Metrics of Quality and Efficiency

RELEASE MANAGEMENT & CONTINUOUS DELIVERY 43
Release and Deployment Management
Release Planning
Deployment Planning
Configuration Management
Continuous Delivery

PURE THEORY 50
Complexity in Software

GLOSSARY 51

Preface to the second edition

Dear Reader,

I wrote the first version of this Primer in about 2000, but didn't publish it until some years later.

Since then I've had dozens, possibly hundreds, of people contact me to say it has been extremely useful for them in their day to day work. I'm extremely happy and humbled that it has helped people in the industry and in education understand the craft of software testing.

But fifteen-plus years is a long time in any industry, let alone software development.

So not only have my own views on the subject changed, but the industry has moved on in leaps and bounds. Just when I thought the IT industry had become stagnant and was slowly subsiding into a mediocre blend of brute force and simplistic aphorisms – along came a paradigm shift, the like of which I honestly believe we have never seen before.

Cheap storage and virtualisation led to the cloud revolution which led to continuous integration which led to devops... well it hasn't been quite like that, but you get the idea. A few people are still underestimating the magnitude of this change in IT but they are in a rapidly shrinking minority.

In a lot of ways the recent developments in in software development mirror my own career.

In 1997, with the help of colleagues from Curtin University in Perth, I helped publish a book on incremental and iterative development. Some of the ideas we espoused shaped my own ideas of software development and you can probably find the traces of them still in this book.

But after that I spent the next fifteen years of my working life as a test manager in various organisations, trying to hold back the a tide of excrement flowing down the waterfall lifecycle. It was a fruitless and often thankless task.

It wasn't that I didn't enjoy it or that I wasn't successful – by all the benchmarks I've had a remarkably successful career – it's just that I always felt it could be so much better.

Software development has always seemed far too hard and far too costly to me... until now.

One of the major influence in my recent working life is from Lean and the Toyota Way. By learning first hand from Lean disciples some of the principles I've been able (like others) to apply them to software development and learn a lot from the process. I find agile, while its heart is in the right place, to be a simplistic interpretation of software development principles (although I've been heartened by recent evolutions – like devops and CI/CD).

The confluence of cheap storage, cloud, automation and modern toolsets is setting the scene for a new era of the software revolution, although a few heads will roll in the process. The software development industry is going to see the same kind of disintermediation that Uber did to taxis or Airbnb to accommodation – and the world will benefit.

This new landscape demands a new kind of tester... or a new kind of way of thinking about testing.

I've tried to distil all I've learned from Lean and Devops, and agile, into this version of the primer.

I hope you like it and you find the Primer useful in your work,

Nick Jenkins, Jan 2017.

Introduction

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 be "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 of failure increases. In software of any reasonable complexity the risk quickly reaches unacceptable levels.

How do you assess the level of risk?

How do you know if your software works before you inflict it upon your customers?

Testing is a search for the truth.

As Kem Caner, Brett Pettichord and James Bach said, testing is 'applied epistemology' – the applied search for justified belief. In philosophical terms, you can think you know something, but without reason or evidence, that belief is not justified; it is akin to false hope. You need evidence, logic or a reason to transform your belief into what a philosopher would call 'knowledge'.

You can hope your software works or you can *know* it works – through testing.

So testing is also a mindset and an attitude.

This book is for anyone that wants to develop a testing mindset: a critical, evaluating mental process that mirrors the principles of the scientific method. But this book extends beyond that to a consideration of all kinds of quality assurance in software development.

This book is for anyone that wants to make good software.

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:

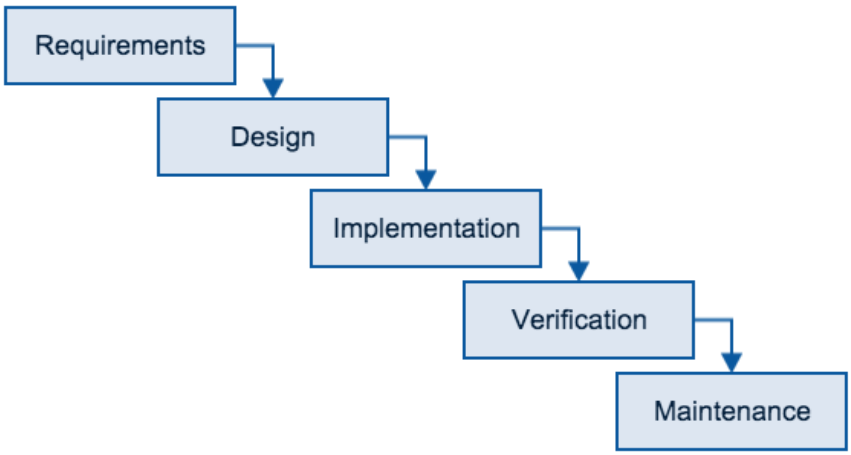


Figure 1: The Waterfall SDLC model

The five steps outlined are :

- *Analyse* the requirements of the software and decide what it is supposed to do
- *Design* a solution to meet these requirements
- *Implement* the design into working software
- *Verify* the finished software against the design (and requirements)
- *Maintain* the software 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. 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 "requirements" to "implementation" 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.

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.

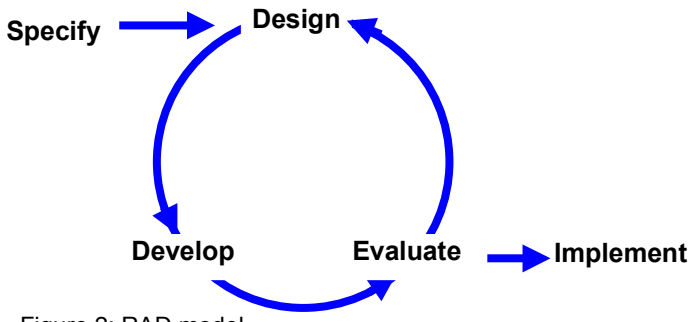


Figure 2: RAD model

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.

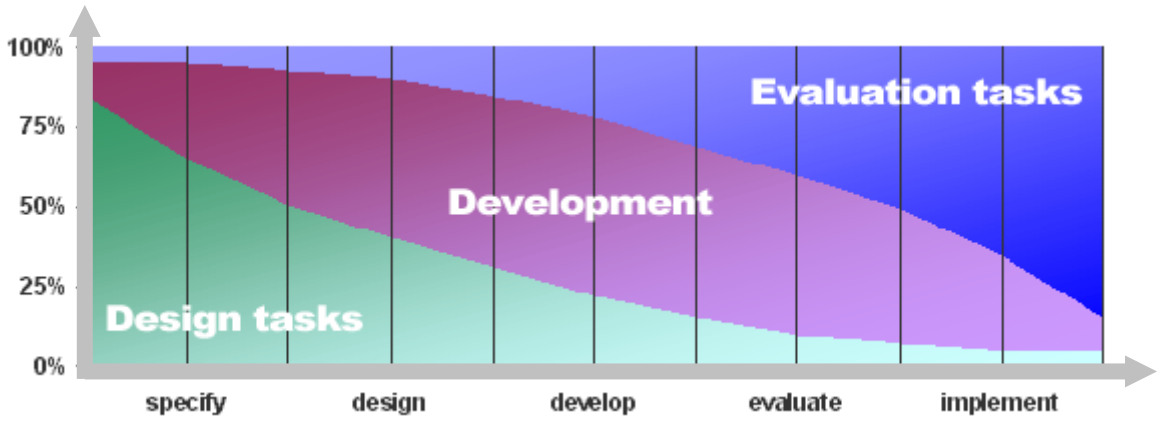


Figure 3: Incremental development

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 emphasis is on testing or evaluation to ensure that what has been developed meets requirements and is solid, stable and bug free (ha!).

Agile and Scrum

The 80's and 90's were the age of the methodology with hundreds of competing methodologies likes the Rational Unified Process (RUP), Team Software Process (TSP) and Cleanroom software engineering.

In 2001 a group of prominent software development practitioners published the *agile manifesto* in a reaction to these 'heavy-weight' methodologies. They called for more emphasis on the human elements of software development and a lighter touch methodology. The manifesto itself was not intended to be a prescriptive methodology in-and-of-itself but rather a set of guiding principles.

Ironically these principles have given way to a series of detailed methodologies, perhaps the best know of which is Scrum (they also include DSDM, extreme programming and the aforementioned RAD).

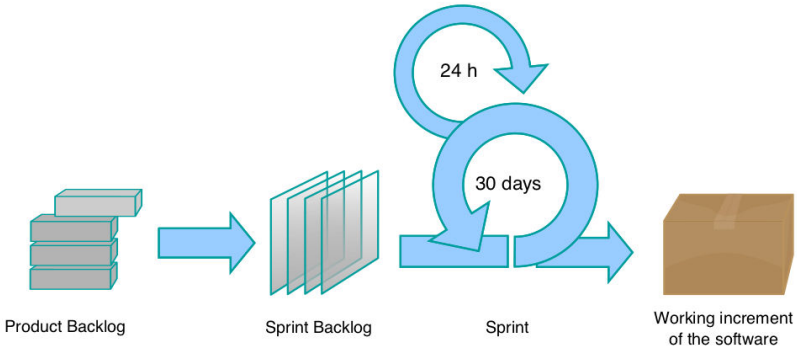


Figure 4: The Scrum Development Model

Based on the agile principles, Scrum specifies the delivery of software in a series of incremental *sprints*. Each sprint is planned, developed and deployed independently and feedback is constantly sought from the customer (or *product owner*) in order to refine the target deliverables.

Scrum teams are generally lead by a 'coach' called a *Scrum Master*. Work is prioritised in a *backlog* of items which are then selected for particular sprints. Completed sprints are demonstrated to the product owner in a *sprint review* and the team conducts a *retrospective* to examine its working processes.

> 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.

Over the years I have substantially changed my view on testing and quality. I used to argue that testing was the best way to improve quality; not so any more.

What changed my mind was a philosophy called 'Lean' or the 'Toyota Way'.

Developed over many years at Toyota and explained to the western world by people like Jim Womack and John Shook as 'Lean' - it is a philosophy born in the manufacturing industry that is finding new roots in different service industries, like healthcare and software development.

Traditional wisdom in car manufacturing had it that defects were inevitable and quality problems were discovered and resolved by inspection – by testing.

Even before they started making cars, the people at Toyota knew this was wrong. Finding a defect and resolving it does not remove the reason that caused it. Only careful problem analysis and process improvement will resolve issues at source, preventing defects.

Most testing does not get this far.

Value, Waste and Quality

Central to this is the concept of value and waste in a process.

In Lean terms, *value* is anything the customer values – anything they will pay for. *Waste* is simply the opposite.

If you are making a product there are three types of task you will engage in:

1. Those that add value
2. Those that don't add value but are necessary
3. Those that don't add value and are unnecessary

To be efficient you should *maximise* type 1, *optimise* type 2 and *eliminate* type 3.

Testing is type 2 task, it doesn't add value in and of itself but it is necessary (to some extent).

A customer won't pay for testing. If you add more testing to a product they're not going pay you more or perceive that the product is better value. They will pay more for a higher quality product, but testing does *not* improve the quality of the product by itself.

Waste	Manufacturing	Knowledge Work
Transport	Moving physical product around	Task switching and lack of focus
Inventory	Product sitting in queues	Code not in production unused documents
Motion	Movement of individuals	Searching for necessary information
Waiting	Waiting for product to arrive	Waiting for code to arrive
Over processing	Work not necessary to the final product	Paperwork not necessary to delivery
Over production	Producing more than the customer required	Features which are not required
Defects	Errors in the product which require rework	Features which don't meet the user's needs

Figure 5: Seven typical wastes

Only if testing leads to changes in the process or systematic improvements does it lead to higher quality. Removing one bug or defect does not improve quality, because it has not removed the cause of the bug. Removing one bug simply stops product quality from getting any worse. This is an open ended cycle – a self reinforcing death spiral.

If you do nothing to remove the source of defects, but you invest in more testing, you're inevitably going to find more bugs. And more bugs to fix means more work for developers and if nothing else changes you will quickly overload your system and your projects will overrun, your software will be buggy and your customers will be unhappy. You need to fix the system, not the symptom.

And testing is a cost – a significant one – so you must optimise it.

You need to uncover the maximum amount of information at the lowest cost and feed that information back into the product cycle to remove the source of defects.

Built-In Quality

In manufacturing they have long known you can't 'inspect-in' quality, it must be 'built-in'.

The goal is not to fix the defect – it is to diagnose how the defect was introduced and to engineer ways to avoid similar defects in future. Fixing the defect once delivers a tiny benefit. Fixing the process or removing the source of the defect delivers a compounding return that pays for itself many times over.

Suppose you are making cars, and a new car arrives at the end of the production line with three wheel-nuts on one wheel. One way to 'fix' the problem is to send off someone to find a 'lug wrench' and an extra wheel-nut to. But that will tie someone up in a pointless task, and the car will have to sit somewhere while it is being fixed. If it happens a lot you might need someone on standby with a wrench and a box of nuts!

By why not stop it from happening? Why not figure out why cars make it down the line with only three wheel nuts, and find a way to prevent that from happening in future?

But how do you fix something at source?

The temptation is always to rush to an immediate fix. First you have to stop and find the source, you have to find the root cause. Stop and ask yourself "why" the defect occurred? What caused it? Was it a simple error or was there confusion or misunderstanding? Why did that occur?

In Toyota this concept was manifested by "andon cords" or stopping the line.

Figure 6: Andon cord display

Traditionally in car manufacturing, since the days of Henry Ford, keeping the line going was sacrosanct. The line meant production, it meant volume and stopping it for problems meant lost production and downtime.

But at Toyota they implemented a system whereby any production worker could pull a cord if they notice a defect – and the line would stop. In fact they *encouraged* workers to pull the cord (the actual system in factories is a little more complicated than that).

Why? Because they realised that a problem (a defect) was an opportunity to remove it's cause from the process; problems are the signposts to their causes. Ignoring them means ignoring the causes and postponing dealing with them means information will disappear. If you shift your thinking from dealing with the defect, to dealing with its cause you gain a long term benefit.

Then you have to have a process for implementing a countermeasure or improvement, for evaluating if it worked and finally you have to have a way of standardising the change if it proves successful.

In 'Lean' this is known as the PDCA cycle or Plan-Do-Check-Act. When you detect a problem (and diagnose the cause) you *plan* a simple countermeasure to avoid the problem and you predict what will happen. Then you *do* the change. You then *check* the results and see if you get the outcome you expected, if not you can try again. If your change gave you the outcome you wanted, then you can *act* to roll out the change across your production line.

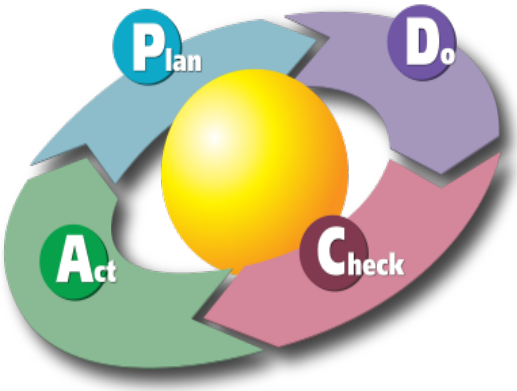


Figure 7: The PDCA problem solving cycle

Another way of understanding this is to think about feedback loops – the shorter a feedback loop the faster you learn.

The longer the time between when a defect is introduced and when it is found, the more rework you incur and the less you learn. If you shorten these loops then the contextual knowledge is fresher and leads to greater insight. You can then apply that knowledge to 'error-proofing' the process.

The Role of Automation

Toyota has a unique perspective on the role of automation in manufacturing. They refer to 'autonomation' or 'automation with a human touch'. The role of automation is to assist humans to do what they do best (solve problems) and to prevent mistakes.

A good example from the production line is an infrared beam placed on a box of wheel-nuts. If a worker doesn't reach into that box four times per wheel, then a signal is lit and the worker has the chance to override it before the line is stopped. If the error is a trivial one, the worker can correct it and work proceeds, but if it is serious he has the option to stop the line and seek help. The automation is designed to help the worker do their job, to succeed.

Another example comes from the world of health care. In a hospital they had a problem with basic hand sanitation in the emergency ward. They put up signs, educated the staff, located hand cleansing stations everywhere and nothing helped. Then a lean expert took a look at the problem and came up with a simple solution. They connected the hand sanitiser pump to the mechanism that opened the door to the emergency ward – to open the door, you had to clean your hands.

Automation in this context is not intended to replace human workers but to support them. It removes tedious repetitive tasks or prevents error.

Automation in Software Development

Many people in software development believe that it is fundamentally different to to anything that has gone before and there is little to learn from other industries. Software development they say is a 'creative discipline' and unlike manufacturing or engineering. They decry attempt to systemise, standardise or automate tasks, claiming that there is no benefit to doing so and instead we must 'unleash human potential'.

Not only is that breathtakingly insulting to other industries, it misunderstands the role of standardisation and automation. It is undeniable that there are many repetitive tasks in software development. And there is plenty of evidence that these repetitive tasks are the source of many software errors.

The way software is built and deployed, the way that infrastructure is configured, how changes are deployed and the way a myriad of different software components are utilised are all repeated a thousands times a day by a thousand different developers.

A good way to standardise a repetitive task is to give it to a machine.

And while there is an element of irony in having software test software (see my section on automated testing for contrast) the tools and practices have evolved to a point where they actively enable the development effort rather than hinder it. Also many of the best tools are free or nearly free to use.

So there is no longer any rational objection to automation in testing or development.

Further the landscape of software has changed so vastly that the complexities of micro-service, cloud or API architectures mandate the use of automation. There is simply no economic way to do tasks manually any more nor any rational justification for doing so.

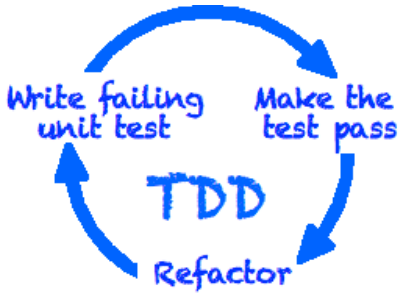
> In some testing literature, James Bach and Michael Bolton have labelled the difference as between 'real' or human *testing* and automated *checking*. Machines they argue can't test because they are deterministic and algorithmic and simply repeat what they've have been told to do.
>
> While I agree with the sentiment I can't agree with appropriating common words to support your argument. Testing vs checking? Meh. A test is a test, no matter who conducts it.

Test Driven Development (TDD)

In recent years I've come across one practice which exemplifies short feedback loops and an emphasis on verifiable quality in software - that is Test Driven Development (TDD).

TDD is a software design practice developed by Kent Beck around 2003. In it, developers write unit tests *before* they write the application code. There are two rules in TDD: First, never write a single line of code unless you have a *failing* automated test; second, eliminate duplication.

TDD encourages developers to think through the success criteria and hence the design of the code they are trying to write, and it ensures that only code sufficient to write the test is added. This provides the simplest solution to a software problem. It also provides instantaneous and constant feedback, through the automated unit test.



TDD creates short feedback loops through unit level regression and means that the code is effectively self documenting through it's embedded tests. Not only can a programmer see what the code does, he gets a sense of why. It also leads to cleaner simpler code through the elimination of duplication in the refactoring step.

Like all software development practices, TDD has its adherents and its opponents.

Behaviour Driven Development (BDD)

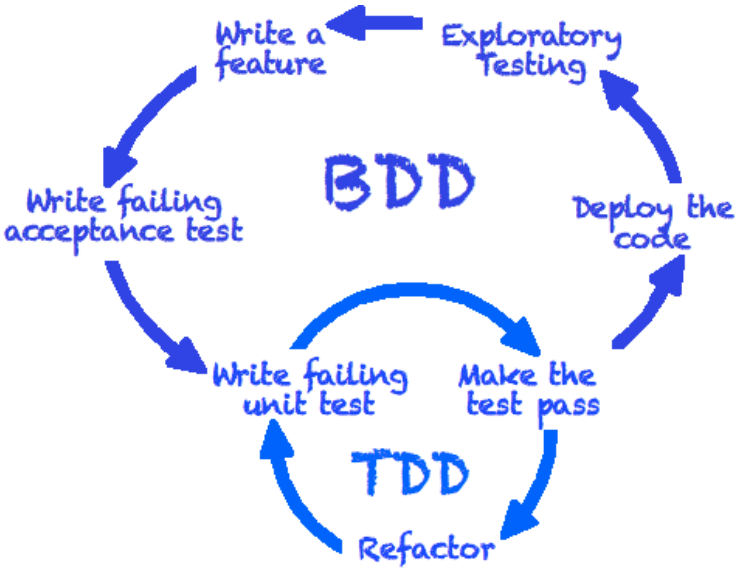
The basic concepts of TDD have been extended to include 'higher' level tests (like acceptance tests). This became known variously as 'specification by example', Acceptance Test Driven Development or Behaviour Driven Development (BDD).

In BDD, there are two nested loops : the lower level TDD loop and a higher level loop that deals with verifying requirements (in the form of tests or *scenarios*). First you write a requirement and it's associated scenarios (usually in a form like FIT or gherkin - qqv), then you do as much TDD as you need to satisfy the test and then you deploy it, possibly with additional manual testing if required.

Typically BDD tests are automated using a tool which automatically interprets the requirement and checks it against the application code, hence they may be referred to as 'executable specifications'.

In this way you are building a tight feedback loop between requirements, tests which verify the behaviour of the software and the software itself.

This eliminates or reduces problems with keeping automated tests up to date (qqv), it provides quick feedback loops and a mechanism for continuous improvement. It also ensures that only the minimal amount of code is written to satisfy a requirement (enough to pass the test).



BDD is a natural fit for practices like continuous integration and deployment (qqv CI/CD).

It gives teams with fast deployment pipelines the confidence to commit small changes, knowing they have a level of verification already.

The Testing Mindset

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

Designers and developers often approach software with an optimism based on the assumption that the changes they make are the correct solution. How could they do anything else? But they are just that – assumptions. Someone with a testing mindset is aware of their assumptions, and the possible limitations.

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

By taking a sceptical approach, we offer a balance.

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

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

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 (waterfall).

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. This is a well understood tenet of software development that is simply ignored by the majority of software development efforts.

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

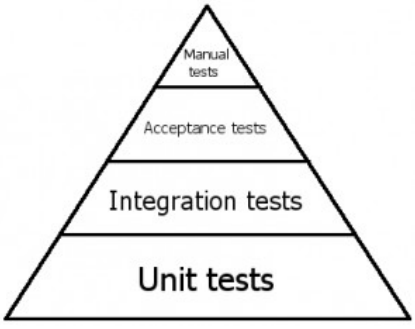


Figure 8: The test pyramid

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

Nor is a single pass of testing enough.

Your first past at testing simply identifies where the defects are. 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 is inadequate.

Regression vs. Retesting

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

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

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

A bug fix could in fact be introducing bugs elsewhere.

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

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

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

> 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. This is particulary true of systems with 'legacy code' - i.e. anything the current developer didn't write.
>
> 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... but we probably wouldn't need much testing.

White-Box vs Black-Box testing

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

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

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

Unit testing is a classic example of white-box testing. In unit testing developers use the programming language they are working in (or an add-on framework like xUnit) to write short contracts for each unit of code. What constitutes a unit is the subject of some debate but unit tests are typically very small, independent tests than run quickly and assert the behaviour of a particular function or object. For example a function that adds numbers together might be tested by checking that 1 plus 1 returns 2.

While this might sound trivial, unit tests are useful for debugging code, particularly complex code that changes frequently. A developer may make a change that they they think does not affect the output of a function, but a well written unit test will prove that the functions contract has not been broken. Unit tests also help design and refactor complex code as they force programmers to break down the logic of their application in to manageable chunks.

Verification and Validation

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

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

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

> On a large scale project I worked on as a test manager, we complained to the development team that our documentation was out of date and we were having difficulty constructing valid tests. They grumbled but eventually assured us they would update the specification and provide us with a new version to plan our tests from.
>
> When I came in the next day, I found two programmers sitting at a pair of computer terminals. While one of them ran the latest version of the software, the other would look over their shoulder and then write up the behaviour of the product as the latest version of the specification!
>
> When we complained to the development manager she said "What do you want? The spec is up to date now, isn't it?" The client, however, was not amused; they now had no way of determining *what the program was supposed to do* as opposed to *what it actually did*.

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

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

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

A note on the 'V-model' : There exists a software testing model called the V-model, but 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.

Errors will happily propagate from phase to phase, chewing up time and effort (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.

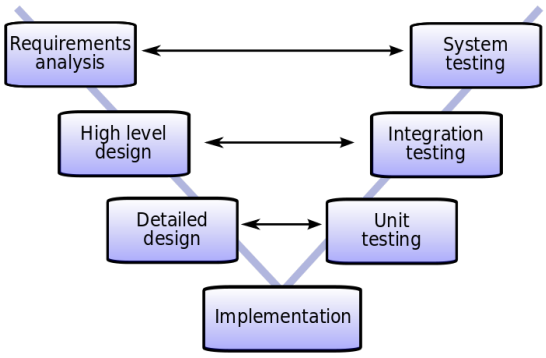


Figure 9: The (horrible, horrible) V-Model

The Role of Functional Testing

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. Getting a shared understanding of what a requirement actually means is central to building good software and is the obsession at the heart of many practices.

Alpha and Beta

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

Typically these milestones are known as "alpha", "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.

> In the days before the Internet and the World Wide Web there was a third stage of testing, the "gold master". It has fallen out of favour but this was the version of software that was considered complete and accurate enough to press the 'gold master' or CD master from.
>
> All your shipping CD's would be copies of this golden master and therefore had to be as close to perfect as possible. In the days of sneaker-net and physical media, this was the last chance to get things right before you incurred the substantial cost of shipping your software to all your customers. Not everybody got it right.

White Box testing

White-box or glass-box testing relies on analysing the code itself and the internal logic of the software.

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 or security flaws.

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 *pairing* or *mob programming*.

Originating from the Extreme Programming (XP) practice known as peer programming, pairing sees modules of code 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 time.

There have been studies into whether pairing does in fact achieve higher quality code or deliver results faster but the results are largely inconclusive. But coding is a largely solo discipline and there are some undeniable (although unproven) advantages to pairing. As a learning or teaching tool it is very effective. When there is a new method be learned or the complexities of new code to be explored, pairing is an excellent way to share that knowledge and develop a shared team practice. Pairing also appears be useful in in promoting refactoring and thoughtful design since they force individual assumptions to be examined.

> Many programmers are self taught, both initially and then on the job. My own opinion is this leads to an independence that is both worthy and worrying. Worrying because their seems to be a resistance to any knowledge coming from outside and asking for help is seen as a weakness. Coders are so used to 'working it out for themselves' that they tend to resist external ideas. Pairing might be a neat way to break that mindset.

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 code 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.

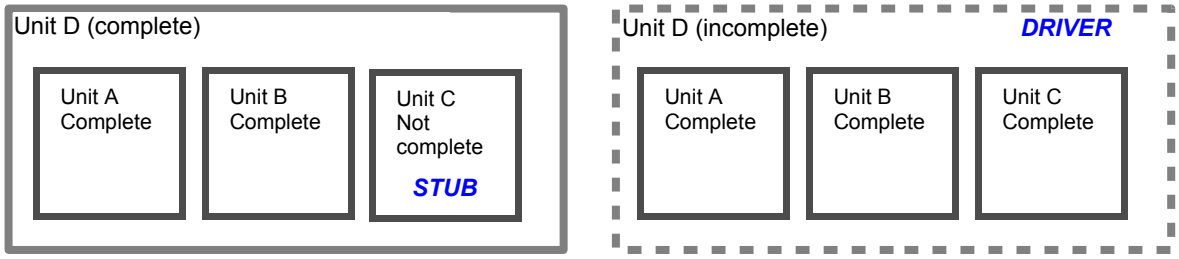


Figure 10: Integration Testing

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.

> In some 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 as one wag told me : "Save It, Testing!"
>
> 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.
>
> Even more bizarrely some organisations have a dedicated UAT team. UAT as we shall see stands for "User Acceptance Testing" and should be a validation check carrier out by actual users to receive the software into production. But organisations often find the distraction of large scale testing to disruptive to their operational teams and so they set up dedicated test teams to 'represent' the users – as if the other teams couldn't do that.
>
> In truth, parallel test teams often points to a lack of trust between silos of the organisation.

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 accepted by the client or customer.

This is when 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 extend to processes outside of the software itself to make sure the whole solution works as advertised in the context in which it is intended.

Other forms of acceptance testing include *Factory Acceptance Testing* (FAT) where the software is tested by a vendor to the client's specification before it leaves the factory or vendor's premises. *Site Acceptance Testing* is the next phases where the test is conducted at the customer's site (on their infrastructure in software terms). Some time around SAT, ownership of the software transfers to the customer, cheques are signed and the software becomes theirs (but subject to whatever support agreement is in place). Sometimes this is part of SAT, some time it is a separate step called 'commissioning' or similar.

While other forms of testing can be more 'free form', the acceptance tests 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. Often there are contractual treatments required for defects found during an acceptance test, so accuracy and detail are at a premium.

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 or low quality software development, the acceptance test becomes the only 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, to evaluate properly.

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 tools that 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 newly minted 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 but it also costs 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 Hidden Cost

The hidden costs of test automation is 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 fairly constant but the underlying functional code changes then your tests will still run and 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 to automate the testing for a piece of software through a rapidly changing interface is a bit like trying to pin a jellyfish to the wall.

If your product has an API or some kind of service layer then there's a much better chance that you'll be able to test core functionality through that interface, but it's still no guarantee. Only by building the maintenance of automated testing into the mainline process of software development (qqv TDD/BDD) can you hope to keep your tests in synch with your application code.

> The funniest business case I have ever seen for automated test tools ran like this :
>
> 1. Using manual testing, we find X number of defects in our software
> 2. It costs $Y to find and fix these defects each year (= tester time)
> 3. We can buy an automated test tool for $Z/year
> 4. Since $Z < $Y we should buy the tool and, *as a bonus, it will find less 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! This was someone's idea of a real business case!

What is Automated Testing Good For?

Automated testing is particularly good at :

- *Load and performance testing* – automated tests are essential for load and performance testing; it's generally not feasible to have 300 users manually test a system simultaneously.
- *Smoke testing* – a quick and dirty test to confirm that the system 'basically' works, if coupled to an automated deployment process they can be triggered with every build or release.
- *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* – including manual tasks that are tedious and prone to human error (e.g. checking account balances to 7 decimal places).
- *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 if they have altered functionality delivered in previous increments).

Note that this last point is only true where the interface code that the automated tests uses is fairly stable. If it is the interface code that changes all the time then it your tests will always require refactoring, adding overhead to your testing cycle. The optimum scenario is where the underlying functional code changes, but the interface remains stable (and not interface does not necessarily mean GUI, it might mean API or service).

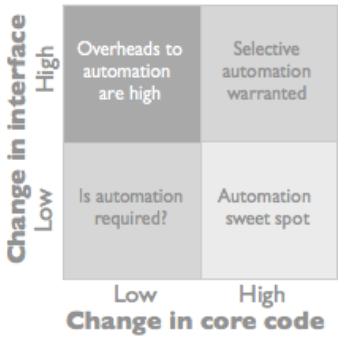


Figure 11: Auto testing payback

Principles of Test Automation

Test automation should be treated like software in its own right.

Automated tests must be managed just like your application code (in fact, exactly like your application code). They must be source controlled alongside the code they test, they must have standards and be reviewed, they must be deployed consistently and correctly versioned.

The 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.

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 wrong, 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 *scale well* and be able to run quickly, otherwise they quickly use their utility. Automated tests which take a long time to run should be refactored into smaller independent units that can be run quickly.

Test code should follow rigorous *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.

If at all possible the *tests should be written in the same coding language as the application code* – this reduces cognitive friction in the team and increases reusability. If this is not possible then tests should be written in *a generic programming language* to avoid the pitfalls of a proprietary format.

Testing the design

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

The purpose of evaluating a specification is threefold:

- To make sure they are accurate, clear and internally consistent (verification)
- Making sure they are consistent with all previous and subsequent documents
- To evaluate how well they reflect reality and match end-user expects (validation)

If the requirements are poorly defined then you cannot rely on them for testing and must seek other sources of truth.

In a specification or statement of requirements there should be a clear distinction between general discussion and the requirement itself. Each requirement should contain a statement which indicates what the software "should" or "must" do – this is called an assertion.

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.
- *Measurable* – a requirement which uses comparative words like "better" or "faster" must specify a quantitative improvement with a specific value (100% faster or 20% more accurate).
- *Testable* – from the stated requirement it must be clear how it can be proved true of false (see *acceptance criteria*). A requirement that is unprovable is irrelevant to testing.
- *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 design specs should not talk about "why" to do things.

> This is once again why I'm growing to love TDD/BDD (qqv). In BDD your requirements are your tests and your tests directly inform your code, so you have the tightest of all possible feedback loops. If a requirement is inaccurate or poorly specified or out of date, it will quickly become highlighted as tests fail. Because it is coupled to code it is easy to do a 1:1 comparison between specification and code and keep them in synch.

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. But as technology becomes more ubiquitous the bar is getting higher; users expect an intuitive interface.
- *Selection of end-user evaluators is critical* –evaluators must be 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.

Research has shown that you need no more than four or five evaluators in a session. 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 it could be put down to personal quirks, with five it is beyond doubt.

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.

> I enjoy watching developers who take part as observers in usability studies. 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 design than your highly-paid self.
>
> Typically developers sit through the performance of the first evaluator and quietly snigger to themselves, attributing the issues to 'finger trouble' or 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".

A/B Testing & Testing in Production

One form of 'usability' testing that has become popular in the cloud era, is the concept of A/B testing.

In A/B testing a feature is rolled out to a subset of the population to monitor their interaction with it. This can then be tweaked so that it delivers maximum value before it is distributed to the wider population. Or two version of the feature may be rolled out simultaneously to two different user groups in order to capture data and decide which implementation is more effective.

Because you are testing with your real-world end users the quality of feedback is unimpeachable. It is no long someone's opinion as to whether or not a feature has value. The users will tell you by their adoption or rejection. This can lead to rapid experimentation and hypothesis testing that will allow and organisation to respond very quickly to changes in market or strategy.

Needless to say this requires small changes and rapid deployment pipelines (and equally rapid backout). This is the area of DevOps and continuous delivery known as *blue/green deployments* (qqv).

Using this method you can quickly and easily deploy small features to a subset of your customers and test their interaction with it. More sophisticated versions of this allow the target set of customers to be tailored by geography or usage or some other significant metadata.

> As an interesting aside, this has led several large scale companies to experiment with 'testing in production' or 'canary testing'. In this changes are deployed straight to a small production user set on a very rapid basis (multiple times per day) with very little testing and their impact monitored. If the overall impact is detrimental, they are backed out and discarded or reworked. If the impact is positive the change is rolled out to all customers.
>
> At first it may sound crazy, but think about it. If you can guarantee that a change will affect no more than 1-2% of your customer base (or even less if you prefer). Then how much (internal) testing do you really need to do to assure yourself it won't have an adverse impact? And how long can you really wait before you get direct feedback?
>
> I can just about hear traditional waterfall project/product managers minds exploding as I write this. "What about risk, what about reputational damage?" they cry, clutching their copies of Prince2 to their hearts.
>
> What they fail to recognise is that in the modern 'digital' landscape, failing to respond to a customers needs, or to a competitor's innovation, may constitute lethal reputational damage.
>
> As a good friend of mine said, "speed is the new black".

Performance Testing

One important aspect of modern software systems is its performance in multi-user or multi-tier environments. To test the performance of the software you need to simulate its production 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. In this case a production-like system must be used and performance in production extrapolated from this.

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.

And having a baseline is important. Without knowledge of the 'pre-change' performance of the software it is impossible to assess the impact of any changes. "The system can only handle 100 transactions an hour!" comes the cry; but if it only handles 50 transactions an hour at the moment, 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.

Types of Performance Testing

Load testing is the simplest form of performance testing. A load test is usually conducted to understand the behaviour of the system under a specified load. This load can be the expected concurrent number of users on the application performing a specific number of transactions within the set duration. This test will return the response times of all the important business transactions. The database, application server, etc. are also monitored during the test to assist in identifying bottlenecks in the application architecture.

Stress testing is normally used to understand the upper limits of capacity within the system. This kind of test is done to determine the system's robustness in terms of extreme load and helps application administrators to determine if the system will perform sufficiently if the current load goes well above the expected maximum.

Soak testing, also known as endurance testing, is usually done to determine if the system can sustain the continuous expected load. During soak tests, memory utilization is monitored to detect potential leaks. Also important, but often overlooked is performance degradation, i.e. to ensure that the throughput and/or response times after some long period of sustained activity are as good as or better than at the beginning of the test.

Spike testing is done by suddenly increasing or decreasing the load by a very large number of users, and observing the behaviour of the system. The goal is to determine whether performance will suffer, the system will fail, or it will be able to handle dramatic changes in load.

From Wikipedia: Software Performance Testing

The Purpose of Test Planning

As an organised activity, testing should be planned, to a reasonable extent.

But test planning represents a special challenge.

The aim of testing is to find bugs in the product and so the aim of planning is to plan how to find the bugs in the product. 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 (see A/B 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, budget and your risk appetite allows.

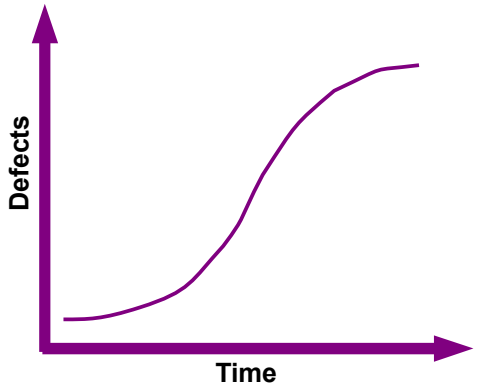


Figure 12: Typical defect discovery rates

Risk in Software

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 product. 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?

As already mentioned, complexity is a good proxy for probability – the more complex the code the more likely it is that an error will be introduced. In existing code, the historical location of defects can be a good guide to likely areas of complexity. In new code, effort or duration can be good proxies for complexity – the more complex or difficult the code, the longer it will take to write.

But impact or criticality is much harder to measure. Business value can be a proxy for criticality, but not always. Business users might not regard the phone or the internet as highly valuable but they are critical pieces of infrastructure that they would struggle to operate without them.

What we need therefore is a recursive model of the software to help guide our testing.

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.

Testing can then be seen as an attempt to evaluate as many of these axes as possible.

Remember we are no longer seeking the impossible 100% coverage but merely 'best' coverage, based on risk and budget.

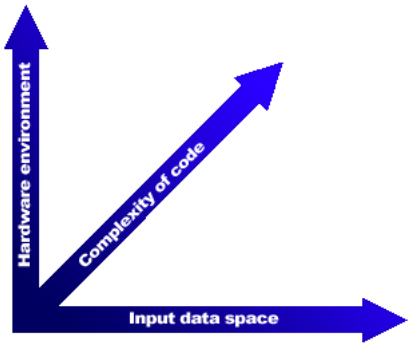


Figure 13 : Software dimensions

Outlining Taxonomy

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 enumerated as much of each axis as you possibly can (you can always add more later).
4. Recurse into each sub-area of each axis if necessary in order to build a taxonomy of your 'testing space'. For example if you have a large monolithic application it may warrant breaking it down into two to three levels of functionality until you reach the right 'granularity'. Each axis will have a different taxonomy, platform for example might map browser, O/S and hardware combinations.

Common Axes

The most common starting point for test planning is a functional decomposition based on a technical specification or an examination of the product itself 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 some other reference
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/software platforms, form factors
Configuration elements	Any modifiable configuration elements and their values
Performance	Different aspects of performance in different scenarios

Test Identification

The next step is to identify tests 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 tests 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.

For now, don't worry too much about the detail of how you might test each point, simply decide 'what' you will test.

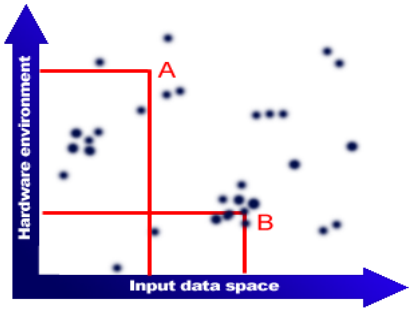


Figure 14 : Test case identification

Test 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 tests "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.

You must try to achieve a balance.

Your aim should be to provide a broad coverage for the majority of your '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 get a good enough estimate by assigning an average execution time 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 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? what is the organisation's risk appetite? 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.

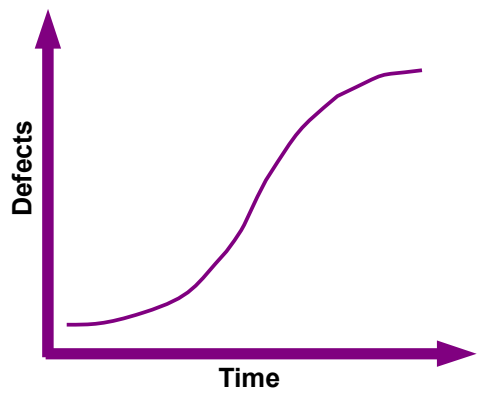


Figure 15: Defect discovery rate again

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. The maximum amount of information is generated at a failure rate of about 50% - if your tests are failing less than 50% of the time then they are not adding maximum value.

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.

This is where automation can be your friend. If the incremental cost of running the test is small then you can continue to run it without loss (regression testing being the prime example). If the cost of the test is high (say with intensive manual testing) then you should consider diverting your effort elsewhere.

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* active while you focus the *deep coverage* 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 and risk appetite)
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 or contractual obligation for example) mean that a script-heavy approach is mandated.

On the other hand, in consumer software development, a looser approach is normally taken. Since speed-to-market is more important than the risk of a single 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 unscripted or *exploratory 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. Often the amount of time spent on scripting can actually exceed the amount of time in execution. If you have an experienced tester with the right set of tools and the right mindset, it is more effective and more cost to turn them loose to find some bugs right away.

This concept is almost heresy in some camps.

There is also an important legal aspect to consider 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. Whether or not you script every case is immaterial. What matters is how you can demonstrate that you prepared your testing in a methodical and professional manner.

> I sit somewhere in the middle of this debate.
>
> Preparation is essential. Some scripting is good, but too much is not – even when it is mandated. 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 child could then execute the scripts and find defects.
>
> This is a dubious conclusion at best.
>
> But sometimes all you have are children and it is a more efficient use of resources to allow your experienced testers to script, and use the kiddies to execute many times over.
>
> In the end, don't let adherence to a particular methodology blind you to the possibilities of other approaches. Train your testers in all the possibilities and let them use their judgement.

Test Cases

Let's assume you have to document your tests; a test case documents a test, intended to prove a requirement or feature.

The relationship is not always one-to-one, sometimes 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 usually 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, 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 it would 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 have brains they will attempt all the possible input values in an attempt to provoke an exception.

Elements of a Traditional Test Case

The following table lists 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.

Acceptance Criteria

In the agile world, test cases have given way to "acceptance criteria'. Because requirements in agile are captured as 'user stories' or feature descriptions on cards, the traditional form of test cases are deprecated in favour of short acceptance criteria (often written on the back of the card).

Broadly speaking these are logical statements that can be proved true when the story/feature has been delivered successfully. As with the original story, agile prizes discussion over documentation so the acceptance criteria stand as signposts for a wider conversation. Only through conversation, so the logic goes, can consensus be achieved and so it is better to provoke the conversation than avoid it by writing down an incomplete set of assumptions.

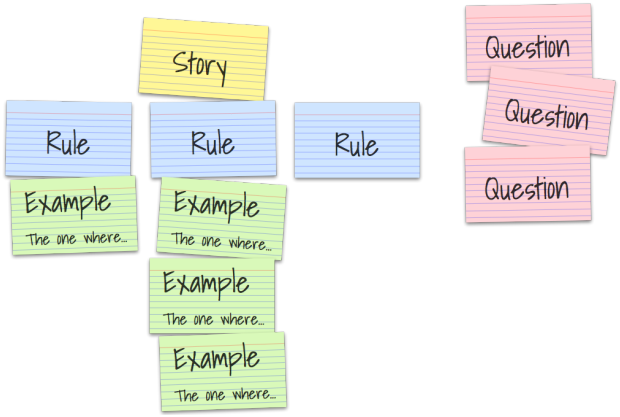
Many new agile teams struggle with acceptance criteria and specifying them at the right level of detail. Too much detail and acceptance criteria just become highly scripted tests, too little detail and it can be unclear exactly what constitutes a successful implementation of the requirement.

Example Mapping

A useful and structured process for delivering the right amount of detail is *Example Mapping*.

Example mapping js a collaborative process in which a team uses three constructs to define requirements:

- concrete *examples* which illustrate the characteristics of a rule
- *rules* that summarise a group of examples
- *questions* that denote areas of uncertainty with rules or examples



The group uses index cards to map out the requirement and its associated rules and examples visually, thus providing a 'map' of the requirement. Requirements that have more than 5 rules for example should probably be split up.

Fit(nesse)

The Fit framework was [developed by Ward Cunningham] as a collaborative testing methodology and evolved into an automated integration testing framework (Fitnesse). Fit revolves around the use of exampled mapped out in decision tables that can be tested in the application. The Fitnesse interprets these example tables using 'fixtures' written in a common programming language to verify the behaviour of the application under test. The most common implementation of Fitnesse uses a wiki to create the decision tables, making it easy to create and update.

Basic Employee Compensation

For each week, hourly employees are paid a standard wage per hour for the first 40 hours worked, 1.5 times their wage for each hour after the first 40 hours, and 2 times their wage for each hour worked on Sundays and holidays.

Here are some typical examples of this:

Payroll.Fixtures.WeeklyCompensation			
StandardHours	HolidayHours	Wage	Pay()
40	0	20	$800
45	0	20	$950
48	8	20	$1360 *expected* $1040 *actual*

Figure 16: A sample FIT table from http://fit.c2.com/

Gherkin's Given-When-Then

One framework for specifying executable specifications that has become popular is Gherkin's *Given-When-Then* construct. Gherkin is a plain english pattern language used to specify examples which illustrate the rules behind a requirement.

An example is known as a *scenario;* the structure of a scenario is:

> *Given <a set of pre-conditions or context>*
> *When <an action or trigger is initiated>*
> *Then <the program produces a result or an outcome>*

Each of the clauses in a scenario can have multiple parts and advanced Gherkin includes the option of example tables similar to the FIT tables explained above. '

```
Feature: Refund item

  Scenario: Jeff returns a faulty microwave
    Given Jeff has bought a microwave for $100
    And he has a receipt
    When he returns the microwave
    Then Jeff should be refunded $100
```

Figure 17: A sample Gherkin scenario from cucumber.io

Scenarios in Gherkin can be coupled to an interpreter and the programmer can write 'step code' to interpret the scenarios in the context of their application and generate automated tests (see right).

The aim of Gherkin/cucumber is to deliver *executable specifications*; requirements which can be automatically proved in the application. When linked to development methodologies like BDD this provides a virtuous feedback loop where the specification directly validates the application's behaviour via an automated test.

This minimises maintenance issues with automated tests, provides a set of tests that are directly traceable to requirements and gives developers a way to constructively refactor their code.

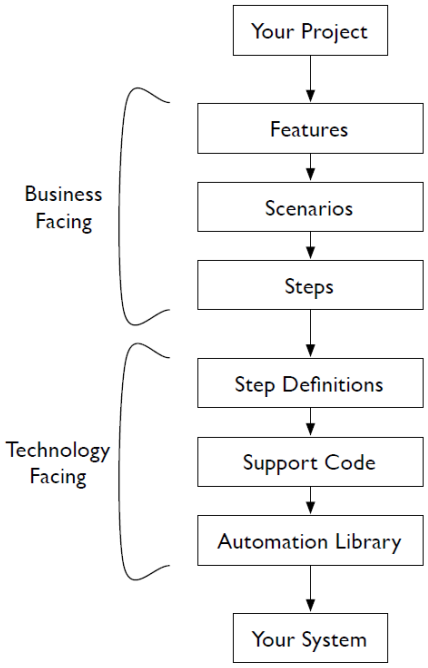


Figure 18: The Cucumber testing stack

Exploratory Testing

In exploratory testing, an individual literally explores the software without benefit of a pre-scripted plan.

> I think it might have been Cem Kaner that said *test plans/scripts are like a tour bus, they can be nice way to see what's there, but don't forget to get off the bus and take a look around*.

The term exploratory is used to differentiate the activity from the random walk of an average users. Explorers have purpose and a method. They keep a map of where they've been and make conscious choices on where to spend there time. So it is with exploratory testing.

Using the outlining technique explained in test planning (qqv) the exploratory tester can record functions, acceptance criteria and context as they methodically explore the software.

Normally exploratory testing is broken up into time-boxed sessions with specific intent. The intent of the first session could be to map the overall functionality, the second to deal with user registration etc, etc. The intent of an exploratory session is captured in a *test charter*. The charter normally defines the goal of a testing session, any setup or pre-requisites required (especially data), any specific references (like requirements or specifications) and contextual information (target platform, background jobs etc).

in this way, exploratory testing becomes a repeatable process without imposing artificial constraints on the initiatives of individuals doing the testing.

Session Based Testing

Session based testing is a way of organising exploratory testing that addresses some of its limitations, notably measuring progress and record keeping. In addition to the pre-test charter an session based tester keeps a 'session log' of their testing, laying the foundation for recursive test planning.

A typical session log will contain details of areas covered (in comparison to the charter), risks identified, bugs logged, questions raised and any metrics, like the time spent on each activity. This log then can be used in a debrief and as the basis for the next session with this charter.

The cumulative output of session logs can be used to estimate progress and to identify areas of focus or concern. If you have completed some kind of test outline or plan then you can tick off progress against this plan by test session.

```
CHARTER : Analyze MapMaker's View menu functionality and
report on areas of potential risk.

#AREAS : OS | Windows 2000 | Menu | View

START : 5/30/00 03:20 pm

TESTER : Jonathan Bach


TEST NOTES: I touched each of the menu items, below, but
focused mostly on zooming behavior with various combinations
of map elements displayed.

BUGS : 1321; 1331

ISSUES/RISKS :

How do I know what details should show up at what zoom
levels?

I'm not sure how the locator map is supposed to work. How is
the user supposed to interact with it?
```

Figure 19: From "Session Based Testing" by James Bach

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 of your 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 and are 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 the wrong 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
- Someone moves the end date due to business or market pressure
- The customer or the product owner changes their mind
- The company/world/industry/market changes

There are three basic ways to deal with this :

1. *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.

2. *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).

3. *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 everyone aware 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 sometimes 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".

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

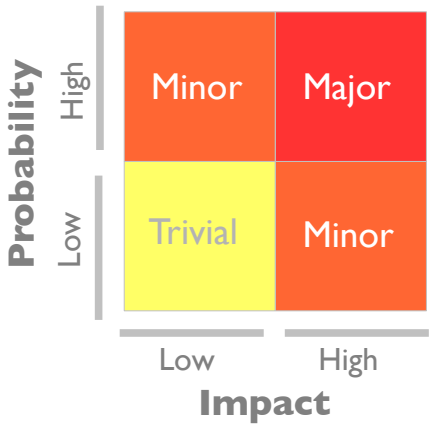


Figure 20: Relative severity in defects

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 :

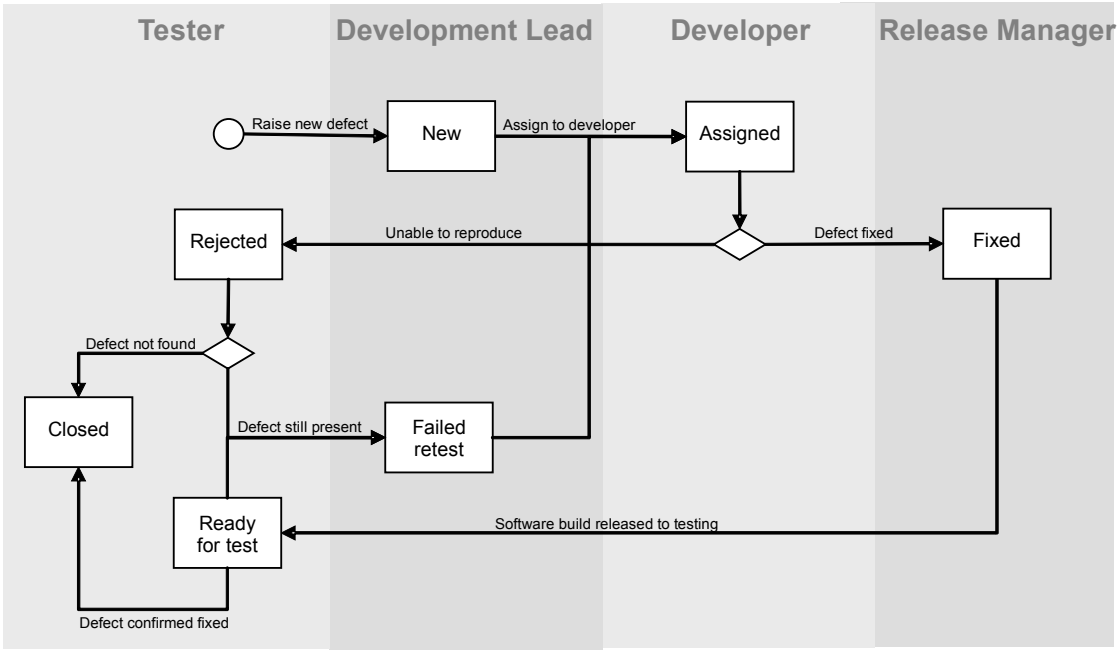


Figure 21: Defect life cycle swim lane diagram

Elements of a Defect Report

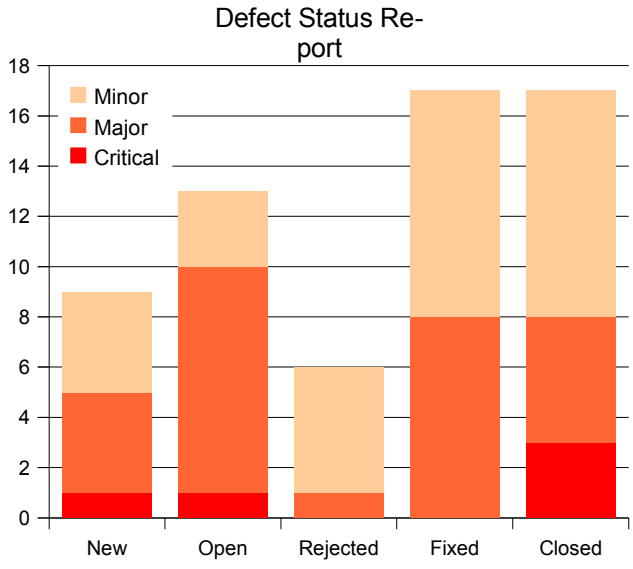
ITEM	DESCRIPTION
Title	A unique, concise and descriptive title for a defect is vital. It will allow the defect to be easily identified and discussed. *Good : "Closed" field in "Account Closure" screen accepts invalid date* *Bad : "Closed field busted"*
Severity	An assessment of the impact of the defect 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"

Defect Reporting

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.



Major	4	9	1	8	5
Minor	4	3	5	9	9

Figure 22: Defect status report

By far and away, the best defect report 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.

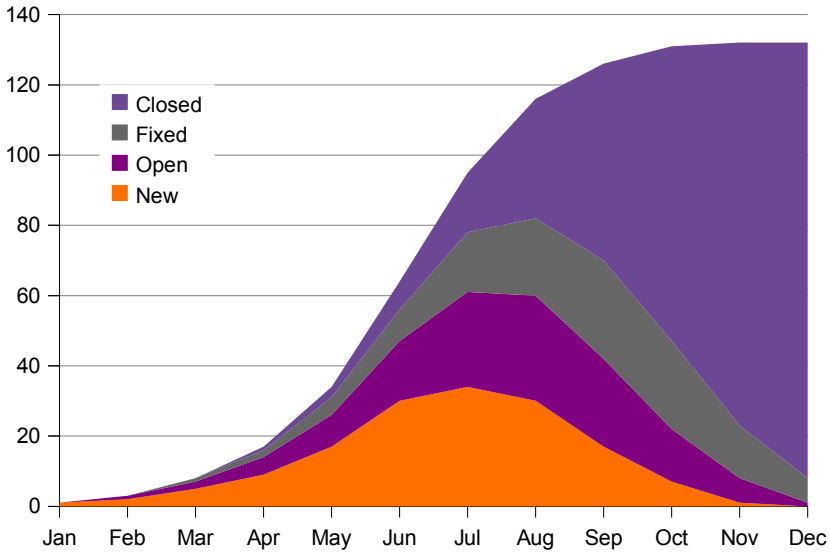


Figure 23: Defect trends over time

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 24: 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 of Quality and Efficiency

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, an organisation or an individual. Far too often they are applied without any deep understanding of the metric, general statistics or organisational psychology. There has been a strong realisation in recent years that metrics should be useful for individuals as well. The use of personal metrics at all levels of software development allow individuals to tune their habits towards more effective behaviours.

> Edwards Deming, the father of quality processes in manufacturing reputedly said "When a measure becomes a target, it stops being a measure". What this means is that if you link a persons reward to a particular target, they will strive to hit that target regardless of the cost.
>
> Metrics are useful measures for guiding informed decision making, but they shouldn't be confused with KPI's or performance targets. By observing the trend of particular metrics teams can adjust their approaches to optimise their processes. But tying metrics to personal or team performance measures is likely to end in disaster.

Defect Injection Rate

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 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".

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?

This cannot be used as measure of developer skill. Each problem in software is different and potentially each solution is different, so using this number as a performance indicator will only lead to problems.

> If you truly want to measure developer performance, consider measure how well they follow your defined processes. It is might contention that the single largest source of defects is people failing to follow well ordered processes in software development.

Defect Discovery Rate

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 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 efficiency of individual testers.

In a 'script-heavy' environment notions 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; or 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 *remove* 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 provide a comparative measure for a particular test effort is easy – simply divide the total number of hours spent in testing by the total number of defects (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.

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.

> 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.

Cycle Time and Takt Time

If you grant the premise that the purpose of software development is to deliver working software to the customer, then measuring how long and how often you deliver that software is of value. Coupled with other raw numbers *cycle time* is a powerful metric to help you streamline your development and testing process.

Cycle time for the overall software development system can be further broken down into component parts to identify and resolve bottlenecks. These subsystem should be broad brush phases like design time, coding time and testing time. The overall system will run no faster than the slowest of the components.

In conjunction with cycle time you should also measure (and manage) *queue lengths*, i.e. how many items are sitting waiting for any particular step. If queues grow to long you should consider purging low value items from the queue. If this happens regularly you should consider establishing alternate paths or lanes for different items to break the deadlock.

If you have large queues, either before development or in subprocesses, it might be worth considering another metric: *takt time*. Takt time is a simple measure of customer demand. If takt time exceeds your cycle time then your software development process will never be able to keep pace with customer demand. Takt time is calculated by dividing the available work time by the number of units of demand in that time.

For example if you deliver software on a weekly basis than the takt time would be 40 hours, divided by the number of requests per week. So in an organisation where users log 200 requests per week, the take time would be 5 hours. This means that the cycle time of the software process must be less than 5 hours, or the team will never keep pace with demand.

Note that both takt time and cycle time are not related to the amount of effort required, but are measures of elapsed time. Often the delays in a process are mostly waiting in queues as a result of bottlenecks.

First Pass Complete & Accurate

Another useful measure (this time of quality) is first pass complete and accurate. This is a measure of how often, on average, a work item arrives at a process step with all the information required to complete it. Normally expressed as a percentage, it is a good measure of the quality of your process as it reflects all the hidden rework buried in your system.

Normally expressed as a percentage, the first-pass-complete-and-accurate score for each sub process can be multiplied together to give you an overall first pass value for your software development process. A score near 100% means your process is high quality, with

Combining cycle time and first pass produces a *value stream map* of your process:

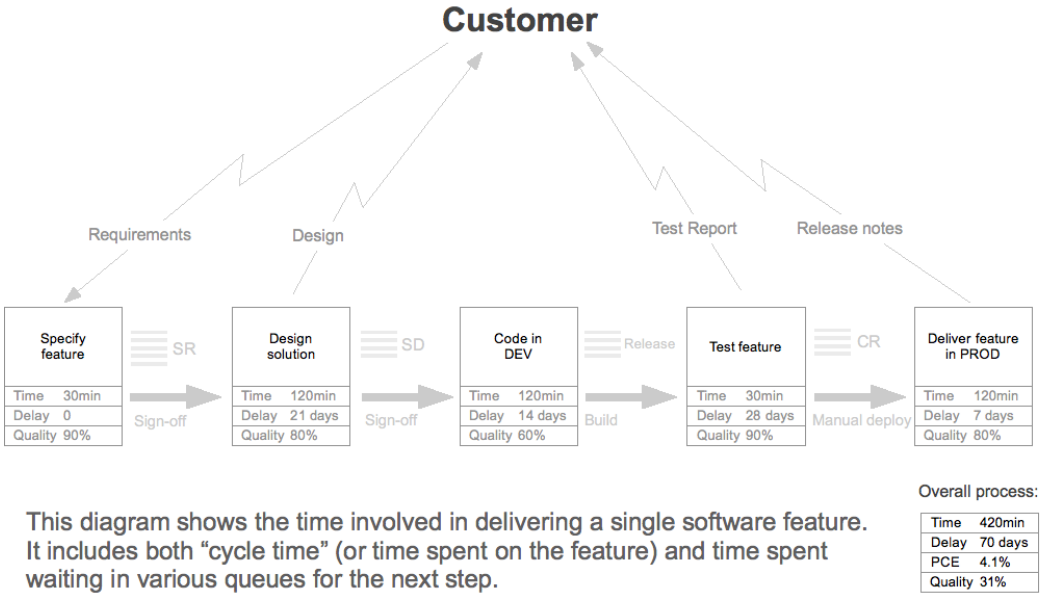


Figure 25: A value stream map of the software development process

Release and Deployment Management

Releasing your software is the process of introducing it to your customers and, like most first dates, it can be traumatic for both sides. You need a strategy and a plan for releasing software in a controlled manner.

Deployment is the technical aspect of moving your software to a new environment. A release to production from a technical point of view is just a deployment to the production environment.

When every change is automatically deployed to production, you've reached the nirvana of continuous delivery.

Release Strategy

How and when you release your software to customers forms the basis of your release strategy. At a basic level it should describe how often software is released to customers and what those releases should contain (e.g. a bug fix release every week and a quarterly feature release containing major functionality).

A release strategy will also detail the approaches, mechanisms, approvals and context of releasing software to give your team a shared understanding of how releases should be approached.

Things you should consider in your release strategy:

Item	Description
Release Cadence	How often do you plan to release? What will each release roughly contain? Are their specific release windows to be observed (so as not to adversely affect customers)?
Deployment Pipeline	What are the stages the software must progress through to reach production? Often these are specific environments, with each environment representing another step in the deployment process.
Authority	Who has the authority to approve the deployment to a specific stage in the pipeline? How is the request made and approval given?
Configuration Management	How is your software's configuration at deploy time captured? How is the configuration of the underlying infrastructure captured? How will this be updated during a release?
Business process	What business processes are related to a software release? What communications or marketing must be done? What training? What change management processes or audit principles must be observed?
Disaster Recovery	In the even of a serious issue during deployment, what is the plan to recover the software to a known-good state? What information needs to be supplied to ensure the DR capability is maintained?
Logging	How will changes during a release be logged? How will release activities be coordinated?
Monitoring	What monitoring is expected at each stage in the pipeline? Where does this information go? Who consumes it?
Transition to Support	What assets or information does the support team require in order to support the new release into production? How will they be given this information? Will they have the chance to review it prior to release?
Capacity Management	How will the release impact production performance? What sizing or capacity information needs to be considered to upgrade production to ensure performance is adequately maintained?
Defect or Warranty plan	After the release, if users report errors, what is the process for fixing them? Is there a special warranty process in place or does it follow the normal find-fix routine of support calls?

Release Planning

Release planning consists of allocating potential changes to a release.

This can occur at many different stages in your software development lifecycle, depending on your release strategy. A just-in-time strategy may see changes being developed and tested and released at the next available opportunity. A more structured release strategy may see changes being allocated to scheduled releases based on complex criteria like business need, interdependency and economies of scale.

A change which is being considered for a release is known as a *candidate*, the list of candidates in a release is known as the *manifest*.

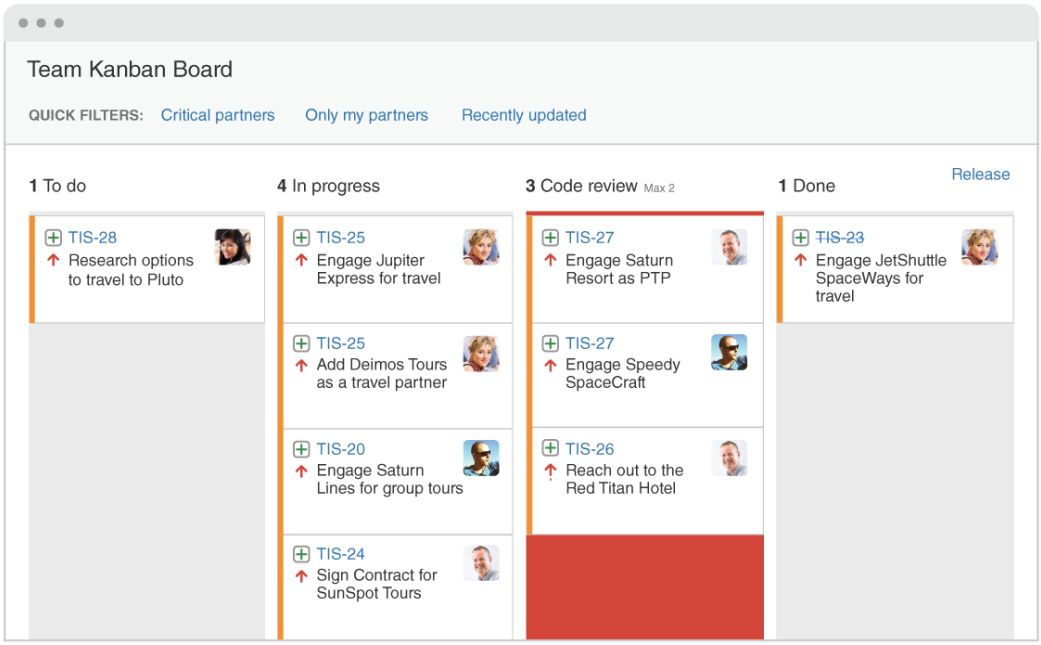
Based on your release strategy, each candidate will have to achieve a number of milestones or criteria to be included within the release (this is true whether your release is just-in-time or pre-planned). Typical milestones include different levels of testing and quality control or some kind of business approval. Scheduled based releases will also have to meet time constraints.

When a candidate does not meet the criteria for a release there are a number of options:

1. The release can be delayed to allow the candidate to 'catchup'
2. The candidate can be dropped from the release and allocated to a later release
3. The criteria can be relaxed to allow the release to proceed with the candidate

Often candidate changes have interdependencies and so the decision to drop a particular candidate must be taken carefully as it may effect the overall release.

A release plan tracks the key dates, criteria and candidate status for each release.



This is a screenshot from JIRA of a *kanban board*. Kanban is a just-in-time scheduling tool that has been used in lean manufacturing for many years. It is based on the principle of 'pull' workflow, where customer demand determines what is delivered and scheduling information flows backward up the value chain (as opposed to being centrally controlled).

In this board software changes enter the workflow at left and 'flow' left to right across the board through defined states until they reach the 'done' column. At that point they are ready to be released (the Release button above 'done' triggers an automated build and deployment).

The pull-based system works by imposing Work-In-Progress (WIP) limits on each state and work is only allowed to move to the next state, when there is capacity available. So nothing else can enter "Code Review" until TIS-27 or TIS-26 move to the "done" column. Work items are pulled from the right by capacity, not pushed from the left by a pre-destined plan.

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.

A release should be verified for completeness as well.

Often releases can be deployed 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.

> 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 same principle can be applied in software development.

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 Required 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...

Deployment Planning

Any piece of software is part of a complex system of dependencies, including it's internal components and those of whatever platform it runs on. Building that platform and deploying the software must be done in a logical sequence to address all of the dependencies.

A deployment plan specifies the sequence of activities required to build the application on its platform environment and who should complete them.

A deployment plan should include details of :

1. Any pre-requisites or preparatory steps that must be complete before deployment begins
2. How to deploy the software for the first time to a new environment
3. How to migrate any existing user data or configuration data to the new application
4. How to smoke test the software in the current environment (qqv)
5. How to backout or rollback the software if the deployment fails
6. How to start, initiate or bootstrap the application once it has been installed

Good deployment planning should assume a blank slate, and thus be able to recreate the software in the correct state *no matter what has gone before*. A good, clean deployment process should require little or no manual intervention in order to deploy software. Every manual step introduces the possibility of human error or variance and makes it unlikely that it will be reproducible in the future. At it's logical extension it is possible to achieve a *one-click hands-off deployment* using an automated deployment pipeline and the principles of infrastructure as code (see below).

In general the same deployment process or plan should be used in all environments (with only minor variations). The greater the variance between deployment for pre-production environments and for the production environment, the greater the risk of defects arising due to mismatched configuration or mistakes made during the deployment process.

Rollback

Deployments will go wrong, mistakes will be made. So identifying the process for *rolling back*, to a previously known-good version of your software is essential. One method of rolling back is to take a snapshot or image of various configuration items before deployment starts. Rollback is then a 'simple' process of deploying the snapshot (this is very common with databases).

Another method of rollback is to trigger a previous deployment from scratch. If your deployment is automated and your last deployment succeeded then rollback can be as simple as re-deploying the previous version (although user data and configuration make this a little more complicated).

A third but far less frequent method is to reverse individual components dependent upon the problem.

Each option has its advantages.

The snapshot method is quick and simple, but a snapshot is often a binary image with no real granularity of detail in the configuration items. Thus, if an item has been configured manually, it will be impossible to reproduced, say in a disaster recovery situation on new hardware. This leaves you with an unreproducible production system which is a risk in itself.

The re-deployment method is more difficult and takes more time but ensures a robust and repeatable process. A variation on this is where infrastructure is software configurable (e.g. virtualisation) and the entire (old) platform is deleted when a new version is deployed.

> An anti-pattern common in many organisations is the manual fix in production, otherwise known as a patch or truth be told, as a hack or kludge. Fixing things directly in production breaks the chain between your production environment and your development/deployment pipeline. If you are subsequently required to make changes to production or to rebuild it for some reason you will have no reliable way to do so. Such manual patching has a horrible tendency to grow like barnacles on an otherwise smooth deployment process and drag the whole ship down.

Configuration Management

How do you track and control all the nuts-and-bolts of a complex piece of software?

Take for example a system where ten developers are working in parallel on pieces of code.

When you go to build and release a version of the system how do you know what changes have been made and tested? How do you know what version of code is in what server? When you release multiple versions of the same system to multiple customers and they start reporting bugs six months later, how do you know which version the error occurs in?

Configuration management tracks and controls all of the assets required to deploy your software. This includes source code, binary code, configuration and data files and, in some cases, even infrastructure.

At the code level a *version control* system will have label each component of your source code with a unique identifier and be able to identify the changes made to it at any point in time. This can then be matched to binary or compiled versions of the software so that you know which version of your application contains which changes or fixes.

Sample version control table :

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)

Version control is very important in the testing process as you need to be able to confirm when and where a defect occurs (and when and where it was fixed!).

According to Jez Humble and David Farley in the seminal "Continuous Delivery", a good configuration management strategy allows you to:

1. Exactly reproduce any of your environments, including application software;
2. Easily make incremental changes to any of the items in these environments;
3. See each change that occurred in an environment and see who made it and when;
4. Satisfy all the compliance or audit requirements that you are subject to;
5. Make it easy for every member of the team to get the information they need and to make the changes they need to make.

Infrastructure as Code

One very important development in recent years has been the evolution of the concept of *infrastructure as code*. With the advent of cloud computing and advance virtualisation, it is now possible to treat hardware as 'ephemeral'. That is, a virtualised server can be spun up in no time at all from a template or set of configuration files. It can also be turned off or deleted when not in use.

The advantages of this approach are multiple: each server becomes 'immutable' or an exact copy of the template – no longer do you have to worry about the inconsistencies of individual 'snowflake' servers; the deployment process can be automated down to the infrastructure level; unused servers are simply turned off, saving money and maintenance; and finally it enables autoscaling to meet demand, where every new server is simply a boilerplate copy of the template.

Continuous Delivery

Continuous delivery (CD) is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time (from Wikipedia).

In continuous delivery, any change to software that a developer checks-in and that passes all the required checks, can be deployed to production *at any time*. Continuous delivery relies heavily on certain design and automation practices that ensure software quality.

Paraphrasing Jez Humble and Dave Farley's book "Continuous Delivery" : *Continuous delivery represents a paradigm shift. Without continuous delivery, your software is broken until someone proves it works, usually during a testing or integration stage. With continuous delivery, your software is proven to work with every new change – and you know the moment it breaks and can fix it immediately. Teams that use continuous delivery effectively are able to deliver software much faster, and with fewer bugs, than teams that do not.*

The practice is a logical extension of the premise that smaller batches produce higher overall throughput (the proof of which is beyond the scope of this primer). By delivering small working batches rapidly to production development teams can respond quicker, reduce risk and deliver more.

The key elements of a successful continuous delivery pipeline are:

1. Your *software design and development process must encourage minimal complexity, refactoring and test coverage* (see TDD/BDD)
2. *All the items required to deploy your software must be version controlled* in your repository so they can be deployed by script;
3. *An automated build process* that can be run from the command line and can *clearly communicate the status of the build to all stakeholders*;
4. Changes to code should be *checked-in to the trunk or mainline at least several times a day*. Any less than this means increments are more likely to break the build and your continuous delivery will fail;
5. In order to ensure the quality of your builds you need a *comprehensive suite of automated tests that run with every build*. Without this you have no way of knowing that your application works and can be deployed to production.

Continuous Integration

Continuous delivery developed from the principles of *continuous integration*.

In traditional large-scale waterfall developments, testing the software development lifecycle would be back-loaded in the schedule. When a development team delivers large increments of code, the level of regression testing increases. Each change has the potential to conflict with each other change or with existing code and therefore must be cross checked. The larger the increment the larger the requirement to cross check.

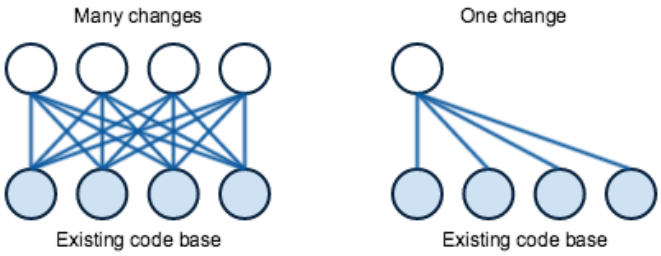


Figure 26: Regression overhead by size of change

This can be improved by delivering smaller increments which can be cross-checked (regression tested) and merged into the combined code base. Smaller increments allow confidence in the code base to be gradually increased to the point at which it can be released. Continuous integration or continuous delivery reduces those increments to the smallest logical size, a developer's code change.

Of course the system can be manipulated and developers can make wholesale changes over long periods of time before they check the code back in. But this behaviour becomes self correcting as the developers bear the pain of reintegrating their code with the *mainline*.

Blue/green deployment

One of the perennial problems of software development is how to deploy changes to a live production system. Most modern systems are in use 24 hours a day or at least during business hours when the development and operations team work. This often forces them into deployments in unenviable hours with complicated release strategies. With a move to continuous delivery and frequent small deployment this becomes untenable.

One solution for this is *blue/green deployments*.

In a blue/green system the application is built in distinct slices that traffic can be routed to. This is common in autoscaling or micro-services architectures where each node has to operate independently. In these instances traffic is load balanced amongst a number of machines running identical code.

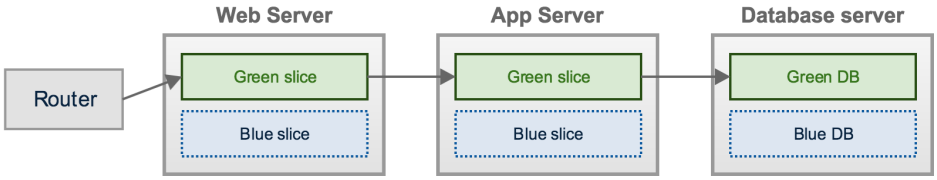


Figure 27: Blue/Green deployment: Continuous Delivery by Jez Humble and Davide Farley

Normally load balancers are designed to evenly split the users over a set of nearly identical application servers. But the routing tables can easily be adjust to divert, say 5%, of your users to a specific application server (or servers) with a specific feature set on it. This allows *A/B testing* in production (qqv).

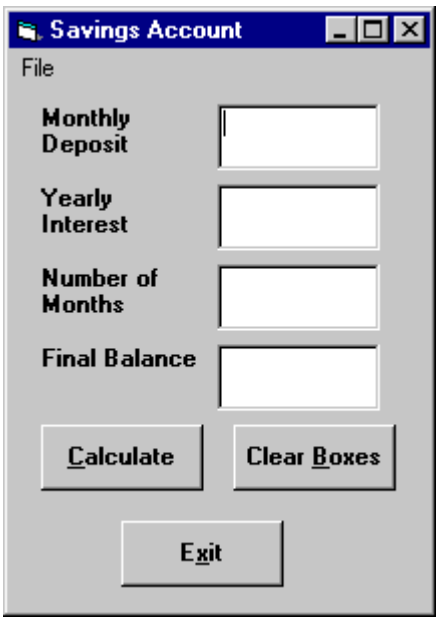
But the proportion of users for a particular slice of the application can be set to 0, effectively taking that slice out of production. Then your new code can be deployed to that slice and (once tested) the routing tables adjusted to shift production to the new servers. The old servers then go out of service and can be updated, bringing your whole application up to specification.

This system also allows near instant backout as the routing can be flipped back in the event of an issue in production. The are many variations on the theme of blue green releasing, *shadow releases* for example would use your most production-like test environment as one leg of the blue-green set.

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. To use 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. This means 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. We need 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 tester who 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. Used in 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