CHAPTER 1

I. Principles of Testing

1. Testing terminology

There is no generally accepted set of testing definitions used by the world wide testing community.

To address this issue the BCS SIGIST – British Computer Society Special Interest Group in software testing has created glossary of testing terms i.e. British Standard 7925 – 1.

Although this is a British standard it is neither mandatory nor universally used. There still exists different understanding of the testing terms.

The more the standards used by the testers and clients alike, the fewer misunderstandings there will be.

Clients will assume you know what they mean, so you must always ensure you are talking about the same thing. If there is any doubt, spell out to your client exactly what you mean by a term when you or they first use it.
Reference to BS 7925 – 1 can be made at this time so as to fully clarify the position.

2. What is testing?

The definitions of testing are varied and are changed over time. There is no one definition that covers all aspects of what testing is. Some definitions include

Execute a program with the intent to certify its quality

Establishing confidence that a program or system does what it is supposed to do.

Executing a program or a system with the intent of finding errors

An activity aimed at evaluating the attribute or capability of a program or a system and determining that it needs its required results.

One of the most complete is probably the process of exercising software or a system to detect errors, to verify that it satisfies functional or non functional requirements and to explore and to understand the status of the benefits and risks associated with its release.

3. Why testing is necessary?

3. A. Errors and faults

One of the main functions of testing is to detect errors and faults. What is meant by these terms?

Error: A human action that produces an incorrect result

Fault: A manifestation of an error in software (Also known as defect or bug)

Defect: The departure of quality, characteristics from its specified value resulting in a product for service not satisfying its normal usage requirements.

Note that although defect is a common name used for a fault, there is a slightly more specific definition. The term defect is less emotive than the term bug, which in developmental circles normally implies a mistake made by developers in producing code, which may not be the reason for a defect.

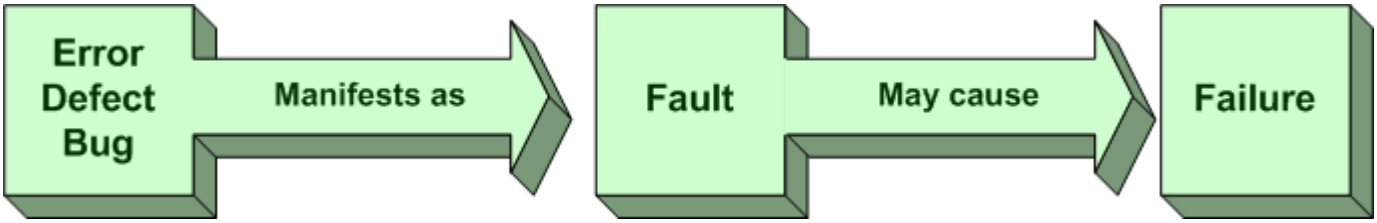
All failures are caused by faults, not all faults cause failures – some can remain hidden because the conditions to bring about the failure that are not present.

For example – if a software specification states that variable A should be set to equal the sum of variables B & C and a developer actually programs A = B – C the coding is an error, and will probably produce an incorrect result when that line of the program is run.

Failure:

A deviation of the software from its expected delivery or service. Notice the use of the word probably in the above example if the value of the variable C is 0 then the error introduced by the programmer will cause not cause an incorrect result because the value of the variable A would be the same (Correct) value whether the formula was entered A = B +C or A = B – C. So a fault may or may not cause a failure. This is an important point.

To summarize a (human) error (defect or bug) manifests as a fault in software which may cause failure.



3. B. How error occurs?

- No one is perfect
- Mistakes are easily made under pressure
- Poorly defined requirements or specifications
- Redefined but undocumented objectives during development
- Insufficient knowledge
- Poor training
- Poor communication
- Incorrect assumptions

3.C. The causes of bugs in software

Some figures show that the number one cause for software bugs is the specifications, with over 50% of bugs being traced to inadequate, misleading or simply wrong statements in requirement specifications. The next common cause of bugs are errors during design (About 25%).

Programme code only accounts for only 15% of bugs in software. This shows that testing can be very important in the early stages of development.

3.D. The cause of errors

Another reason why testing is necessary is that there is an associated cost with errors. Detecting and repairing errors can save this cost.

A single failure can cause nothing or a lot. Software and safety critical systems can cause death or injury if it fails. So the cost of failure in such a system may be in human lives.

A single failure can cost very little or nothing.

The wrong font in letter may just look strange. If the font specified is Times New Roman (Bold) and the font actually used is Times New Roman(Regular) it is an error. But not one that will probably cause too much of a problem and may only cost only a few pennies to correct.

- A single failure can cost millions.

NASA Mars Polar Lander lost 3rd December 1999.

The NASA Mars Polar Lander used a parachute to slow its descent to mars. After deploying the parachute 3 legs would open in preparation of touch down. At 1800 feet above the surface, it was to release the parachute and ignite landing thrusters. A switch was incorporated in one of the feet of the craft to switch off the fuel on touchdown by setting a data bit in a computer. It was found that on opening the legs, vibration caused the switch to operate, turning off the fuel and causing the craft to plummet 1800 feet to the surface.

The Lander was tested by multiple teams- one for the leg opening process and one for the rest of the landing process. The first team did not check to see if the landing bit was set (not their area) and the second team always cleared the bit before testing.

- A single failure can cost lives

US patriot missile defense system, 1991
A software bug was found that caused a small timing error in the patriot's tracking system. 28 US soldiers were killed when the tracking system (which had been accumulating errors for 100 hours) caused a patriot to miss an incoming missile.

3.E. Exhaustive Testing

If safety critical systems are so important, you would think they must be tested absolutely exhaustively. For example, nuclear power installations, air traffic control systems and fly-wire systems. Indeed, it is important that these systems are tested more thoroughly than most, but exhaustively testing is not possible.

Example

A simple function adds two 32 bit numbers and returns a result
Assumption: 1000 test cases can be executed per second
How long will it take to exhaustively test this function?

Answer: 585 million years
(2 to the power of (32+32) / 1000 * 60 *60*24*365.25 = 584,542,046 years)

Exhaustive testing would in most cases take an enormous amount if resource and time and is therefore usually impractical (all projects have constraints of time, money and resource).

So, how much testing is enough?

3.F. How much testing is enough?

On what basis is the decision made to stop?

Stopping testing too early risks leaving severe errors in the system;
Continuing testing too long can delay live release resulting in:

- Loss of revenue
- Loss of business (which may or may not result in loss of revenue)
- Damage to corporate reputation (which may or may not result in both of the above)

Testing and Risk:

So, when should testing stop?
The answer is depends on the risk to the business.

The amount of testing performed depends on the risks involved. Risk must be used as the basis for allocating the test time i.e. available and for selecting what to test and where to place emphasis.

In fact, testing can continue after release of the program to the live environment – it doesn't' t necessarily stop on final release.

Testing usually stops when management has enough information to make a reasoned decision about the benefits and risk of releasing the product and decides to either release the product or cancel the product.

What do we mean by the term quality?

A quality product is 3.G. Testing and quality:

One displaying excellence. Quality is the totality of the characteristics of something which has the bearing on the capability to satisfy the requirements.

Does testing a program or system improve its quality?

No. Testing a program doesn't in itself improve the program's quality. Testing identifies faults whose removal increases the software quality by increasing the software's potential reliability.

Reliability

The probability that software will not cause the failure of the system for a specified period of time under specified conditions.

Quality and reliability are not the same things. Just because a program or a system is stable, dependable and reliable does not necessarily mean it is of high quality. Reliability is only one aspect of quality.

Testing is the measurement of software quality.

We measure how closely we have achieved quality by testing the relevant factors such as correctness, reliability, usability, maintainability, reusability, testability, etc.

- Correctness: how well does the software perform in accordance to the specification and requirements.
- Reliability: how long can the software be used, how or how many transactions can be performed, before failures are encountered.
- Usability: how easy is it to use the software or system by all those who have interactions with it.
- Maintainability: how easy is it to make modifications or repairs to the system, or to upgrade with new releases of the application or operating system? How long does maintenance stay, and can this be done without closing the whole system down?
- Reusability: is the program or system modular in style, and can elements of it be used in future systems with similar requirements?
- Testability: are the requirements and specifications fully documented, upto date and ambiguous? Is the program or system easy to test?

3.H. Why testing is necessary?

Other determining factors for why testing is necessary?

Other factors that may determine why testing is performed include:

- Contractual requirements
- Legal requirements

These are normally defined in industry specific standards or based on agreed best practice (or, more realistically, non-negligent practice).

Contractual requirements:

A certain level of testing may be built in to a contract- for example, a testing contractor may agree to test 100% of business critical functions of a system and 50% of non-critical functions and to continue testing until there are no severe errors in these functions. These functions would need to be categorized and agreed upon “severe” defined before the contract is signed. No matter how much time or resources needed, the testing would continue until the contract would fulfill.

Legal requirements:
For example, aviation safety-critical or nuclear safety systems may be the subject of legal requirements to continue testing of 100% of functions until only few lowest category errors remain.

Industry specific standards:

The motor industry software reliability association (MISRA) has developed a set of guidelines for C programming of safety related embedded automotive systems. Another example is DO-178B, “software consideration in airborne systems and equipment certification”.

Agreed Best Practice:

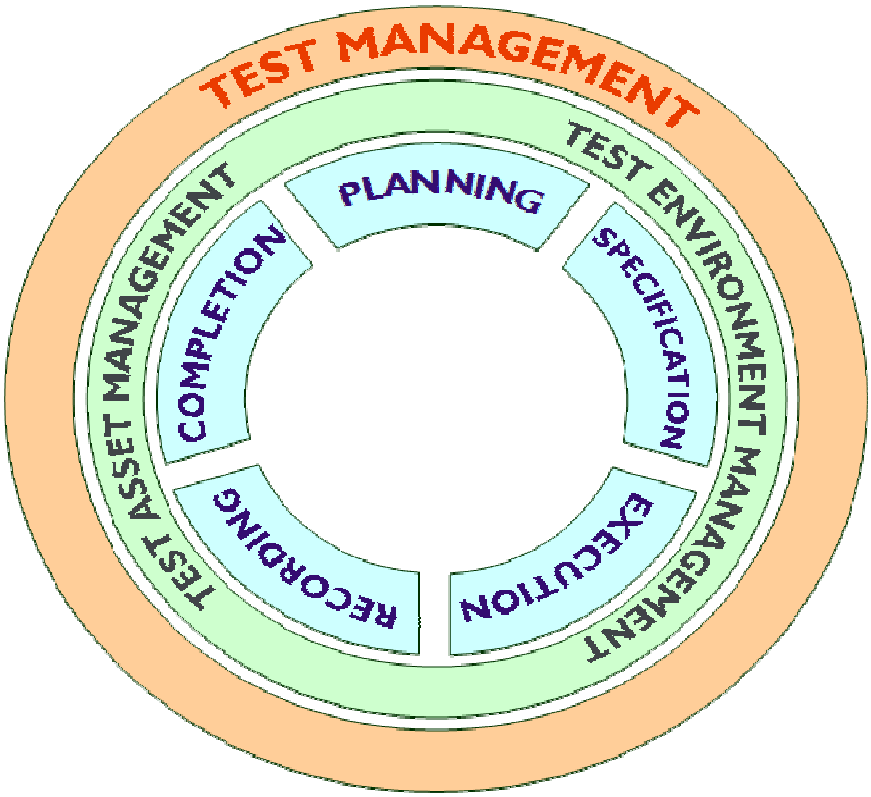
This differs from organization to organization. Some will have well defined in-house standards and best practice documents covering all aspects of testing. Others may have very sketchy outlines of what they consider to be the best way of performing testing.

SUMMARY

- The main purpose of testing is to discover faults
- Once discovered, faults can be fixed leading to better quality software
- Testing is the measurement of software quality
- Testing enables risk to be quantified and managed
- It is difficult to determine how much testing is enough

4.0 Fundamental Test Process

4.A. The stages in the fundamental test process



The fundamental test process comprises:

- Test planning
- Test specification
- Test execution
- Test recording
- Checking for test completion

All stages are governed by test management (which equates to managing risks) and are also the subject of test asset management and test environment management.

The five stages in the inner circle of the diagram shown above are the ones to be studied for the ISEB foundation certificate in software testing.

The test process is iterative, as is emphasized by the circular diagram.

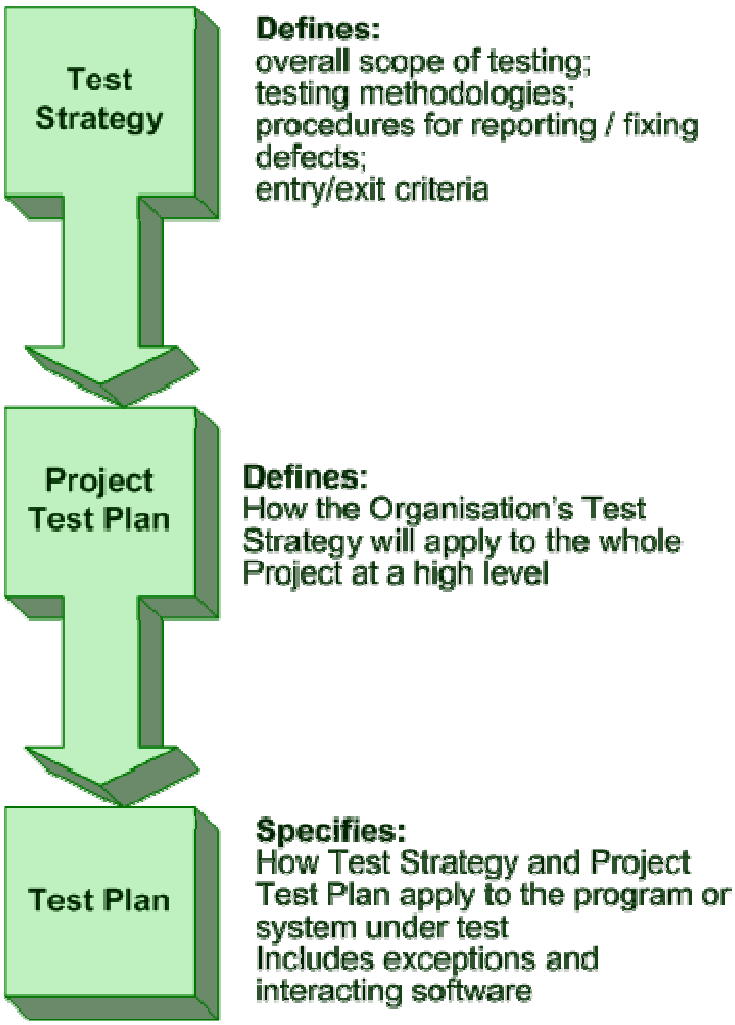
4.B Test Planning

The first part of the test process, test planning is vital. The test planning goal is to communicate the software test teams:

- Intent
- Expectations
- Understanding of the testing to be performed, includes techniques

The main deliverable of the planning stage is the test plan, which should specify how the test strategy and project test plan apply to the software under test.

The test plan should include identification of all exceptions to the test strategy and details of all software with which the software under test will interact during test execution such as drivers and stubs.



The test plan prescribes the scope, approach, resources, and schedule of the testing activities. It defines the items being tested, the feature to be tested, the testing tasks to be performed, the personnel responsible for each task, and the risk associated with this plan.
Assuming the specification and requirements are available, the planning process can (and possibly should) be the most lengthy stage in testing.

4.C Test Specification

The second part of the test process, is defining what to test and creating the test to do the job.

- Preparation: this is to ensuring that all required resources are to hand- including specifications and requirements and the project plan and test plans for reference.
- Analysis: this involves looking at the program or system documentation with the view to picking up what to test. There are many methods of doing this: for example, reading requirements and specifications, highlighting testable phases and from these creating open “one liner” test ideas of test conditions. The one liner then may be expanded into mare detailed test cases. All should be numbered and cross referenced back to the original documentation. Selecting test cases is an exceptionally important task. Selecting the wrong cases can waste time and effort, and hence money. It can result in testing too much or too little or testing the wrong thing completely.

Creating Test Cases:

This is the process of converting test cases into scripts that can be followed easily.

Test Case- Definition

A test asset that describes a set of inputs, actions and expected results used to prove a feature or a function of the software under test.

- Define initial conditions: are there any special conditions that need to be in place before this test case can be executed. For example should a overnight run have been completed successfully or should a particular server and its back-up both be in operation, should external communication links be up and running?
- Define standing data: data that will be referenced during the test- for example, dates of birth, names, addresses, bank transactions amount.
- Specify actions: what needs to be done by the person performing the test. These need to be clear, unambiguous and, given sufficient time, written in such a way that anyone can understand them, not just the test author.
- Specify expected results: see below

- Defining expected results: identifying what the outcomes of the tests should be. All relevant outcomes expected from the running of the tests need to be specified, for example, screen updates, or change to tables. Each separate action will have a expected separate result. The state of the application after the test may also be specified (still running, closed, awaiting another input) and the state of transactional data may be specified (account balances, name changes,etc.). Why must expected results be defined? Because **if no expectations are defined there is no way of telling if the test passed or failed.**

4.D Test Execution

Each test case should be executed however before execution of tests the following tasks should be completed.

- Check the test execution schedule. How long have we got? What resources (testers) are needed?
- Determine the test to be run : what tests are to be carried out now?
- Check that the environment is configured and ready: ensure that standing data and transactional data is in place.
- Ensure that all required resources are available: are testers present and able to perform their allotted tasks?
- Check that any required backups have been made: it could be disastrous if it was not possible to restore an environment having changed it irrevocably during testing.
- Ensure that recovery procedures are in place. For the same reasons as the previous points.
- Perform a confidence test(also known as a sanity check or health check) –this checks that the basics are in place and that the system is able to, for example, start-up normally and perform some predefined tasks that show it is operating in such a way that tests can be run.

Run the tests

- Each test should be run
- Tests either pass or fail- if the test is partially passes and partially fails (assuming a test case is proving more than one thing) then the whole test fails. There are no in-betweens where test results are concerned.

What is a successful test?

- The reason for running a test is to detect faults
- The successful test is one that does detect a fault

This is counter-intuitive, because faults delay progress: a successful test is one that may cause delay. The way to view this apparent paradox is the successful test reveals a fault which, if found; later, may be many times more costly to correct. So in the long run is a good thing.

4.E Test Recording

Having run a test, the results of the test (outcomes) need to noted in the test records. Test results should be recorded immediately i.e. as soon as the test has run. It is virtually impossible to remember individual detail of test results after only a short time when many tests are being run.

The test records for each test case should unambiguously detail:
- The identities and version of the software under test.
- The test specification
- The actual outcome

This is because it should be possible to establish that all of the specified testing activities have been carried out by reference to the test records.

The actual out comes should be compared to the expected out come. Any discrepancies found should be logged and analyzed in order to establish where its cause lies and the earliest test activity that should be repeated, for example, in order to remove the faults in the test specification or to verify the removal of the fault in the software. This is also useful for producing metrics for use in the test management activities.

In addition to the other items it may also be useful to record the test timings.

The test coverage levels achieved for those measures specified as test completion criteria should be recorded.

Below is a possible test result recording form for a manual recording system.

TEST RESULT RECORD FORM					
Project ID:					
SUT Title:		SUT Version:		Testing Phase:	
Test ID:		Date of Test:		Time of Test:	
Observed Test Results:					
Test Results Category:	☐ Pass ☐ Fail				
Error Description:					
Tester:					
COMPANY: AHSTS			CREATOR: Adrian Howes		

4.F Checking for test completion

Having run the tests and recorded the outcomes the test records should be checked against the previously specified test completion criteria or exit criteria.
Exit criteria should have been defined for example:

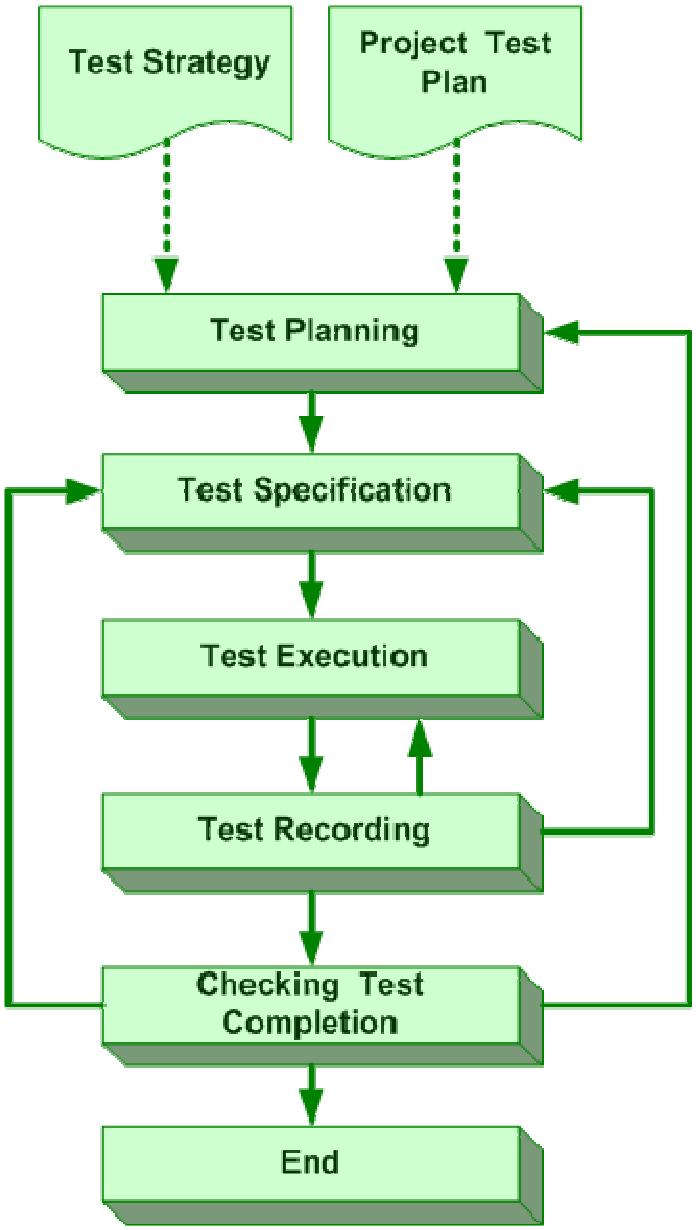
- Test requirement coverage- this could refer to user requirements or most frequently used transactions. All the requirements for the program or system may have to have been verified. May be just the higher risk requirements need to have been covered.
- Test code coverage- say, a percent of all branches or lines of code. 100% of the code may be specified as been covered. More likely a lower figure in most cases.
- Defect detection metrics- some times exit criteria are defined in terms of the number or rate of detection of defects, or of the number of more severe defects remaining.

If these criteria are not met, the earliest test activity that must be repeated in order to meet the criteria should be identified and the test process should be restarted form that point.

This may be necessary to repeat the test specification activity to design further test cases to meet t he test coverage target. Completion or exit criteria are used to determine when testing is complete. These criteria may be defined in terms of cost, time, faults found or coverage criteria.

Coverage criteria are defined in terms of items that are exercised by test suits, such as branches, user requirements, most frequently used transactions and so on.

Summary



5.0 Psychology of testing

The aims of this section

- To identify the mindset of a good tester
- To understand the mindset between testers and developers
- To understand the importance of tester/developer communication
- To appreciate the right and wrong ways of presenting faults
- To examine the different levels of testing independence

5.A. Testing as a Destructive process

Testing is performed with the primary intent of finding faults in the software, rather than of merely proving that it works correctly according to specification.

Because of this negative approach, testing can be perceived as a destructive process, required mindset of a tester may be perceived as likewise being distructive.

Developing, in contrast can be seen as a constructive process and accordingly the mindset of a developer may be more constructive and artistic in nature.

Thus the required mindset of a tester is different than a developer.

5B. Attributes needed by testers

Testing is wide ranging and demanding profession. The attributes needed by the testers are accordingly wide ranging and the role demands a high level of skills in many areas.

Analytical skills

Bug finding, focus on essentials

- How to find bugs- planning, preparation and execution of tests
- How to understand system from design documents and data models
- How to read and understand test and system specification
- How to extract testable functionality
- How to work efficiently for example in correctly prioritizing tests
- How to focus in essentials, for example, key components and business critical area.

Practical skills

Intellectual self organization

- Ability to absorb and work with incomplete facts for example to make accurate predictions and assumptions where absolutely necessary.
- Ability to learn quickly on many levels foe example testing often involves the requirement to learn new applications, their context and new skills in a very short time
- Good verbal and written communication, for example explaining clearly and concisely what you require of others and conveying the results and progress of your tests in a manner easily understood by all is essential.
- Ability to priorities not only what tests to carry out first, but also what tasks take precedence over others. For example, should a meeting with developers to discuss a level defects take priority over a weekly progress meeting.
- Self organization for example- time management, planning and preparation.
- Accuracy- among other things, the ability to follow instructions and record results precisely.
- Common sense- a lot of testing comes down to intelligence and common sense.

Interpersonal skills

Communication

- Communication with developers through effective defect reporting
- Communication with managers through metrics
- Questioning skills

- Diplomacy- there are right and wrong ways of presenting faults to authors and management. For example wrong way is “the software is rubbish, this application is badly designed and does not work and the function is faulty”. Better ways are “there is a possible defect in this function, the application appears to run slowly when used this way, the system seems to exhibit anomalous behavior..
- Pejorative terms such as fault and failure should be avoided where possible. Kinder terms should be used such as variance, inconsistency, incident, which do not seem to point the finger of blame so much. Having said this the job of the tester is still ultimately reporting defects in the products. One company spent thousands of dollars and weeks of time discussing the change of name of their product anomaly report to product incident report. It is not know if it made any real difference to productivity
- It is important to have good, effective communication between developer and tester for example, passing on in good time changes to the applications or menu structures that might affect the tests where the developer thinks the code might be buggy, where there might be difficulty in reproducing the reported bugs

Integrity

- The nature of the role is such that the testers will inevitable come in contact with information of sensitive nature, whether it is personal or financial. In this respect the tester is in a position of trust and such information is always be treated a confidential. The tester is in a privilege with the fate of the business often resting at least partly on the testers’ shoulders.
- The tester should always act with integrity of all aspects of the role.

Knowledge

Testing projects and business systems

- How projects work
- How computer systems and business needs interact
- Testing techniques
- Testing best practices
- Operating systems
- Web browsers
- IT in general

Also needed is an ability to think outside of a system specification, because specifications do not always cover everything completely. For example, what happens if an age range of 1-60 is specified and an age input of 61 is made?

5.C. Tester developer communication

A working dialogue between testers and developers is essential

There must be good communication channels between the two parts of the partnership. Testers should be able to approach developers to inform them of discovered potential defects. However, just informing them verbally will not be sufficient. Records must be kept and the proof of the defects if vital.
A good relationship helps enormously.

Being able to speak of the record with developers might prevent or reduce raising of defects incorrectly, thereby saving time and resources. Developers are more likely to listen to testers that they respect than those that just annoy them.

Whenever you raise a defect you are effectively criticizing a developer's work. If the developer realizes that both of you are working to improve quality in a professional and competent way, they are more likely to appreciate you raising defects because they will realize you are doing it partly to save their embarrassment if a defect gets through to the live systems.

5.D. Tester management communication

Management needs information about test results. They make decisions about risks based partly on test results and test metrics, and testers are vital in providing feedback on test results to feed the requirements of management for decision making data.

One of the ways for testers to pass information to management is progress reports. These can be daily, weekly, monthly or any other intervals based upon the urgency of need.

Metrics are good way to provide testing progress. They can be presented in a tabular form, or more usually in the form of graphs. Metrics can include:

- Number of test plans
- Number of tests run
- Number of tests passed
- Number of tests failed
- Defect analysis (severity and time to fix, etc).
- Time to execute tests.

Formal reports are not the only means of communicating with management. Often it is more effective to have face to face communication, especially when the matter is urgent. What ever means is used for communication, a record should be kept of what was said and what was decided if appropriate.

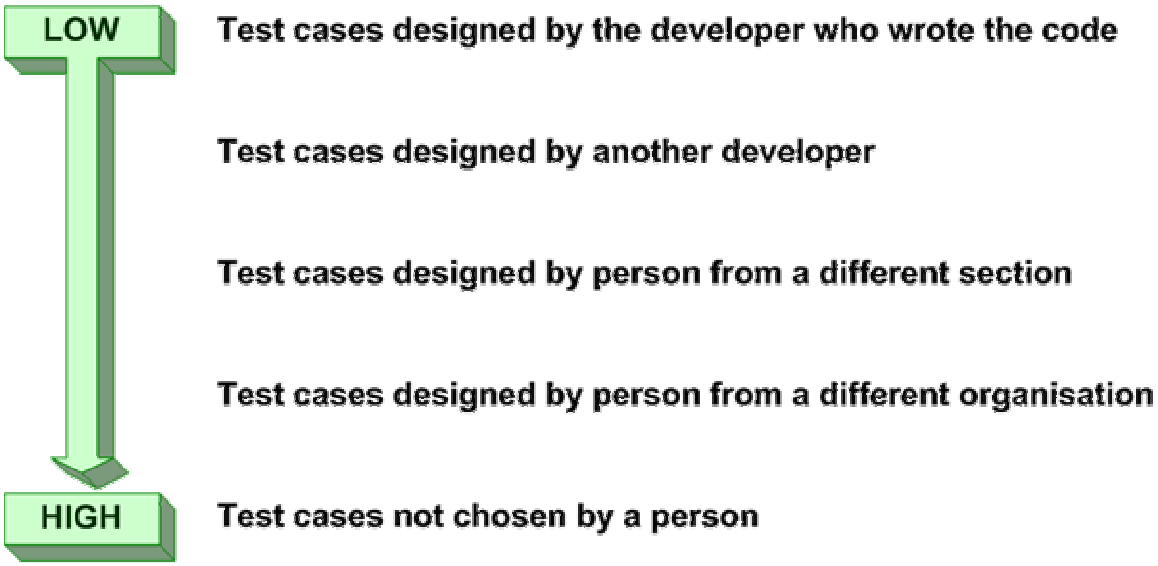
5.E. Testing independence

Generally it is believed that objective, independent testing is more effective. If an unbiased view is required, an unbiased observer must carryout the measurement. Developers may find it difficult to be objective about their own code.

If the author of a product tests their own output then the assumptions they made during development are carried into the testing. People see what they want to see, there can be an emotional attachment to ones own work, and there may be a vested interest in not finding faults.

Developers may concentrate proving the program works (positive testing or test to pass). By contrast, independent testers will also carryout negative testing also known as test to fail.

Levels of Independence



Levels of Independence

Test cases are designed by person who writes the software under test.

This is bad. If a specification says to accept an age between 18 and 68, a developer may take this literally and code for ages 19 to 67 (literally between 18 and 68). If this is the case the developer will almost certainly design a test for exactly the same conditions. The software will pass the test but will contain a defect.

Test cases are designed by another person in the development team

This is better. But still not particularly good. The same assumptions may be made by a similar minded developer. There may be a vested interest in not finding faults

Test cases are designed by a person from a different section.

Again, an improvement over the previous two scenarios, but once more there may be vested interest preventing a selection of valid tests.

Test cases are designed by a person from a different organization

The best of the manual ways of producing tests:

A different organization will normally have no eggs to grind and can be completely objective in choosing the test cases and subjecting the application under test to a rigorous test set. In most cases the separate organization will specialize in software testing and will apply the appropriate test techniques, including negative testing methods. A counter argument for using an external organization is that they may not understand the product as well as one of the developers or may not have such commitment to the final product as someone from business. The first of these arguments is countered by engaging external test organizations early enough in the life cycle to

ensure they can learn the product sufficiently well to design effective testing. The second argument is valid, but applies equally to all service industries not just testing.

Test cases are not chosen by a person

This is utopia for test independence. A special piece of software determines what cases should be chosen and the parameters should be used based on a list of specifications and expected results entered into the tool's database.

Summary

- Testing is considered a destructive process
- A testers mindset is different to that of a developer
- Testers require particular skills
- Testers-developer communication is vital
- Independent, objective testing is considered more effective

6.0. Retesting and regression testing

6.A. Retesting

The purpose of testing is finding faults. Faults must be corrected and software re-released as a new version of the application under test. Once the new version is ready for release, tests are re-run to ensure the previously found faults are actually been fixed. This is re-testing.

It is rare that a test is used only one time. The reuse of old test is useful- it saves time, shows up changes and it is important for consistent testing.

It is important to write tests so they can be reused. Test should not have to be re-written each time they are run. Re-writing is not only a waste of time, it provides more opportunity for introducing errors into test scripts.

When carrying out re-testing it is useful to also consider additional testing for similar and related faults.

Re-testing should be created for in plans and designs

Re-testing should be included in test schedules

- It is very easy to omit retesting when estimating the total testing effort required for a project. When scheduling, estimates need to be made as how much retesting will be needed and when it will take place.

Test should be easily run

- This may need substantial thought before actually writing test cases and scripts. It is often worth discussing the possible problems of retesting with the software designers and developers, who may be able to provide some insight into the potential problems.

Test data should be re-usable

- In some applications, data cannot be reused due to non-changeable indexes or automatically generated keys. If data is not reusable, test scripts may not have to be

amended for each rerun or written in a template fashion where the script is generic and data is used each time, held independently of the test scripts. The purpose of retesting is to show that faults have been fixed, so tests need to be run in identical circumstances (as far as possible) – i.e. same data, same actions and same environment.

Ensure test environment is restorable

- To enable the same test to be rerun, the conditions under which the test was conducted need to be in place. Environments can be difficult to restore to their original state. In some cases it can be impossible to reset data or environments if the environment is shared with other users conducting their own tests.

6.B. Regression testing

"retesting of a previously tested program following modifications to ensure that faults have not been introduced or uncovered as a result of the changes made"

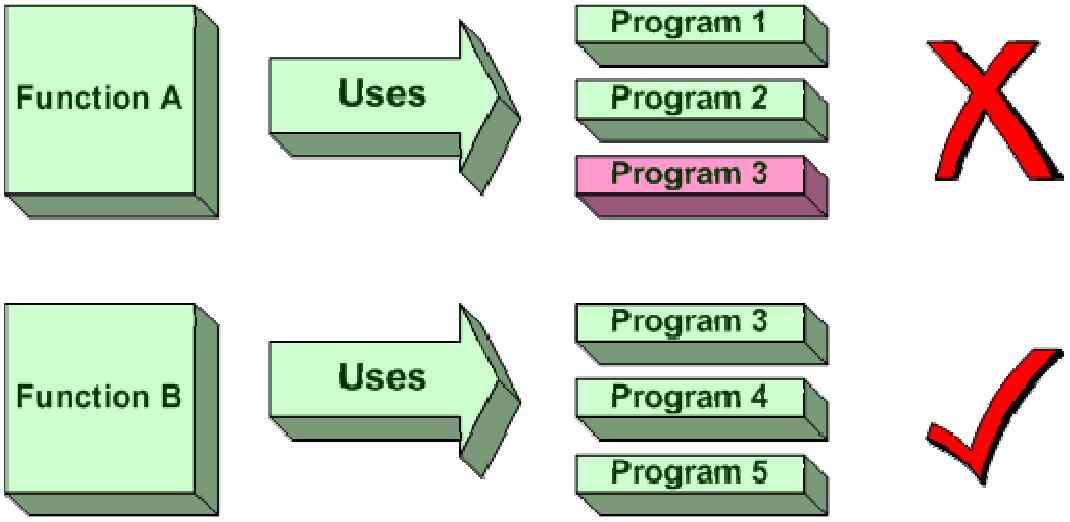
Note the difference between this and retesting.

Retesting and running tests again to show that faults have been fixed. Regression testing goes further, showing that other functions have not been affected by the changes.

Regression testing should show only a small number of faults, since the main purpose of this type of testing is to prove that nothing has changed.

Although still aimed at finding faults, it is more oriented to engendering confidence in the software and producing a quantifiable risk assessment.

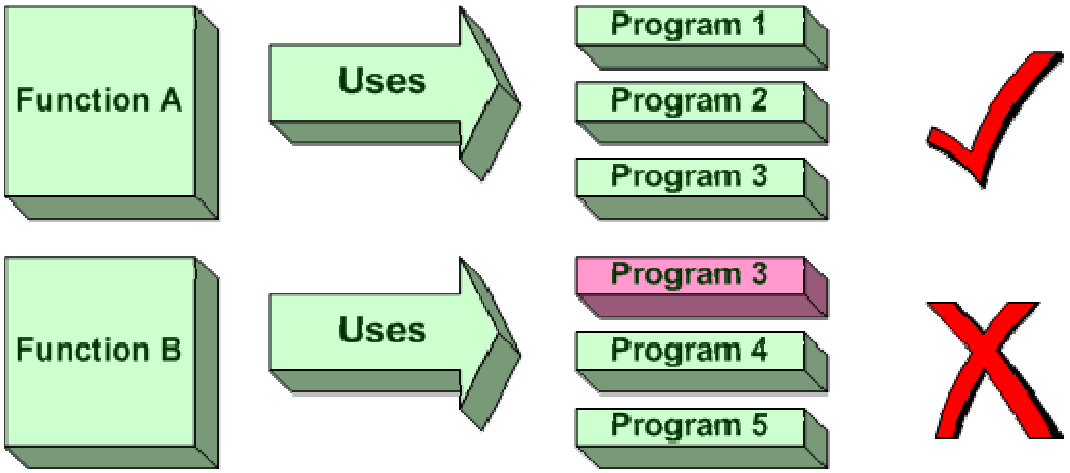
Regression testing is necessary because it is possible to fix one thing and while doing so, break another which at first seem unconnected. The example below shows this



In the diagrammatic representation above function A uses programs 1, 2 and 3. function B used programs 3, 4, and 5.

Testing and subsequent investigation has shown that function A does not work as specified due to a problem in program 3.

Changes are made to program 3 in an attempt to correct the problem.



As a result, function B does not work because of the changes made to program 3.

Retesting would have shown that function A works after the fix. Regression testing would have shown that function B does not work, although it did before the fix to program 3.

6.C. When to regression test

Regression testing should be carried out

When ever a change is made to the software

- In the application under test
- To other software (for example, antivirus software)
- To the OS (for example upgrades)

When ever a change is made to the environment, for example:

- Replacing one technology with another
- Changing printer type

Regression tests are carried out to prove that any changes are not uncovered new bugs.

Regression testing is a very important part of the testing process.

6.D. What to regression test

Regression testing should be carried out by selecting test cases that cover:

- Safety critical functions
- Business critical functions
- Areas subject to frequent change
- Areas or functions with a history of high level of faults

As with all testing, business and safety critical functions should be the main concern, but as the prime reason for regression testing is proving that change has not produced or uncovered new bugs, areas subject to frequent change should also to be focused on.

If a website that runs 24hrs a day is being tested, regression testing will be required when updates are carried out or if emergency fixes are made. For example, a full regression may be carried out every two weeks, mid range tests every two days and an ongoing regression test may be carried every two hours.

- Regression testing ideal for automation
  Because regression tests are likely to be run identically time and time again, they are the ideal subjects for automation. However since an application may change or evolve in other ways, this must be taken into consideration when designing automation tests and an in-depth knowledge of application under test is vital. Regression test suites are run many times and generally evolve slowly, so regression testing is ideal for automation. If automation is not possible or the regression test suites are very large then it may be necessary to prune the test suites. Repetitive test may be dropped, the number of test on fixed faults may be reduced, test cases combined, some tests may be designated for periodic testing, etc. A subset of the regression testing test suites may also be used to verify emergency fixes.

  In general for automation of regression testing:
  - Proper test case selection is important
  - Good knowledge of the application is needed
  - Application evolution should be anticipated

Summary

- Software must be retested when faults have been fixed so that:
  - New faults introduced may be found
  - Previously existing but undetected bugs may be uncovered
- Tests are written with reuse in mind
- Retesting is running again previously failed test when a fix has been made to prove the fix
- Regression testing is checking for bugs in code that is unchanged as a result of changes to the other parts of application

7.0 Expected Results

What are expected results

Expected results are synonymous with the expected out comes. In other words what is predicted by the requirements and by the specification as result of particular inputs to the application under test.

Expected results are not the same as outputs- outputs are not predicted.

If expected results have not been defined then a possible, but a erroneous result may be interpreted as the correct one. Expected results must therefore be defined prior to test execution.

Without defined expected results, miss interpretations may occur. If you do not know the expected results, how do you know if the result is correct.

7.A. The sources of expected results:

Expected results can come from:

- Knowledge or experience of the application
- Specifications
- Requirements
- The business

But not:

- The code- since the code could have been written incorrectly or based in incomplete or erroneous assumptions. (The only exception is when testing a trusted system i.e. many years in operation without error and before any changes have been made.
- The results of tests first run - generating expected results from a first run of a test are not valid. Since you are assuming that the application is correct to begin with.

7.B the oracle assumption

"The oracle assumption" is that the tester can routinely identify the correct outcome of a test.

An oracle may be for example, the existing system (for a bench mark), or a specification, or an individual's specialized knowledge, but not the code- again because assumptions cannot be made that the code is correctly written.

7.C Why do tests need to be prioritized

Exhaustive testing is not possible

There is never enough time to do all the testing you would like to do. As a result, in all but most trivial systems, some faults will inevitably get through to the live system. Therefore tests must be prioritized to ensure that the best testing is carried out within the available time and budget.

Prioritize tests so that whenever you stop testing you have done the best testing in the available time. There will be aspects that must be tested, and those that would be nice to test if there is time. Decisions must be made regarding what are the most critical tests.

Ranking criteria used to prioritize tests:

The ranking criteria used to prioritize need to be identified, such as severity, probability, visibility of failure, the priorities of the requirements to be tested, what the customer wants, change proneness, error proneness, business criticality, technical criticality and complexity.

- Severity of failure
  Major system failure or minor inconvenience? If a likely outcome of a particular failure is that a system will crash and loose a business of hundreds of thousands of pounds, that is quite likely a candidate for high prioritization. If the all that would result from a failure is an extra sheet of blank paper at the end of the weekly report, a low priority would no doubt allocated. In practice the decisions about prioritization are rarely as clear cut and many conflicting factors may need to be taken into consideration.

- Probability of failure

  The likelihood that a particular function or configuration will contain or be susceptible to faults. For new hardware, talk to the developers. Previous experience is useful here. For the probabilities of faults occurring in particular functions, talk to the users.

- Visibility of failure

For in house applications there may be less visibility, so a lesser priority can be assigned. For public web applications visibility will be much higher, hence higher priority.

- Priorities of requirements

  Start with those requirements that are essential to the business needs, then move on to the aspects that are not so vital, the “nice to have requirements”.

- Customer wants

  The customer’s perception of what is important, which is the same as the business requirements is not necessarily. For example, the ability to access personal information or to move money from account to account on the same day.

- Liability to change

  Frequent changes produce more likelihood of faults. Identify functions or features that are most subject to change. The more frequently new versions are produced the more likely the introduction of errors. Functions that are subject to change on a frequent change are similarly affected.

- Error proness

  Likelihood of error in a particular function. This can be gleaned from non-issues, so reference the project issues log. This is linked to probability and complexity. For example a particular developer may have been known to make mistakes in a certain coding technique, a communication link is known to fail regularly and cause problems with roll back transactions.

- Business criticality- what must work for the business to function?

  What must work so as not to compromise business image.

- Technical Criticality- what parts of the system are most important technically?

  For example modems in communication system or servers for websites.

- Complexity- the more complex a function the more likely it is that error will occur. More complex coding techniques can lead to higher error rates.
  Organizations some times prioritize by types of faults- in other words certain types of faults may be fixed before others. It is important for the tester to understand how this will affect re-testing and the time to spend on different types of faults.

- Availability of resources: the more testers there are, the more can be tested. Similarly, the longer they are available the more can be tested.

  Organizations sometimes prioritize by the types of faults- in other words certain types of faults may be fixed before others. It is important for the testers to understand how this will affect re-testing and the time to spend on different types of faults.

CHAPTER 2

MODELS FOR TESTING

1.A V, V & T

Verification, validation and testing

The definitions for V, V & T come from the British standard 7925-1:1998- software testing vocabulary.

Verification – definition: confirmation by examination and provision of objective evidence that specified requirements have been fulfilled.

Verification of any computer system is the task of determining that the system is built according to its specification. Verification asks the question "is the system built right?"

Issues raised during verification include:

- Does the design reflect the requirements?
- Are all of the issues contained in the requirements addresses in the design?
- Does the detailed design reflect the design goals?
- Does the code accurately reflect the detailed design?
- Is the code correct with the language syntax?

When the program has been verified, it is assured that there are no bugs or other technical errors. In other words verification is the process of determining:

- Are we building the system correctly?

Validation:

Confirmation by examination and provision of objective evidence that the particular requirements for the specific requirements have been fulfilled.

Validation is the process of determining that the system actually fulfills the purpose that it was intended.

In other words ensuring that the software correctly reflects the user's needs and requirements.

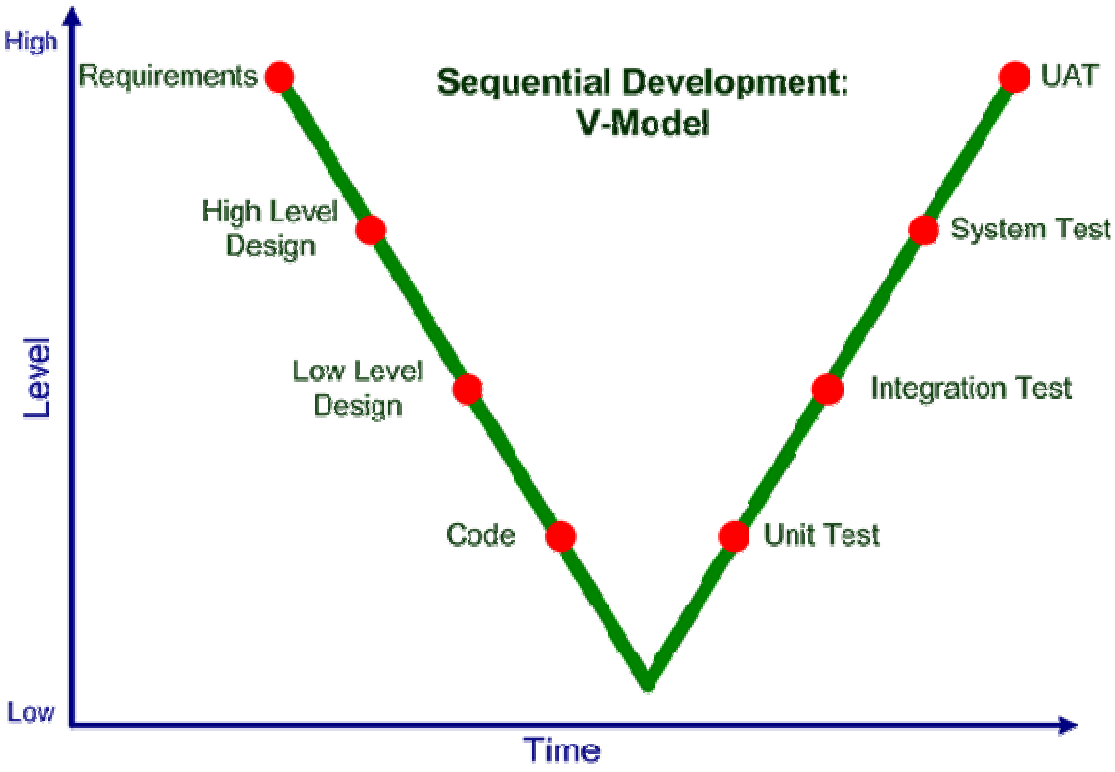
Validation answers the question "is it the right system?"
"Is the knowledge base correct or is the program doing the job correctly?"
Thus, validation is the determination that the completed system performs the functions in the requirements specification and is usable for the intended purposes.

The scope of the specification is rarely prescribed, and is particularly impossible and it is practically impossible to test a system under all the rare events possible. Therefore it is impossible to have an absolute guarantee that a program satisfies its specification, only a degree of confidence can be obtained that a program is valid.

Testing: process of exercising software to verify that it meets its specified requirements and to detect errors.

V Model

The V model is so called simply because it is shaped like V.



This is a simple v model which maps the different development phases to the appropriate test phases.

So we have

- Requirements mapped to UAT
- High level design mapped to the system test
- Low level mapped to the integration test
- Code mapped to unit test

The V model proceeds from left to right, depicting the basic sequence of the development of testing activities

The model is valuable because it highlights the several levels of testing and depicts the way each relates to a different development phase.

- Unit testing is coded based is developed primarily by developers to develop the smallest piece of executable code functions suitably
- Integration testing demonstrates that 2 or more units or other integrations work together properly and tends to focus on the interfaces specified in low level design.
- When all the units and their various integrations have been tested, system testing demonstrates that the system works from end to end in a production like environment to provide the business functions specified In the high level design

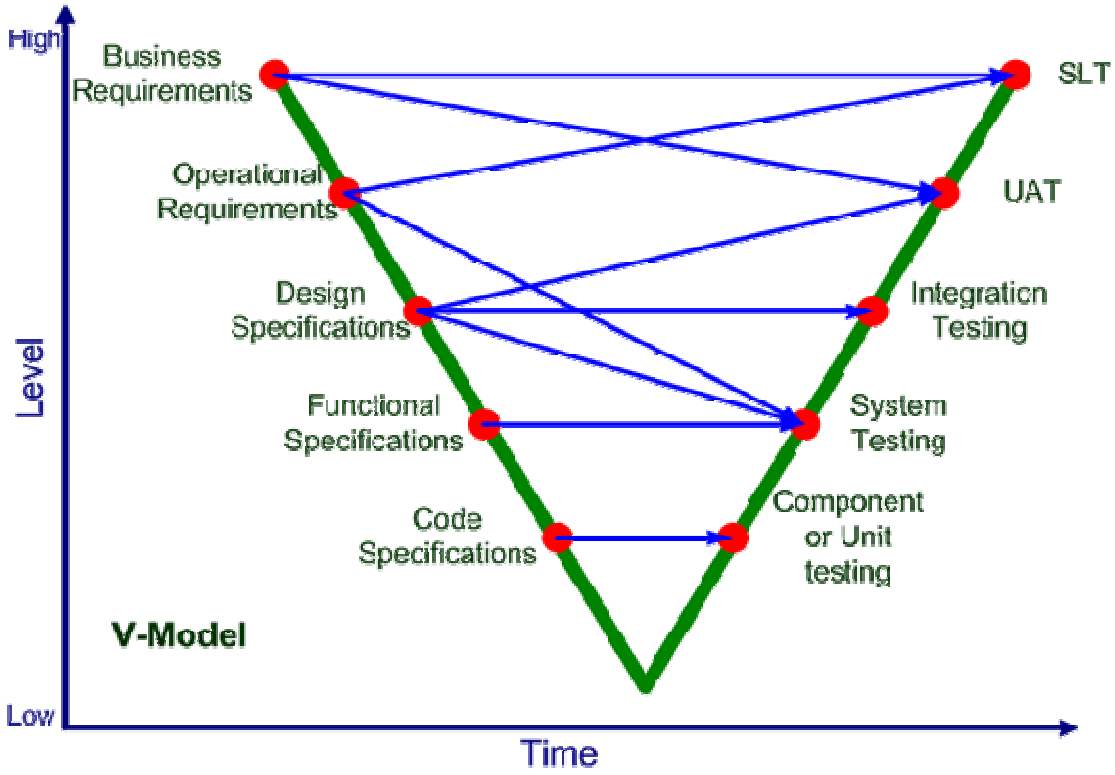
Finally, when the technical organization has completed these tests, the business or user performed acceptance testing to confirm that the system does, in fact, meets their business requirements.

Integration testing is broken into 2 stages;

1. Integration testing in the small (link testing)
2. Integration testing in the large (where the system or application under test is tested for communication for other applications or networks

V- Model

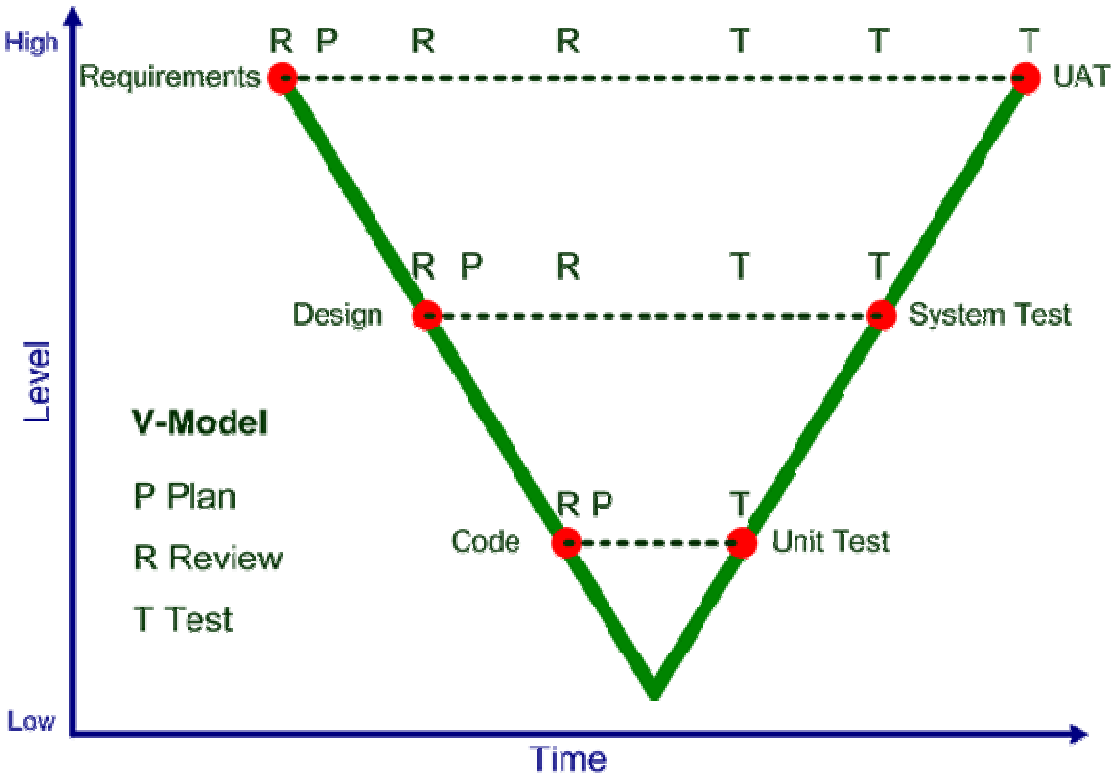
Below is another version of V model with lines indicating from where information for tests can be gained. This is not exhaustive. Note that SLT is service level testing.



V Model

If testing is only carried out when the development cycle reaches the right hand side of the v model the time for the user acceptance testing may be reached only to discover that the application doesn't match the design or the user requirements . to change the user system at this stage would be expensive and time consuming and may not be practicable at all. To prevent this happening testing should start as early as possible. The earliest the deliverable can be reviewed as soon as it has been produced.

The v model here shows where reviews (a type of testing) should be carried out. Planning for testing can also be commenced once deliverables have been reviewed



U can see here that the V model identifies base lines (both h testing and development) and deliverable which should be tested at each stage of the development life cycle

Other models:

Other models used for testing are the waterfall model (also used as a name for one type of development methodology), the W model and the spiral model. None of these need to be studied in detail for the ISEB foundation course but be aware that they exist.

Summary:

- Two common models for testing are
  - V, V and T
  - V- Model
- Most common is the V model
- V-Model identifies baselines to be tested at each stage of development.

The cost of defects

Software is produced by way of development life cycle. Effects or bugs can be found at any stage in this cycle.

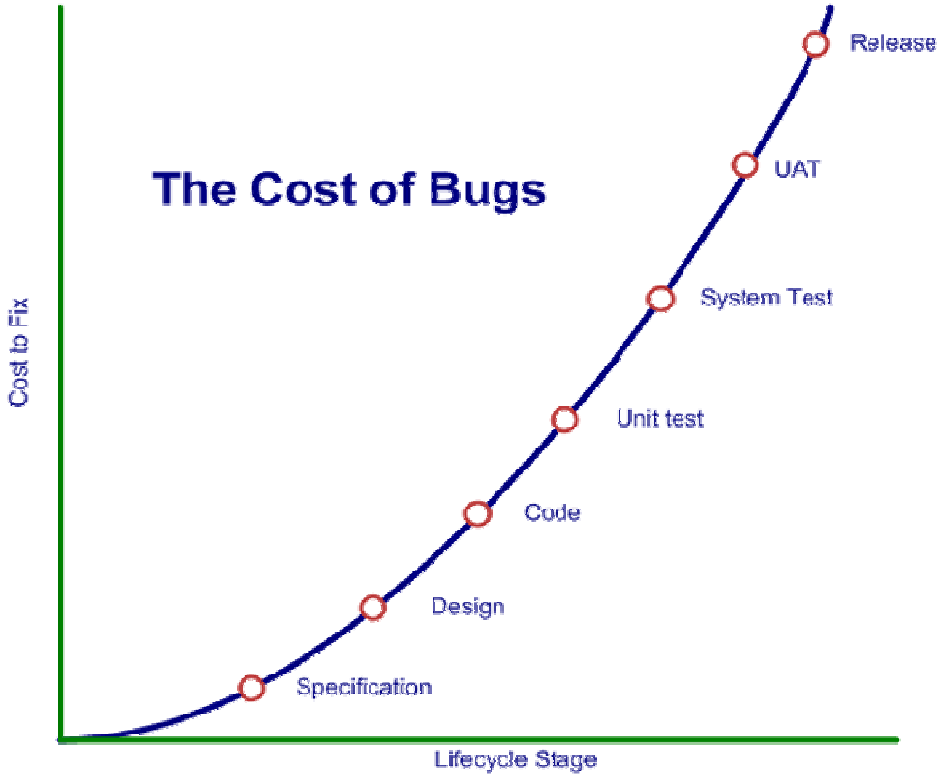
Requirements

- Design
- Coding
- Unit testing

- Integration testing in the small
- System testing
- In the large user acceptance testing

The cost of fixing bugs grows over the course the development life cycled

The graph shown here is one company's model of the relative cost involved of fixing bugs discovered at the various stages of the development process



Why is the graph the shape it is

If a bug found in requirements before any other work is carried out the only change needed may be to a document costing a pound or two in time. If the same bug is found in UAT substantial rewrite of s/w may be required costing many hundred s or thousands of pound.

It is estimated that the cost of fixing errors goes up by 10 times from one stage to another i.e. exponentially. One large insurance company estimates a 50 fold increase at each stage

Cost of defects:

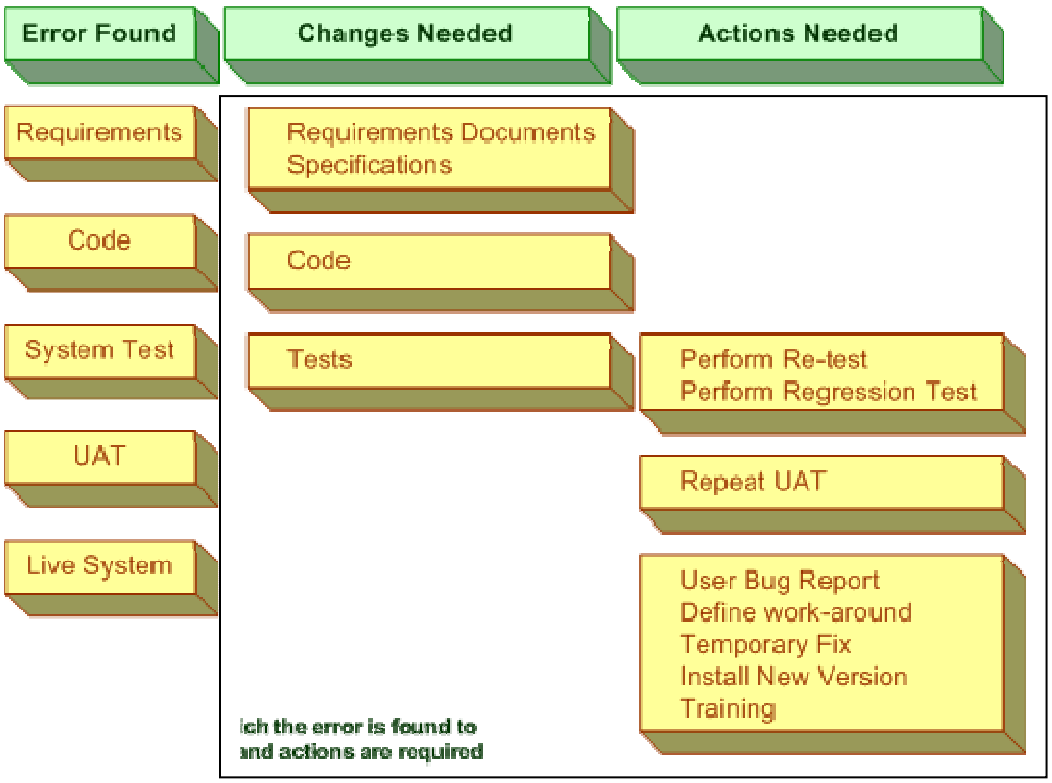
The earlier the fault is found:

- The cheaper it is to put right
- The easier it is to put right

The cost of faults escalates as we move the product development towards field use. If a fault is detected just before field use, the cost of rework to correct the fault increases dramatically because more than one previous stages of design, coding, and testing may have to be repeated.

The diagram below shows the increase in work as defects are detected at the later stages in the development progression.

Cost of defects.



If the faults occur during field use the potential costs of the field might be catastrophic. If the faults that are present in the documentation go undetected, the development based on that documentation might generate many related faults which multiply the effect of the original one

Additional costs of error sin the live systems may include:

- A loss of business
- Extra training for the users to cope with any substantial changes

Reviews are economical

Many defects can be found by reviews.

- Of requirements
- Of design
- Of code
- Of tests themselves

Examples of design faults found by review

A major insurance company was developing a new website and associated call centre s/w. The design called for database to hold the details of products purchased to either route. The designer or developers of the call centre application assumed they were responsible for the main database

to hold the customer purchase information and the web designers or developers thought they were.

It was only when, half way through the development phase testers reviewed the documentation as their first step after having been asked to involved that the question was asked “who is responsible for synchronization fog the 2 database”

This dumb founded both sets of developers who had from their own views just their own part of the documentation that there as only one database.

The discovery set back the project by over 2 months and caused 40000 pounds.

If the documentation had been reviewed and testing planned at the beginning of the project instead of half way thru the cost would have been merely the price of employing the tester fro few extra weeks..

Easy test deign is economical and beneficial.

Early test design

- Can highlight errors in specification; when test case are being designed it is common for specification and requirement errors to be picked up (since test case deign requires a detailed analysis of its requirements and specification). Even if the documents have been already been reviewed the test case design may find more anomalies.

- Can prevent fault multiplication; if specification errors are missed the anomalies that get through u to late stage can make their way into many modules causing multiple defects from the one error. Detecting the fault at early stage means dealing with only one error at early stage

Cost of testing vs. cost of not testing

- Testing is always seen as expensive as those who have to pay for it
- The cost of fixing faults especially at later stages is very expensive
- The cost of testing is generally lower than the cost associated with major faults (such as poor quality product and / fixing faults) although few organizations have figures to confirm these

Summary:

- The cost of faults escalates towards field use.
- Early test design can prevent fault multiplication.
- Cost of testing is generally lower than fixing major faults.

High level test planning

Test plans

- What is the purpose of a test plan?

  To prescribe the scope approach, resources, and schedule of the testing activities. To identify the items being tested the features to be tested the testing task to be performed , the personal responsible for each task and the risks associated with each plan, IEEE standard 829-1998

Ultimately the purpose of the test plan is to be a document that helps a test effort to be successful by being a distillation of what to be tested , how and by whom and being available to all those involved in the project sp they may have a compete understanding of the testing intentions

The result of the test planning process should be a clear process and concise definition of the quality and reliability goals that are the important of the project

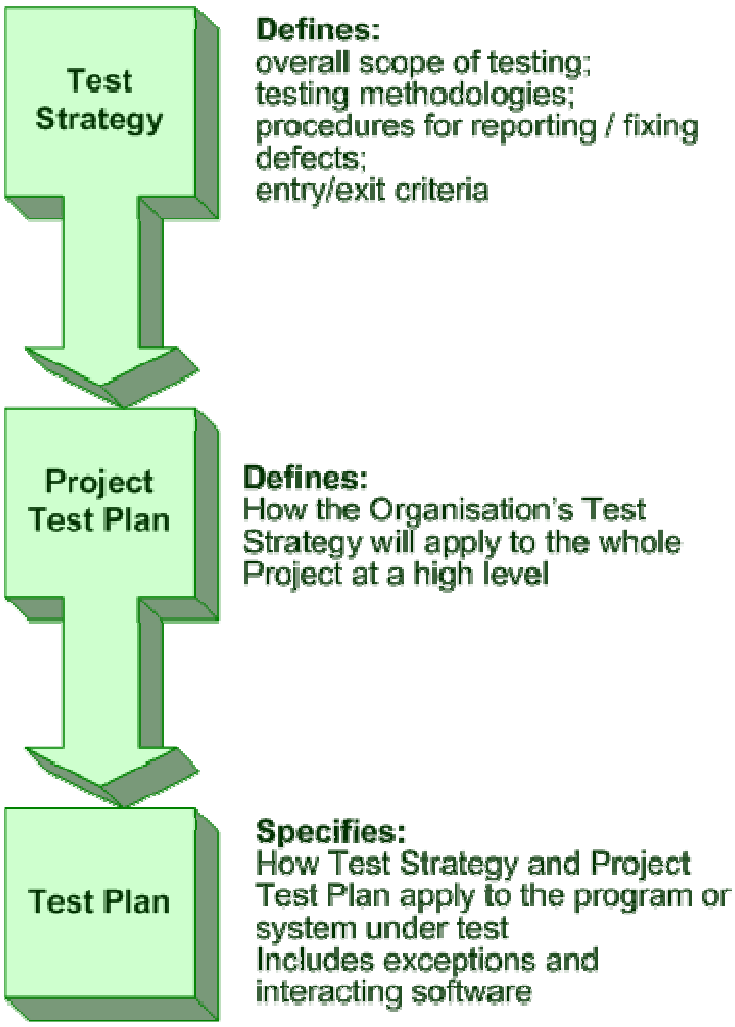
All these should be agreed by all parties – and the test plan is the document that all parties will use as a refining achieving this.

What is a test plan?

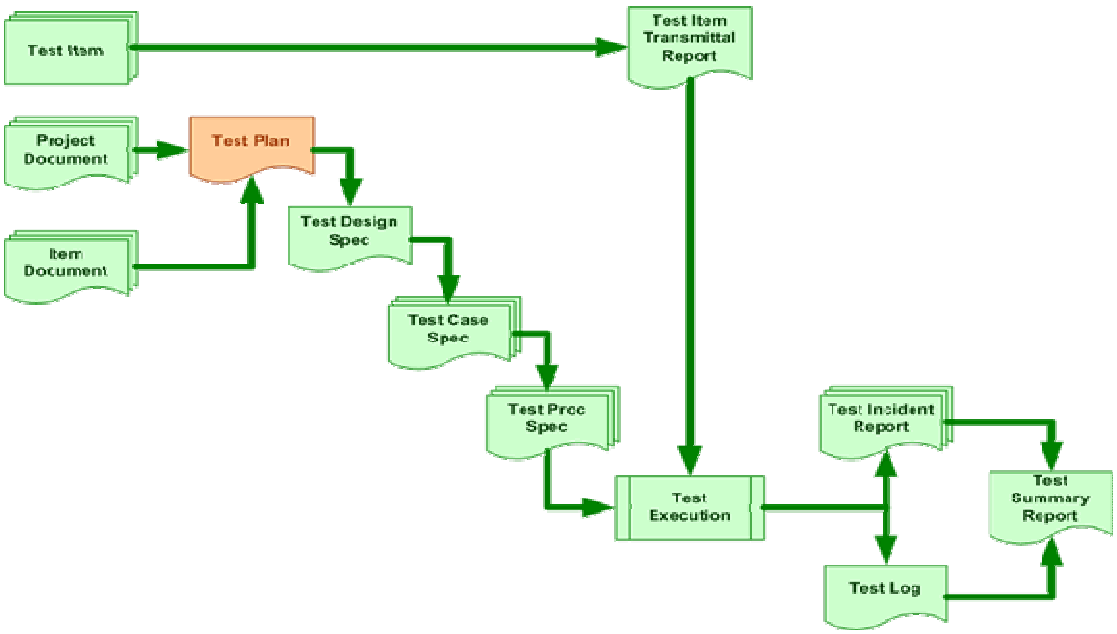
A document describing the scope, the approach, the resources and the schedule of the intending testing of activities. It identities the test items the features to be tested, the testing task, who will do each task, an d any risks requiring contingency planning IEEE 829 – 1998

Position of the test plan in the development life cycle:

The diagram below shows where the test plan fits in the test strategy and project test plan. The test strategy feeds into the project test plan and both feed into the test plan.



The diagram below shows the test plan in context with other documents in the testing process (based on IEEE 829-1998 standard for software test documentation).



It can be seen from above that the test plan is an important document. The test plan is a document that changes over time. This change particularly risks, schedules, staffing and training needs.

Test plan items

According IEEE 829 – 1998 standard for s/w documentation the test plans consists of 16 items

1. Test plan identifier
2. Test plan introduction
3. Test items
4. Features to be tested
5. Features not be tested
6. Approach
7. Items pass/fail criteria
8. Suspension or resumption criteria
9. Test deliverables
10. Testing task
11. Environmental needs
12. Responsibilities
13. Staffing or training needs
14. Schedule
15. Risk/contingency
16. Approvals

Test plan identifier:

This is the unique identifier assigned to this test plan. Most of the organizations have a lot of paper work and it is very important to be able to identify exactly which document if dealt with or referred d to. The test plan identifier should uniquely distinguish the test plan.

The plan may well go through a number of versions. The identifier may remain well the same throughout these changes and the versions detailed in the tables at the front of the document or the name of the document may change. Example, by the addition of the version numbers.

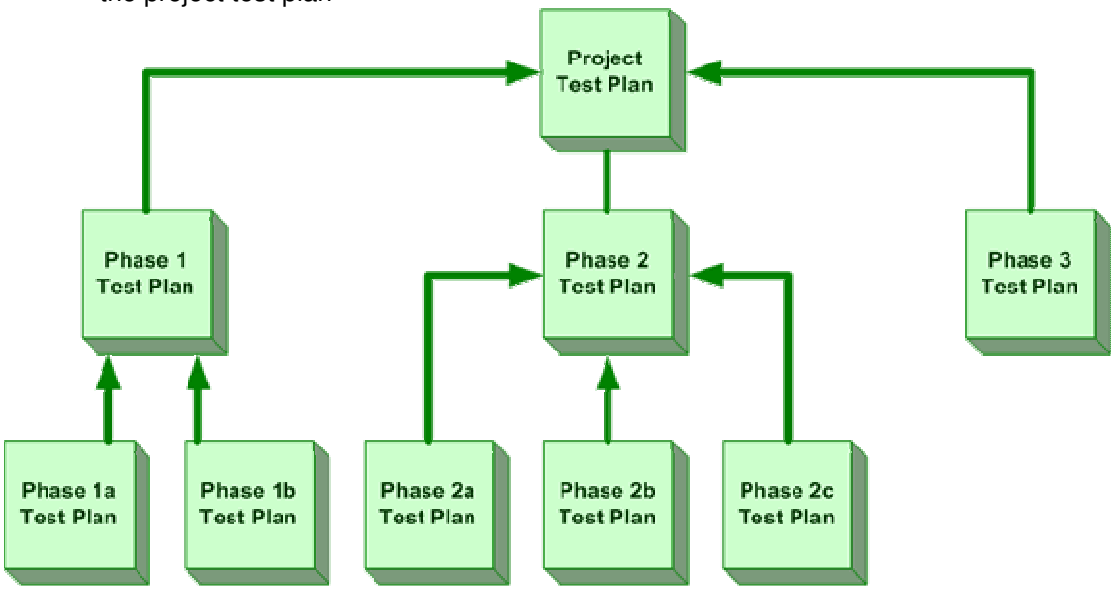
Test plan introduction:

This is basically an overview of the test plan summarizing the retirements, what needs to be achieved and why testing is necessary

In the introduction should be a summary of the software items and the s/w features to be tested. Then need for each item and its history may be included

There should also be reference to the following documents when they exist in the highest level of testing

1. project authorization; what person or group , authorized the project and its associated expenditure
2. project plan – as we have seen the project plan feeds into the test plan
3. quality assurance plan – how the quality of the desirable of the projects is top be monitored and by whom
4. configuration management pan – how and by whom the configuration management asks will be carried out and monitored
5. relevant policies – for example the best practices adhered to by the organization
6. relevant standard – for example software testing standard of legal standard that must be followed
7. in multi level test plans each lower plan must be reference the next higher level plan as demonstrated in the diagram below
8. here there are numerous test plans for the various level of the project
9. the phase 1a test plan would reference the phase 1 test plans which in turn would refine the project test plan



Test items:

A test item is a software item which is to be the object of testing so could be a program, an object, a module or a function. Test items a re also known as test assets (along with lot of other tings).

In this selection of the test plan:

- Identify the test items including tier version and/or revision levels. Also specify the characteristics of their transmittal media that impact hardware requirement sot indicate the need for a logical or physical transformation before testing can begin) for example if programs must be transferred from tapes to disc).
- Supply references to the following test documentation if exists:
    - A. requirements specification
    - B. Design specification
    - User guide
    - D. Operation guide
    - E. installation guide
- Reference should be made to any incidents reports relating to the test items. Any items that are to be specifically excluded from testing may also be identified.

Features to be tested

in this section of the test plan , the identify all software features and combinations of all the software features to be tested

Identify the test deign specification associated with each features an each combination In other words list all those features of all the program and the system to be tester “in scope”

This acts as a good checklist when completing the exit criteria.

Features not to be tested

In the section of test plan identify all features and significant combination of features that will not be tested and the reasons.

In other words identify those features that are “out of scope”.

Approach

In this section of the set plan the overall approach of testing is described.

For each major group of features or all combinations of features the approach is specified that will ensure that the feature groups are adequately tested

Specify the major activities, techniques and tools that are used to test the designated groups of features. Such things as performance tools, win runner (an automated test running tool or test director (another tool use for incident management, specification coverage, etc)

The approach should be described in sufficient details to permit the identification of the major testing task and the estimation of the time required to do each one.

The level of testing may also be discovered

Specify the minimum degree of the comprehensiveness desired an identify the techniques that will be use that will judge the comprehensiveness of the testing effort. for example determine which statement have been executed at least once. Techniques could be requirement based – for

ex. A certain percentage of requirements must be covered or that critical functions have all been covered.

Specify an additional completion criterion – the frequency of errors. The techniques to be used to trace requirements must be specified.

Identify significant constraints on testing, such as test items availability, testing resource availability and deadlines.

All of these are fairly at a high level (we are after all talking about a high level test planning) but should be in sufficient detail to enable estimation of test efforts in terms of time and resources

Item pass/fail criteria

Specify the criteria to be used to determine whether each test item has passed /fail testing.

We are a talking about test items here not individual test cases, so the past criteria could , for example, be defined as a percentage of test cases for the item having passed (say 95 %)

A second example could be that no severity A defects should be detected and not fixed i.e. remain outstanding. This metric could also include faults estimated to remain in the s/w but not yet detected.

Suspension / resumption criteria

In this section of the test plan, specify reasons why testing could be suspended;

Specify the criteria used to suspend all or a portion of the testing activity on the test items associated with the test plan. These could be, for example, if a major fault was uncovered that caused the system to be so unreliable that the further testing could be a waste of time.

Also, specify the testing activities that must be repeated on resumption;

Specify t he testing activities that must be repeated, when testing is resumed after suspension. The steps, resources, configuration necessary to enable resumption should be detailed.

Who would decide when testing should be resumed?

There is no definitive answer but it is likely to be the test manger or project manager, depending on the size of the project and the way responsibilities have been allocated.

Test Deliverables

Every thing that is produced on a testing project is in one way or another test deliverable. All are test assets that should be documented.

Identify the deliverable documents. The following documents should be included;

- Test plan
- Test design specification
- Test case specification
- Test procedure specification
- Test item transmittal reports
- Test logs

- Test incident reports
- Test summary reports

Test input data and test output data should be identified as deliverables. Test tools (for example; module drivers and stubs) may also be included.

Testing tasks

In this section of the test plan, identify the set of task necessary to prepare for and perform testing. This could include

1. test case identification
2. test case design
3. test data preparation including storage
4. creating a restorable baseline

identify all inter task dependencies and any special skills required, for example; automation (WinRunner) skills, spreadsheet skills.

Environmental needs

In this section of the test plan specify both the necessary and desired properties of the test environment.

This specification should contain the physical characteristics of the facilities including

- the hardware
- the communications (for ex: internet access)
- system software

The mode of usage of the test environment should be described (for ex: stand alone, network), and any other software or supplies needed to support the test should be listed.

Also specify the level of security that must be provided for the test facilities, system software and proprietary components such as s/w , data and hardware. For example, it may not be possible to log onto a network without passwords or sufficient access levels. It may not even be possible to install software unless in possession of administrator privileges.

Identify special test tools needed. Identify any other testing needs (example; applications or office space). Don' t forget to specify things that may be taken things for granted. For example; a desk for 4 testers will probably be enough space. One PC per tester may be a minimum.

Identify the sources for all needs that are not currently available the test group. Note that

- Different terminologies may be used to describe test environments; model office, test harness etc.
- The planning that goes into this area must be detailed. It may be better to have a separate very detailed test environment plan and referred to it as a test plan.
- Sometimes environments need t be shared causing problems with testing and resting baselines.

Responsibilities

In this section of the test plan, identify the groups responsible for

- Managing, designing, repairing, executing, witnessing, checking, and resolving
- Providing the test items identified and the environmental needs identified.

These groups may include the

- Developers
- Testers
- Operation staff
- User representatives
- Technical support staff
- Data administrative staff
- Quality support staff

It is important to understand the roles needed and the skills required. Allocating those responsible will be a process of matching those with appropriate skills to an appropriate task. Taking into account their availability and cost.

With larger projects there will be separate testing teams responsible for their own stage of testing (for example; SIT, UAT and so on).

Responsibilities need to be allocated to be most efficient use of these teams, keeping duplication of effort to a minimum.

Staffing and training needs.

In this section of test plan, specify test staffing needs by skill level.

Try not to specify a particular, named person. Always specify a position, for example, manager, team leader, test analyst or automation analyst.

Specify for each post identified the skills required; excel, access, win runner, etc

Identify training options for providing necessary skills; in-house, mentor, external, refresher courses.

Schedule

In this section of the test plan

- Include time scales, dates of testing tasks
- Include test milestones identified in the software project scheduled
- Detail all items transmittal events
- Define additional test milestones needed
- Estimate the time required to do each testing task
- Specify the schedule for each testing task and test milestone
- Specify periods of use for each testing resource (including facilities, tools, and staff)
- Specify test staffing needs by skill level

The schedule helps others plan for example those in charge of environments.

Risks and contingencies

In this section of the test plan identify the high risk assumption of the test plan. For ex ample, assuming that there will be no illness among the testers, there will be no catastrophic failures or major show stopper bucks found in certain modules.

Specify contingency plans for each assumption that is the actions to be taken to minimize the impact on testing. If the risks materialize for example delayed delivery of test items might require increased night shift scheduling to meet the delivery dates.

Approvals

In this last section of h test plan specify the names and titles of all the persons who must approve the plan.

Provide space for the signatures and dates of the approvers.

This is important as it shows that the document has been reviewed, agreed and that it has backing of those in charge of their relevant stages of the project.

Entry and exit criteria

- Entry criteria

Conditions that have to be fulfilled in order for testing to commence on a particular test item or group of items.

Entry criteria could include for example the availability of a certain resource (a person with particular knowledge – a particular piece of hardware or are reliable communication link – although the later may come under the environmental needs). Other criteria may be that training has to have been carried out or the realistic deadlines must have been specified.

Entry criteria must be defined before the design starts

- Exit criteria

Conditions to be met to allow testing to stop.

For example;

- Test pass rate must be met or surpassed
- All major defects / anomalies must have been resolved
- Performance targets must have been achieved
- As with entry criteria, exit criteria must be defined and agreed upon testing starts

Sources of test data

New systems may not have existing standing data, so it must be created in some manner. This can be a very specialized job, particularly if database relations are complicated. Some organizations have whole teams dedicated to database designs and data populations.

Newly created data may not be represented;

- It has not been in a live system, so may not look like real data

Copied data has particular problems

- Data protection issues – data must be anonymous. One insurance company used real data testing of a purchased system. Be cause of the way the system was built the test environment has to be linked to the live system and thousands of customers were sent

bills for the ISAs they has purchased during testing. This caused embarrassment to the company and a financial loss as they had to pay compensation in many cases.
- Conversion issues – special routines may be need a existing data. Again, this is a specialized task which can take a huge amount of time depending on the size of the complexity of the database involved.

Data subsets

- Because databases can be huge – subsets of data are often used. Data needs to be chosen careful to ensure it is representative and has sufficient coverage of test cases.

Documentation requirements

All project documentation should be

- Standardized
- Stored centrally
- Secure
- Up to date
- Accurate

Organized and complete documentation s a very important pre-requisite in any project and no more so in the context of high level test planning.

One of the most frustrating aspects of testing project (indeed of any project) is not being able to find a document relating to a problem that you are trying to resolve or having found the relevant document not being able to understand because of its complexity or the fact that it is patently out of date.

Ideally, all documentation produce should be in a form prescribed by the organization – i.e. standardized, so that the reader become familiar with the basic structure of documentation and find information more easily within individual documents (document author, date of wiring, date of any changes, document file name, location, etc should be easily found).

Documentation should be kept in a central location available to relevant parties. There is no point placing a test plan or project plan on a password protected standalone PC in a locked room. It should be on an accessible directory on a network or if this is not possible and up-to-date hard copy at least be available at all times.

If possible all documentation should be in the charge of the project librarian who ensures that all documentation is up to date and available and who performs the function of protecting documents from unauthorized changes.

Summary

- Test plans include
    - Scoping the test
    - Risk analysis
    - Test stages
    - Entry and exit criteria
    - Test environment requirements

The test plans that change over time. Things change – particularly risks, schedules, staffing and training needs.

Acceptance testing

What is acceptance testing?

Formal testing conducted to enable a user, customer or other authorized entity to determine whether to accept a system or component.

BS 7925 – 1; 1998

There are different types of acceptance testing

- User acceptance testing
- Operations acceptance testing
- Alpha testing
- Data testing

UAT – User acceptance testing

None usually by acronym – it is exactly what it says it is – testing the system for its acceptability by the future users.

- It covers all areas of a project, not just the software system.
- It is also known as business acceptance testing or business process testing
- It is carried out by user representative with the help of the test team. Tests are designed with the users perspective in mind using real data and real scenarios.
- UAT is the final business validation that the application is ready for release.

UAT Approach

Once common approach uses a Model Office. The setup will be exactly as the working environment – including such external interfaces such as telephones (for software controlled call centre applications).

It uses a black box approach to testing – i.e. testing without reference to the s/w code

It uses thread testing techniques – to verify high level business requirements in practice typical tasks are carried out during testing as they would be in normal usage, end to end.

User acceptance testing not only applies to s/w, it can also be applied to system documentation such as user guides.

UAT Test case selection

Test case may be selected by appropriate system tests but may need to be extended to cover end to end process. Before final selection is made discussion with user representatives is essential.

The requirements of he system under test should be reviewed to identify important areas or functions to be proved. Additional tests may be implemented such as those covering

- Usability
- Documentation (user manual)

- Help facilities (both online / document based)

If the system test was not run in the life environment then any system test concerned with links with other systems must be re run under user acceptance testing. For example credit card validation routines where links with validation services may need proving or user privilege validation where links with higher networks may be required.

Any system tests checking interoperability or compatibility should also be re run under UAT.

UAT data requirements

The use of life data is preferable – but this poses problems with security or data protection legislation. Clearance may be required if data is sensitive even if not protected for legal reasons (for example it may contain trade secrets or secret formulae)

There are ways around these problems

- Special s/w tools are available to scramble or de personalize the data
- Copies or subsets could be used – but beware of the problems of using non-representative data

Recreating data can be a specialized job , particularly with complex data models and building up histories is very time consuming and difficult to make realistic.

Operations acceptance testing

This is distinct from UAT. OAT is used to confirm that the application under test confirms to its operations requirements – in terms of

- Installation of a upgrades
- Back up and restore of the system s/w or the data
- Archiving of old data
- Registration of new users
- Assigning privileges to users

Contract acceptance testing

This is a demonstration for the acceptance criteria which will have been defined in the development contract. Before the s/w is accepted testing is performed to show that the s/w matches its specifications. A percentage of contracted development fees may be paid as functions are proved.

Alpha testing

Simulated or actual operational testing at an in-house site not otherwise involved with the s/w developers

BS – 7925 – 1/ 1998

Customers could perform this type of testing. A stable version of the s/w is required. s/w is used as if it had been bought of the shelf.

Alphas and beta testing – in both alpha and beta tests when the s/w seems stable people who represents the market use the product in the same way that they would if they bought the finished

version. Alpha testing is usually unsupervised and unaided, but as it is carried out in house, unobtrusive observation and note taking during the tests is a possibility.

Beta testing

Operational testing at a site not otherwise involved with the s/w developers

BS – 7925 – 1/1998

Beta testing is almost the same as alpha testing but it is carried out at the users of customers own site i.e. completely away from the developers.

When the s/w is stable people who represent your market use the product in the same way that they would if they bought the finished version and provide their comments. Beta testing is totally unsupervised and unaided although a helpline may be provided. New versions of the Microsoft windows are often tested like this. It provides a greater variety of configuration that could be simulated in house at a much reduced cost. Quite often beta testers are not paid but offer their service because it gives them a chance to be the first to see the new software. Feedback from the beta testers can be gathered, for example by questionnaires or online defect reporting systems.

Summary

- Acceptance testing is final test of functionality by the business
- Use should be heavily involved
- Contract acceptance testing – demonstration of criteria defined in the contract
- Alpha tests are performed at the developer's site while beta tests are performed at the testers sites.

Integration testing in the large

A couple of definitions from the BS – 7925 -1 / 1998

Integrations – process of combining components into larger assemblies BS 7925 -1 /1998

Integration testing – testing performed to exposed faults in the interfaces and in the interaction between integrated components. BS 7925 – 1 /1998

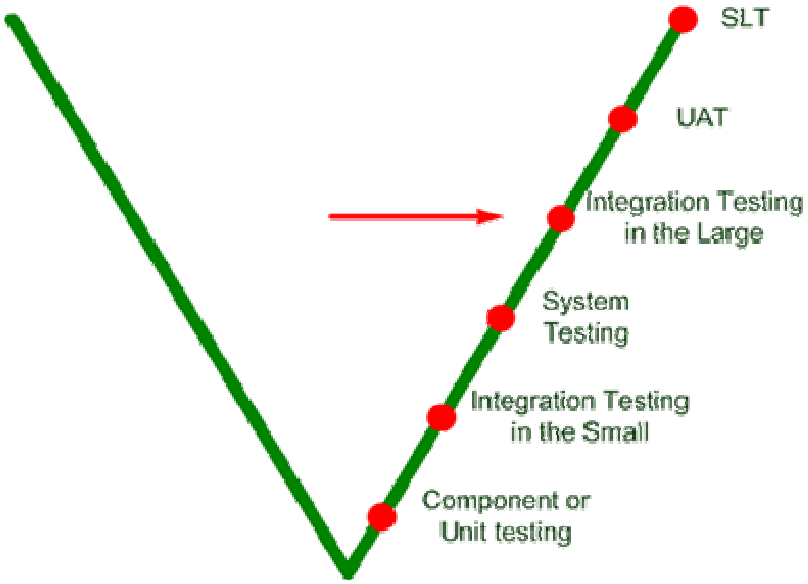
Integration in the context of "integration in the large" means integration with other (complete) systems. It is also known as system integration testing and compatibility testing.

- Objectives – to provide confidence that the application under test is able to inter operate with of he s/w systems.
- To prove that the application doesn't' t produce faults in other systems

The application may well correctly in the test environment standing alone, but the protocols and the messages used when interacting with other systems may fail. Of the s/w systems may be those that are co-resident in the memory not just those that are completely external.

The position of the integration testing in the large in development life cycle.

In the v model of s/w testing integration testing in the large normally comes after system testing



Approach

Set up the tests

The first activity is to review interoperability requirements.

Refer to the requirements specifications and identify the communication requirements. What does the communication between the systems need to do and how it is supposed to do it. What are the business requirements that utilize the communication channels. Are there performance requirement – for example a minimum speed of communication or transactions per minute? are there specific demands for system performance – for example are maximum delays defined for when retrieving ? what are the security issues revolving around interoperability. Next, review the life system environment;

This may highlight particular issues to be tested – compatibility between operating systems with translation s/w (middleware such as CORBA) or with data synchronization which means databases.

Lastly, create test cases and scripts;

Having performed the reviews and identified the risks associated with interfaces to other systems, tests can be designed and returned. It is sometimes possible to reuse system test scripts or those from other phases of testing. (For example, unit or link testing).

Environment

The system should be tested in a live environment when testing otherwise than in the life environment it is extremely difficult to have confidence in the results of the tests.

If the life environment is not available for whatever reason – a test environment will have to suffice. This could be a replica of a live environment i.e. a model office – which should be a faithful copy of the live environment.

If external interfaces are not available – then use stubs or drivers. A stub is dummy procedure that returns a value when called for. A driver is supporting code and data that is used to provide an environment for testing heart of a system in isolation.

Types of approach

Incremental approaches

Top down

In the top down approach to integrate testing the component at the tope of the component hierarchy is tested first with lower level components being simulated by stubs.

Tested components are then used to test their next lower level components. The process if repeated until the lower level components has been tested.

In the bottom approach to integration testing the lowest level components are tested first and are then used to facilitate the higher level components. The process is repeated until the component at the tope of the hierarchy is tested.

Non – Incremental approach.

Big bang

With the big bang approach – no incremental testing takes place prior to systems component being combined to form the system. This tends to be more common than the incremental approaches.

Summary

- Testing of integration with other (complete) systems
- Includes the identification of, and the risks associated with , interfaces to these other systems
- There are incremental and non incremental approaches to integration
- Incremental
    - Top down
    - Bottom up

    - Non-incremental
        - Big bang

Non-functional system testing

Introduction

Non-functional system testing

Testing of those requirements that don't relate to functionality for example; performance, reliability and usability
BS 7925 -1/1998
Actually defined as non functional testing or technical requirements testing in BS 925

Users normally concentrate on the functionality of the system. Can it retrieve the correct records? Can it add up correctly? Non-functional system testing looks at those aspects of the system which affect the use in other ways. For example: how quickly does it run? Is it secure? Is it easy to use?

Why carry out non-functional system testing?

Functionality is important, but other factors can be vital to the success of the application. A system may be able to produce a insurance premium quote correctly but if it takes 2 hours to do so it may not be of much use to the business.

If an online banking system can only handle only 1 transaction at a time it will probably not cope with the expected number of concurrent users.

There may also be contractual targets for the non-functional aspects of the system that need to be proved , speed , security .

Non-functional test techniques

There are non-functional system test techniques.

- Load
- Stress
- Volume
- Performance
- Storage
- Security
- Usability
- Installability
- Recovery
- Documentation

Load testing

The simulation of business data volumes and multiple users performing typical business procedures or

Testing geared to assessing the ability of the application to deal with the expected throughput of the data and the users

In other words, performing tests to see if the business processes function properly under heavy load. Will the system remain stable under the expected load of 2000 concurrent users?

Very often the terms load and performance are used interchangeably, to ensure that everyone involved knows exactly what they are talking about.

Stress testing

Testing conducted to evaluate a system or a component at or beyond the limits of the specified requirements

BS 925 – 1 / 1998

Stress testing is pushing or loading the system or the component until it reaches its limits of operability. It assesses component of systems to and beyond the limits of expected use.

Examples:

Egg; when egg banking was launched in 1999 there was an initial surge of online requests to join. The system couldn't cope with the demand. New requests were turned down until measures were out in lace to enable the system to cope with the demand. This was expensive rework in itself but it is impossible to assess how much business was lost by potential customers not coming back.

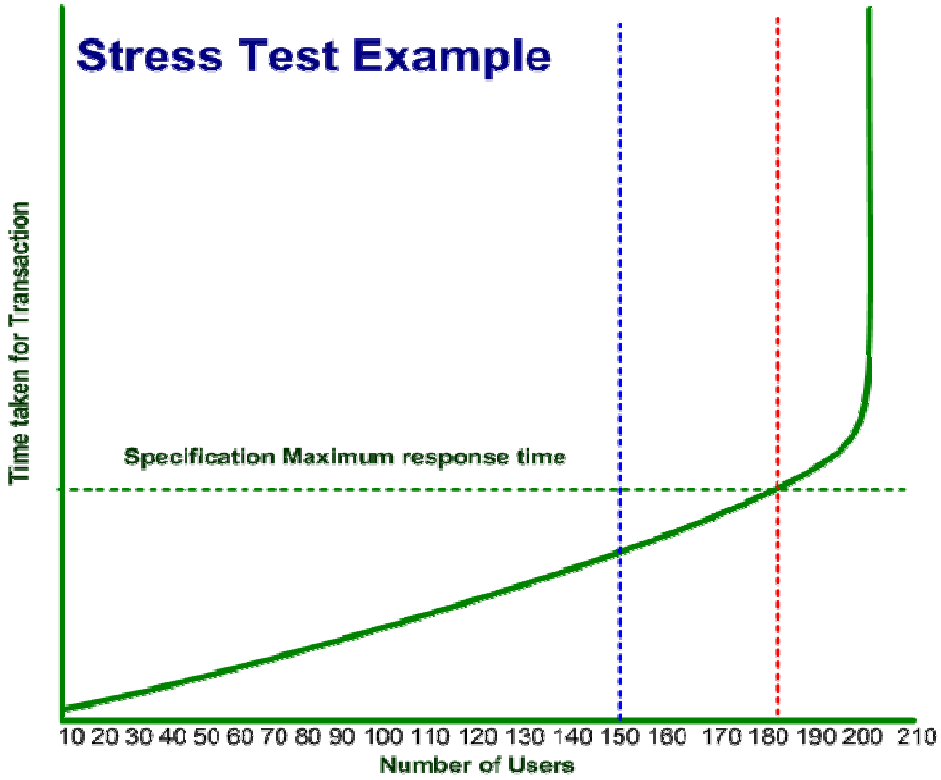
2001 census website; this was overwhelmed with the number of people trying to access the site. It took many months before the system was upgraded to be able to be responded t the public. Stress testing involved subjecting the program to unexpected loads and stresses. Don't confuse stress testing with volume testing. A heavy stress is a peak volume of data encountered over short time. An analogy is an appraisal of a typist. A volume test determines whether the typist can cope with the draft of a large report whereas a stress test determines whether the typist can type at a rate of 50 words per minute.

Another ways of stressing a system is to run it under less than ideal conditions. For example;

- With lower than usual memory
- With smaller disk space
- With slower cpu's
- With slower modems
- Or communications lines

Stress testing

The diagram below shows the graphical representation of results of a stress test. The time taken for a transaction was measured for different numbers of concurrent users of a system. As the number of user's increases the systems response time increases eventually (at just over 180 concurrent users) the maximum response time allowed in the specification is reached (indicated by the red dotted line). If the number of concurrent users is more than the maximum expected (in this case 150, indicated by the blue dotted line) the in the system exceeds its specification and passes the test. The system is then pushed beyond the limits until at above 200 users it appears to produce infinite response times (where the graph line is vertical), at which point the system may have crashed. Further investigation may be required into this response.



Volume testing

Testing in which the system is subjected to large volumes of data.

BS 7925-1:1998

Volume testing is the simulation of high volumes of data over a long period of time and is also known as soak testing or smoke testing.
Note the inclusion of the time element. Functional system testing and load testing does not subject the system to prolonged use. Volume testing ensures that the system is put through its paces over substantial times. This may help to show such errors as memory leagues, which may cause system crashes over long periods (for example, if programs do not clear either themselves or unused variables from memory).

Also known as endurance testing, it is particularly useful for testing web systems that are online 24hrs a day.

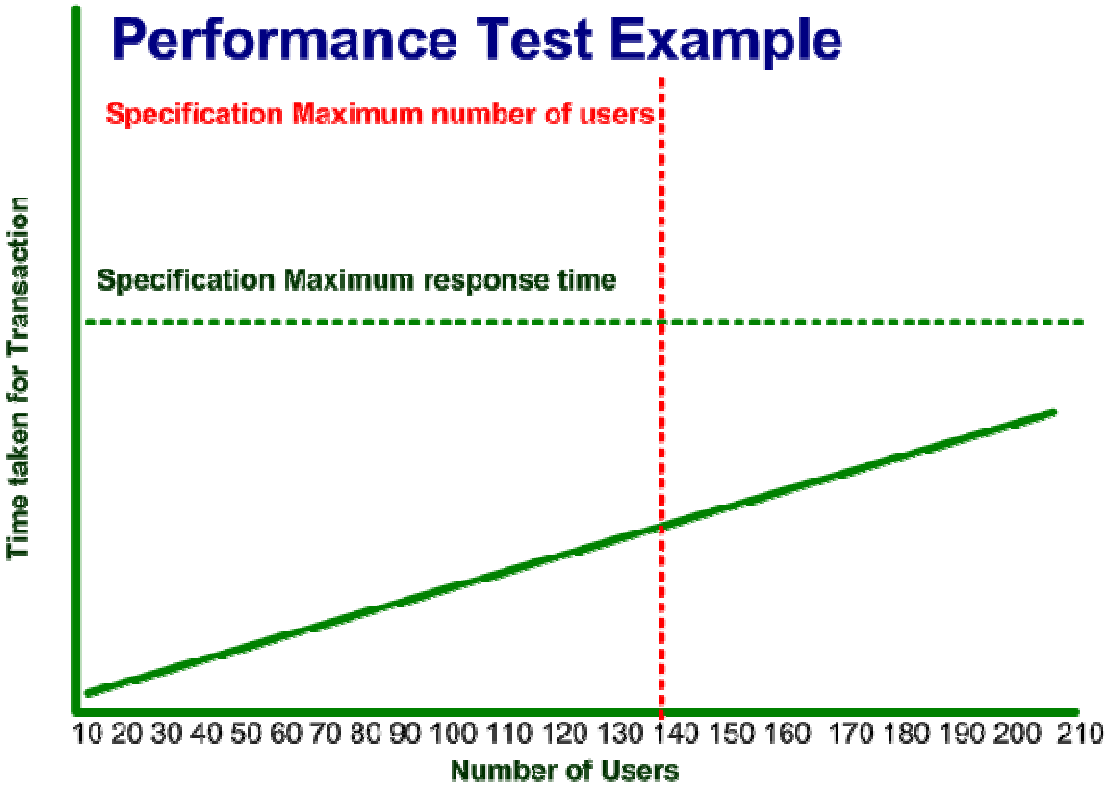
Performance testing

Testing conducted to evaluate the compliance of a system or component which specified performance requirements BS 7925-1:1998.

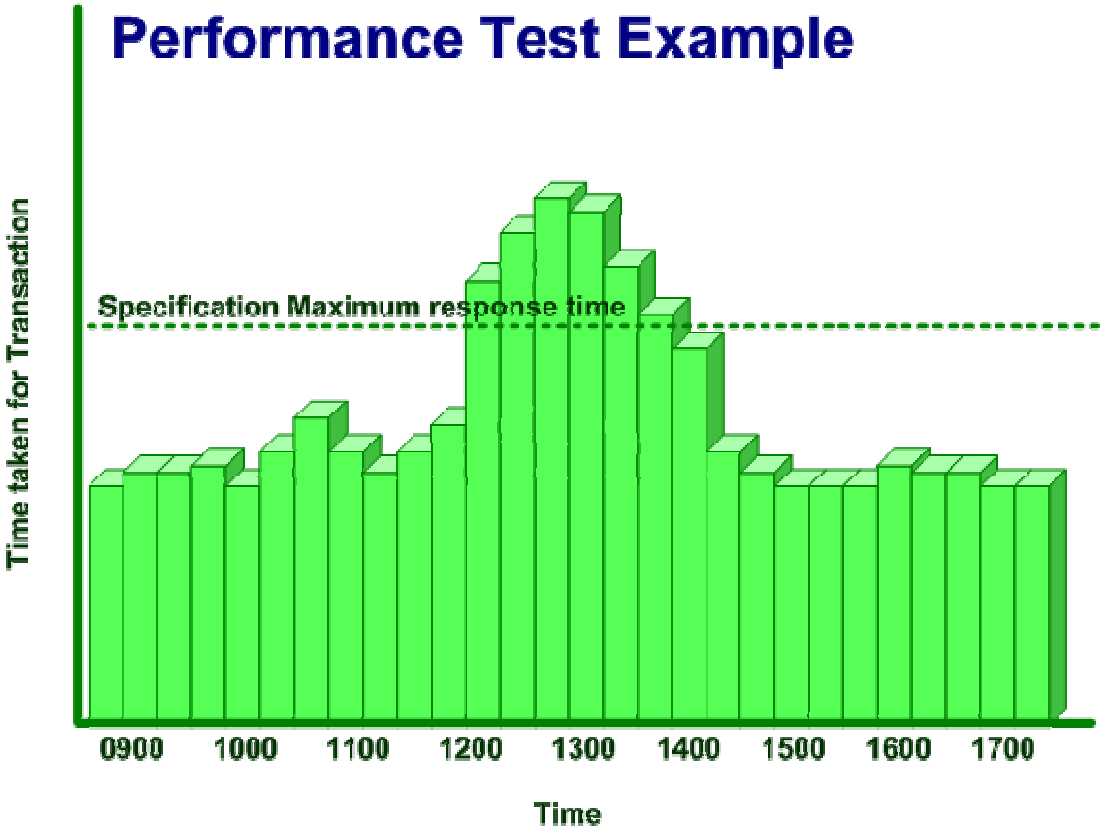
It is possible to test a single component. Anything from a component to a full process could also be tested. For example, specifications may say that the time taken to process an order must be less than 20 seconds. This process could be timed for a number of data inputs, for example, with one user and a number of users.

The graph below shows the results from a performance test. The maximum number of users that the system is expected to be able to handle is 137 shown by the vertical red dotted line. The

maximum response time specified for a particular transaction is shown by the horizontal green line. The system us performance tested by timing the transaction in question for different number of concurrent users and the results plotted (the solid green data line). In this case it can be seen that when the maximum number of users are concurrent the transaction time is well below the maximum specified, so the application passes this test.



Below graph is another example of performance testing. In this instance showing the time taken for a particular function at various times during the day. Notice the response times are longer in the middle of the day (presumably when more users are on the system over launch) and are in excess of the maximum specified time in the requirements, so the system fails this particular test.



Storage testing

Testing whether the system meets its specified storage objectives. Storage testing relates to the amounts of main and secondary storage used by the program. This depends on use of memory by code (for system memory) and the size and number of records for hard-disk space. Other aspects that can be tested are, for example, the sizes of required temporary of spill files.

Results of this type of tests may feed into later stages of capacity planning. Test cases should be devised to show if the storage objective have been met.

Usability testing

Testing the ease with which users can learn and use a product.

This is a very wide ranging subject. Usability testing is an attempt to find the human factor or usability, or problems in systems and applications. It is an important and difficult area because, in general, insufficient attention is placed on studying and defining human factor considerations of applications and the assessment and measurement of usability is highly subjective. The following is the list illustrating the kinds of considerations that might be tested under this heading:

- Has each user interface been designed with a reference to the intelligence and educational background of the end user? Have environmental pressures of the end user been taken into consideration
- Are the outputs of the program meaningful and not abusive? Are they devoid of “computer gibberish?

- Are all errors diagnostic? (For example error messages) clear, informative and straight forward, or is a PhD in computer science required to comprehend them? For instance, does the program produce such messages as “by the k 1228 open erupt on file CIS in a bend code equal 102”?
- Does the total set of user interfaces exhibit sufficient “conceptual integrity? Does it have an underlined consistency and employ a uniformity of syntax, conventions, semantics, format, style and abbreviations?
- If accuracy is vital (for example, in non-online banking systems), is sufficient redundancy present in the input (account number and customer name)?
- Does the system contain an excessive number of option or options that are unlikely to be used?
- Does the system return some type of immediate acknowledgement for all inputs?
- Is a program easy to use? For instance, does entry of a command require repeated shifts between upper and lower case characters or a need to press three widely spaced keys at once?

Documentation testing

Testing concerned with the accuracy of documentation

Why test documents?

If installation instructions are user instructions are wrong, they will not only be seen as bugs by the users, but they may lead to further errors in use. Document testing can help to improve usability and reliability and can help reduce support costs.

What documents can be tested?

- Help files
- User manuals
- SQU messages
- Error messages
- Any form of output from the system

Installability testing

Testing concerned with the installation procedures for the systems. When installing software systems, normally number of options must be selcted and files and libraries must be allocated and loaded. Install ability testing is concerned with helping to ensure that these procedures work correctly.

At the same time, a valid hardware configuration must be present and the programs must be connected to other programs or systems. Another purpose of installation testing is to find any errors made during or as a result of the installation process. Other aspects of install ability that can be tested are:

- Does the installation actually work? I.e. does the installation actually produce a working application that appears to function correctly?
- Does the installation affect any other software for example Netscape can affect internet explorer and vise versa?
- Is it easy to perform? Or is it complicate and require a depth of knowledge unlikely to be possessed by an average user?
- Can installation be undone and leave system exactly as it was before?
  Many pieces of software leave unused files or registration details behind after un-installation.

Recovery testing

Testing aimed at verifying the system's stability to recover from varying degrees of failure.

Operating systems, database management systems and teleprocessing programs often have recovery objectives. This means it has been defined exactly how the system should recover from such events as programming errors, hardware failures and data errors. Recovery testing is used to prove these recovery functions.

Approach:

- Programming errors can be purposely injected into an operating system to determine if it can recover from them.
- Hard ware failures for example memory parity errors and I/O device errors can be simulated.
- Data errors (for example, communications line noise or invalid pointers in database) can be created or simulated to analyze the reaction of the system.
- Testing can ever be as simple as unplugging the PC for a few seconds or turning of the network servers.

Summary:

Functions are not all that need to be tested. Other considerations are:

- Load, performance stress testing
- Security
- Usability
- Storage
- Volume
- Installability
- Documentation
- Recovery

Functional system testing

Introduction

Functional requirements

A requirement that specifies a function that a system or system component must perform.

This can be virtually anything from a requirement that an entry should be able to made on a screen to a detailed description of a complete calculation.

System testing

Process of testing an integrated system to verify that it meets specified requirements.

System testing allows unit or link testing (testing in the small in the V model).

Combining the two definitions thus gives a definition of functional system testing which is:

The process of testing an integrated system to verify that it meets specified functions that it must perform.

Problems with functional requirements:

The following demonstrates some of the problems in defining and interpreting requirements:

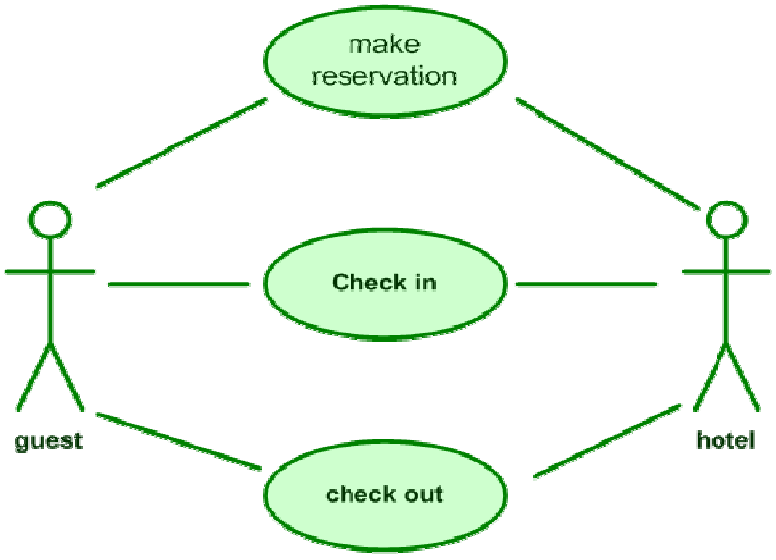
What the user asked for?
What the analyst specified?
What the programmer wrote?
What they finally implemented?
What the user really wanted?

This scenario is very common in software development. Specifying exactly what is required is difficult without being ambiguous. Misunderstandings can arise at all stages. A small mistake at early stage can multiply. It is important to review specifications at all stages to ensure what is being built is correct. Even if the function for what the user asked for is correct, it may not accord with exactly what user wanted.

Business process based testing:

This is the selection of test cases based on expected user profiles provided by:

- Scenarios: described by users based on their expected use of the systems.
- Or
- Use cases: these are more formalized and very common in systems using the UML and object oriented developments. Users of the system are identified together with the tasks they might undertake with the systems. Information is also sought about which tasks are most important so the tests can be planned and prioritized accordingly.



Use Case model for hotel checking in / out system

Summary:

- Functional system testing uses functional requirements on which to base the tests.

- Business process based testing employs scenarios or use cases from a user prospective on which to base tests.

Integration testing in the small

Introduction

Integration testing is defined as:

Testing performed to expose faults in the interfaces and in the interactions between the integrated components.

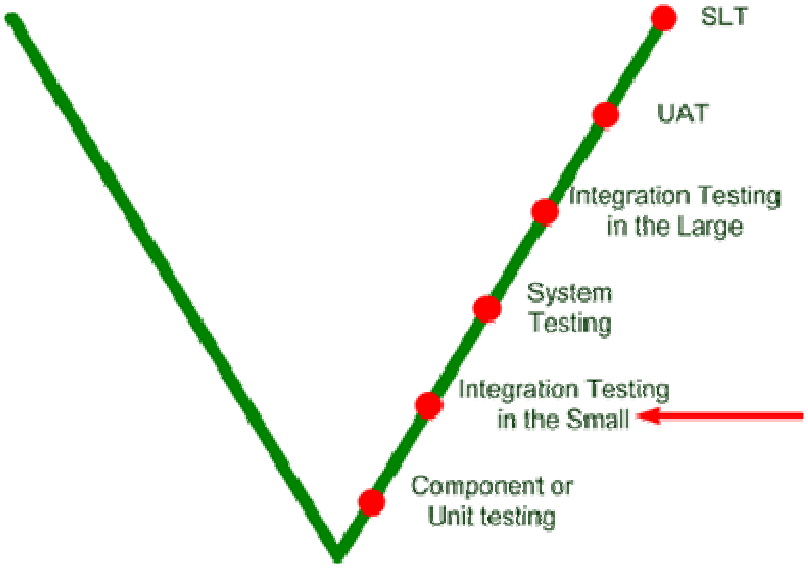
The objectives of the integration testing in the small (also known as link testing or compatibility testing) are:

- To provide confidence that individual software components are able to interoperate with other software components.
- To prove that the components do not produce faults in the other components.

The reasons for carrying out integration testing in small is that the application under test may well work correctly in the test environment standing alone, but the protocols and messages used when interacting with other systems may fail.

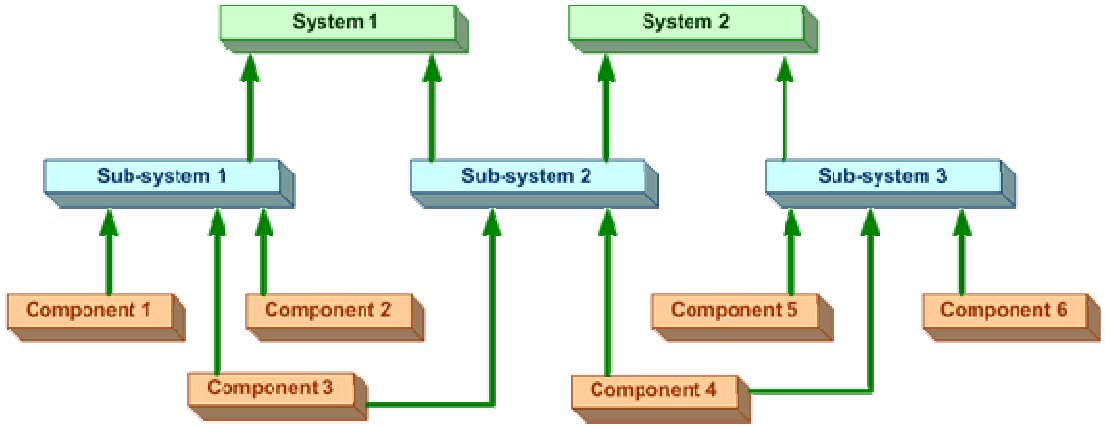
When we talk about other software systems in this context we also include those that are co resident in memory with the component under test, not just those that are completely external to the physical or logical systems.

Integration testing in the small usually takes place after component or unit testing in the V-Model.



Combination (integration) of software components:

The diagram below illustrates that software components are combined to make up sub-systems. In turn integration of sub-systems makes up systems.

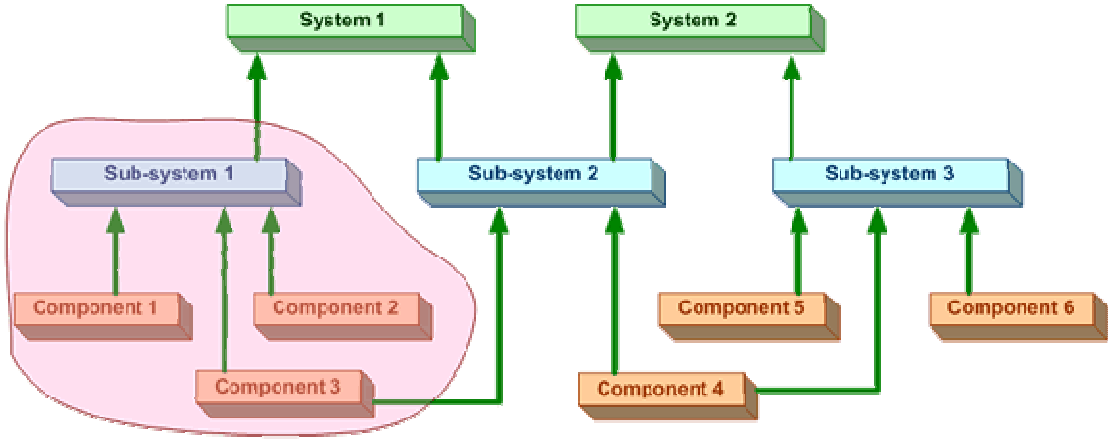


Note that component 3 is used in sub-systems 1 and 2. And also ends up both in system 1 and 2. Components are defined as a minimal software item for which a separate specification is available.

Note: component as defined here does not correspond to software unit as defined in VS ISO/IEC 12207 which requires separate compilability.

Example:

In the diagram below the shaded area shows which components may be tested at a particular stage of integration testing in the small? Components 1,2 and 3 are tested together to prove they interact without problems when combining to make up subsystem 1.



Stubs and drivers:

Stubs and drivers are frequently used when carrying out integration testing in the small.

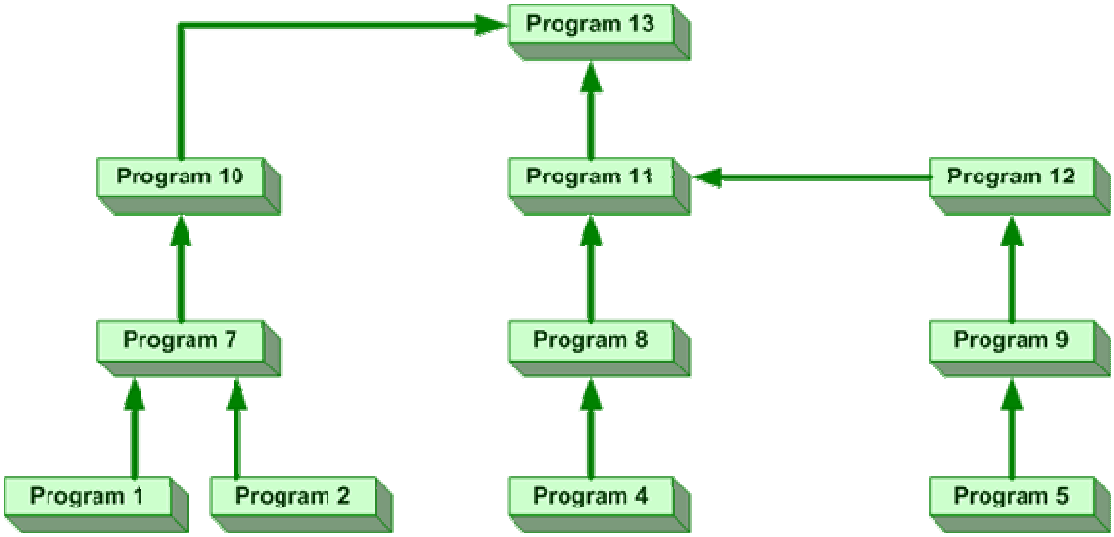
Stub:

Skeletal or special purpose implementation of a software module, used to develop or test a component that calls or is otherwise dependent on it. In other words it is a dummy procedure that returns a value when called for.

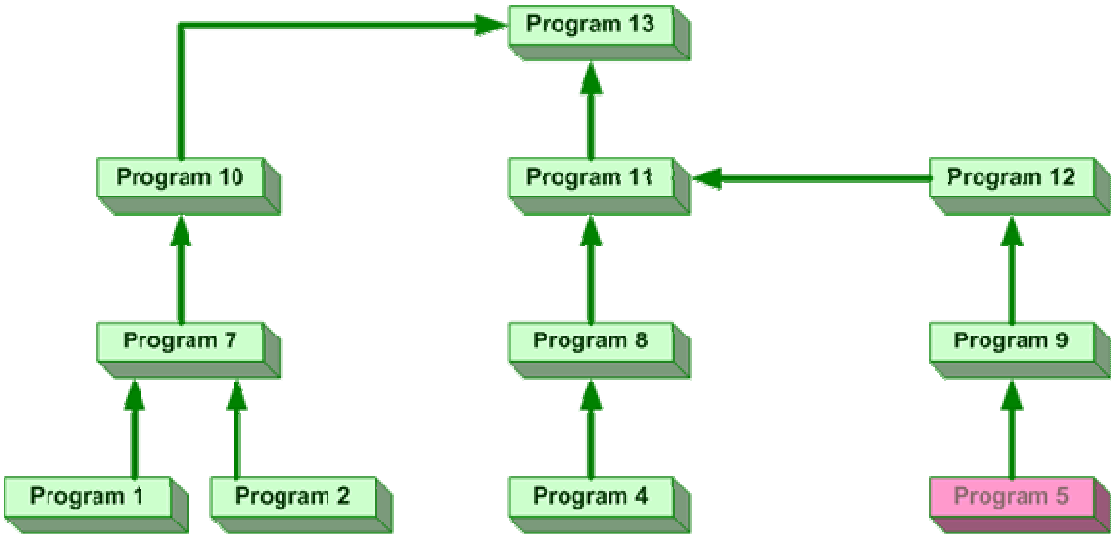
Driver:

Supporting code and data providing an environment for testing as part of a system in isolation. Do not confuse this definition with the BS7925 definition of a test driver.

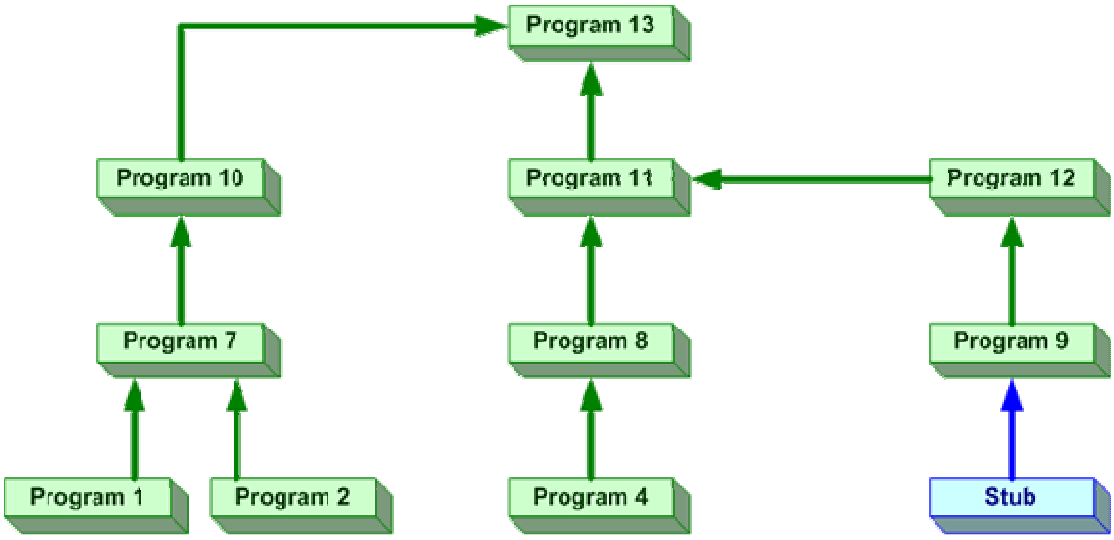
Stubs example:



Programs making up an application rely on one another to request values or to pass data. In the diagram above for example program 7 may receive request from program 1 and 2 and these in turn pass values back to program 7 in response. Without programs 1 or 2 program 7 cannot function fully.

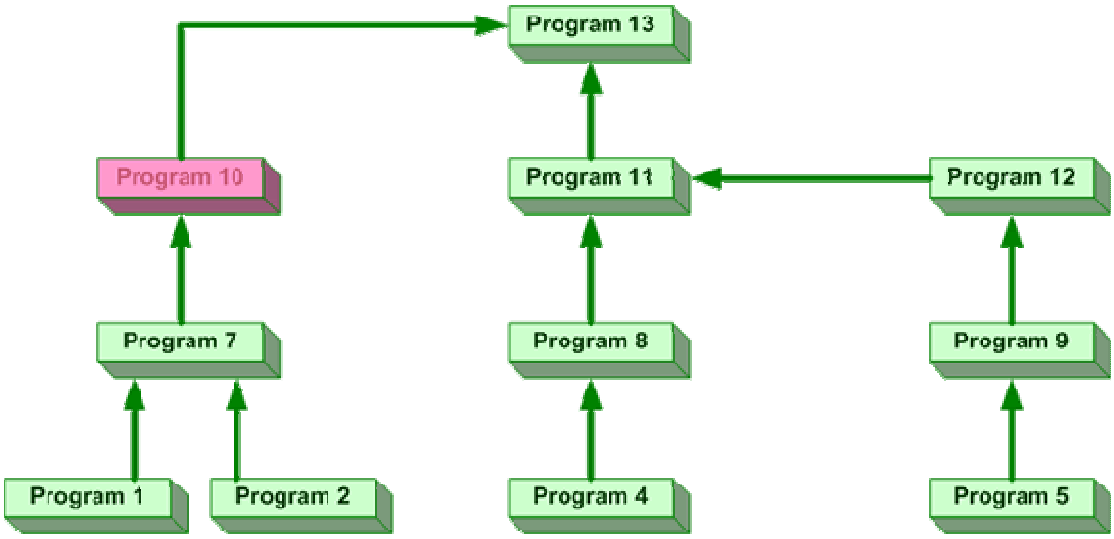


In this case all programs are completed except program files. Since this required directly by program 9 and thus by program 12, program 11 and program 13, the application cannot be thoroughly tested.

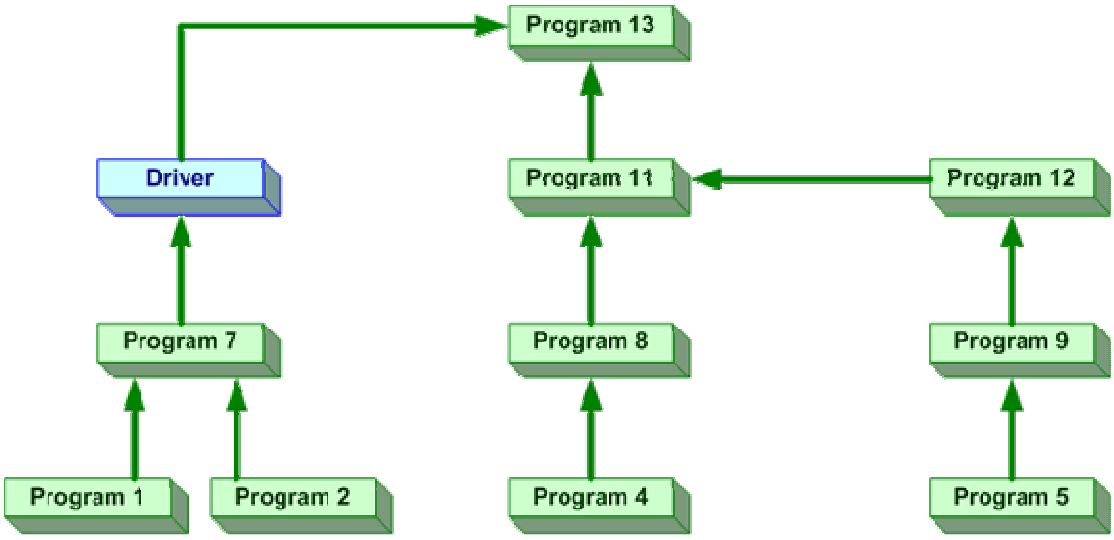


We overcome this difficulty by creation of a stub. The stub is a piece of software written by developers which does not contain all the functionality of the missing program. But it simulates by providing a value when requested by a program 9 in this case.

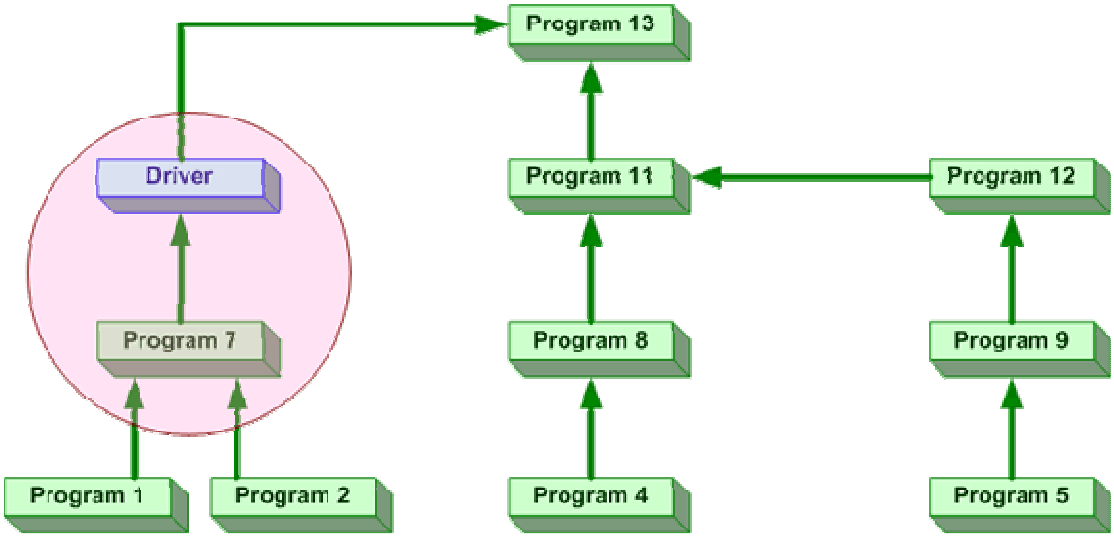
Driver example:



Similarly for drivers program 10 is the only one now not complete. We want to test program 1,2 and 7.



In this case we have a driver written this will simulate program 10's functionality. It may supply data or may send messages to program 7.



In this way we can test program 7 and 1 and 2 without having a full system. Drivers can also take a place of external systems when they are not available for some reasons.

Incremental approaches

Top-down testing

Approach to integration testing where the component at the top of the component hierarchy is tested first with lower level components simulated by stubs. Tested components are then used to test lower level components,

Bottom-up testing

Approach to integration testing where the lowest level components are tested first then used to facilitate the testing of the higher level components.

Note: in bottom-up testing the process is repeated until the component at the top of the hierarchy is tested.

Functional integration

This is integration testing where system components related to particular functions are integrated into the systems one at a time until the entire system is integrated.

Non-incremental approaches

Big-bang testing

Integration testing where no incremental testing takes place prior to all the system components being combined to form the system.

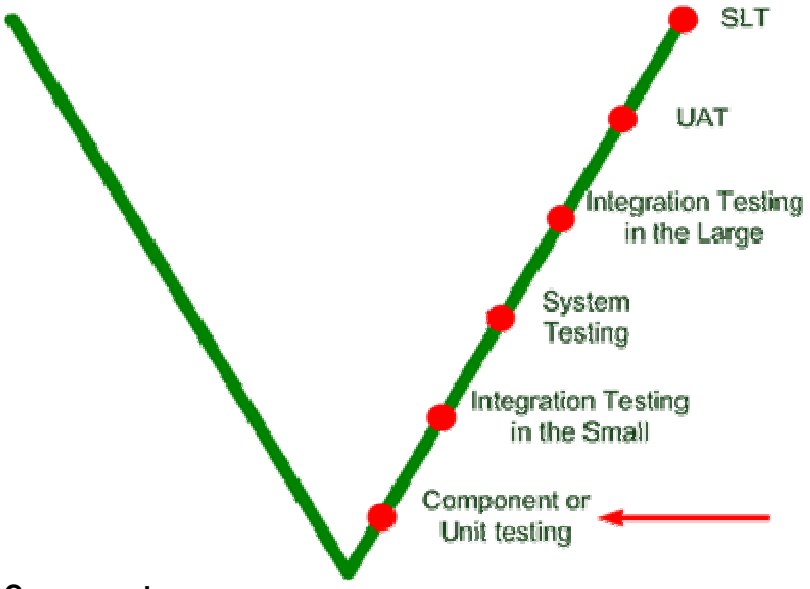
Summary:

- Components are assembled into sub-systems and systems
- Stubs and drivers replace missing components.
- Approaches to integration testing in the small
    - Incremental
        - Top-down, bottom-up, functional
    - Non-incremental
        - Big-bang

Component testing

Introduction

Component testing is the testing of individual software component. Also known as unit testing, module testing and program testing. Component testing is the first type of testing on the right-hand side of the V-model.



Component

Minimal software items for which a separate specification is available.
Note: component as defined here does not correspond to software unit which requires separate compilability.

A component may be just a few lines long or 10,000. it depends on the level of detail at which the components are specified. The British standard for software component testing is BS7925-2:1998.
Objective: the stated objective of the standard is to enable measurement and comparison of testing performed on software components. This will enable users of the standard to directly improve the quality of their software testing and there by improved the quality of their software products.

The standard:

- Prescribes characteristics of the test process.
- Describes the number of techniques for test case design and measurement, which support the test process.

What the standard covers:

Specified components

A software component must have a specification in order to be tested according to the standard. Give any initial state of the component, in a defined environment, for any fully defined sequence of inputs and any observed outcomes. It must be possible to establish whether or not the component conforms to its specification.

Dynamic execution

The standard addresses dynamic execution and analysis of the results of execution.

Techniques and measures

The standard defines test case design techniques and test measurement techniques. The techniques are defined in order to help users of the standard design. Test cases and to quantify the testing they perform. The definition of test case techniques and measures provides for common understanding in both the specification and comparison of software testing.

Test process attributes

The standard describes attributes of the test process that indicate the quality of the testing performed. These attributes are selected to provide the means of assessing, comparing and improving test quality.

Generic test process

The standard defines a generic test process. A generic process is chosen to ensure that the standard is applicable to the diverse requirements of the software industry.

What the standard does not cover?

Types of testing:

The standard excludes a number of areas of software testing, for example,

- Integration testing
- System testing

- User acceptance testing
- Statistical testing
- Testing of non-functional attributes such as performance
- Testing of real-time aspects
- Testing of concurrency
- Static analysis such as data flow or control flow analysis.
- Review and inspections

Complete strategy for all software testing would cover these and other aspects.

CHAPTER 3

Dynamic Testing

Black box testing

Black box (or functional) testing is concerned with testing the specification of a system or a program. It deals with what the system does, not precisely How it does it.

Functional or Black box testing

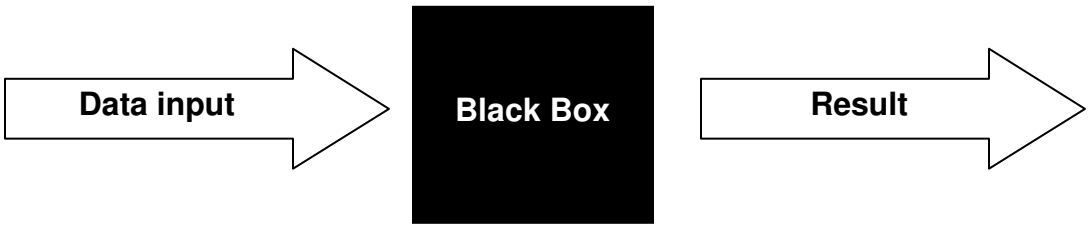
Test case selection that is based on an analysis of the specification of the component without reference to its internal workings

BS7925-1:1998

All attributes of the system are looked at with a view to defining acceptance criteria that are objective and measurable.

Black box testing is also known as facility testing or feature testing.

The idea behind black box testing is that the test inputs (data) disappear into a black box, which performs actions on the data and which produces as result as an output.



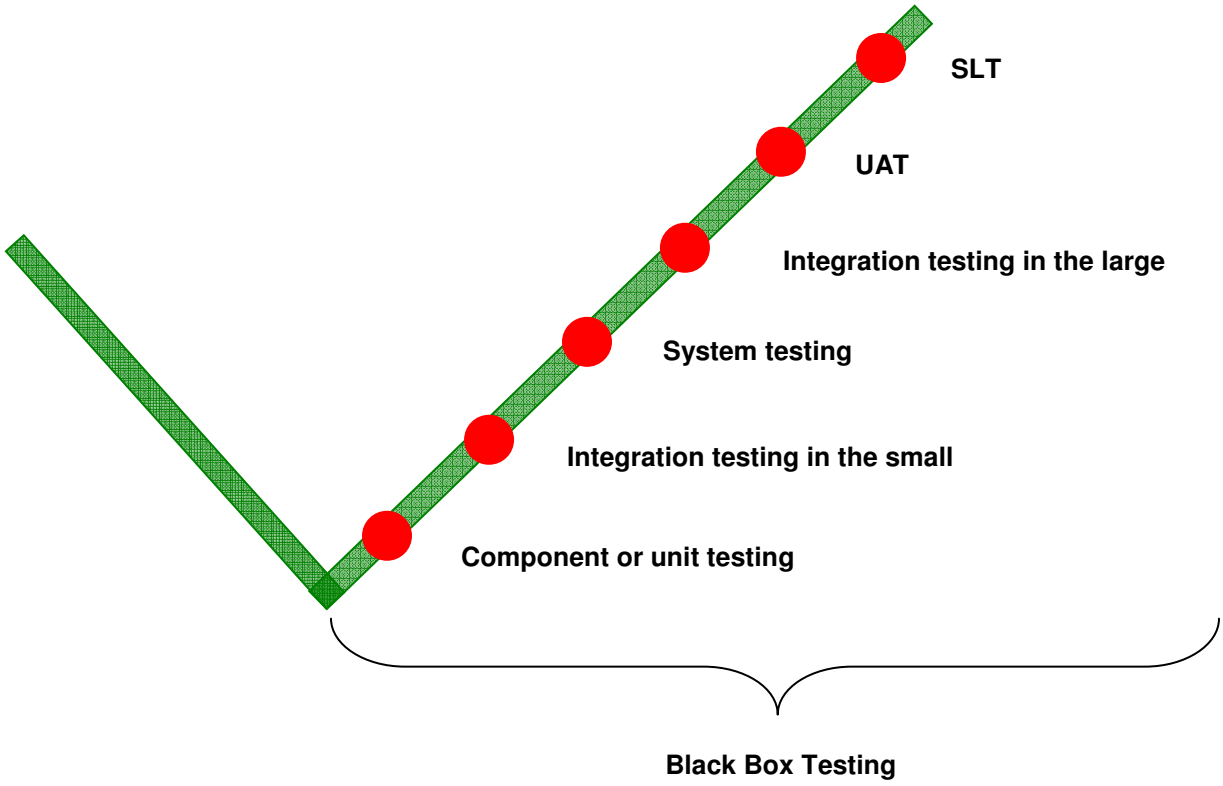
It is not important what happens inside the black box, but if the actual result obtained is the same as the expected result, then the test had passed. A basic knowledge of the code may help, but it is not vital.

Items tested by Black Box Techniques

Items tested by Black Box test case design techniques include

- Requirements
- Functional specifications
- Clerical procedures
- Designs
- Programs
- Subsystems
- Full systems

Black box testing is relevant throughout the lifecycle, as illustrated here – it can be used from unit testing through to service level testing (and maintenance testing)



White Box Testing

Test Case selection that is based on an analysis of the internal structure of the component

BS7925-1:1998

In block-box testing no knowledge of the code or of how the component under test is required. In white-box testing, it is essential to understand the structure of the code or have an understanding of the programming techniques used.

White Box testing is also known as:

- Structural test case design
- Glass box testing
- Logic-coverage testing
- Logic-driven testing

White box testing relates to testing the construction details of the design. Tests are built from knowledge of the internal structure of the program. The main concern of the testing is how a particular piece of code works, although functionality is still important.

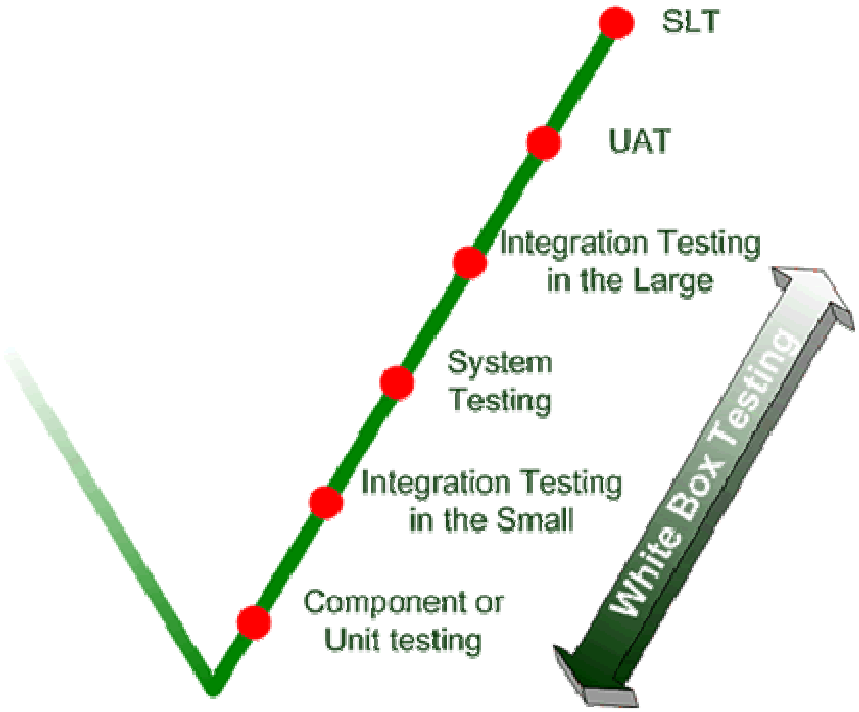
The testing effort is directed towards testing how the system has been built. Testing is done from the programmer's perspective. Tests can be performed during the build process, but normally best after positive testing had shown that the component does what it is asked of it in the specifications (programmers usually do this as they are working)

Any program or component that has a procedure such as "If this Do that" or "While this Do" can be tested. Data is input to the conditions, decisions are made by the code based on the data and results are obtained. By using various techniques, various paths though the code are tested.

Items tested by white-box test case design techniques include:

- Programs
- Program components
- Forms
- HTML pages
- Error handling

Black-box testing is relevant throughout the life cycle whereas in general, additional white box testing is appropriate for sub system testing (just testing, link testing) but becomes progressively less useful towards system and acceptance testing, as illustrated below. This is because system and acceptance testers will tend to focus more on specifications and requirements than on code.



White Box Test case design techniques

A test case design technique is simply a method used to derive or select test cases (BS 7925-1:1998)

A process to select and create a good test case provides a set of rules to aid in the identification and definition of tests. Advantages of using defined techniques include that a structured method is given which will help to reduce the test coverage redundancy. Test should not overlap, so it is good practice to ensure that the maximum test coverage is achieved with the fewest number of tests. Defined techniques also provides a repeatable method to determine test criteria.

There are 2 main types, systematic and non-systematic. Non-Systematic techniques include error-guessing systematic techniques each has associated measures including:

- % Statement coverage
- % Decision coverage
- % Condition coverage
- % Function coverage
- % Code coverage

For each of the systematic techniques (which we will go into in more detail in a later session) there is an associated measure which enables the amount of testing coverage achieved to be qualified. This provides one means of deciding when a suitable amount of testing has been performed.

Use of test tools

The process of white box testing can, with the code of any great size, be almost impossible and very consuming as programs may have thousands of lines of code and millions of different paths. Tools are available to help in this process.

Most tools provide statistics on the most common coverage measures, such as statement coverage or branch coverage. This data is typically used to:

Monitor the state of testing
Provide management with information regarding the level of testing
Enable testers to generate more testers to cover code not yet run

Some tools include: Source Purify and VSlick, but even most compliers carry out some white box testing processes.

There are also some tools available for black box testing – such as tools that automatically generate functional test cases.

The proper use of tools increases testing productivity and helps to give more confidence in he applications under test.

Summary

- Black box testing
    - Testing without the knowledge of the code
    - Focuses on functionality

- White box testing
    - Testing with the knowledge of code

- Systematic techniques and measures are used to provide confidence

- Use of tools increases productivity and confidence in the product

Black Box test techniques

Black box test case design techniques in BS7925-2

BS7925-2 concerns software component testing. Amongst other subjects, it refers to the following black box test techniques.

Equivalence partitioning:

In this technique a large input value range is split into partitions of input and output values. Each partition contains a set of values, chosen so that all the values can reasonably be expected to be equivalent (i.e. based on the fact that testing one of these is representations of testing them all).

Both valid and invalid values are partitioned in this way.

Boundary Value analysis:

This takes equivalence partitioning a stage further. Not only is the range split into its equivalent partitions and one value tested from each partition but also the values at the edges of the partition are tested.

State transition testing:

In this technique the different states in which the component may exist and the transitions between those states are used. Events cause transitions between states.

Syntax testing:

This is a systematic technique that uses the definition of the input data to produce test conditions. It uses a formal method to describe the input data, known as Backus-Naur form notation. This allows the full range of input data (both valid and invalid) to be documented and transposed into test cases.

Syntax testing example

Suppose we want to define an "operator" (a piece of data to be entered as input to a function, for example) to be either a left parenthesis '(' followed by 2 to 4 letters and a plus sign or to be 2 right parenthesis. The operator could look something like:

(AAAA+

Or

))

Using Backus Naur Form notations (BNF)

OPERATOR:=(MNEMONIC +/))…….the / is an OR symbol
MNEMONIC:=LETTERS42……this means 2 to 4 letters inclusive
LETTERS:=A/B/C…..X/Y/Z…….this means letters can be any uppercase alphabetic

Cause effect graphing

This assists with the mapping of complex systems where a particular cause or combination of causes may have one or more effects.

How to write tests using cause effect graphing:

- List and label all causes and effects for a module, in no particular order
- Draw cause-effect graph which relates various causes with their effects
- Develop a decision table from the graph
- Convert each row of the decision table into a test scenario

Random testing

This is the selection of random values for testing from the set of all possible input values. The input distribution used in the generation of random input values is based on the expected distribution of inputs found in normal use of the application.

Where the operational distribution is not known then a uniform input distribution is used. For example, if a field accepts a single alpha character, then the test case would be a random selection of a character between A and Z.

Don't confuse random testing with ad-hoc testing, which is testing with no formal structure.

Error guessing

This is the technique of using experience to postulate errors.

All of the above listed techniques, except for Random testing and error guessing, have an associated measurement technique.

Equivalence Partitioning

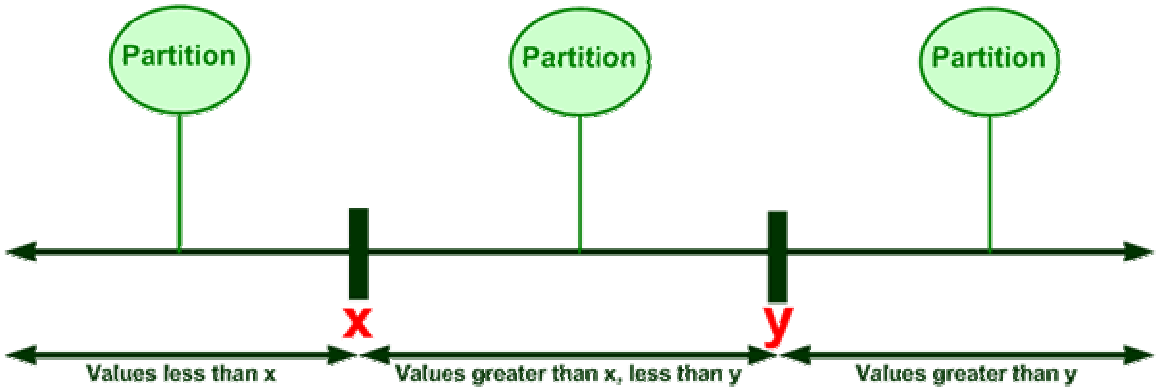
Equivalence partitioning is based on the idea that the inputs and outputs of a software component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component.

Equivalence partitioning use a model of the component that partitions the input and output values of the component.

The model comprises partitions of input and output values. Each partition contains a set of range of values, chosen such that all the values can reasonably be expected to be treated by the component in the same way (i.e. they may be considered 'equivalent')

The input and output values are derived from the specification of the component's behavior.

The diagram below shoes a representation of a requirement for a particular function. Values less than x are treated in one way, values greater than x and less than y are treated in a second way, values greater than y are treated in a third way. (This is a bad specification because it does not say what happens at values that equal x or y). Any value less than x could be used as input data test for that particular partition and similarly for other partitions.



FIGURE

In practice, a value from the middle of the partition is used. This is because if a middle value doesn' t work, there is not much point testing other values.

Example:

Adapted from BS7925 – 2)

A component, generate_grading, has the following specification

The component has passed an exam mark (out of 100) from which it generates a grade for the course in the range 'A' to 'D'. The grade is allocated from the overall mark which is calculated as the sum of the exam and the coursework marks as follows

- Greater than or equal to 70 – ‘A’
- Greater than or equal to 50, but less than 70 – ‘B’
- Greater than or equal to 30, but less than 50 – ‘C’
- Less than 30 – ‘D’
- Where a mark is outside its expected ranger then a fault message (‘FM’) is generated
- All inputs are passed as integers

CHAPTER 4

Reviews and the test process

Reviews in context

Reviews are part of testing known as static testing

Static testing is the testing of an object without the execution on a computer

Why review?

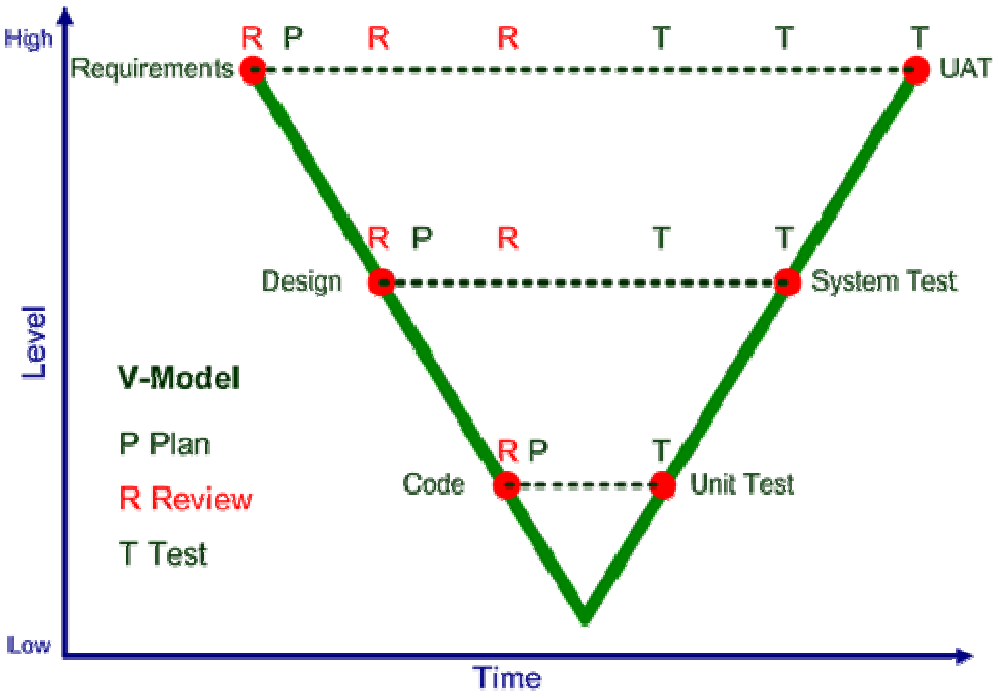
- Reviews are a very effective way to find issues early in the design or development process
- Early discovery of errors helps to ensure that the system produced adheres to the specification that was originally defined
- This reduces the likelihood of design issues arising in the live environment, thereby reducing the cost of development and maintenance of the product
- The review process leads to further process improvement
- Ultimately, better quality software closer to the requirements of the business is obtained. Development should be more productive, there should be a reduction in time taken to finish the product, the total cost should be reduced and there should be fewer failures or defects

Overall, reviewing reduces the cost and the risk of software testing and lowers the dependence on quality assurance.

When to review?

As early as possible in the development lifecycle. The reason for this is that issues are thereby found sooner rather than later and are not carried forward to later stages of development where they can be much more costly to rectify.

Wherever in the lifecycle, objects should be reviewed before they are formally published and begin to be used. As can be seen in the diagram below, most review activity tends to take place on the left hand side of the v-model, although not exclusively.



Figure

What to review?

Anything can be reviewed:

Specified documents

Specifications can be reviewed, including user requirements, functional and technical specifications and user guides.

Code and program designs

Walkthroughs and reviews should be carried out on all program design documents. This ensures that development standards are being adhered to and that the most effective solutions are being implemented.

Testing Documentation

Test documents should be reviewed by the test team (to ensure that testing standards are being followed) and reviewed by the business to ensure that the business functionality is being tested correctly.

Test plans should be reviewed regularly by the test manager.

It is estimated that 25% of test cases are written incorrectly at first draft. So, test cases are good candidate for reviews. Testers can also review test results, defects and metrics.

Who performs the reviews?

The project manager should decide who performs reviews on various objects, based on an individual's ability experience and seniority.

For test documentation, the test manager should have input on who performs the reviews, if not full control.

Cost of reviews

It is essential that ongoing review costs only about 15% of the total development budget. This includes the costs of the reviews themselves and their associated activities, such as analysis of metrics and implementation of process improvements

This figure may sound high, but up to 70% of the project issues can be uncovered during the review process, thereby making the review process pay for itself probably many times over in anticipating problems before they escalate.

Possible problems

No-one likes criticism. If you have ever had some work of yours analyzed and have had it found not to be up to the level of quality that the reviewer expects you may have felt slighted.

Wrongly, handled, reviews can cause resentment or fiction. Project members should be aware that reviews are there not to pick out individuals for ridicule, but to identify errors earlier than they otherwise would be in order to help improve the quality of the product. Even so, when errors are found, it should be treated as a good thing rather than bad, since an error discovered means it can be dealt with and should not reappear later in the project lifecycle.

Having said all this, some people will still resent having their work criticized, and may react badly. Diplomacy is required in handling all reviews.

Other possible problems include:

- Inadequate preparation before reviewing
- Lack of appropriate follow-up of uncovered defects
- Inappropriate choice of reviewers
- Complacency: just because a product has been reviewed it does not necessarily mean it is 100% correct. When using all documents or objects that have been reviewed, an open mind should still be kept and the usual tester's analyzing should be employed.

Summary

Reviews are known to be cost effective through

- Increased development productivity
- Reduced development time scales
- Reduced testing time and costs
- Reduced fault levels and lifetime costs
- Anything can be reviewed
- The earlier the better
- Cost of reviews is about 15% of development budget including metrics analysis and process improvement

Objectives of reviews

For each review technique the primary objective is to find faults in the object being reviewed

Other goals are

- To perform verification and validation against speculations and standards
- To obtain some process improvements
- To obtain consensus

Four main types of review are

- Informal review
- Walkthrough
- Technical review
- Inspection

Informal review

The informal review (buddy review or sanity check) is the least formal and usually least expensive of all reviews.

It is not documented but still an effective way of reviewing documentation. It is probably the most common for of review and very useful.

Informal reviews are simply one or more reviewers (other than authors) looking at a product and providing comments on it. They rarely involve formal meetings.

There is normally no statement of objectives documented for an informal review and they are characterized by there being no requirement for the results of the review to be documented.

Purpose

To provide informal feedback on a product's:

- Content
- Style
- Approach
- Completeness
- Correctness

These are the most likely attributes that an informal review will cover. There are probably others that will be addressed depending on the type of product being reviewed – for example, the process may just be a look to see if there are any grammatical or spelling errors in a report.

Walkthrough

Purpose

The purpose of a walkthrough is to evaluate a software product:

Educating an audience regarding the software product

IEEE 1028:1997 Standard for software reviews

Like a presentation, a walkthrough is usually based on a series of scenarios or dry runs in which a peer groups attempt to identify errors in the logic of a product or to highlight situations the author has omitted

Overview

- The meeting is led by the product author
- It is performed by one or more reviewers (who are subject-matter experts) who provide comments on the product.
- Outcomes and actions are formally recorded
- The meeting may be chaired by a walkthrough leader who resolves issues when consensus is not achieved

Also known as a structures walkthrough, a walkthrough is a

Review of requirements, designs or code characterized by the reviewers sufficiently in advance so they can familiarize themselves with the product before the meeting

The author also acts as the presenter and leads the reviewers through the product, often using scenarios for requirements and design models and dry run through code.

Roles

The IEEE identifies the following specific roles for walkthroughs

Walkthrough leader
Recorder
Author
Team member

Walkthrough leader

The walkthrough leader (most of ten the author) conducts the walkthrough, handles the administration tasks pertaining to the walkthrough (such as distributing documents and arranging the meting) and ensures that the walkthrough is conducted in an orderly manner. The walkthrough leader also ensures that the team arrives at a decision or identified action for each discussion item, and issues the walkthrough output.

Recorder

The recorder notes all decisions and identified actions arising during the walkthrough meeting. In addition, the recorder notes all comments made during the walkthrough that pertain to anomalies found, questions of style, omissions, contradictions, suggestions for improvement, or alternative approaches.

Author

The author should present the software product in the walkthrough

Team member

Provides detailed input to the review

Objectives

The main objective, as of nay type of review is to discover errors, for example

To find errors in function, logic or implementation for any representation of the software
To look for errors in or weaknesses of style
To verify that the software meets the requirements

Other objectives are
To ensure that the software has been developed according to pre-defined standards
To make projects more manageable
To achieve software developed in a uniform way

What can be reviewed by a walkthrough

The IEEE identifies the following software products as examples of those that may be subjected to walkthroughs

- Software requirements specifications
- Software design descriptions
- Source code
- Software test documentation
- Software user documentation
- Maintenance manuals
- System build procedures
- Installation procedures
- Release notes

Preparation

The walkthrough activities should be scheduled once the source material is complete. Allow about 2 hours for preparation. The all through itself should last no longer than 2 hours. After the length of time it is difficult for most people to maintain concentration and enthusiasm

The meeting

The author's overview

Reviewers should understand the product without any assistance, since the author' s overview may otherwise push reviewers into making the same logical errors the authors may have made.

The walkthrough leader should schedule the walkthrough when the author is ready

It should be noted that although the author is an essential part of the walkthrough, care should be taken to ensure that the author does not try to brainwash the walkthrough team through justification of the author's solution. The overview should be just that – an overview of the product, not seen as a medium for trying to impose the author's views.

Technical Review

Overview

A technical review (otherwise know as peer review) is a semi formal type of review, where the sole purpose of to find faults in the software code.

Purpose

To evaluate a software product by a team of qualified personnel to determine its suitability for its intended use and identify any discrepancies from specifications and standards

IEEE 1028:1997 standard for softwares

The meeting

The technical review is strictly documented. The review group usually consists of peers and technical experts. Management do not usually participate in this type of review The meeting:

- The meeting is chaired by the presenter
- The meeting is attended by one or more reviewers (subject-matter experts) who provide comments on the product
- The product is reviewed sequentially
- Consensus is reached on actions to be carried out subsequent to the meeting.
- Outcomes and actions are formally recorded

The product and source documents are distributed to the reviewers in advance and the reviewers in advance and the reviewers are expected to study the product head of the meeting.

At the meeting the reviewers report back on the documents. The presenter at the meeting is not the author of the product, and generally leads the meeting through the product sequentially. Reviewers discuss the issues raised and come to a consensus about what should be done next.

More reviews may not be involved in technical reviews than in walkthroughs or inspections.

Technical review roles

The IEEE identifies the following specific roles

1. Decision maker
2. Review leader
3. Recorder
4. Technical staff

Plus optionally

1. Management staff
2. Other team members
3. Customer user representative

Decision maker

The decision maker is the person for whom the technical review is conducted. The decision maker must determine if the review objectives have been met.

Review leader

The review leader is responsible for the review. This responsibility includes performing administrative tasks pertaining to the review, ensuring that the review is conducted in an orderly manner sure that the review meets its objectives. The review leader also promulgates the review outputs.

Recorder

The recorder is responsible for documenting anomalies, action teams, decisions and recommendations made by the review team.

Technical staff

The technical staff actively participates in the technical review and in the evaluation of the software product

Management staff

The management staff may participate in the technical review for the purpose of identifying issues that require management resolution, but normally don't get involved

Customer or user representative

The role of the customer or user representative should be determined by the review leader prior to the review

Technical review objectives

The objectives of a technical review are to determine for the product

- That it conforms to the specifications
- That it adheres to requisite regulations, standards and guidelines and accords with plans
- That changes are properly implemented
- That changes affect only system areas identified by the change specification

What can be reviewed?

Examples of software products subject to review include

- Software requirements specification
- Software design description
- Software test documentation
- Software user documentation
- Maintainace Manual
- System build procedures
- Installation procedures
- Release notes

Inspection

Purpose

IEEE 1028:1997 Standard for software reviews states that the purpose of an inspection is to detect and identify software product anomalies. An inspection is a systematic peer examination that:

- Verifies that the software product satisfies its specifications
- Verifies that the software product satisfies specified quality attributes
- Verifies that the software product conforms to applicable regulations, standards, guidelines, plans and procedures
- Identifies deviations from standards and specifications

- Collects software engineering data
- Uses the collected software engineering data to improve the inspection process itself and its supporting documentation (for example, checklists) again, this is optional.

Method

The inspection method explained here is based on Fagan inspections.

The inspection process is formally defined and followed rigorously. The outcome of the inspection is based on whether the product meets pre-defined targets rather than the consensus of the reviewers.

An initial meeting is held so that all participants understand their responsibilities and also understand the product and source documents. The time to be spent in individual checking is planned and the parts of the product to be checked in more detail by a particular reviewer (also know as inspector or checker) may be identified.

Each reviewer inspects the product individually and reports back on issues raised.

A logging meeting may be deemed necessary dependant on the number of issues identified by the individual performing the checking or on the likelihood of finding additional issues during the logging meeting as predicted by measurements from previous inspections. The issues already found may be documented at this meeting and new issues may also be identified. Discussion of whether raised issues are properly dealt with. The author is not allowed to present the product at a logging meeting.

The moderator collects statistics about the time spent by each participant and the faults found.

Estimates are calculated for the remaining faults per page and the time and cost saved bye the process

The IEEE also states that individuals holding management positions over any member of the inspection team must not participate in the inspection.

Inspection roles

The moderators' responsibilities are

- To manage the inspection
- To ensure a positive and non-combative approach
- To keep the inspection on target and to ensure that all checks are completed
- To generate statistics on preparation time and errors found

Success of an inspection is dependant on the successful moderation of the meeting. Specialist training in inspection moderation should be considered for all inspection moderators.

Author

Answers questions about the product from anyone in the meeting

Reader

The reader reads the product aloud

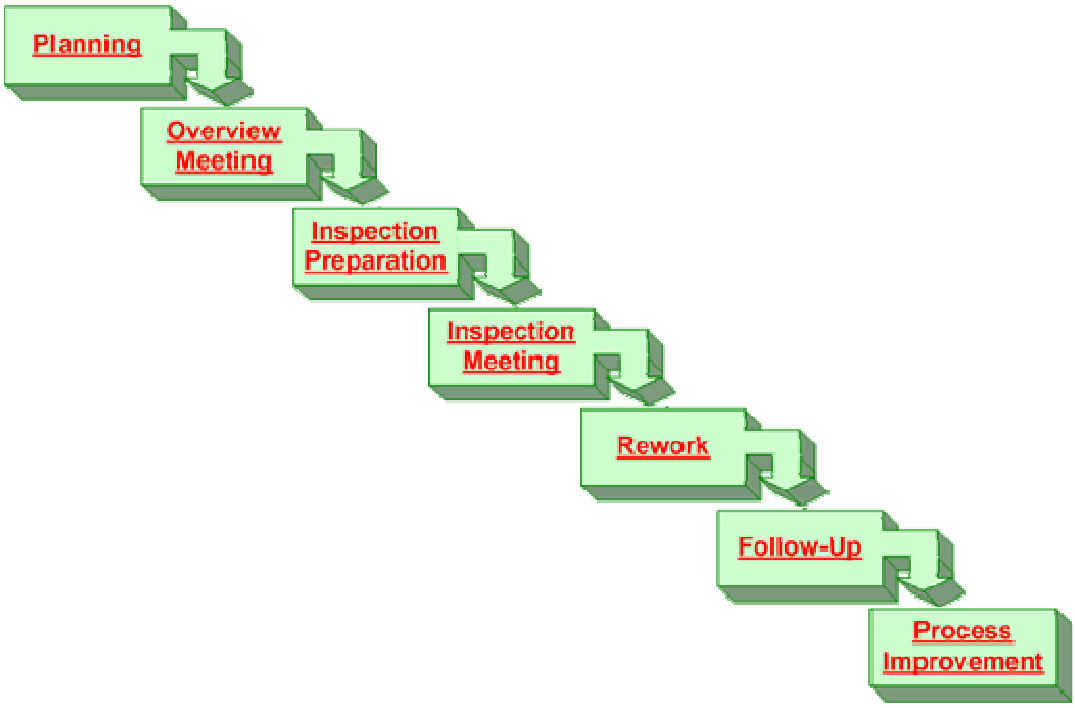
Inspector

An inspector is an independent assessor of the product
Recorder

The recorder documents the inspection outcomes.

Inspection activities

The diagram below shows the flow of activities comprising the inspection process



FIGURE

Planning

Planning includes identification of the inspection team, distribution of inspection materials and the schedule for the review process

Overview meeting

The overview meeting is an introductory meeting at which the product author provides information on the product to be inspected. This is a relatively informal part of the process. It should be used by inspectors as an opportunity to ask questions about the product in order to gain an understanding of it and how it fits with other related products. This is not an inspection meeting and so the identification and discussion of errors should be kept to a minimum.

Inspection preparation

Inspectors should use this phase to review the solution and identify any defects. Fagan specifies that defects fit into one or three categories.

A missing requirement or feature is not present
A wrong requirement or feature has been incorrectly implemented
An extra feature has been introduced with the product and identify those parts of the document o code that can be used to summarize the product

Inspector meeting

At the inspection meeting, the product is reviewed, ensuring that all defects identified during the preparation phase are covered. The recorder should note all the defects that have been identified. The inspection meeting is a defect identification meeting not a defect resolution meeting and so the moderator should minimize any discussion about solutions

There are two approaches to defect recording; either collate all the information from individual inspectors prior to the meeting or capture it during the inspection meeting

The moderator should record statistics from the inspection using standardized forms.

The final actions of the meeting is to decide whether to

- Accept the product
- Accept it with provision
- Reject it

Rework

Rework is the phase where the product author addresses all the defects raised in the inspection meeting

Follow up

During the follow up the moderator reviews the revised product to ensure that all defects have been corrected. In extreme cases and where agreed at the inspection meeting, a further inspection should be performed.

Process improvement

Through the analysis of the statistics recorder by the moderator, particular areas of weakness can be identified and addressed.

Summary of types of review

Type	Approach	Led By	Documented	Usual Activities / By Whom
Informal Review	Informal	No formal leader	No	Look at product Provide comments on it
Walkthrough	Less Formal	Author	Yes	Product presented Peers find errors
Technical Review	Formal	Presenter	Yes	Find faults in code Peers and technical experts
Inspection	Most Formal	Moderator (not Author)	Yes	Product reviewed sequentially Peers find errors

Review deliverables

The review process can produce

- Product changes
- Source document changes
- Process improvement

The main deliverable from a review is a list of changes to be made to the object that has been reviewed

General roles and responsibilities summarized

Moderator

- Usually specially trained
- Facilitates the review process
- Ensures review is focused and goals are reached

Author

- Produce document to be reviewed
- Ensures all reviewers have documents before meeting

Reviewer

- Prepares for the review
- Must not be dogmatic in approach
- Must have an open mind
- Review the document not the author

Management

- Promotes and encourages the process
- Must be open to the resulting ideas

Review pitfalls

There are a number of aspects of reviews that can prove problematic. The following points should be followed in order to reduce the diversity of those problems

- All members must be suitably trained in the review process. Moderators in formal inspections must particularly but everyone who has an input t a review should understand their role and their responsibilities.

- There must be sufficient documentation to review

- Management must be fully behind the review process. If there is a lack of support, for example not allowing staff to attend the meeting, the process will not succeed.

- Reviewers must have sufficient expertise or experience – particularly in technical aspects of the item being reviewed.

- Teams should be of a reasonable size: small teams can cause problems – there may be insurmountable friction between protagonists or it may be difficult to get a consensus when only two people are involved and they have diametrically opposing views. On the other hand, a team that is too large will be unwieldy and promote endless discussion without achieving resolution.

Summary

All types of review must be planned

The main purpose of all reviews is to find defects

The four main types of review in order of increasing formality are:

- Informal review (buddy review)
- Walkthrough or presentation
- Technical review
- Inspection

Static Analysis

Analysis of a program carried out without executing the program

BS7925-1:1998

For example, one type of simple static analysis is the examination of program logic for errors.

Static analysis provides information about the quality of software by examining the code, rather than by running test cases through it.

What can static analysis find?

Possible faults in code:

- Unreachable code: analysis can determine that parts of code can never be executed, this is also known as dead code
- Undeclared variables: for example, trying to use the statement ‘a=b*c’ without declaring the variables ‘b’ or ‘c’ before hand.
- Parameter type-mismatches: for example, if the expected type of a parameter to a function is an integer and another function is sending it a real number
- Infinite loops: when running a program if you wither have a flood of output that will not stop or a hug cursor it may be attributed to a mistake in a loop construct. An example could be a forgotten counter increment in a FOR statement in a FOR statement causing the FOR loop to run infinitely
- Uncalled functions and procedures – redundant code can make programs run slower and take up memory unnecessarily
- Possible array-bound violations

Benefits

Provides a wide ranger of objectives which can be used in the assessment of the code quality
Identifies basic coding errors as detailed above
Is relatively cheap and can be carried out anywhere – it requires only time, no hardware or computer resources

Pitfalls and limitations

It is not possible to check declared variable values – this needs a program run

- It can identify errors which would not necessarily cause failure, so may waste time focusing attention on defects that are not important
- Code is not executed, so it is unlike real operating conditions
- The only real pitfall of psychological rather than real. Metrics gathered may include errors which will not cause failures when the software is run, which may move focus from the real errors which need to be addressed.
- However they can improve overall quality by increased awareness, and probable removal of errors.

Complier-generated information

Compilers convert program-language code into machine-readable (computer readable) code to give faster execution. They perform certain checks (such as code syntax testing and variable declaration) as they run.

Most modern compliers help with static analysis as they can:

- Check syntax
- Check variable declaration
- Check misspelling
- Produce a memory map
- Cross-reference variables

Compliers highlight these issues in the course of compilation

Information on variable use can be very useful during maintenance of the software

Some higher languages have the ability to by-pass some compilation rules. For example, Microsoft visual basic for applications does not require declaration of variables explicitly unless a particular option is set. In these cases static testing becomes essential for trapping these program errors

Any faults found by compliers can be found by static analysis

Data flow analysis

In data flow analysis, data is followed through paths it takes within a section of code and attempts are made to find anomalies such as

Definitions of variables with no subsequent or intervening use
Attempts to use a variable after its removal from memory

The use of variables is defined as one of three types or 'uses'

Definition use (d-use) where the variable is defined for example a;=6
Computational use (c-use) where the variable is used in a computation, for example, a/4
Predicate use (p-use) where the variable can be related to the use of control flow graphing where nodes are the d-use and c-use and edges are the p-use)

CHAPTER 5

Organisation

Organizational structures for testing

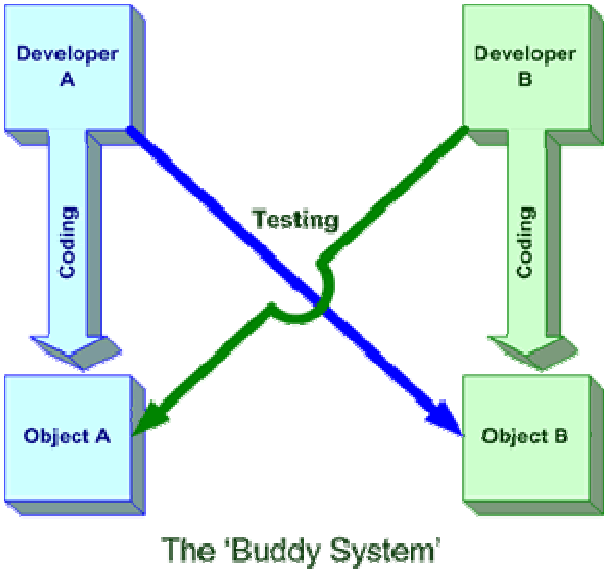
In any areas of business, management is very important. Without proper management things can go wrong all too easily. Testing is no exception to this general rule. One of the tasks in the management of testing is to organize the infrastructure, in particular the organization of the staff involved, their relationships to each other and how they will interface with the others involved in the project.

Business organizations may implement different testing structures depending on

- The testing philosophy subscribed to by the business
- The commitment to testing that the business has
- Budget constraints imposed
- The resources available
- The size of the project the subject of testing

For small projects, or for businesses that don't have a well developed testing strategy or philosophy, it may be that all testing is carried out by the developers alone.

At the very lowest level of testing the application may be tested by the person who wrote the code (in addition to the normal testing that they would carry out as they write it). A slightly better structure is the buddy system where one developer tests the work of another.

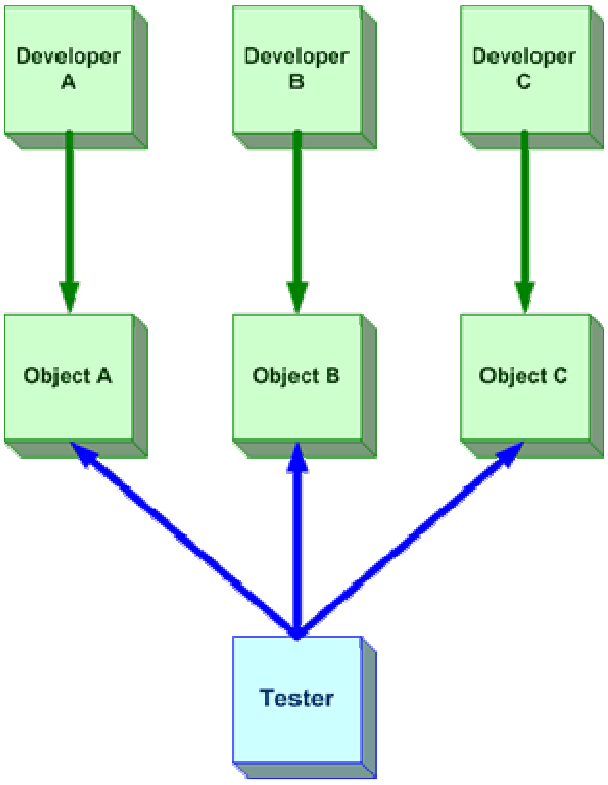


The 'Buddy System'

FIGURE

Neither of these testing structures is particularly robust, not least because of the problems associated with the lack of testing independence.

A better type of organization than the buddy system is this

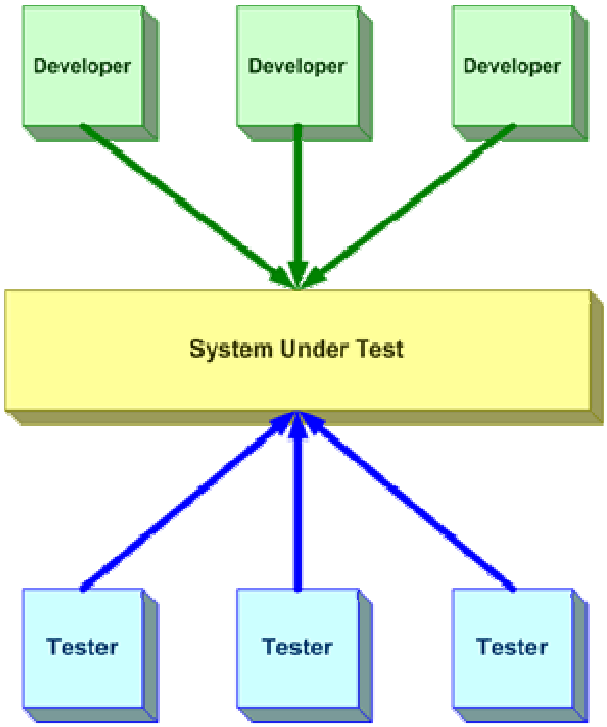


FIGURE

Here, one tester is responsible for testing the programming output from a team of developers. It has a higher degree of testing independence than the system in which developers test their own work but still enables the tester to be involved throughout the development lifecycle.

One possible disadvantage is that the tester may become too close to the development and lose a degree of objectivity.

In larger organizations, and in those which have a greater commitment to testing, it is normal to find a dedicated team (or multiple teams) responsible for the testing process

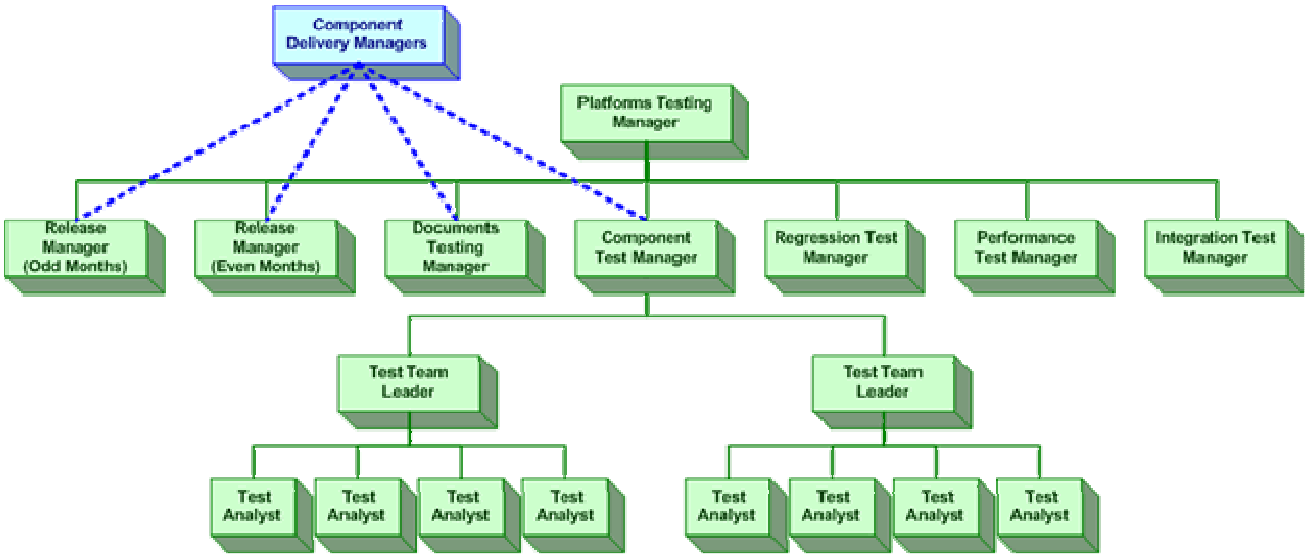


FIGURE

The testers will probably be employees of the organization, although the team may include one or more external testers (possible contractors working either for themselves or for dedicated testing companies)

The larger the team of testers, the better should be the quality of testing, because of a greater independence, objectivity and breadth of knowledge (of the system under test in particular and of testing in general)

The diagram below shows the organization of an actual business. Only the hierarchy for below the component test manger is shown because of lack of space but each of the other managers would have their own teams similarly organized with a test team leader responsible for four test analysts



FIGURE

The component delivery managers have limited managerial input to some of the testing managers but the platforms testing manager (the overall testing manager in this organization) has the final say in what will be tested and how, and is ultimately responsible to the board of directors for the business.

Other types of structures

Internal test consultant

Internal test consultants

For particular stages of the testing process some organizations may not have sufficient testing skills available. In this case a business may wish to bring in test consultants on a temporary basis to bridge the gap.

Separate testing organizations

The ultimate in the test independence (at least when considering testing by people as opposed to completely automated testing) is the use if a separate, external organization to which is given the responsibility for testing the whole or part of a project. One of the downsides of this cost – it can be quite expensive. Another is that the business may feel that an people brought in from an external company will not have the same dedication to the product that an employee of theirs would have.

Other considerations

Team composition changes over the life cycle

The composition of the testing team may change throughout the life cycle, for example at the beginning of the project there may be a substantial requirement for senior analysts who have the ability to examine a system specification, analyze the requirements and produce estimates for testing effort required.

When system testing gets underway the testing requirements may change the need for more personnel who are able to run test rather than analyze and design them. At certain times there may be more need for load and performance experts

Additional roles

Additional roles that may be required at some point are business analysts, trainers, auditors and so on

Other organizational tasks

Deciding reporting lines

When organizing the team structure, it is important to define the structure of reporting lines. Members of the testing team must understand clearly to whom they are responsible for reporting progress or problems. It is also vital that the testing teams know the lines of liaison with, for example user representatives, operations representatives and of course the development team.

Agree deliverables/set milestones and timescales

Not only is it important that staff know to whom they are responsible but it is also important for them to know what it is that they are expected to produce and when they are expected to produce it.

Deliverables that the testing team might be expected to produce include

- Test specifications
- Test scripts
- Test plans
- Progress reports
- Test logs
- Incident reports
- Test completion reports

Team composition

A multidisciplinary team with specialist skills is usually required. Most of the following roles are usually needed for any project of reasonable scale:

- Test analysts
- Test automation experts
- Database administrator (DBA)/ designer
- User interface experts
- Test environment manger
- Load and performance experts
- Project management experts

The skills required will depend on the complexity of the project as well as on what it is that the project is producing. Testing a website may include the requirement for very detailed load and performance assessments for example.

Required skills are not always available when they are needed, and lack of money may make it impossible to engage the right person for a particular task. In this case compromises will have to be made

Summary

- There are different testing structures
- Self testing
- Buddy system
- Test team
- Internal test consultant
- Separate organization
- Testing requires a wide range of specialist skills

Configuration Management

Introduction

What is configuration management?

Configuration management is management method used to prevent unauthorized changes to any asset system.

An asset is any item created for use with the software system, for example a database, test data, use guides, operation manuals, automated object naps, test scripts, results and plans.

Configuration management is defined within the Prince 2 project management methodology as

"the method used to ensure the validity and consistency of baselined products either in isolation or when used as component for other products. All products, once baselined are entered into the configuration library and are then controlled by the configuration librarian to ensure that all recipients of products are recorded to ensure they can be recalled and replaced in the event of change….

Configuration management also manages the modification of baselined products to ensure that different changes are not made to the same product at the same time"

A **baselined** product is one that has been approved for inclusion within the configuration library.

The **configuration library** is a repository containing all configuration items and which has access restricted to prevent unauthorized modification

BS 6488, Code of practice for configuration management states that:

"Configuration management identifies, in detail, the total configuration (i.e. hardware, firmware, software, services and supplies) current at any time in the life cycle of each system to which it is applied, together with any changes or enhancements that are proposed or are in course of being implemented. It provides traceability of changes through the lifecycle of each system and across associated systems or groups of systems. It therefore permits the retrospective of a system whenever necessary"

Typical symptoms of poor configuration management

When configuration management is not carried out well in a project certain telltale signs begin to become apparent. These include:

Source and Object code mismatches

This is normally a version control problem. The version of software running in the test environment must be correctly matched against the version of source code in the program library. When faults occur in testing it is imperative that the developer fix the right version of code**.**

Inability to identify source code changes

Changes made to source code should be traceable back to the person who made them. If there is no record or log kept of who was allocated a piece of work or who carried it out, then changes could be made with no accountability. In the worst case, this situation leaves the system open to damage by those with malicious intent

Simultaneous changes made to the source code

This occurs when two developers unknowingly take responsibility or changing a piece of code at the same time. They both make changes and save or compile their code back to the system. Unfortunately, the changes made by the developer who saves his work first are over written by the changes made by the developer who saves his work second.

Lost changes

This is one symptom of the problem of simultaneous changes

Inability to identify the complier version used

Problems can often occur during or as a result of compiling. For example, one version of a computer may allow a piece of code through while another version is less tolerant. If new faults suddenly appear in parts of the system which have undergone no change, it is possible that a different version of the compiler has been used which ahs introduced the defects. If there is no record of the complier versions used at various stages, it will be difficult, if not impossible, to identify the cause of the problems.

Configuration identification

To be able to apply configuration management, all items to be the subject of the method have to be identified and their versions recorded. For example

All details of the items must be noted, such as:

- Unique identification number
- The name of the item
- The physical form of the item
- The functional characteristics of any code
- Version numbers of code

Each object that is recorded become a configuration item.

Configuration library

The repository containing all the configuration items is known as the configuration library. This can be in electronic form (a database, a spreadsheet, or even a word processing document) or just a simple paper system

Security

Access to the configuration library must be restricted to prevent unauthorized entries or modifications. The library can be electronically protected, for example by passwords or by restricting access to certain directories on a network. Paper systems should be kept under lock and key when not supervised.

Organisation

The configuration library should be structured to ensure that all configuration items can be readily accessed in a logical way. If using an automated configuration management tool, such as PVCS, the structure needs to be defined to reflect the relationships that exist between the individual configuration items. This will help to ensure that the integrity of the product is not compromised when items are updated.

If an automated tool is not used, the configuration librarian must ensure that these relationships are noted in the configuration library register.

Configuration library register

Entry details

For each entry in the configuration library register, the following information should be recorded.

- Part number: the part number of the product to be entered into the library. Where the project has not allocated part numbers to products, the configuration librarian should consider adding library part numbers to simplify the identification of parent and children products
- **Name:** the name of the product
- **Owner:** The name of the person (or role) responsible for the product
- **Parent:** where the product is a component of a larger product deliverable, the product at the next level up in the hierarchy should be identified.
- **Children:** Where the product has lower level products associated with it record these here
- **Change status:** Record whether the product is released for use or frozen for modification for example
- **Current change information:** details of any current or pending change requests
- **Version:** the current effective version of the product
- **Change history:** details of any chances made to the product since the initial baseline should be kept
- **Distribution:** the details about who has been issued with the current version of the product.

Accepting products into the configuration library

Products should only be accepted into the configuration library when the following criteria have been met.

- **The product has been approved:** no item may be enter the configuration library for the first time without approval of the project or team leader or manager

- **All changes must be complete:** when a product has been updated as the result of a change request, it may only be accepted back into the library when all required modifications have been satisfactorily completed.

- **The item meets the product descriptio**n: it is worthwhile confirming that the product conforms (or still conforms) to the product description.

Configuration control

Configuration control concerns the maintenance of the items in the library

Issue of items

Once a product has been entered into the configuration library it can only be issued for use by the configuration librarian.

Recording changes

When the configuration librarian receives an authorized change request or products that are the subject of the change should be frozen in order to prevent further issues of the product or further changes being made at the same time.

The configuration librarian is responsible for managing the change process and should monitor that the changes are being made in accordance with the specified timescale. Any anticipated deviation from this should, if necessary be escalated to the manager responsible for resolution.

Once modifications have been completed to a product released for change, the configuration librarian must ensure that all modifications detailed within the change pack have been satisfactorily completed. This includes ensuring that all necessary approvals have been received. After the preliminary checks have been completed the revised product should be received back into the configuration library and the status of the products amended to “released”.

Configuration Management and Testing

Testing not just development, must also be conducted under effective configuration management.

It is important that the results of testing be reproducible, allowing testers to recreate the conditions of a previous test and be able to repeat the tests exactly, for example it may be necessary to repeat a particular test in response to an error raised against the system when it is being used’ live’. Configuration management can help to keep track of test and conditions.

Testing configurations items

Test items that can be the subject of configuration management and thus become configuration items include

- The test plan

- Test specifications
- Test data
- Supporting software – for example, test harness, stubs and drivers
- Test cases
- Test scripts

Configuration Auditing

Configuration auditing is the process of checking on the contents of libraries for example for their compliance to standards.

The process involves the determination of to what extent the configuration item reflects the actual physical and functional characteristics, ensuring that the configuration items reflect the definitions in the requirements.

Configuration management tools

Configuration management can be complicated particularly in environments where mixed hardware and software platforms are being used. Fortunately, sophisticated cross-platform configuration management tools are increasingly available – for example Meranr SCM or Opsware.

Summary

- Configuration management provides version control, traceability and enables reconstruction
- All configuration items and their versions must be known and changes recorded.
- Symptoms of poor configuration management include
    - Source code and object code mismatches
    - Non-repeatability
    - Lost changes
- Configuration management can be complicated but there are tools available to help

Test estimation, Monitoring and Control

Test Estimation

For reasons of planning for the resources that will be required in a project, the testing effort must be estimated. Te main outputs from estimation are:

- The overall time required for testing
- The resources needed to carry out the testing
- An allowance for rework

Test estimation should start by focusing on the high level test plan because the effort required to perform activities specified in the high level test plan must be calculated in advance. There are many dependencies in testing and only by knowing in beforehand the likely time and resource requirement can a reasonably accurate prediction be made as to when a product may be ready for the live environment.

Rework must be taken into consideration. No software passes all tests the first time round and re-testing (i.e. rework) will be required to ensure that defects have been rectified.

Factors to consider when estimating testing effort

- **Risks:** probably the most important aspect in testing is assessing the risks. Questions that must be addressed include: what is the most critical part of a system? Which parts of the application need to be tested and to what extent? What are the risks of failure? Should more testing be carried out if the failure is likely to result in a human fatality?

- **Complexity of code:** the more complex the code, the more likely it is that errors will creep in and hence the more testing may be required.

- **Stability of source code:** the more unstable the source code, again the more testing is likely to be required

- **Coverage required:** does the test plan require, for example, 100 percent coverage of business functionality or does it allow for the coverage of just business critical functions?

- **Complexity of hardware solution:** once more, a very complex hardware solution on the leading edge of technology may indicate a need for extra testing.

Test Estimation outputs

The estimation process should produce a number of quantifiable outputs which can be fed into test and project plans including:

- The number of tests to be written

- The time required to write and execute them.

- The environment required and time needed to et it up. Time must be allocated to set up the environment (PC's databases and so on)

- Resources and skills required

- An allocation of time for investigation and recording of faults: testing is very dependant on other activities and can easily be delayed. Testing itself can ruin schedules – finding a critical fault may force the development team to spend time fixing them. The tests must then be re-run and regression testing must be carried out before resuming mainstream testing.

- Time for re-testing and regression testing. Every fault found needs to be retested. The environment may need resetting and other test may be required to be executed before the test in which the fault was found can be run. Every new release and every change to the environment requires a regression test.

Test monitoring

Testing should be quantitative and not subjective. Using metrics to report progress reduces subjectivity and helps to produce consistent reporting. Metrics also enable management to predict when products may be ready for release to their customers.

Various measures can be used for tracking progress, including:

- Number of test cases written
- Number of scripts completed
- Number of tests run
- Number of tests passed.
- Number of defects raised
- Number of retests

Since these metrics are used by the test manager and the project manager as a basis for making decisions, it is important that the measures used are accurate.

The test manager may have to report on deviations from the project plan or test plans, such as sunning out of time before completion criteria are achieved. Analysis of test metrics makes it easier to predict when things are going wrong, particularly if the metrics are produced on a regular and frequent basis.

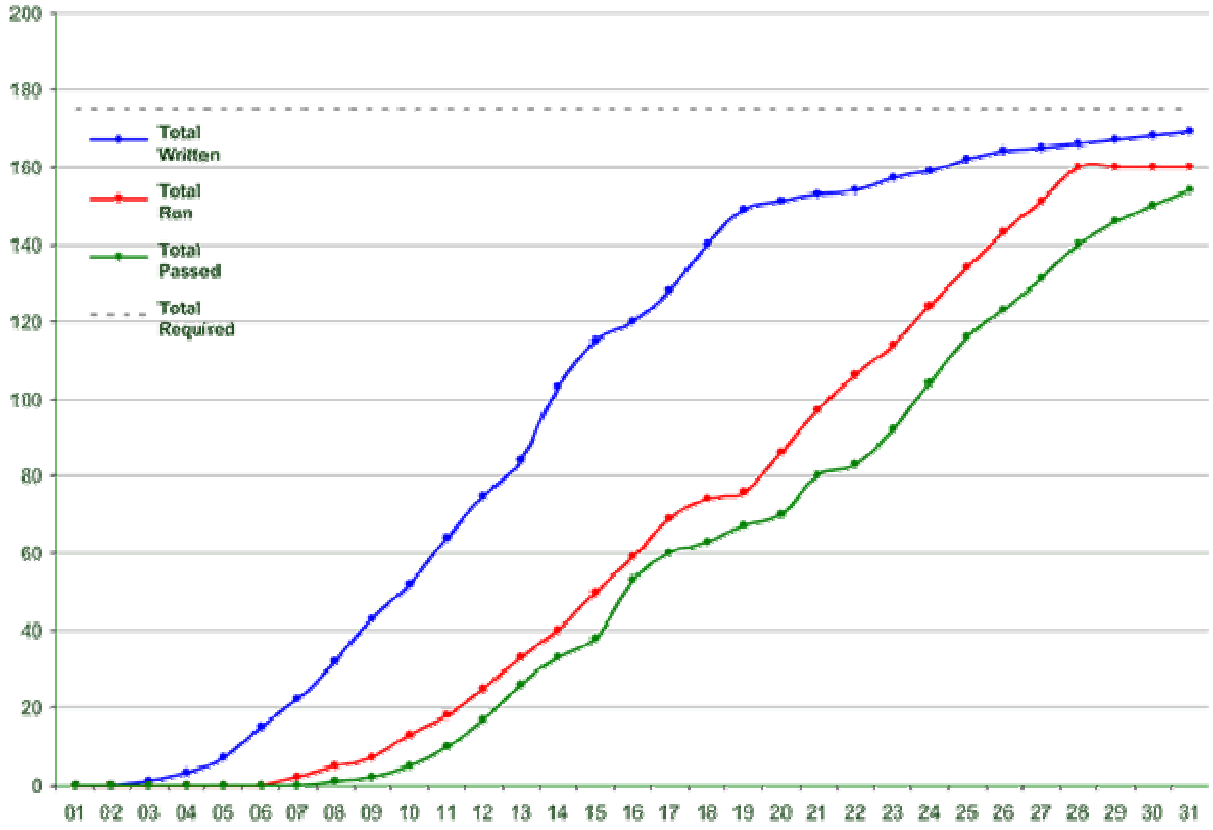
Other quantifiable aspects to monitor in order to facilitate estimation and to gauge current progress are:

- Time taken to write tests
- Time taken to run tests
- Categories of faults found
- Time taken to remedy defects

The information gathered during the monitoring activity must be collected and presented in a useful way to those who will use the data as a basis for decision making. Most usually, charts or graphical representation of the data is considered the best way of representing it.

The diagram below is one example of a test monitoring chart prepared for the presentation of information to the management. The horizontal axis is time and the vertical axis is the number of tests.

The chart plots three discrete pieces of information. The running total of the number of test scripts written, the running total of scripts run and the running total of those that passed.



FIGURE/GRAPH

This graph enables management to see the progress of the testing effort over time. In this case the deadline for the test completion is 31st may. The graph shows that the testing effort has not been enough to complete in time. Test management should have been able to predict from the slope of the graph of total tests written that 100% was not going to be achieved by the end of the month. Reallocation of resources to increase effort on writing scripts may have avoided the failure.

Test Control

The better that testing is monitored, the easier it will be for management to exercise control over the testing process.

It is important for management to have the overall picture at their disposal. This is gained from an understanding of the business risks and how they relate to the current progress in terms of what risks have been covered by adequate testing. If there are areas that have not been tested adequately (or not at all), management will need to make decisions to rectify the situation by exercising control measures on:

- Resources
- Schedules
- Environments

- Completion criteria

Resources

- **Assign more:**

  It may be necessary to obtain more testing staff to cope with extra work or to help catch up on schedules that are running later than anticipated for whatever reason. However, it is not always the best response to late-running schedule, particularly towards the end of a project, as it takes those new to a project some time to learn the system and to et up to speed.

  Adding new testers may also require the addition of new hardware – they will probably need at least a computer each and access to any networks. Doubling the number of testers does not double the output of work. There are overheads for example, in terms of communications and dissemination of information that produce a diminishing return of work output for the number of people employed on a project.

- **Reallocate:**

Reallocation of testers between different areas of the project may be a solution. It is a better solution in as much as they should at least be au-fait with the project. Of course the part of the project they are moved from may suffer as a result, so the impact of the change needs to be assessed. Normally testers would be moved from a low risk area or from one in which the testing is not due to start for some time. Decisions are made with the reference to the business risks.

- **Train:**

If there is a lack of resource due to inadequate skills, training could be an option – but this takes time to organize and time to carry out. However on longer projects (and if the monitoring is able to predict problems well in advance) this could be effective.

Schedules

- **Postpone**

If testing is not progressing because of failures in a part of the system, schedules may be able to be changed to put back testing of that part of the system until the difficulties are overcome

- **Bring forward**

  Other testing may be able to be bought forward

It is not always possible to juggle with a schedule because of the dependencies between parts of the system. If part of a system fails, the testing plans need to be looked closely to establish.

- What can and cannot still be tested without failed system (if anything)

- What needs to be done to re-establish the testing environment if it has been damaged by the failure

The internal and external dependencies on testing need to be analyzed before any decisions are made and rescheduling put into place.

Environments

- **Add new:**

If testing is falling behind it may be due to a problem with the testing environment. The test environment may be inadequate in terms of its capacity or power. Extra environments may be able to be added but this will probably involve extra hardware (at no extra cost) and may take time to arrange.

- **Reallocate:**

Different environments may be able to be reallocated to different areas of testing in order to reduce interference. For example, some testing may permanently change data that another set of tests need to remain stable. Reallocating one set of testing to another environment may overcome this problem.

- **Reconfigure:**

Reconfiguring an environment may be required if slow response is being experienced or if the environment crashes regularly. It is important that any reconfiguration is done with regard to what the live system will be like. There is little point changing the environment to make the testing go quickly if when the system goes live it doesn't' t work due to an environment incompatibility.

- **Reschedule use:**

To avoid testing clashes, rescheduling the use of the environment may be possible – for example main functional testing can take place Monday to Friday and database back up and restore testing could be carried out at weekends. There are obvious cost and resource implications for any out of normal hour's activities.

Completion criteria

If testing is falling behind and deadlines are irremovable (say due to legally imposed implementation date, such as in tax related software developments) the criteria for completion may be able toe be changes. For example, instead of requiring 100% of all functions to have been tested, the criteria may be amended to requiring only 100% high business risk functions to have been tested with no category A errors.

The decision to change test completion criteria would normally be made at high management level – for example, by the project manager with the approval of the business board representatives.

Summary

- Test estimation
    - The effort required to perform activities in high level test plan and rework must be calculated in advance.

- Test monitoring
    - Measures for tracking progress include tests run, tests passed/failed, incidents raised/ fixed retests
    - The test manager may have to report on deviations
- Test control
    - Reallocation of resources may be necessary (changes to test schedule, test environments, number of testers)

Incident management

What is an incident?

An incident is any significant, unplanned event or unexpected behavior of a system under test that requires further investigation and / or correction. Incidents are variously known as:

- Defects
- Errors
- Problems
- Anomalies
- Faults
- Glitches
- Bug
- Incidents
- Flaws
- Gripes

Or even as opportunities by optimistic project members.

Incidents are raised when expected and actual test results differ. Incidents are also raised against documentation not just against code or a system under test. Incidents should be logged when someone other than the author of the product under test performs the testing

IEEE Std 1044:1993

This standard provides a uniform approach to the classification of anomalies found in software.

The standard provides a definition of anomaly as

> Any condition that departs from exceptions based on documentation [such as specification, design documents, user documents, standards] users perceptions or experiences.

The IEEE prefers to use the word anomaly because it is more neutral in connotation than for example defect

An anomaly is not necessarily a problem in the software product – it can be caused by something other than software. Ana anomaly is basically a potential problem and is a subset of incidents.

Incidents can be anything which needs to be tracked – for example a change request which is not necessarily a problem.

There are many different types of category for the classification of an anomaly. For example:

- Project activity
- Project phase
- Suspected case
- Repeat ability
- Symptom
- Product status

And so on. Some of the categories are compulsory to record, while others are discretionary.

Incident tracking

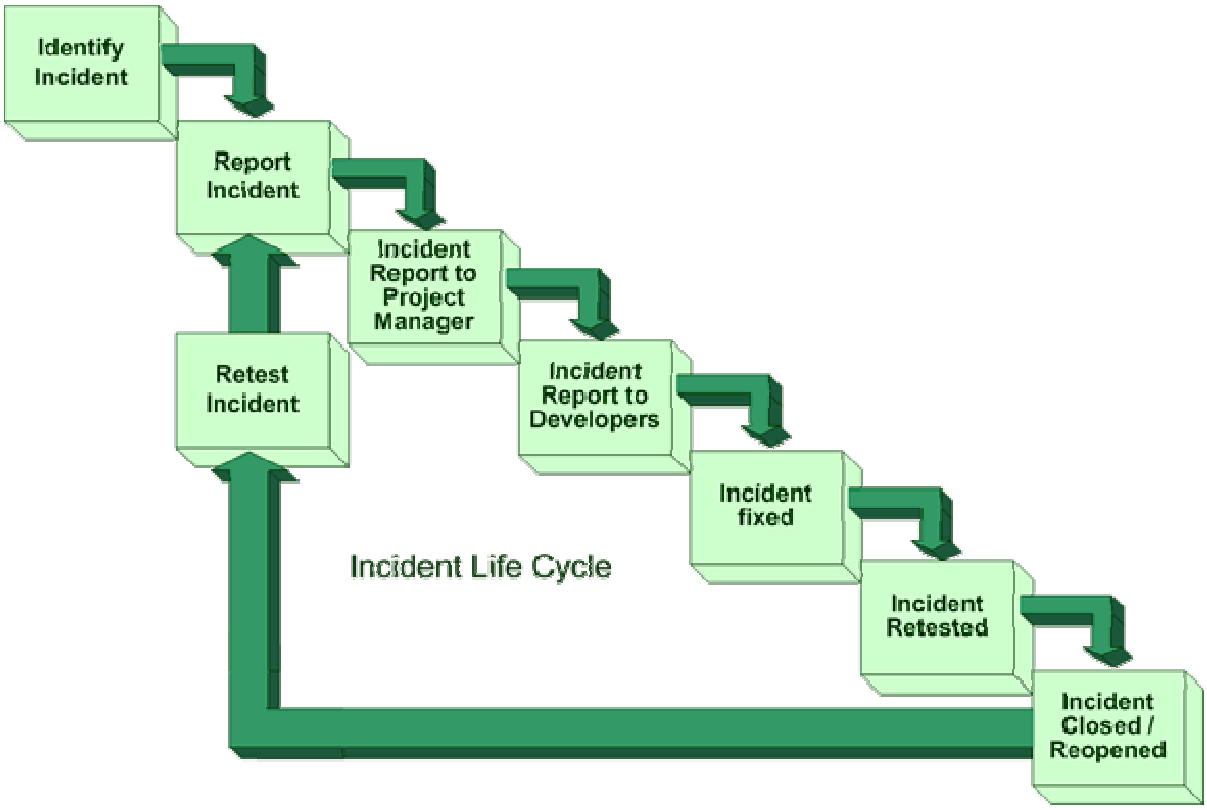
Purposes of incident tracking

There are three main reasons for implementing an incident tracking process:

- It is a mechanism for identifying and recording potential defects in system under test
- It helps to ensure that problems and facts are addressed appropriately and in a timely manner throughout the incident lifecycle
- It provides mechanisms to monitor defects and discover areas in which improvements could be made for a better quality product

Incident life cycle

The incident life cycle can be summarized in the diagram below



FIGURE

This is a simplified version of what may happen in the real world – an ideal situation.

Identification of incidents

While under test (or during general use) a system may exhibit behavior that is unexpected or environmental incidents may occur that prevent testing from taking place. This may be identified by a user, a developer or a tester.

The person who discovers an incident is known as the originator

The originator should raise an incident report. The raising of an incident report formally initiates an assessment of the unexpected behavior. In some testing set-ups the person finding the incident is not necessarily always the same person who writes the incident report. In these cases a person is designated to create the reports, or to enter them on whatever tracking system is being used. However, the originator will obviously have to pass on details of the incident to this person, wither verbally or in writing.

Investigate the incident

Once an incident has been identified, it is wise to investigate the circumstances in enough detail to provide a clear and full description of it – including the sequence of events that led to the incident taking place. In many cases it is prudent to refer to the specifications and cross check

with the test scenario to ensure that the incident is indeed unexpected. In some cases this can prevent time wasted on incidents raised in error.

Reporting the incident

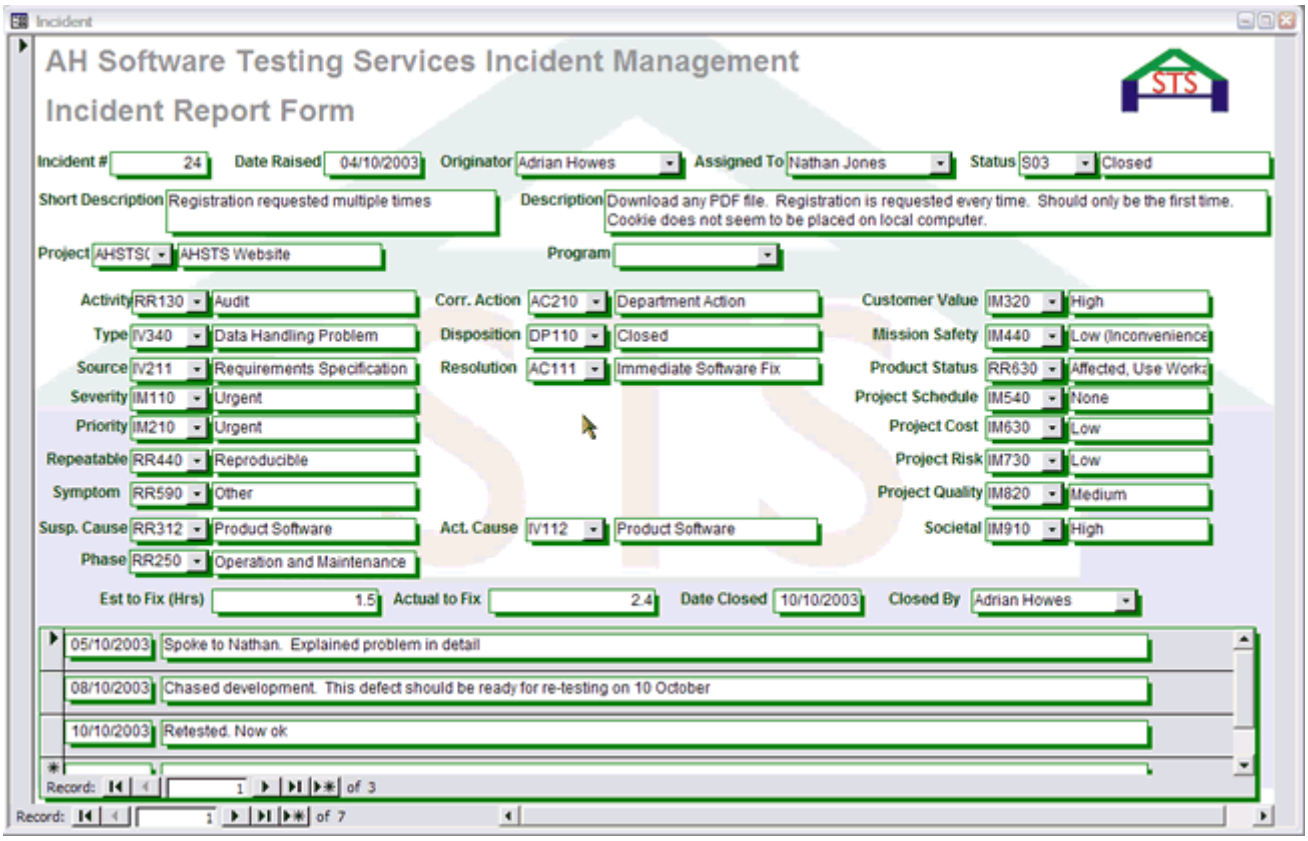
The incident should be reported, preferably on to an automated tracking system. The entry can be made directly by the originator or co-coordinator or entered by someone else (for example: support staff)

Automated systems are not always available and a variety of other methods can be used for tracking, ranging from a simple folder containing loose-leaf sheets of paper to a computer spreadsheet.

Incident reports can come in a variety of forms, for example

- Paper based – many companies have their own designs
- Electronic form – can be filled over a network (test director for example) or online

The form shown below is a from a proprietary incident management application that complies with IEEE standard in terms of the ability to classify all the required (and optional) categories.



There is a hierarchical nature to the classification of anomalies. The top levels of the hierarchy consists of the categories

- Requirements
- Design

- Implementation
- Test
- Operation and maintainace
- Retirement

Within these top level categories, more detailed classifications are listed. For example:

- Code RR210 requirements
- Code RR211 concept evaluation
- Code RR212 System requirements
- Code RR213 Software requirements
- Code RR214 Prototype requirements
- Code RR220 Design
- Code RR221 System design

And so on. This system of classification is flexible in that users can extend the classifications to meet their own needs.

Assessment

Each incident raised is assessed (by project management or an assessment panel) to determine whether it is to be considered a defect that requires resolution and to determine the impact of the unexpected behavior. The assessment normally consists of:

- Evaluating incidents
- Prioritizing incidents appropriately. This means making value judgments about the most important aspects of the defect at that time
- Adding appropriate comments
- Allocating to an appropriate person for resolution

Alternative actions may be

- Return the incident to the originator for more detail if there is insufficient to be able to make an immediate decision
- Mark as deferred, as designed or not reproducible according to the circumstances
- Depending on the phrase in which the incident is found and its severity, testing may also be reviewed to see why the error was not discovered earlier in the cycle

One of the purposes of incident assessment is to determine whether the development process could be improved to help prevent similar defects being introduced in the future.

Allocation for resolution

Incidents are allocated to an appropriate person for resolutions. The revolver (normally developer or business analyst or other responsible person) investigates the incident in order to:

- Determine how the defect may have originally been inserted into the system
- Fix the cause of the incident or

- Explain why the incident should not be fixed
- Determine whether there is a need to ask for more information from the originator
- Determine why the defect may have been previously overlooked.

Resolution

The method of resolution depends upon the type of incident. If it is a software defect, the system is modified to remove the defect and resolve the anomaly. Resolutions may involve modification to

- **Code**
- **Data**
- **Software configuration**
- **Documentation**
- **Screen designs**
- **Test environment**

Incidents may be resolved in a number of ways. They can be

- **Set to pending** – in this case the incident has yet to be resolved. This may be because of a technical difficulty for which the solution is not immediately available
- **Fixed** – the incident has been eliminated and the system is ready for retesting
- **Deferred** – in this case the incident has yet to be resolved. This may be because of a technical difficulty for which the solution is not immediately available
- **Marked not reproducible** – the incident cannot be recreated for whatever reason
- **Marked as designed or reject** – in this case the incident is thought not to be an error, but reflects the intended operation of the system.
- **Marked withdrawn by originator** – if the originator feels that the incident report should never have been raised – possibly a simple mistake
- **Marked need more information** – originator is requested to provide more information. An incident should not remain in this state for long. The defect is not yet finally resolved and should receive the attention it deserves depending on the severity and priority assigned to it
- **Marked 'disagree with suggestion'** – in this case a response to a change request where no change to the design will be made. The incident report can say that the operation is faulty and a change request raised as it is working as expected but the system is effectively unusable in the present form. If this is so, it should be designated as a change request and a change request cross reference given.
- **Set to duplicate** – making an incident in this way closes it because the anomaly is being taken care of another incident report. Cross references should be provided between the duplicates.

- **Marked as closed** – incident is fixed and retested. Only a tester should set an incident to close and preferably it should be the originator, although the latter is not always possible.

Validation

After modifications have been made during resolution and the incident is allegedly fixed and marked as fixed or ready to retest the repaired system is retested in order to check that:

- The incident was actually resolved
- Other problems related to the incident have been considered in the resolution
- The resolution has not unexpectedly affected other parts of the system

If the modification passes the retest, the incident status is set to confirmed fixed or closed. The incident should only be closed if a proper description of the issue has been given. If the description the resolver gets just says fixed then it prevents further analysis to determine error trends

If the modification fails the retest the status is reset to open. Some organizations choose to use a status of re-opened for this scenario. However the status reopened really only applies to incidents that have been closed at some point and have reoccurred. As long as all concerned in the incident management process know exactly what each status means, the exact wording is not crucial but standardization of terminology is a good thing.

If other problems are found in retest, a new incident is raised and cross references with the original defect.

Deferrals and the appeal process

A deferral course occurs when an incident is recognized but the decision is made not to fix it yet. The decision is based on type and severity and also if the functionality affected is on critical path of the project. The explanation for a deferral should be made in the incident report comments

Deferral review meetings are conducted to review deferred problems prior to releases. All these meetings not only should project management be involved but also should include marketing, technical support, development so that appropriate representations may be made as to why these deferred defects should be addressed more quickly.

Unaddressed incidents

If incident management is not carried out correctly, incidents may go unaddressed.

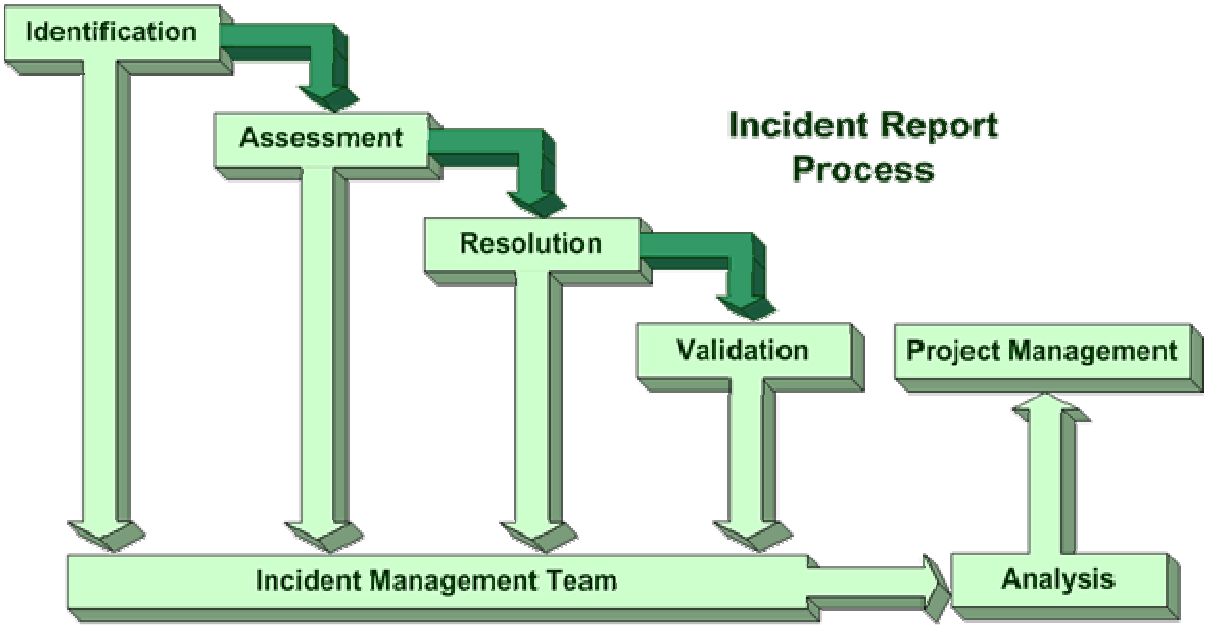
This may be because incident reports are

- Lost
- Deliberately or inadvertently set aside
- Assigned a low priority and forgotten

To help prevent this, summary reports should be circulated regularly to remind all involved of not yet-resolved problems. Also, summary reports should be reviewed in details prior to release.

Management role

The incident report process involves identification of the incident, its assessment, its resolution and validation that the resolution has been effective in removing the defect or other problem



At all stages reports are made to and by management. Management should receive reports concerning individual incidents which they can review for prioritsation. Normally there is an incident review panel which includes the project manager, the test manager and the development team leaders or a similar mix.

Management is responsible for co-coordinating the whole process

Incident analysis and summary reports

Incident analysis reports can include the following information

- Number of incidents found through the project so far
- Number outstanding
- Number deferred compared to number fixed
- Number found by testers and by others
- Progress each week

All of the above can be analyzed by various categories (such as severity, priority, type of error, stage at which discovered and so on)

These reports help the management to

- Evaluate the quality if programming effort
- Assess the current reliability of the product
- Assess the reliability of different aspects of the system under test
- Assess the effectiveness of the testing effort

- Rate of discovery of new incidents compared to rate of fixing incidents (or project likely completion date)
- Ensure that all defects are addressed as soon as practicable
- Assess the progress of defect resolution

The frequency of incidents summary reports is a matter to be decided by project management. A frequency of weekly is common, but a daily report enables a project management team to identify problems before they escalate. Close to project deadlines and with many incidents coming in there may be a shortening of reporting cycles. Frequency may also depend on the development method or project lifecycle being used.

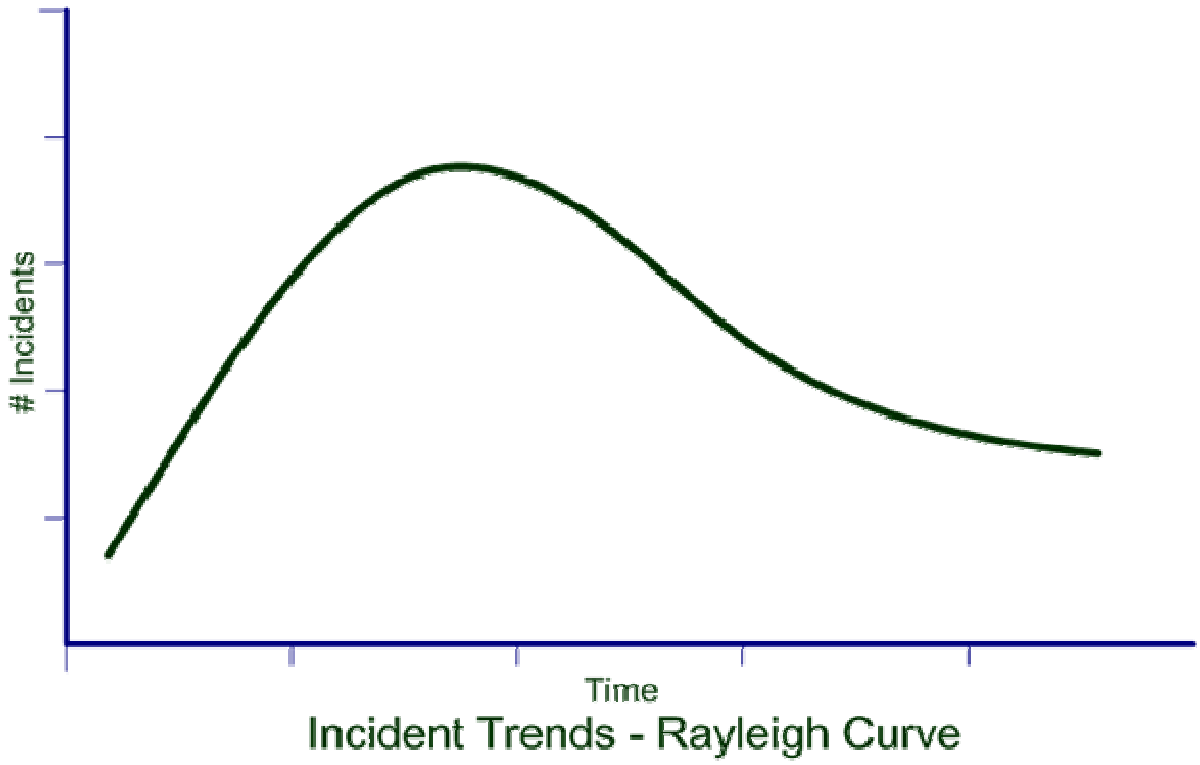
It is very useful to be able to generate reports on demand rather than having to wait for a predetermined interval.

Many different types of incidents summary reports may be generated. Some are illustrated in the following pages

Project management have the responsibility for the overall generation and assessment of reports summarizing the collected defect data.

Incident trend reports

Rayleigh curve

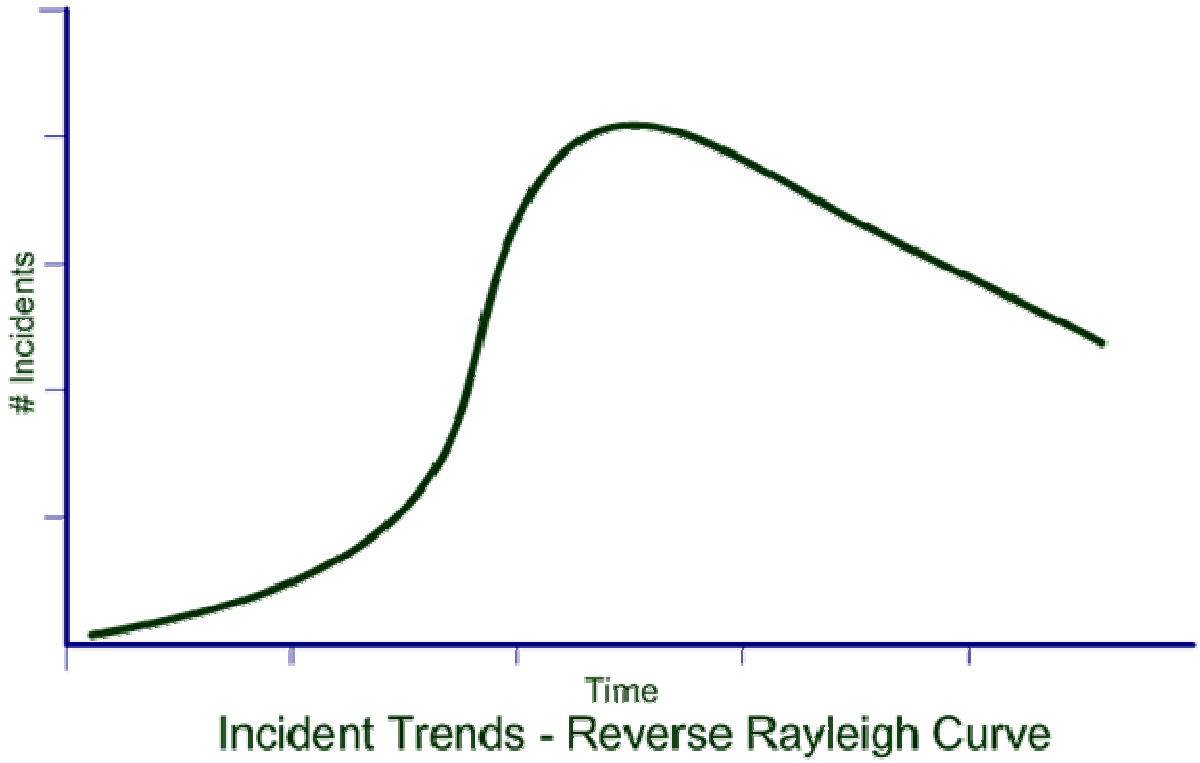


FIGURE

The Rayleigh curve is the expected shape for incident numbers detected over the course of a project. Its characteristics indicate that:

- Tests are being executed steadily
- Tests are being spread evenly across system
- No major changes are being implemented during testing
- There is a learning curve at the start
- Builds to a maximum
- Then a slow continuous decline as it becomes increasingly unlikely that new defects will be detected

Reverse Rayleigh curve

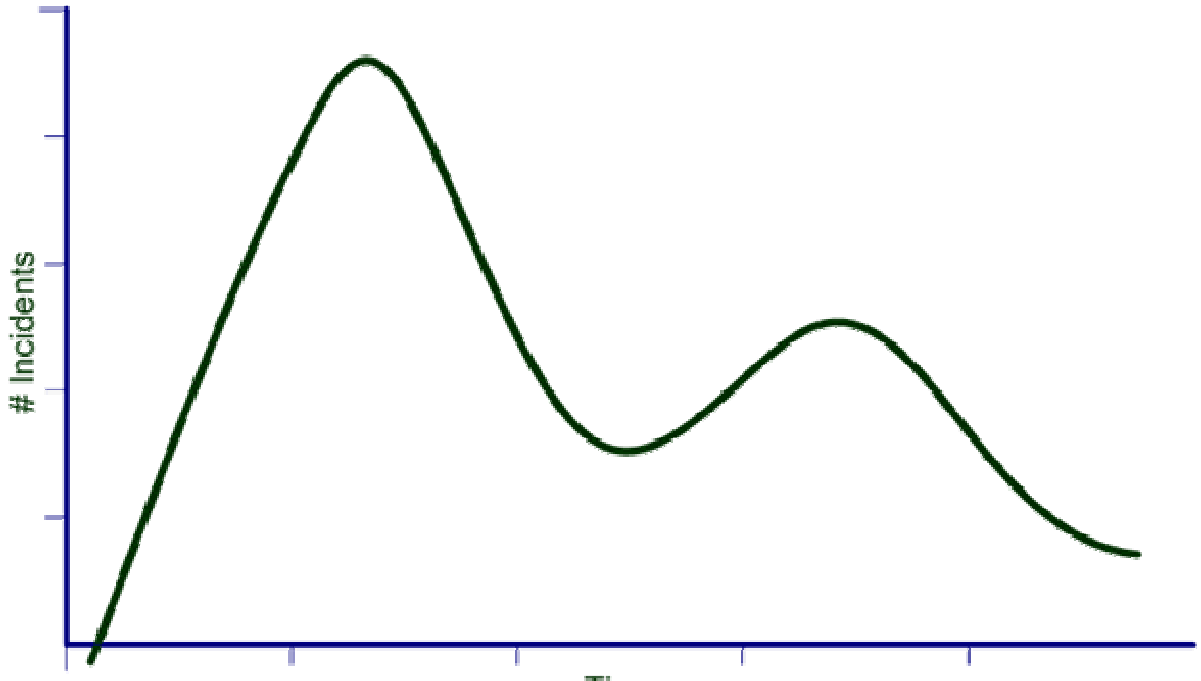


Figure

The reverse Rayleigh curve illustrate here that :

- There were problems in starting testing
- Few defects were detected early on the cycle – which could have led to premature confidence in the system
- Lot of defects were detected later in the cycle – which may create a panic to meet deadlines

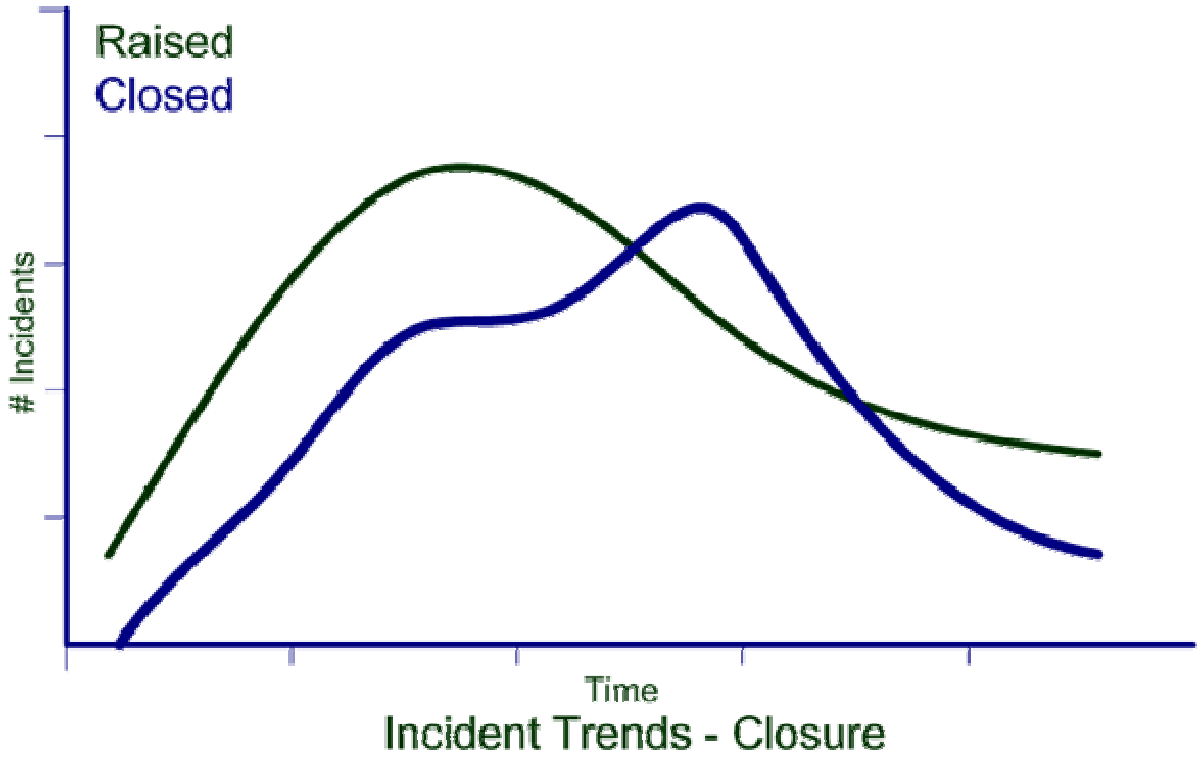
Squiggle curve



Incident Trends - Squiggle Curve

The squiggle curve may indicate simultaneous testing and modification
Later increases are produced by detecting incidents resulting from modifications which have introduced additional defects

Closure

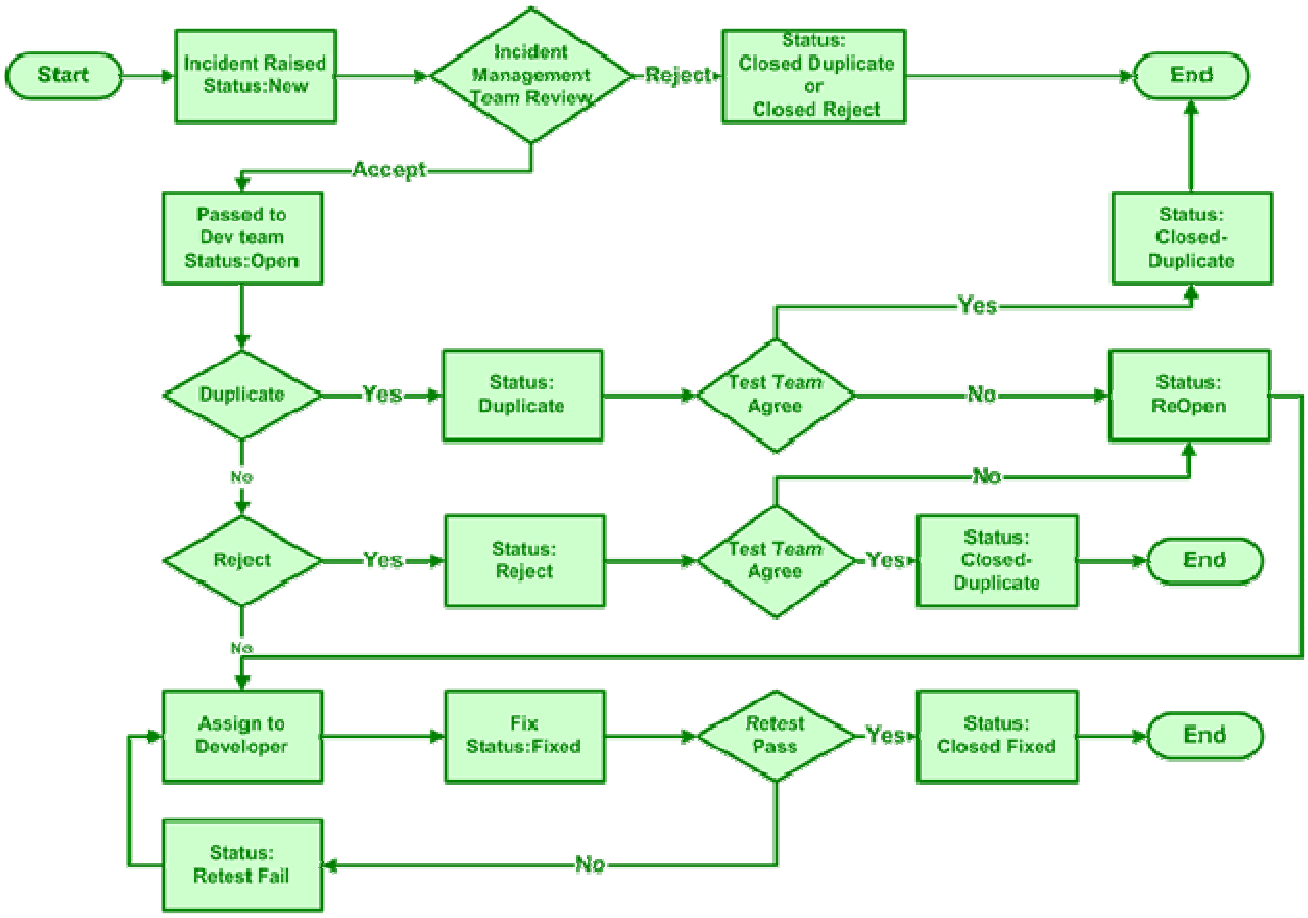


Incident Trends - Closure

The closer that the incident fixed curve is to the incidents raised curve the more efficiently defects are being fixed after being raised. This type of graph is useful to give an indication of whether

- Sufficient resources have been assigned to debugging and maintenance
- There has been effective debugging
- The code is demonstrating a reasonable degree of maintainability

The flow chart below represnts just one (simplified) example of a defect only lifecycle management process. There are almost as many detailed processes as there are organizations using them.



Incident tracking system requirements

To make the incident tracking process more manageable, it is useful to have a tracking system in place, whether it be a manual or a computerized system

An incident tracking system should be ale to support activities involved in

- Capturing
- Assessing
- Resolving
- Analyzing defects

The tracking system should also be able to

- Maintain a defect repository
- Record the history of individual defect resolutions

Security issues

Security is important in incident tracking systems to prevent loss of information or fudging. In particular

- Closed incidents should not be able to be changed

- Only the person or area the incident is assigned to should be able to change the incident report at that time
- Developers should never be allowed to close reports unless they opened them in the first instance

Search and browse facility

Incident tracking systems should provide all capability to query and browse through defects. Ideally the ability to query and browse be available to all users, including developers and testers so that when incidents are discovered a database may be searched for similar incidents which have been raised in the past to enable duplicates to be easily identified.

Reporting

Users should be able to extract defect statistics easily. The facility to generate pre-defined and customizable tabular reports or charts are an advantage, since these make it easier to quickly generate evidence which project management may use for the prediction of future defect numbers.

The tracking system should provide the capability to include notes, comments and attachments (such as screen shots, text taken from source code) in the description of the incident.

Link to other systems

The tracking system should support the co-ordination of defect activities amongst different project staff. Of developers and testers can both use the system simultaneously, for example with adequate record-locking procedures in place, this would be an advantage.

It would also be useful if the tacking system could link defects to other project attributes such as:

- System versions
- Test case sub-components
- Who is the author of the original code
- Who is responsible for fixing
- Which tester found the defect
- Which tester re-tested the defect
- System documentation

The provision of external interfaces to email and the internet aids in keeping communication of defects details efficient.

Integration with version control tools would be helpful. Change management is a vital part of the project and can also deal with incidents. Configuration management should be kept in mind here in connection with version control since an incident could be caused by something not being configured (such as the environment, the database, the PC and son on)

Standards of testing

Introduction

What is a standard?

The deliberate acceptance by a group of people having common interests or background of a quantifiable metric that influences their behavior and activities by permitting a common interchange

A document established by consensus and approved by a recognized body that provides for common and repeated use, rules, guidelines or characteristics for activities or their results, aimed at the achievement of the optimum degree of order in a given context. – ISO

The three types of standard

Quality assurance standards – are produced by specific bodies
Industry specific standards – are the result of a combination of practices within an industry
Testing standards – are produced by independent bodies

The purpose of testing standards is to provide discipline and structure for the testing process

Quality assurance standards

Quality assurance standards are written by specific bodies and state that testing should be performed

Examples include

ISO 9000-3:1997 – Guidelines for ISO 9001:1994 the development, supply, installation and maintainace of computer software

IEEE 1012: 1998 – Standard for software verification and validation

Industry specific standards

Industry-specific standards specify what level of testing to perform and are written by the industry concerned or by government

Examples include:

DO-178B: Software considerations in airborne systems and equipment certification
Railway signaling standards

Testing standards

Testing standards specify how to perform testing

Examples include

- **BS 7925-1** Software testing vocabulary
- **BS7925-2:1998** Software component testing
- **IEEE 829-1998** Standard for software test documentation

Ideally, testing standards should be referred from both QA and industry specific standards

The language of standards

Standards tend to contain phrases which have very specific meanings. In order to be able to use and follow the standards correctly an understanding of the language employed is essential. Key phrases include:

Shall/shall not: these terms don't indicate requirements that are to be followed strictly in order to conform to the standard and from which no deviation is permitted

Should/should not: these terms indicate that among several possibilities one is recommended as particularly suitable, or that a certain course is preferred but not necessarily required

May need not: these terms indicate actions permissible within the limits of the standard

Can/cannot: these terms are used for statements of possibility and capability, whether material, physical or casual

Standard bodies

A number of bodies exist which define and promulgate various standards. Some examples are:

ISO – The international organization for standardization

IEEE – the institute for electrical and electronics engineers

ITU international telecommunications union

Specialist groups – these exist for particular industries more often than not legally mandated and controlled, for example the nuclear and aeronautics industries

Benefits of standards

Standards can educate and inform. Often a great deal of extra information and background to the subject is included in standards. BS 7925-2 (Software component testing) for example includes a number of examples the use test technique.

By increasing understanding, standards can improve the quality of work that users of the standards produce.

Standards require compliance with a document, formal rigorous, disciplined and repeatable process. If any process is run under these conditions the products of the process will have a better chance of being improved quality.

Possible problems with standards

Some software standards are not standards. Not everyone uses them, and those that do use them do not always adhere to them one hundred percent. The worst case is mistakenly to assume that you are adhering fully to a standard but in fact you are implementing only part of it or implementing it wrongly.

It is practically impossible to measure conformance to software standards. There are no real objective tests although some methodologies are available for testing, such as the capability maturity model (CMM) or SPICE which is the software process improvement and capability determination. Unfortunately, these are not specifically for measuring conformance to testing standards as such, more for measurements of the use of a reasonable testing procedure.

Some software standards prescribe, recommend or mandate the use of technology that has been validated objectively. It is all very well stating that a particular technology should be used, but many of those technologies are suspect in themselves.

Many software standards are too big. For many projects, the sheer scale of some standards and the detail and the complexity they encompass makes them impractical for full compliance.

Summary

- There are 3 types of standards for testing
    - QA standards – which tell you to do it
    - Industry specific standards – which will tell you at which level to do it
    - Testing standards – which will tell you how to do it
- Ideally testing standards should be referenced from the other two.

CHAPTER 6

Types of CAST tools

Introductions

What are CAST Tools?

- CAST stands for **Computer Aided Software Testing**. CAST tools are pieces of software which to a greater or lesser extent automate many of the tasks associated with the testing process.

- They can be time saving; computers can normally outperform humans, but only as long as the tools are set up and used correctly.

- They can perform tasks impossible to do manually; for example they can examine locations in computer memory an analyze individual software or hardware component performance

- They can help to reduce boredom; computers can repeat a task over an over again almost as infinitum very accurately, quickly and precisely and without tiring.

Requirements testing tools

Keeping track of requirements is vital to systems development. In a volatile business environment, system requirements fluctuate and expand all the time. Requirements management is usually the first area to suffer in such an environment. Without clear and structured requirements management, chaos can ensue. As business processes and system requirements

increase in complexity, human error in requirements management can cause recurrent and costly problems in development.

There are many tools available which provide automated support for the verification and validation of requirements. Some are detailed below:

Consistency checking tools

Tools that help to check consistency of code style, design format and so on.

Requirements change management tools

- **Requirements classification tools:** They allow classification of various requirements types that include functional, non functional, interface, performance, security, system requirements, etc.

- **Automatic requirements identification tools:** They enable speedy unique requirement identification through automatic requirement tag generation

- **Automatic version control tools:** They can maintain all versions of your entered requirements and changes. This means you can revert back to any version of a particular requirement very easily. Some tools have an automatic version history facility. This change management feature also allows you to generate a report on your requirements-change-history at any time.

Requirements Analysis tools

Requirements are defined in the tool or a tool-accessible database. The tool reviews the requirements based on pre-defined rules. An example is Rotational Requisite Pro.

Requirements Traceability tools

Requirements can be stored separately, in a relationship (database) or linked to design objects. Using traceability features, you can document and identify lower level requirements derived from parent requirements. Traceability features allow you to study the impacts of requirements changes. You can generate data on and examine the impact of change on lower level requirements. You can run different traceability reports which allow you to evaluate the change from perspectives of the business, of the application and the user.

Animation tools

Formal methods allow implementations to be built that are correct with respect to a formal requirements specification. However, this doesn't mean that the initial specification adequately captures the user's requirements. Animation is one way of validating the specification against these needs. Animation executes a specification and gives the developer a means of testing a specification. An example is Possum, an animator used to support testing, concentrating on reuse of animation processes and products during testing.

Static Analysis Tools

Static analysis tools provide information about the quality of the software by examining the code of the application, rather than by running test cases through it. Static analysis tools usually

provide objective measurements of various characteristics of the software, such as the cyclomatic complexity measure and other quality metrics.

Below is a list of some of the functions provided by Static Analysers:

- Programming Standards Verification
- Structured Programming Verification
- Complexity Metric Production
- Procedure Interface Analysis
- Variable Cross-Reference
- Unreachable Code
- Reporting Static Dataflow Analysis
- Code Reformatting
- Loop Analysis
- Analysis of Recursive Procedures

Other measures can include:

- Depth of nesting
- Coupling (linking of modules, transfer of information)
- Cohesion (grouping of modules, functional, sequential, procedural)
- Comment code

Our example of Static Tool Analysis is LRDA Testbed.

Dynamic Analysis Tools

Dynamic Analysis Tools provide run-time information on the state of software as it is being executed. They are most commonly used to:

- Monitor allocation, use and de-allocation of memory
- Flag memory leaks
- Identify unassigned pointers
- Find pointer arithmetic
- Find divide-by-zero errors

Dynamic Analysis provides for the coverage of:

- Statements
- Decision paths
- Loops

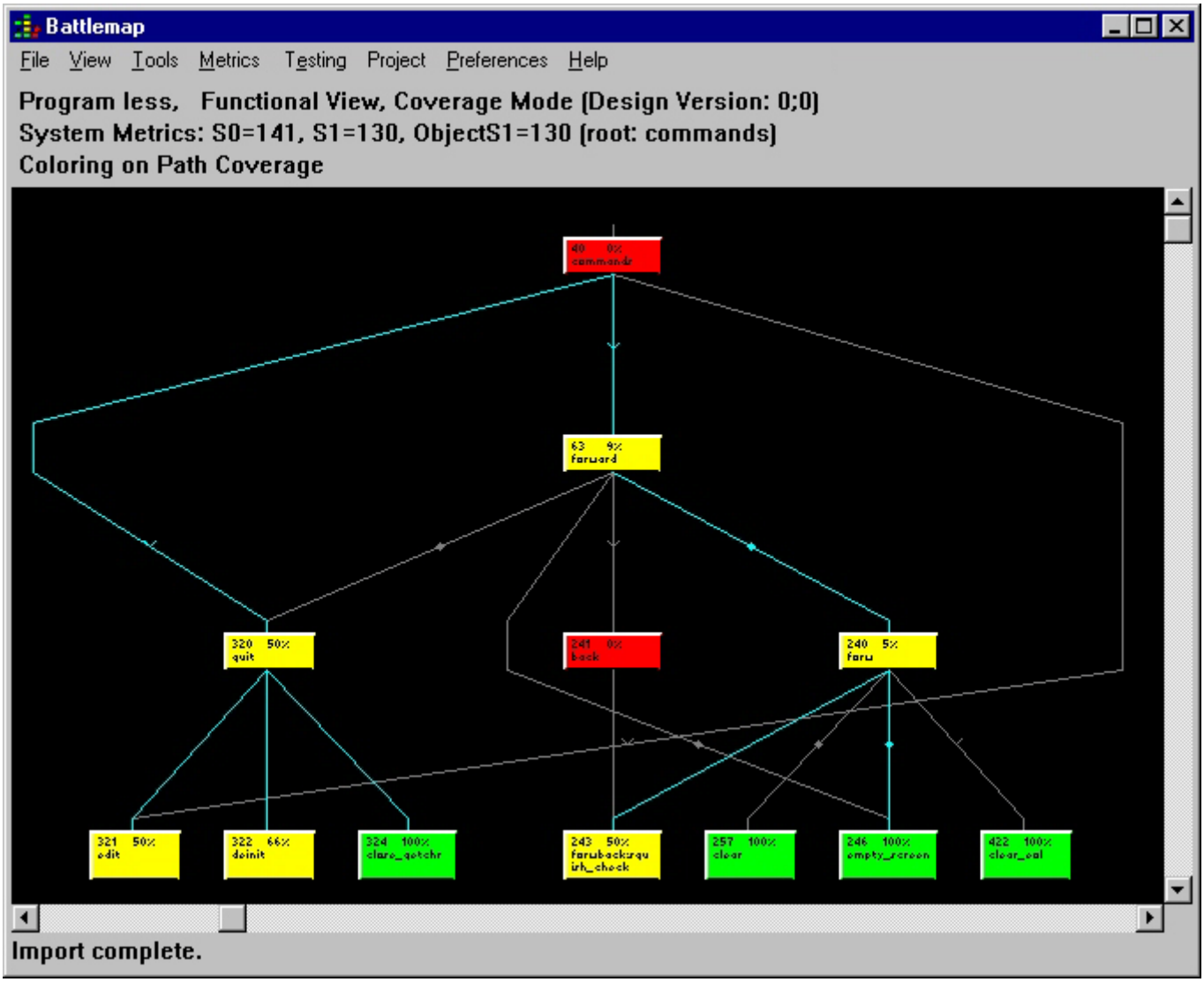
This would not be possible using static analysis methods.

The program or system must be stable before dynamic testing can be carried out.

Test Design Tools

Test Design tools generate test cases from a specification that must normally be held in a CASE (Computer Aided System Engineering) tool repository or from formally specified requirements held in the actual tool itself. Some tools generate test cases from any analysis of the source code.

An example is McCabe test. Below is an example of test cases in diagrammatic form in McCabe test. From a set of requirements the test tool analyses the structure and produces a set of tests to provide pre-defined coverage.



FIGURE

Test Data Preparation Tools

Test data preparation tools enable data to be selected from existing databases or created, generated, manipulated and edited for use in tests. The most sophisticated tools can deal with a range of file and database formats.

Creating test data often takes as long as the defining tests themselves, particularly when testing systems with substantial sized databases. Database applications require lots of testing and they also require lots of test data. Often, highly paid database architects or programmers take hours keying in banal customer names such as “John Doe” and insipid addresses such as “123 abc street”. They do this because they know the database and the types of data it expects. This is not

a very good use of anyone's time – particularly for highly paid professionals. In any case, it is a very boring task to sit and keyboard typing in large amounts of data that no one will actually use.

Test Preparation tools are designed to generate test data for a wide variety of DBMSs (Database Management Systems) – you can get them for use with just about anything for which you can get an ODBC driver. Even if you can't get an ODBC driver for a particular DBMS, you can still generate 'flat files' such as common delimited files, which you can later import.

Most of these tools generate random test data based on the data types of the defined fields – for example, valid dates for date fields, numbers for numeric fields and characters strings for character fields. You tell it how many rows to generate and it takes care of the rest.

TestBytes is one example of a test data preparation tool.

It is possible to use simpler, more generic tools, such as spreadsheet or word processor.

Nonsensical (automatically generated) data is of questionable value – it can make determining whether your application is working correctly difficult. (Is the peculiar data being output the fault of the application or of the test data?)

Character-Based Test Running Tools

Capture/replay tools enable recording and replaying of a test script. They also enable the capturing of selected test outputs for comparison against future replays.

They can either be intrusive or non-intrusive in operation. An intrusive system resides on the same machine as the system under test; a non-intrusive system can reside on a separate computer or is an extra piece of hardware connected to the test system.

Character base test running tools provide test capture and replay facilities for dumb-terminal based applications – for example, early PC-DOS based systems and those running the UNIX operating system. They simulate user entered terminal keystrokes and capture screen responses for later comparison. Test procedures are normally captured in a programmable script language. Data, test cases and expected results may be held in separate test repositories.

This type of tool is most often used to automate regression testing.

GUI Test Running Tools

GUI (Graphical User Interface) test running tools provide test capture and replay facilities for GUI based applications (For example Windows and MAC)

The tools simulate mouse movement, button clicks and keyboard inputs and can recognize GUI objects as windows, fields, buttons, and other controls.

The states of objects and bitmap images can be captured for later comparison; objects such as Windows, boxes, text labels, input boxes; states such as enabled/disabled, boxes checked or unchecked and so on.

Test procedures are normally captured in a programmable script language – TSL in the case of Mercury Interactive's WinRunner. Data, test cases and expected results may beheld in separate repositories – in the case of WinRunner, it can use Excel spreadsheets from which to read test data and to which to write results.

These tools are most often used to automate regression testing where consistent data input is vital if the system is being checked for consistent response to the same input.

Disadvantages:

- The system being tested under must be stable
- It can be time consuming to develop scripts – especially when there are multiple paths for choices in the program. Some web based applications dynamically change the name of the objects, causing further problems of misrecognition of objects.
- It can be time consuming to maintain scripts – if requirements change, so must the scripts.
- Inappropriate use can be worse than not using them at all. The software can become shelfware all too easily.

These types of tool should only be used to back up good testing practices already in place, not used as a cure for bad practice.

Debugging Tools

Debugging tools are used mainly by programmers to reproduce bugs and to investigate the state of programs.

Debuggers enable programmers to execute programs line by line, to halt the program at any program statement.

An example is the Visual Basic editor which has an inbuilt debugger.

Test Harness and Drivers

Test harness and drivers are used to exist, but custom-written programs also fall into this category.

Simulators

Definition

Device, computer program or system used during software verification, which behaves or operates like a given system when provided with a set of controlled inputs.

BS7925-1

Simulators are used to support tests where code or other systems are either unavailable or impracticable to use (for example, testing software designed to cope with nuclear meltdowns). They can also be a lot cheaper than the real thing – for example, software for inter planetary spacecraft.

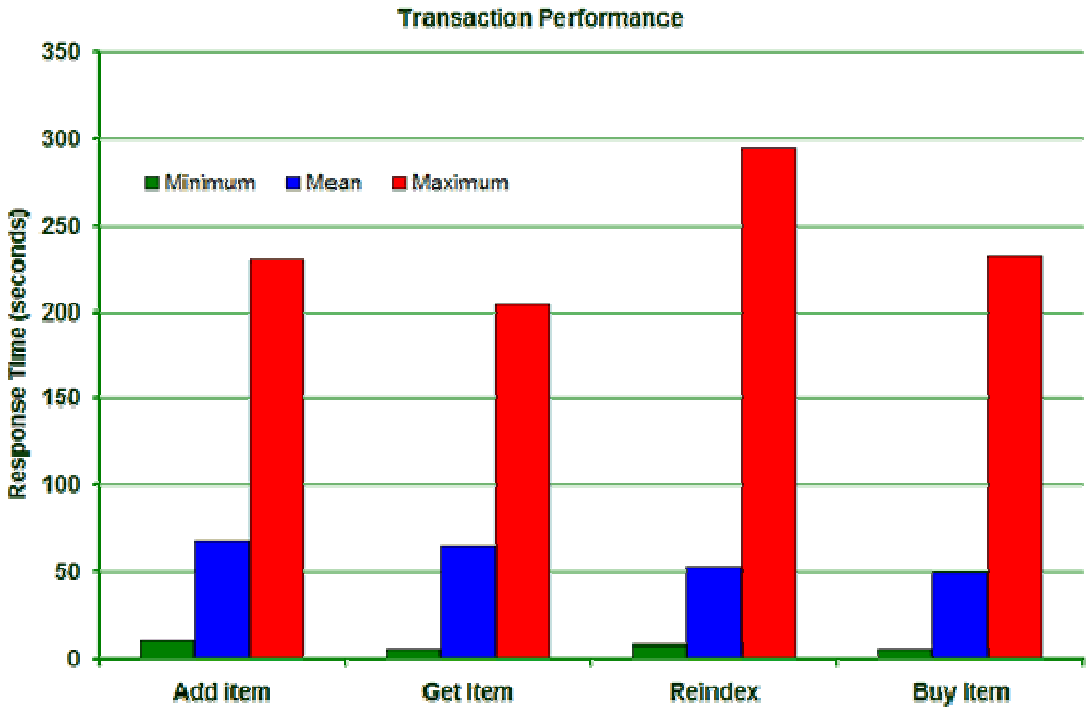
Performance test tools

Performance test tools measure normal system activities. They drive the system and can measure transaction response times. There are two main types:

- Load generation tools
- Transaction measurement tools

Transaction measurement tools

With this type of tool, specific transactions are measured for response times. Single or multiple transactions may be measured at once. Typical transactions for the application are measured; for example, record retrieval or record update. The diagram below is an example transaction graph using timing data produced from a LoadRunner test run.



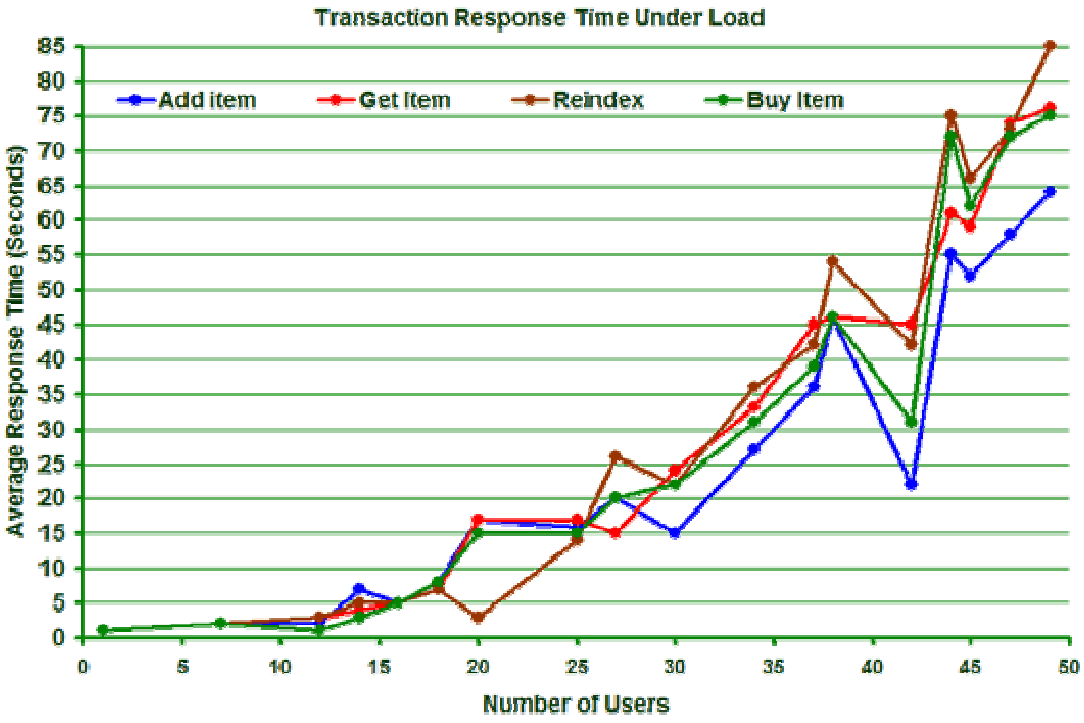
Load generation tools

Load generation tools simulate load on systems (multiple users). This type of performance test tool simulates tens, hundreds or thousands of users accessing the system at once. Examples are LoadRunner, WebLoad or QA Load.

System performance is measured for various loads. Tools can simulate multiple sessions from a single device, in other words a sloe C may be all that is required to simulate hundreds of users. The tools can drive the interface itself or by using test drivers.

Although not always easy to set up and quite costly (usually those able to simulate more users are most expensive), simulating particular loads more cost effective than employing many people and a lot of hardware to reproduce a load.

The diagram below is an example of a load performance graph. Various average transaction times are plotted against number of virtual users to see how performance degrades as load is increased.



Test Management tools

Test management tools are designed to centrally manage and store test assets – scripts, plans, specification, results, incident reports and so on.

There is a wide range of tools available – with a correspondingly large ranger of features and focuses.

The main point is that by storing test assets in a structured, easily searchable and retrievable manner and particularly by linking test material to specifications it enables impact of changes to be assessed and quantified and enables test coverage to be measured more readily.

Some tools support the project management aspects of testing, for example the scheduling of tests, the logging of results and the management of incidents raised during testing. Incident management tools may also have workflow-oriented facilities to track and control the allocation, correction and re-testing of incidents. Most test management tools provide reporting and analysis facilities.

It is common for the management tools to integrate with the other tools – for example, TestDirector may be used to hold scripts and scenarios which may be used automatically by LoadRunner

Examples of test management tools are Mercury Interactive's TestDirector

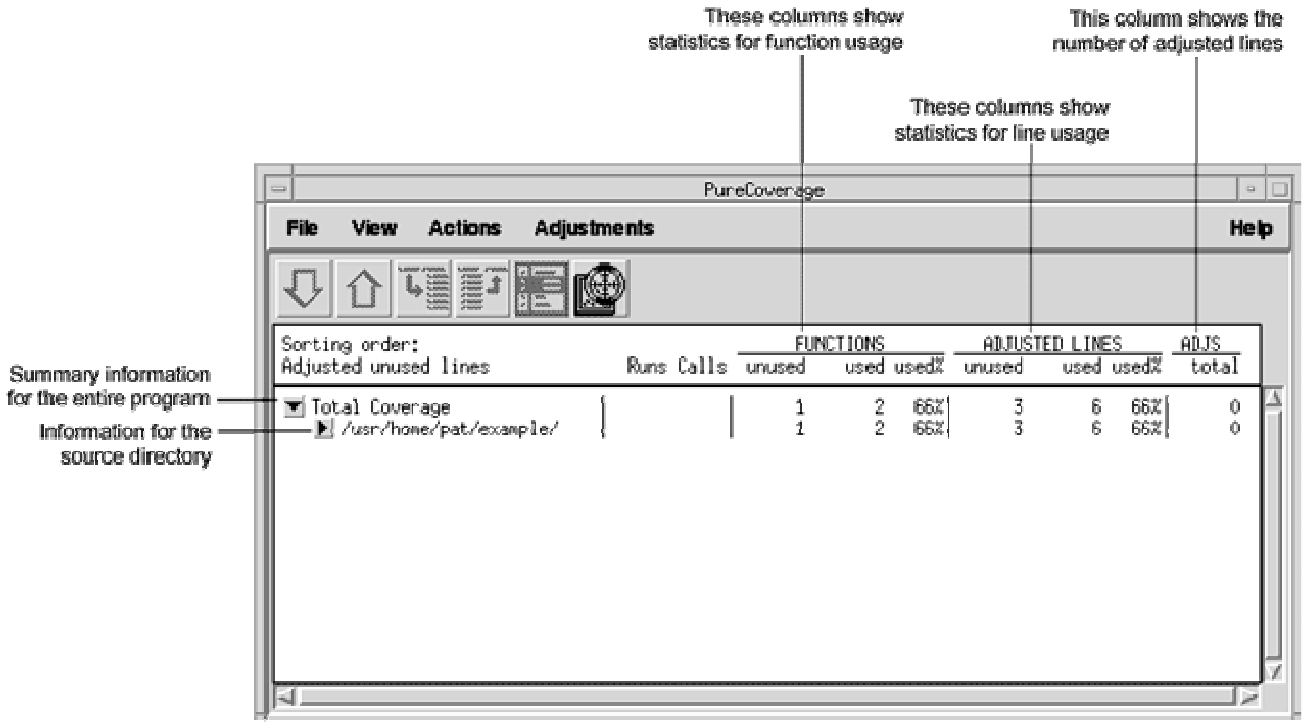
Coverage Measurement Tools

Coverage measurement (or analysis) tools provide objective measures of structural test coverage when tests are executed.

When using these tools programs to be tested are instrumented before compilation. This means that the coverage measurement tool makes itself notes about the code, in terms of the statements and structures (IF's, WHILES's, FOR's and so on) and places these notes in a log file. Implementation code dynamically captures the coverage data in a log file without affecting the functionality of the program under test. After execution of the program, the log file is analyzed and coverage statistics are generated.

Most tools provide statistics on the most common coverage measurement such as statement or branch coverage – enabling identification of the portions of the application that tests have not exercised. More sophisticated tools have the facility to accumulate coverage data over multiple runs and multiple builds of software and to merge data from different programs sharing common source code.

Rational Pure coverage is one coverage measurement tools. Below is an example output from the RationalPure coverage. The report shown here is a simple "Hello World" program with three functions. Two or three functions have been used, giving function coverage of 66%.



Comparison tools

Comparison tools are used to detect differences between actual results and expected results.

Standalone comparison tools normally deal with a range of file or database formats – Excel spreadsheets, Oracle or Access databases, tab or comma-delimited text files and so on.

Test running tools usually have built-in comparators that deal with character screens, GUI objects or bitmap images – for example, WinRunner.

These tools often have filtering or masking capabilities, whereby they can ignore rows or columns of data or areas.

Summary

- CAST – Computer aided software testing
- CAST tools help to automate various areas of testing
- There are tools available for all parts of the testing lifecycle except test planning

Tool selection and Implementation

What can be automated?

Most aspects of testing can be automated

- Test analysis/design
- Test case selection
- Test development
- Test execution
- Comparison of results
- Load and performance

Also, test metrics, defect tracking and reporting and configuration management. There are many test activities which can be automated and test execution tools are not necessarily the first or only choice.

Test planning cannot be automated.

To automate or not to automate?

Before deciding what tools to procure, the decision has to be made whether or not automation is appropriate.

Firstly, false expectations must be overcome:

- It is not possible to automate everything – test planning cannot be automated for example
- Multiple tools may be needed – one tool will not usually do everything that is required, whether it is because different task are needed or that there are different hardware or software environments not all supported by the one tool.
- The test effort does not decrease immediately on using tools – although one of the main aims of introducing automation is to reduce the overall test effort, there is a learning curve associated with their introduction an setting up complicated scripts to make decisions (that a human can make almost without thinking) takes extra effort.
- Test schedules don't shorten immediately – just as the test effort may initially increase, the testing schedule may well be extended. Only after the tool has been properly implemented may there be associated productivity gains.

- Use of tools may require new skills – training may be required, adding to drains on project resources and time.

- Tools wills still not provide 100% coverage – just because automation is being used does not mean that every path through a program or every combination of input to a field (both valid and invalid) will be tested – there are still far too many combinations

CAST Tool requirements

When considering the requirements for CAST tools on a project, it must be borne in mind that many test activities can be automated, but test execution tools are not necessarily the first or only choice. The first actions to perform when specifying requirements are:

- To identify the test activities where tool could be of benefit

- Prioritize the areas of most importance

CAST Readiness

The ease with which CAST tools can be implemented can be called 'CAST Readiness'.

The fit with the test process may be more important than choosing the tool with the most features in deciding whether a tools is actually required (and if so, which one)

The benefits of tools usually depend on a systematic and disciplined test process. If testing is chaotic – tools may not be useful and may actually hinder testing. So, a good testing process should already be in place – without one, ad-hoc and no-repeatable, non-measurable testing may result. If a good test process is not already in place, then the process must improve in parallel with the tools implementation.

In addition, standards should be in place and adhered to – particularly for test scripts, other wise non –repeatable tests could result and produce an inability or re-use them in later builds of the software.

As had been stated, 100% automation is usually impossible – not everything can always be tested by one tool. Attempting to get a tool do something it is made to do is a waste of time and can delay proper testing.

CAST tools must be introduced early enough in the life cycle. Introducing CAST late in the cycle can leave too little time for set-up and proper configuration. Testers must know how to use tools properly – training may be necessary. In any case testers should be advised of the tools they will be expected to use as soon as possible so they can plan accordingly.

CAST Tool Selection Process

Once automation requirements are agreed, the selection process has 4 stages:

1. Creation of a candidate tool shortlist
2. Arrangement of demonstrations
3. Evaluation of selected tools
4. Review and selection tool

Candidate tool shortlist

- Whoever is performing the evaluation (more than likely a test automation specialist) matches the test tool requirements with the list of test tools available to obtain a list of possible candidates. If one or more test tools exist tat can do the same task, the evaluator must determine, according to some functional evaluation criteria which are the best to go forward into a shortlist.

- Criteria to consider include:

- Tools may have interesting features, but may not necessarily be available on the platforms being used. For example, it may work on 15 versions of UNIX but not yours.

- Some tools, for example performance testing tools, require their own hardware as the cost of procuring this hardware should be a consideration in the cost benefit analysis.

- If tools are already being used it may be necessary to consider the level and usefulness of integration with other tools. For example, it may be required to integrate a test execution tool with an existing test management tool (or vice versa)

- Some vendors offer integrated toolkits, such as test execution, test management and performance testing bundles. The integration between some tools may bring major benefits while in other cases; the level of integration is cosmetic only.

Arrange demonstrations

Presentation and demonstration of tools helps by:

- Allowing questions to be asked of product seller – obtaining assurance that this tool will work in a particular environment, that it will do the job expected of it and so on.

- Increasing 'buy-in' by those who have to pay – if the project owners are involved, they can see first hand how the tool may be able to save time effort and money in the long term.

- It encourages interest throughout the project – those who are going to use the tool can see what to expect and can raise issues from a more informed position.

CAST Tool Selection Process:

Evaluation of selected tools

Evaluation of short-listed tools must be against pre-defined functional criteria – after all, the process of evaluation is similar to testing in that you should know before you test what you are looking for. One way of doing this is to use a score card where various characteristics are scored and weighted to give a value. Below is the part of an example evaluation scorecard for a GUI Testing tool

Table / figure

Other characteristics that can be evaluated include:

- General features – availability, support, maturity

- Technical specifications – applicability, compatibility

- Quality attributes – usability, adaptability, reliability, performance

Tool should be exercised against a particular application and environment and the details of the set-up noted for future reference in the event of problems arising

The evaluation will involve expense and a cost in both time and resources – this should also be included in any evaluation criteria.

Review and select tool

This is the final stage of the selection process. Results of the evaluation process feed into the review and the final selection of the tool is made. The tool then goes on to a pilot project

Pilot project

Before making a commitment to implementing the tool across all projects, a pilot project is usually undertaken to ensure that the benefits of using the tool can actually be achieved. The pilot project is a small scale installation to prove the test tool in a working environment. It should not be run in business critical areas but should be complex enough to put the test tool through its paces. If things go badly wrong then if the project is not in a business critical are, any damage should be limited.

The test team needs to obtain experience with an automated test tool on a small project before automation is rolled out on a large scale throughout the Organisation. During this stage it will be possible to identify any changes required to the test process and to assess the actual costs and benefits of implementation

Even though the test tool vendor may guarantee a test tool's functionality, it is often the case that tools do not work as expected in a particular environment. To verify that it will run properly in a particular environment, it may be worthwhile setting up an environment similar to that of the pilot project and trying out the software there before rolling it out to the pilot.

Roll-Out

Roll out of the tools should be based on a successful result from the evaluation of the pilot project.

Roll out normally requires strong commitment from tool users and new projects, as there is an initial overhead in using any tool in new projects.

Summary

- Automation tools can help most areas of testing

- A good test process must be in places before using them

- Care has to be taken in evaluation, selection and implementation

- Stages of the selection process are

    - Create shortlist
    - Arrange demonstrations
    - Evaluate

- Review and select

GLOSSARY

Principles of testing

Testing terminology

The BCS SIGIST standard glossary of testing terms (British Standard BS 7925-1)

Why testing is necessary

Define errors, faults, failures and reliability; errors and how they occur; cost of errors; exhaustive testing is impossible; testing and risk; testing and quality; testing and contractual requirements; testing and legal, regulatory or mandatory requirements; how much testing is enough

Fundamental test process

The test process; successful tests detect faults; meaning of completion or exit criteria, coverage criteria

The psychology of testing

Testing to find faults; tester-developer relationship; independence

Re-Testing and regression testing

Fault-fixing and re-testing; test repeatability; regression testing and automation; selecting regression test cases

Expected results

Identifying required behavior

Prioritization of tests

Test scope and limited resources; most important tests first; criteria for prioritization

Testing throughout the lifecycle

Models of testing

The V, V & T model.

Economics of testing

Early test design; how preparing tests find defects in specifications, cost of faults versus the cost of testing.

High level test planning

Scoping the test; risk analysis; test stages, entry and exit criteria; test environment requirements; sources of test data, documentation requirements

Acceptance testing

User acceptance testing, contract acceptance testing, alpha and beta testing.

Integration testing in the large

Testing the integration of systems and packages; testing interfaces to external organizations (Eg: Electronic data interchange, internet)

No-Functional system testing

Non-functional requirements; non-functional test types; load, performance and stress; security, usability, storage, volume; installability, documentation, recovery

Functional System testing

Functional requirements; requirements based testing, business process based testing

Integration testing in the small

Assembling components into sub systems; subsystems to systems, stubs and drivers, big bang, top down, bottom up, other strategies

Component testing

(Also known as Unit, Module, program testing) overview of BS 7925-2 Software component testing, component test process

Maintenance testing

Problems of maintenance, testing changes, risks of changes and regression testing

Dynamic Testing Techniques

Black and White Box testing

Functional or black testing; structural, white or glass box testing; trend from white box to black box through the lifecycle; techniques and tools

Black box testing techniques

Black box techniques as defined in the BCS Standard

White box test techniques

White box techniques as defined in the BCS standard

Error guessing

Using experience to postulate errors; using error guessing to complement test design techniques

Static Testing

Reviews and test process

Why, when and what to review?; costs and benefits of reviews

Types of review

Types of reviews; goals; activities performed, roles and responsibilities, deliverables, pitfalls

Static analysis

Simple static analysis; compiler-generated information; dataflow analysis; control-flow graphing; complexity analysis

Test Management

Organisation

Organizational structures for testing; team composition

Configuration Management

Typical symptoms of poor CM; Configuration identification; configuration control; status accounting; configuration auditing

Test estimation, monitoring and control

Test estimation; test monitoring; test control

Incident management

What is an incident; incidents and the test process; incident logging; tracking and analysis

Standards for testing

QA standards; industry-specific standards; testing standards

Tool Support for Testing (CAST)

Types of CAST tool

Requirements testing; static analysis; test design; data preparation; character-based test running; GUI test running; test harness, drivers and simulators; performance testing; dynamic analysis; debugging; comparison; test management; coverage measurement.

Tool selection and Implementation

Which test activities can be automates?; CAST tool requirements; which tool types to use? Test process maturity and CAST readiness; selection process; tools, platforms and CAST integration; pilot projects and roll out.