ISTQB Advanced Level

Certification Exam

Self Study E-Book

Chapter –1: Testing During the Life Cycle

Several models currently exist in relation to the Systems Development Life Cycle (SDLC). When referring to SDLC we simply mean the activities of the software development and maintenance. Let's look at some of the most common models for testing in use today:

- V & V
- Waterfall Model
- V – Model
- Spiral
- RAD
- DDSM

V & V

V & V represents; Verification and Validation.

Verification: *confirmation by examination and provision of objective evidence that specified requirements have been fulfilled [BS7925-1]*

Validation: *confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use have been fulfilled [BS7925-1]*

Software Validation and Verification can involve analysis, reviewing, demonstrating or testing of all software developments. When implementing this model, we must be sure that everything is verified. This will include the development process and the development product itself. Verification and Validation is normally carried out at the end of the development lifecycle (after all software developing is complete). But it can also be performed much earlier on in the development lifecycle by simply using reviews.

Verification would normally involve meetings and reviews to evaluate the documents, plans, requirements and specifications.

Validation involves the actual testing. This should take place after the verification phase has been completed.

Verification and Validation, if implemented correctly can be very cost-effective if planned correctly.

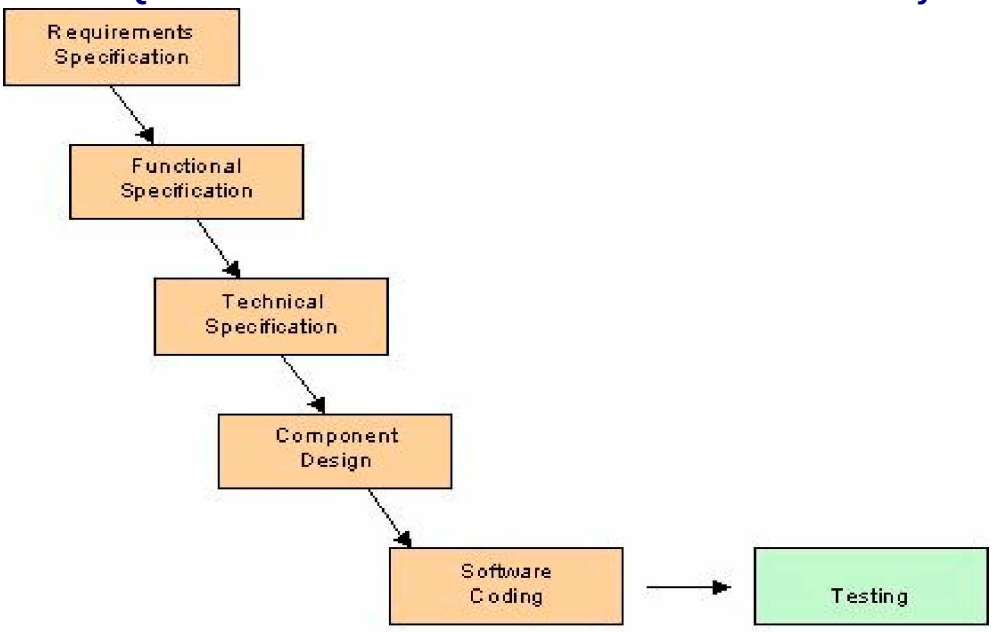
Summary:

Verification: Are we building the product right?

Validation: Are we building the right product?

Waterfall Model

The Waterfall model is also known as the 'Sequential model'. Each stage follows on from the previous one. The testing is performed in 'block' as the last stage.

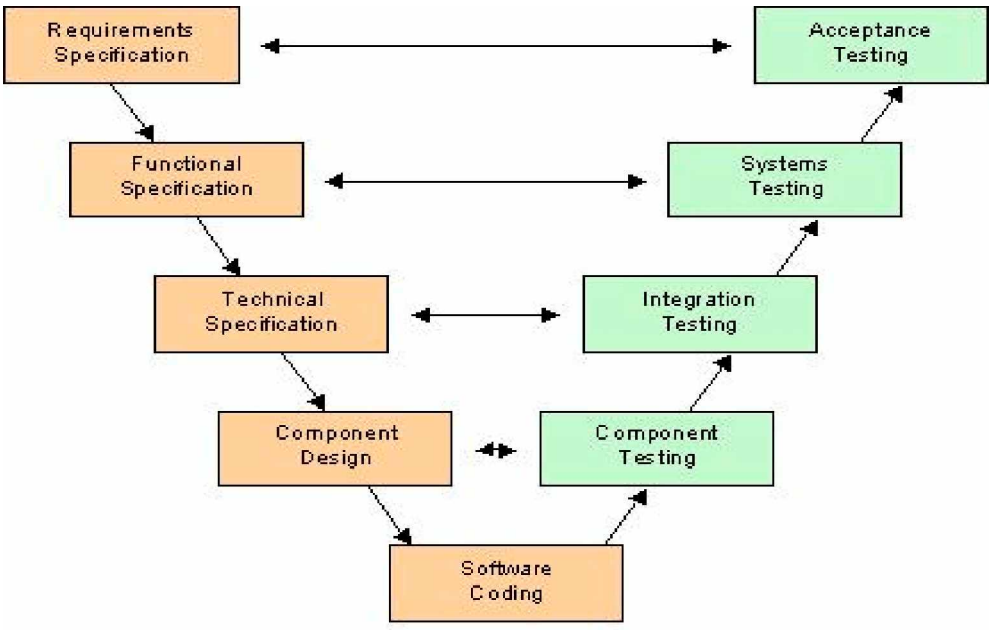


Using the above example, we can see that from a testing point of view, it is simply too little, too late. Planning or Test creation is not considered until the actual software code has been written. This can result in problems being found much later on in the project lifecycle than is desirable.

V-Model

The V-Model is an industry standard framework that shows clearly the software development lifecycle in relation to testing. It also highlights the fact that the testing is just as important as the software development itself. As you can see from the diagram below, the relationships between development and testing are clearly defined.

You will often see different names for each of the software development stages in a V-Model. You may also see more or fewer stages, as this is dependant on the individual product, and are more commonly, dependant on the individual company.



Looking at the above diagram, we can not only see the addition of the kind of testing activities that we would expect to be present. But also, we can see how each testing activity ties in with each development phase, thus verification of the design phases is included. This V-Model improves the presence of the testing activities to display a more balanced approach.

Spiral Model

The Spiral model is an incremental testing approach to both Development and Testing. This is used most effectively when the users do not know all of the requirements. From what is known, initial requirements can be defined. Then from these, the code and test cases are created. As time goes on, more details of the requirements are known and implemented in further iterations of the design, coding and testing phases. The system is considered to be complete, when enough of the iterations have taken place.

RAD

RAD represents Rapid Application Development, and is a software development process that was developed in the mid 1980's. It was developed to overcome the rigidity of such processes as 'The Waterfall Model'. Specifically the problems that existed were for example; the length of developments resulted in requirements changing before the system was complete.

Advantages

One of the advantages of RAD is increased speed. It can achieve this by the increased usage of Computer-Aided Software Engineering (CASE) tools. The CASE tools allow the capturing of requirements and converting them into usable code in a short amount of time. Another advantage is the increased overall quality. This is achieved by the involvement of the user, in the analysis and design stages.

Disadvantages

One of the disadvantages of RAD is the reduced amount of scalability. This is due to the fact that a RAD development will start at the prototype stage, proceeding on to the completed product. Another disadvantage is the reduction of features. This is because a RAD development is strictly governed by time-boxing. This results in any unfinished features to not be included in the released software, only to be deferred to future releases.

Elements

Let's now look at the six individual elements of RAD and what they mean.

Prototyping:
Creating a reduced-feature application, loosely resembling the finished product within a short period of time. This can assist the user with their requirements.

Iterative Development:
Creation of numerous versions of the applications, each one containing more functionality. Each version should produce a requirement to base the next version on.

Time-boxing:
A process of bumping features to future versions to prevent the iteration from running out of time. High standards of management are required to perform this adequately.

Team Members:
Small teams consisting of adaptable, multi-skilled and highly motivated workers are needed to perform a RAD development.

Management Approach:
Highly motivated workers depend on a good manager. The manager must prevent any obstacles, and be heavily involved with the development to ensure iterative cycles stay on track.

RAD Tools:
The latest technologies should be used for RAD sparing no expense. The focus is on the speed of the development process and not on the cost of tools. CASE tools are commonly used for RAD development

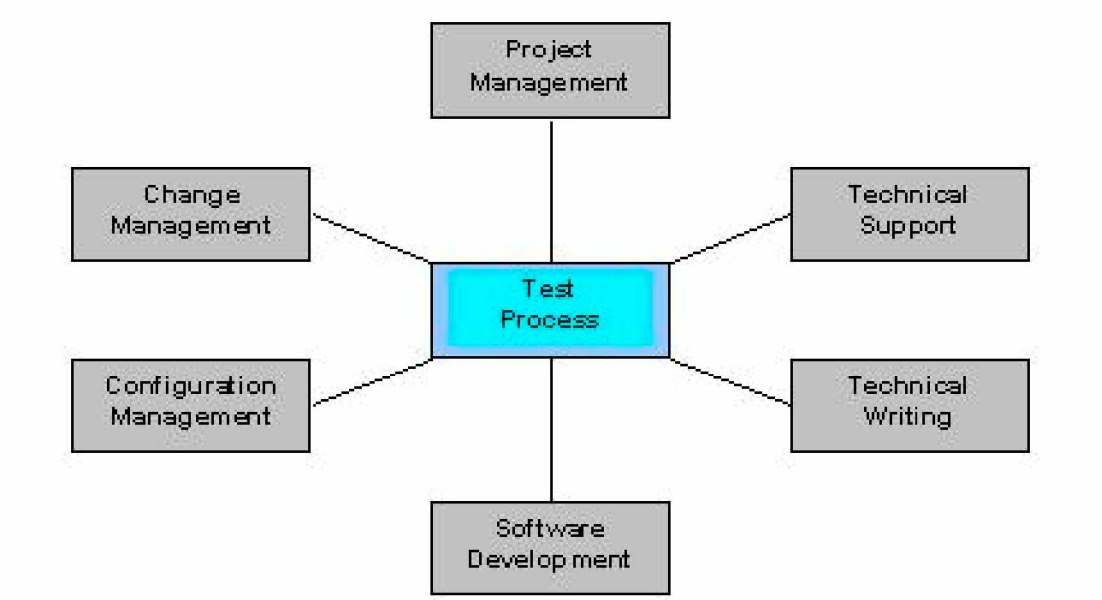
DSDM

DSDM (Dynamic Systems Development Methodology) is basically a high level framework of already proven RAD techniques, and also management controls that are used to increase the chances of successful RAD projects.

An advantage is that the high level framework allows for a process that can be easily modified for an individual projects specific needs. But, this relatively simple framework also results poor implementation due to a lack of detail.

Process Interfaces

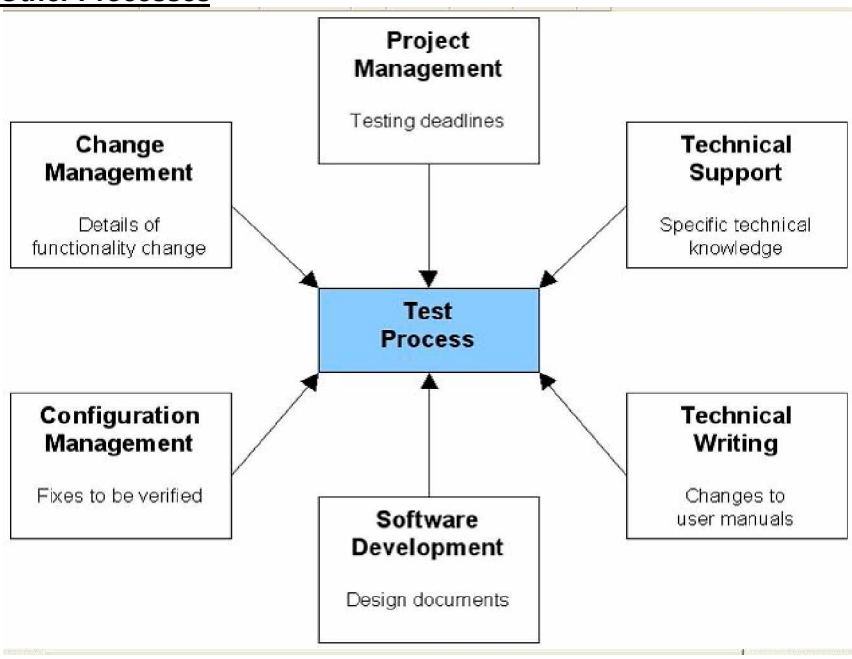
As a Tester, your focus will be fixed on the test process. But we must consider other processes that exist, and also their interaction with the test process. The following diagram illustrates other processes that may well interact with the test process.



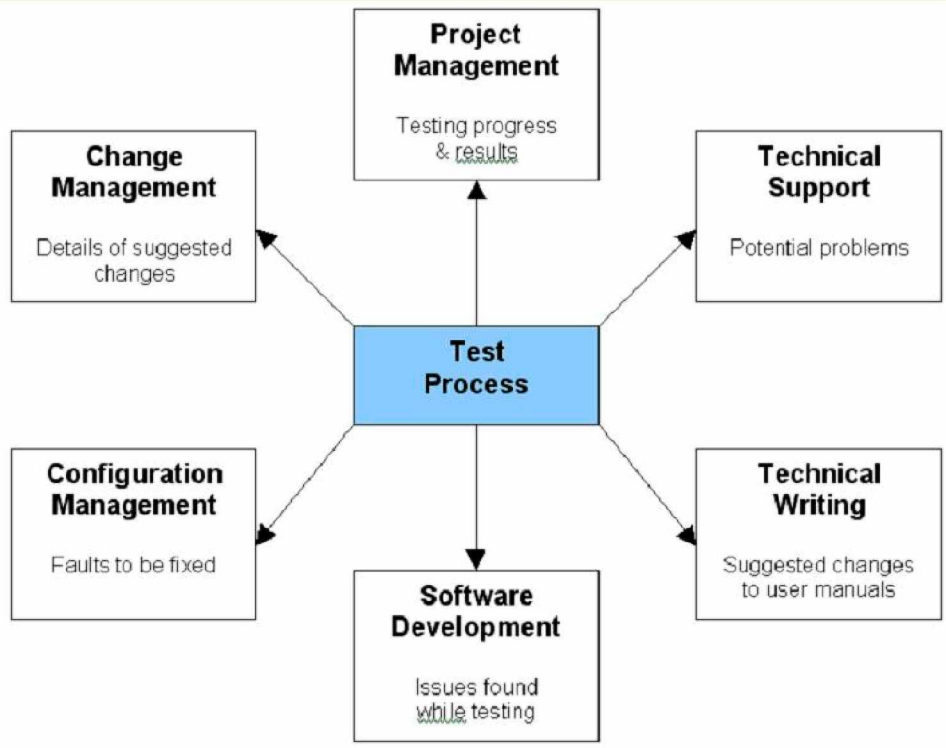
The diagrams on the following page display example information that may be obtained by interfacing with other processes. It also shows that as a Tester, useful information can not only be provided to other processes, but also obtained from them.

Some processes can directly affect the Test Process in an adverse way. For example, if an object-oriented development paradigm is used by the Software Development team. Then this could cause problems as that process, and by design, will not make various types of information visible outside of its originating process. Obvious drawbacks to this would be the creation of test cases, where specific knowledge of the internal workings of the program/system is required.

Input From Other Processes



Input To Other Processes



Component Testing

Component testing is also known as Unit, Module, or Program Testing. In simple terms, this type of testing focuses simply on testing of the individual components themselves. It is common for component testing to be carried out by the Developer of the software. This, however has a very low rating of testing independence. A better approach would be to use what we call the 'buddy system'. This simply means that two Developers test each others work, giving a high rating of independence. In order to effectively test the components, Developers often use Stubs and Drivers.

Stubs and Drivers:

If we need to test a low level module, then something called a 'driver' can be used. A 'driver' is a high level routine that will call lower level sub-programs. The way in which a driver works is to pass input data to the item under test and compare the output to the truth.

In order to test a high level module of a program, then we should use something called a 'stub'. A stub actually works by taking the place of a real routine. This saves the complexity of actually having the real routines configured, and can efficiently show whether or not the item under test is actually interfacing with the routine as expected, without having the real routine present.

Component Integration Testing

This type of Integration testing is concerned with ensuring the interactions between the software components at the module level behave as expected. Component Integration Testing is also often referred to as 'Integration Testing in the Small'.

It is commonly performed after any Component Testing has completed, and the behaviour tested may cover both functional and non-functional aspects of the integrated system.

Functional System Testing

In simple terms, 'Functional Systems Testing' focuses on what the system is actually supposed to do. So how do we know exactly what it is supposed to do? It is commonly defined in what is known as a functional requirement.

The IEEE defines functional requirement as:

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

An example of a requirement may be:

"The system must process the user input and print out a report"

Requirements-based Functional Testing:

Requirements-based Functional Testing is simply testing the functionality of the software/system based on the requirements. The tests themselves should be derived from the documented requirements and not based on the software code itself. This method of functional testing ensures that the users will be getting what they want, as the requirements document basically specifies what the user has asked for.

Business Process Functional Testing:

Different types of users may use the developed software in different ways. These ways are analysed and business scenarios are then created. User profiles are often used in Business Process Functional Testing. Remember that all of the functionality should be tested for, not just the most commonly used areas.

Non-Functional System Testing

Load Testing:

Testing the ability of the system to be able to bear loads. An example would be testing that a system could process a specified amount of transactions within a specified time period. So you are effectively

loading the system up to a high level, then ensuring it can still function correctly whilst under this heavy load.

Performance Testing:

A program/system may have requirements to meet certain levels of performance. For a program, this could be the speed of which it can process a given task. For a networking device, it could mean the throughput of network traffic rate. Often, Performance Testing is designed to be negative, i.e. try and prove that the system does not meet its required level of performance.

Stress Testing:

Stress Testing simply means putting the system under stress. The testing is not normally carried out over a long period, as this would effectively be a form of duration testing. Imagine a system was designed to process a maximum of 1000 transactions in an hour. A stress test would be seeing if the systems could first, actually cope with that many transactions in a given time period, followed by seeing how the system copes when asked to process more than 1000.

Security Testing:

A major requirement in today's software/systems is security, particularly with the internet revolution. Security testing is focused at finding loopholes in the programs security checks. A common approach is to create test cases based on known problems from a similar program, and test these against the program under test.

Useability Testing:

This is where consideration is taken into account of how the user will use the product. It is common for considerable resources to be spent on defining exactly what the customer requires and how simple it is to use the program to achieve there aims. For example; test cases could be created based on the Graphical User Interface, to see how easy it would be to use in relation to a typical customer scenario.

Storage Testing:

This type of testing may focus on the actual memory used by a program or system under certain conditions. Also disk space used by the program/system could also be a factor. These factors may actually come from a requirement, and should be approached from a negative testing point of view.

Volume Testing:

Volume Testing is a form of Systems Testing. Its primary focus is to concentrate on testing the system while subjected to heavy volumes of data. Testing should be approached from a negative point of view to show that the program/system cannot operate correctly when using the volume of data specified in the requirements.

Installability Testing:

A complicated program may also have a complicated installation process. Consideration should be made as to whether the program will be installed by a customer or an installation engineer. Customer installations commonly use some kind of automated installation program. This would obviously have to under go significant testing in itself, as an incorrect installation procedure could render the target machine/system useless.

Documentation Testing:

Documentation in today's environment can take several forms, as the documentation could be a printed document, an integral help file or even a web page. Depending on the documentation media type, some example areas to focus on could be, spelling, usability, technical accuracy etc.

Recovery Testing:

Recovery Testing is normally carried out by using test cases based on specific requirements. A system may be designed to fail under a given scenario, for example if attacked by a malicious user; the program/system may have been designed to shut down. Recovery testing should focus on how the system handles the failure and how it handles the recovery process.

System Integration Testing

This type of Integration Testing is concerned with ensuring the interactions between systems behave as expected. It is commonly performed after any Systems Testing has completed. Typically not all systems referenced in the testing are controlled by the developing organization. Some systems maybe controlled by other organizations, but interface directly with the system under test.

The greater the amount of functionality involved within a single integration phase, then the harder it will be to track down exactly what has gone wrong when a problem is found. It makes good sense to increment the amount of functionality in a structured manner. This way when a problem arises, you will already have a rough idea of where the problem may be. Try and avoid waiting for all components to be ready and integrating everything at the end. As this will normally result in defects being found late in the project, and potentially a great deal of work pin-pointing the problem, followed of course by re-development and re-testing.

Acceptance Testing

User Acceptance Testing or 'UAT' is commonly the last testing performed on the software product before its actual release. It is common for the customer to perform this type of testing, or at least be partially involved. Often, the testing environment used to perform User Acceptance Testing is based on a model of the customer's environment. This is done to try and simulate as closely as possible the way in which the software product will actually be used by the customer.

Contract and Regulation Acceptance Testing:
This type of Acceptance Testing is aimed at ensuring the acceptance criteria within the original contract have indeed been met by the developed software. Normally any acceptance criteria is defined when the contract is agreed. Regulation Acceptance Testing is performed when there exists specific regulations that must be adhered to, for example, there may be safety regulations, or legal regulations.

Operational Acceptance Testing:
This form of acceptance testing is commonly performed by a System Administrator and would normally be concerned with ensuring that functionality such as; backup/restore, maintenance, and security functionality is present and behaves as expected.

Alpha & Beta Testing

Once the developed software is stable, it is often good practice to allow representatives of the customer market to test it. Often the software will not contain all of the features expected in the final product and will commonly contain bugs, but the resulting feedback could be invaluable.

Alpha Testing should be performed at the developer's site, and predominantly performed by internal testers only. Often, other company department personnel can act as testers. The marketing or sales departments are often chosen for this purpose.

Beta Testing is commonly performed at the customer's site, and normally carried out by the customers themselves. Potential customers are often eager to trial a new product or new software version. This allows the customer to see any improvements at first hand and ascertain whether or not it satisfies their requirements, which may, or may not be a good thing. On the flip side, it gives invaluable feedback to the developer, often at little or no cost.

Test Phases

Most software developments will require the testing to be spread out over numerous phases. This can be due to reasons such as; staggered code drops, incremental releases, sheer volume of tests etc.

It is important to ascertain what each phase should provide. Some suggested items of information for a test phase are:

Objective:

Exactly what we want to achieve by executing tests from this phase.

Scope:

What specifically we are testing for, and what we are not testing.

Entry Criteria:

What is acceptable prior to any tests beginning (quality of code for example).

Exit Criteria:

What is acceptable for the testing to be completed (number of outstanding faults etc).

Test Deliverables:

What the tester will provide after testing has completed (a Test Report for example).

Test Techniques:

Functional Testing, Static Analysis and Dynamic Analysis are example test techniques.

Metrics:

Metrics may be required for reports (i.e. tests failed & tests completed percentages).

Test Tools:

Any specific tools required for completing the testing.

Testing Standards:

Any company specific standards or policies to follow. Or any industry standards to follow.

Re-testing and Regression Testing

It is imperative that when a fault is fixed it is re-tested to ensure the fault has indeed been correctly fixed.

There are many tools used in a test environment today that allow a priority to be assigned to a fault when it is initially logged. You can use this priority again when it comes to verifying a fix for a fault, particularly when it comes to deciding how much time to take over verifying the fix. For example if you are verifying that a 'typo' has been fixed in a help file, it would probably have been raised as a low priority fault. So you can quickly come to the conclusion that it would probably only take a few minutes to actually verify the fault has been fixed. If, however a high priority fault was initially raised that wiped all of the customers stored data, then you would want to make sure that sufficient time was allocated to make absolutely sure that the fault was fixed. It is important that consideration of the possible consequences of the fault not being fixed properly is considered during verification.

Another important factor when it comes to testing is when there is suspicion that the modified software could affect other areas of software functionality. For example, if there was an original fault of a field on a user input form not accepting data. Then not only should you focus on re-testing that field, you should also consider checking that other functionality on the form has not been adversely affected. This is called Regression Testing.

For example; there may be a 'sub-total' box that may use the data in the field in question for its calculation. That is just one example; the main point is not to focus specifically on the fixed item, but to also consider the effects on related areas. If you had a complete Test Specification for a software product, you may decide to completely re-run all of the test cases, but often sufficient time is not available to do this. So what you can do is 'cherry-pick' relevant test cases that cover all of the main features of the software with a view to prove existing functionality has not been adversely affected. This would effectively form a Regression Test.

Regression test cases are often combined to form a Regression Test suite. This can then be ran against any software that has undergone modification with an aim of providing confidence in the overall state of the software. Common practice is to automate Regression Tests.

To assist you on what to additionally look for when re-testing, it is always a good idea to communicate with the Developer who created the fix. They are in a good position to tell you how the fix has been implemented, and it is much easier to test something when you have an understanding of what changes have been made.

Re-test:
"Whenever a fault is detected and fixed then the software should be re-tested to ensure that the original fault has been successfully removed."

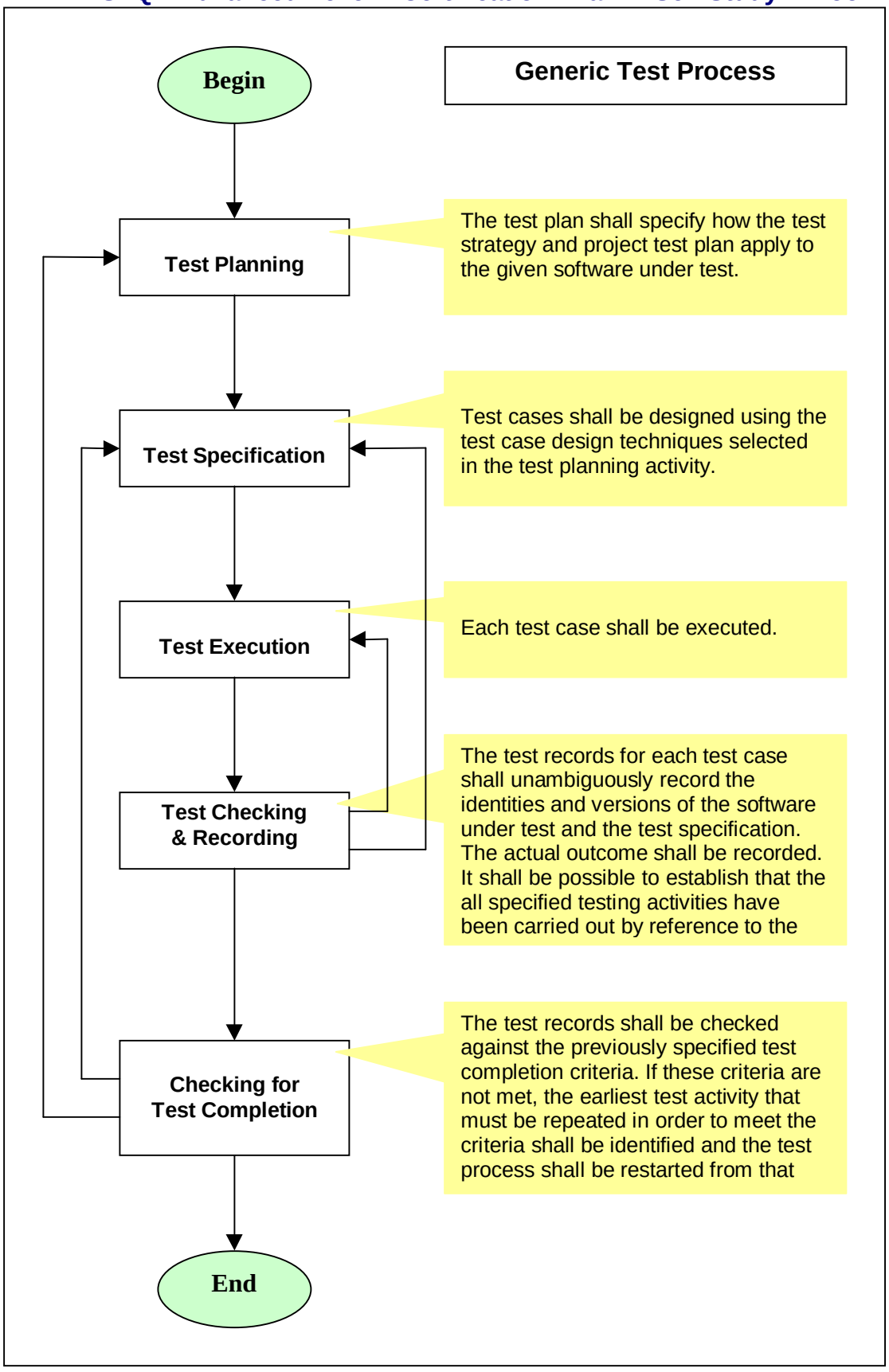
Regression Test:
"Regression testing attempts to verify that modifications have not caused unintended adverse side effects in the unchanged software (regression faults) and that the modified system still meets its requirements."

Chapter –2: Testing Process

GENERIC TEST PROCESS

In order to perform effective testing, the testing must be planned. Once the testing has been planned, it is also important to adhere to it. A common pitfall among Testers is to create a good test plan, but then not follow it correctly. We already know that it is impossible to completely test everything, but with careful planning that includes the right selection of tests and the way that they are tested, we can effectively test the software product to a high standard.

A standard test process that is commonly used exists within the BS7925-2 Standard for Software Component Testing. It is a good place to start if no process currently exists, and can be easily modified to suit an individual companies testing process, for example ‘Test Recording’ and ‘Checking for Test Completion’ are often combined into a single entity.



TEST PLANNING

We will discuss the Test Planning process in a later section. But for a brief overview of a Test Plan document:

A Test Plan should be a single document that basically contains what is going to be tested, why it is going to be tested, and how it is going to be tested. It is also important to clarify what is not going to be tested in the software product too. With regards to using a standard Test Plan layout, then we can look to the advice given by the IEEE(Institute of Electrical and Electronic Engineers) located in the International Standard IEEE Std 929-1998.

TEST SPECIFICATION

When deciding upon which test specification techniques to use, we should take into account the following points:

- The Risks to be mitigated
- Knowledge of software to be tested and project documentation
- Test specification choice should be explained in the Test Plan or Strategy

A good source of reference for definitions of test specification techniques is the BS 7925-2:1998. The design of the test specification is made of the following three components:

- Identification of test coverage items (analysis)
- Creation of test cases based on the identified items (design)
- Organization of the test environment and test procedure/script (build)

The analysis and design stages described above may also include prioritisation criteria identified within risk analysis and test planning.

A good source of test case design techniques are specified in the BS 7925-2:1998 document can be applied to the following types of testing:

- Component testing
- Component Integration Testing
- Functional system testing
- Non-functional System Testing
- System Integration Testing
- Acceptance Testing.

Remember that test cases can be generated from looking at any Business Scenarios or Use Test Cases.

When it comes to the task of actually creating the individual test cases, we should make sure that certain information is always included in them. What we are trying to achieve is to have a standard method of creating the test cases that all testers can follow. This allows any tester to be able to understand another tester's work. The following items are suggested items of information that we can include in our test cases:

Test Input:

This should specify any data received from an external source by the test object during test execution. The external source could be hardware, software or a human etc.

Expected Outcome:

Knowledge of the specification for the software under test will allow us to specify the expected outcome. If we do not have this knowledge available then an alternative source should be specified here, such as a design document for example (We should not perform the test if we do not know what

the expected outcome should be). We should also specify the final state of the software after the test case has been performed here.

Test Rational:

This should include details on test coverage i.e. what functionality will be tested by performing the test case.

Specific Test Case Pre-requisites:

Information relating to the state the test environment or software settings should be in prior to performing the test case will be detailed here.

The test specification should not only list the functional tests, but also list any non-functional tests too. These could be items such as:

- Stress
- Usability
- Maintainability
- Reliability
- Memory Management
- Security
- Recovery
- Disaster Recovery
- Volume
- Performance
- Procedure
- Configuration
- Portability
- Installability
- Interoperability
- Compatibility

Remember that techniques such as static analysis and reviews are also used to detect faults, and so should be listed in the test specification as well.

TEST EXECUTION

This phase is where the actual testing is performed. It can mean running the test cases manually or by use of an automated testing tool.

An important aspect of actually performing the test cases is to ensure a procedure is followed. This is to ensure that the tests are auditable. This will also be of benefit if any re-tests or regression tests are required, as all future tests could be ran exactly the same way if we know how the preceding tests were performed.

Often, when performing tests, a tester will realize that additional tests may be required, so it is advisable to have some system in place where a tester can suggest additional tests to be performed. A tester will often find that the developer of the software under test has a keen interest in the outcome of the test cases. This is to be expected, and also encouraged as a level of confidence in the software and also the test cases can be achieved by allowing the developer to witness some of the tests.

TEST CHECKING & RECORDING

This is where the results of each test are stored. Details such as the software versions tested and the test specifications used should also be recorded along with the actual test results.

Test Checking:

The most important part of running a test is to observe the actual outcome, and comparing it to the expected outcome (Remember that the expected outcome must be known prior to running the test case). On comparison, if we observe a difference between the actual outcome and the expected outcome, no matter how small, then we must log it.

Once a fault is found we should attempt to investigate the probable cause. This investigation will of course be dependant on the Tester's ability and also the time available for investigation. The fault could be with the software, but first check the test set-up and maybe re-run the test. There's nothing worse than finding a fault and raising it as a fault with the software, only to find that it was an error with the test set-up itself. The aim of the investigation should be to pin-point as much as possible the root cause of the problem, note any work-arounds, and note the steps to reproduce the fault (some organizations will insist that the fault is reproducible before attempting to fix it).

Test Recording:

We must ensure that every test phase that is performed should have the result logged. This should include all tests that were executed, passed, failed and even not executed. This allows measuring for test completion criteria to be achieved. The following items are suggestions of the type information that is expected to be recording when executing tests, not all of the items are required as different organisations will have different ideas.

- Date of testing
- Time to complete tests
- Test specification used
- Test specification version
- Item under test
- Software version
- Test set-up used
- Details of faults raised
- Test case numbers
- Test case titles
- Expected outcomes
- Actual outcomes
- Test cases executed
- Test cases passed
- Test cases failed
- Test cases not executed

CHECKING FOR TEST COMPLETION

This involves ensuring that the test completion criteria have been met. If this has not been achieved, then some tests may need to be re-run, or even new test cases designed. If there are legitimate reasons for the exit criteria to not have been met, then subject to management approval, this can be accepted. Changes to accommodate changes to the accepted exit criteria would require formal documentation to make apparent any associated risks and impact to the customer. A test report is often a requirement by most organizations, and will summarize the results and conclusions of the testing performed.

Chapter –3: Test Management

TEST MANAGEMENT DOCUMENTATION

As a tester you will come across all sorts of test related documentation which will inevitably vary from organization to organization. This is due to no formal world-wide recognized documentation standard (as yet). On top of that each individual organization injects there own policies, standards and approaches into the test documentation system.

The following section aims to provide an idea of what each test document listed here should ideally contain. Also, bare in mind that each of the following document types may have been merged or even split up by an organization, but still should contain the expected information.

Overview:

Test Policy	Contains organization philosophy towards testing.
Test Strategy	Contains test phases to be performed for each programme.
Project Test Plan	Contains test phases to be performed for each project.
Phase Test Plan	Contains requirements for performing tests within the phase.

Test Policy

The root of any organizations testing commonly starts with the test policy. It is designed to represent the testing philosophy of the company as a whole. It should apply to both new projects and maintenance work. Normally fairly short in length, the test policy should be a high-level document, and should contain the following items:

Definition of testing:

Example: *“ensuring the software fulfils its requirements”*

The testing process:

Example: *“all test plans are written in accordance with company policy”*

Evaluation of testing:

Example: *“effect on business of finding a fault after its release”*

Quality levels:

Example: *“no outstanding high severity faults prior to products release”*

Test process improvement approach:

Example: *“project review meetings to be held after project completion”*

Test Strategy

Based on the test policy, the test strategy is designed to give an overview of the test requirements for a programme or even organization.

Information relating to risks should be documented here, specifically the risks that will be addressed by the testing, and the specific tests that will be used against each risk. The individual items of information held within the test strategy may not be contained within a single document, and could be

labeled a company test strategy, site test strategy or project strategy etc.

Different strategies are often created for a variety of reasons, such as a children's alphabet program would probably not require strict adherence to a safety standard. Or software that is often updated may focus more on the regression and retesting areas.

The test strategy should show each testing phase which will commonly describe some, or all of the following high-level terms:

- Chosen test process
- Captured metric and measures
- Incident management approach
- Degree of test independence
- Standards and policy compliance
- Test environment details
- Entry and exit criteria
- Testing approach
- Test completion criteria
- Test case design technique
- Test automation approach
- Software reusability
- Retesting approach
- Regression testing

Project Plan

Exactly how the test strategy for a particular project will be implemented is displayed in the project plan. The project test plan will normally be referenced from the overall project plan. In relation to the test strategy, the project plan should detail items from the test strategy that it is complying with, and also items it is not complying with.

The following items of information are also commonplace within a project plan:

- Project costs, timescales for approval purposes
- Identification of test cycles
- Identification of personnel associated with the project
- Identification of project deliverables
- Confirmation of test coverage for management and customers

Phase Test Plan

The specific details of the approach taken by the testing for a specific test phase is shown in this document. It can be thought of as being based on the project plan, but with greater amounts of detail included, such as testing activities based on day to day plan, or expected amounts of man hours to complete individual tasks.

TEST PLAN DOCUMENTATION

When considering creating a project test plan or a phase test plan, we can of course reference the IEEE 829-1998 document which contains details of what types of information should be included.

TEST ESTIMATION

One of the first questions a tester will be asked when presenting their proposed test specification to a manager is "how long will it take?"

The test estimate can be presented in a number of ways, and there is no real set format. But the type of information that should be considered before presenting your figure is:

- Time for test preparation
- Time for iterations of tests
- Time for any training
- Time for test planning
- Time for executing the tests (don't forget that one!)
- Time for any retesting, fix verification
- Time when you're not available to test, holidays etc.

We can use some techniques to assist us in calculating our estimate, including:

- Previous experience
- Intuition, educated guess
- Any company estimating standards
- Formula based techniques

When we talk about formulas, we can use, amongst other things 'metrics' to provide us with a figure of how long it will take to perform certain tasks.

Example:
Take a look at previous similar testing timescales from previous projects. Then work out the average time it took to execute a test. Then simply multiply this time by the amount of test cases and iteration you have for this new project. It's not super accurate, but it may give a rough idea to start with.

The real problems come from providing a test estimate on a project that contains software of the type that has not been tested before. A suggestion here would be talk to the developers for their thoughts, talk to any other testers as they may have tested something similar. Your last resort is to take an educated guess based on your intuition and previous experience. When you find yourself in this position, remember one thing, it is much better to over-estimate than under-estimate!

SCHEDULING OF TEST PLANNING

Test early!

It's extremely important to be involved in the project lifecycle as early as possible. This is because as a Tester you can see any potential problems that may affect your future testing. For example, you discover that a requirement is un-testable. If you are involved early enough, you might be able to influence a change of requirement. You may find that you have a high priority test on functionality that may not exist until later in the development. This information could be used to influence the order of developing certain components in order to help with your planned testing.

Not only can the tester benefit from being involved early, but also other members of the development team can benefit from your early test input too. You would probably be able to organize your testing time better if you could have input as to 'when' certain functionality should be made available. Therefore, possibly reduce your overall testing time resulting in a happy project manager. You may also find that part of the testing to be carried out by a developer overlaps with what you as a Tester had planned. You may be able to off-load some of the work from the developer in this situation, or furthermore you may have time available to assist with their testing.

A useful tip when thinking about releasing a test plan is to release it as soon as possible. Don't wait for all of the information if it means delaying the release of the test plan. The test plan is an important piece of information (even if told otherwise!) which holds information that can impact the completion of the project. There are numerous bodies in the organization that will reference the test plan, so it makes good sense to get it out there ASAP, especially as more often than not, you will want feedback as soon as possible in order to make any alterations. You can always fill in the gaps at a later date as information becomes available.

TEST PROGRESS MONITORING AND CONTROL

Test Progress Monitoring

Once the testing has started, from a tester's point of view all activity will be focused on the actual testing. In a typical scenario, as time goes on, some tests have been completed, whilst others remain to be completed. Then the Project Manager asks the Test Team "what state is the software in?"

In order to properly answer the Project Manager's question, we need some way of monitoring the undergoing tests. There is no 'one' perfect way to document this, as every company will probably have their own way of doing things. But here are some suggested items to document during a test phase:

- Number of Tests Passed
- Number of Tests Failed
- Number of Tests Run
- Number of Tests Remaining

This information is commonly stored on a results sheet for the Test Specification being performed. These details should be updated as much as possible during the testing. This way an accurate picture of the testing can be obtained at any time.

It is a good idea to store the information in a place where other interested parties can view it. This is a step towards more of a greater flow of information. This is also where a Test Matrix can be used to not only store a list of the Test Specifications that will be ran, but also the results, including statistics obtained from the above list of items combined from each Test Specification. For example, if someone wants an idea as to the quality of code or test progress, then they can simply view the Test Matrix for an overall picture. If they are interested in specific test cases, then they can view the individual results sheet for the Test Specification in question.

- Faults Found
- Faults Fixed & Verified

'Faults Found' will be documented by the Testers, but 'Faults Fixed & Verified' details will commonly be controlled by the Development Team or Project Manager.

Often the Tester or Test Manager will have to report to the Project Manager any deviations from the Project/Test Plans. This could be for reasons such as; running out of time to test the product. If the details of the above specified items are readily available, then it will make monitoring the test process an easier task. It is also a good idea to have progress meetings with the Project Manager to ensure not only sufficient test progress is being made, but feedback as to the quality of code is also made.

Test Control

The Test Manager would have detailed in the Test Plan some form of 'Exit Criteria' in relation to the tests being ran for a particular phase, or project testing in its entirety. This will often include an acceptable percentage of test cases that have been completed. If during the project, time is running out and there is a risk of not achieving this target, then the test Manager in association with the Project Manager may have to take action.

Examples of the type of action that occur:

- Changing the schedule to allow more time

- Allocate more Testers
- Reduce test coverage on low priority test cases

In order for any of the above actions to be carried out, it is imperative that any deviation from the Test Plan or potential risks to the successful completion of the testing is highlighted as soon as possible.

Chapter –4: Testing & Risk

INTRODUCTION TO TESTING AND RISK

Risk and Testing

When we talk about risk in relation to testing, what we mean is the chances of something happening, and the effect that it might have when it does happen. We can define different levels of risk by either the likelihood of it occurring or the severity of the impact if it does occur.

Some risks can be based on the project, such as staff shortages, contractual issues etc. It is a good idea to include project risks within the Test Plan. This allows the risks to be seen by any interested parties involved with the testing on the project.

When referring to product associated risks, we are talking about the risk to the quality of the product.

Consider the risk involved of releasing Air Traffic Control software. If that product fails the results could be disastrous, and possibly cause loss of life. Or the risk that a business accounting software package does not do what it was designed to do. Both of these potential risks may affect the end user and the developing company, so nobody wins!

What if we know the risks? We can use the known risks to decide where to start testing and where to concentrate more test effort on. What we could do is prioritise our test cases with a view to test the high risk areas first, and use our remaining testing time to concentrate on testing low risk areas. We could also consider using different types of testing to target different types of risks.

A way to reduce the risks associated with a product is to test it. This may produce faults that subsequently can be fixed prior to release of the product. But what about project related risks? Well, these risks can be reduced by an effective test strategy.

RISK MANAGEMENT

Risk management comprises of the following three components:

- Risk Identification
- Risk Analysis
- Risk Mitigation

Risk management should be a consideration for everyone involved in the project. Let's now look at each risk management activity.

Risk Identification

The following techniques can all be used to identify risks associated with products and projects. The list is by no means rigid, as many organisations will have there own techniques.

- Expert Interviews
- Independent Assessment
- Risk Templates
- Lessons Learned
- Risk Workshops
- Brainstorming and Checklists

Risk Analysis

Many people involved with software development run and hide when they here the words 'Risk Analysis', as they think it is just another management technique that will get in the way of them doing their job. But if they took the time to think about risk analysis they would realise that it is a concept

that can benefit 'everyone' involved with the development. So what does the term 'Risk Analysis' actually mean? It is simply 'studying the identified risks'!

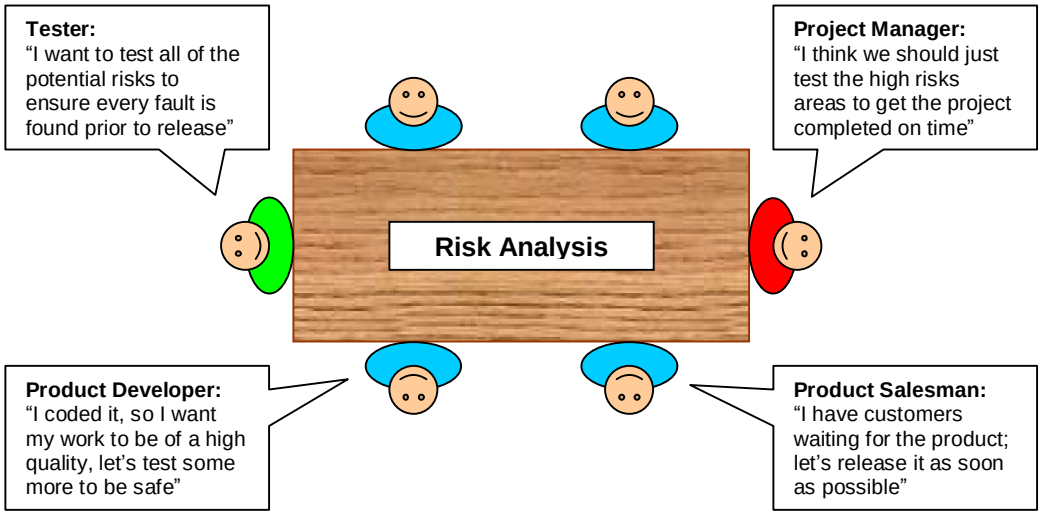
A simple formula can be used to calculate risk:

Frequency (likelihood) X Severity (impact)

By using the above formula we can produce a figure, otherwise known as the 'exposure'.

Although, quite often we find it difficult to assign numerical values to either, 'frequency' or 'severity'. So that leaves us with another way to analyse the risk, which is by using classes and categories to define the levels of risk. Which ever method we use, it is imperative that we use a trusted method. Otherwise any conclusions will be based on a 'perceived' probability and consequence. In other words, any value or class assigned to studied risk will be based on an opinion or guess. Tried and tested metrics for evaluating risks are definitely a 'first choice'. Don't gamble unless you have too!

Remember that the majority of risk analysis is going to be based on personal perceptions. This results (more often than not) in a difference of opinion. The opinion differences are normally based on the persons viewpoint from their role within the organization. So, when taking part in a risk analysis, bare in mind what each participant wants out of the risk analysis. For example, the Project Manager would be keen to get the product completed before the deadline, and not want to plan for additional testing time. Whereas a Tester or Developer would prefer that all possible risks would have been eliminated, as if there are future problems it could look badly on their work.



When assessing the risks of a software product, we can split the risks into two logical groups, which are; non-functional risks, such as security, useability and performance related risks, and functional risks, such as a specific functional requirement, or a particular components functionality.

Risk analysis should not be a one-off item to ensure a tick in the box for a test process. It must be considered as an ongoing process. You may find that the expected amount of faults found in a given area of functionality may have been underestimated. This would prompt for increasing the risk associated with this functionality. A procedure should be in place to allow the modification of risk statuses and levels associated with the development. On the flip-side of that, there may be significantly less faults found in a given area of functionality. Therefore, you may be able to spend more time testing another high risk area of functionality. Again, a process should be in place in order for the tester to be able to raise this issue and effectively act upon it.

Risk Mitigation

Risk mitigation is simply the response to the analysed risks. A choice must be made as what action should be carried out once a risk has been identified. Some possible choices could be:

- Do nothing
- Take preventative action (test it)
- Contingency plan (what we should do if the predicted fault actually occurs)

If the action is to 'test', then we need to determine the level of testing and the type of testing to be performed.

Level of testing:
The level of testing is normally determined from the level of risk. A common approach is to test everything that has a high level of risk, and then reduce the amount of testing in relation to the decreasing level of risk.

Type of testing:
If there is a risk of errors in the graphical user interface, then the testing type could be 'usability testing' for instance. If the risk is of the software 'freezing' when performing transactions in quick succession, then we could implement 'load testing'.

Not only should we consider testing all of the high risk areas, we should also consider the order in which we test them. Chances are, testing a high risk area may show lots of faults. This would mean time would have to be spent investigating the faults, fixing them, and verifying the fixes. This could impact the testing deadline, and so careful consideration should be made as to which high risk areas to test first. You may decide on testing the high risk areas that will be quickest to test, or the high risk area that you feel will show up the most faults. Both are valid approaches and must be used based on an individual projects circumstances. You may also find that low risk areas may not be tested at all. This is quite common in many development projects, and so highlights the importance of the risk analysis process itself.

Chapter –5: Testing Techniques

FUNCTONAL/STRUCTURAL TESTING TECHNIQUES

Equivalence Partitioning

Imagine you were assigned the task of manually testing a software program that processed product orders. The following information is an extract of what is to be processed by the program:

Order Number:	0 – 1000
Product code:	A1 to A99, B1 to B99, C1 to C99

If you decided you wanted to test this thoroughly, it would take absolutely ages. As you would need to enter every number between 0 and 1000 for the order number. You would also need to enter every possible item in the product code ranges. Imagine the combinations of all of the possible entries…….millions?

An alternative to the obviously unachievable goal of testing every possible value is Equivalence Partitioning. What this method allows you to do is effectively partition the possible program inputs. For each of the above input fields, it should not matter which values are entered as long as they are within the correct range and of the correct type. This is because it should be handled they same way by the program (*This would obviously need to be clarified by the developer as it is an important assumption).* For example, you could test the order number by choosing just a few random values in the specified range. For the product code this time choose a small selection from each equivalent partition e.g. A66, A91, B17, B59, C5, C77 etc.

So the point of equivalence partitioning is to reduce the amount of testing by choosing a small selection of the possible values to be tested, as the program will handle them in the same way.

Boundary Value Analysis

By the use of equivalence partitioning, a tester can perform effective testing without testing every possible value. This method can be enhanced further by another method called ‘Boundary Value Analysis’. After time, an experienced Tester will often realise that problems can occur at the boundaries of the input and output spaces. When testing only a small amount of possible values, the minimum and maximum possible values should be amongst the first items to be tested.

Using our previous example from ‘Equivalence Partitioning’. Let’s now enhance the testing by adding some ‘Boundary Value Analysis’ based testing.

Order Number:	0 – 1000
Product Code:	A1 to A99, B1 to B99, C1 to C99

For the order number, we would test ‘0’ as the minimum value, and ‘1000’ as the maximum value. So, those values are what the software program would expect. But let’s approach this in a more negative way. Let’s add tests that are effectively out of range, i.e. ‘-1’ and ‘1001’. This gives us confidence that that the range is clearly defined and functioning correctly.

To complete the testing scenario, we can add tests for the product code, i.e. ‘A0’, A100, and so on.

Classification tree method

The classification tree method is also known as a decision tree method and the terms can be used interchangeably as they mean the same thing. It is a good choice especially when the goal is to generate rules that can be easily understood and translated into a query language such as SQL.

Classification / decision tree learning is a popular method used in ‘*Data Mining’. Here, a decision tree describes a tree structure where the tree’s leaves represent classifications, and branches represent conjunctions of features that lead to those particular classifications.

A decision tree can be learned by splitting the source set into subsets based on an attribute value test. This process is repeated on each subset in a recursive manner. The recursion is completed when splitting is either not possible, or a single classification can be applied to each element of the subset.

*Data Mining - known as Knowledge-Discovery in Databases (KDD), it is the process of automatically searching large volumes of data for patterns.

State Transition Testing

This type of Black-box testing is based on the concept of 'states' and 'finite-states', and is based on the Tester being able to view the software's states, transition between states, and what will trigger a state change. Test cases can then be designed to execute the state changes.

Requirements-based Functional Testing

Requirements-based Testing is simply testing the functionality of the software/system based on the requirements. The tests themselves should be derived from the documented requirements and not based on the software code itself. This method of functional testing ensures that the users will be getting what they want, as the requirements document basically specifies what the user has asked for.

Code Coverage

Code coverage is simply how thoroughly your test cases exercise the actual software code. We can use code coverage to give ourselves a simple form of quality measurement. The quality is effectively the quality of the testing and not the code itself. Code coverage is considered to be a 'white-box' testing technique as the focus is aimed at the internal workings of the software i.e. the code itself.

Code coverage is particularly useful in identifying the paths in a program that are not being tested. It is best tested when the program contains large amounts of decision points.

Statement Testing

This testing method involves using a model of the source code which identifies statements. These statements are either categorized as being either 'executable' or 'non-executable'. In order to use this method, the input to each component must be identified. Also, each test case must be able to identify each individual statement. Lastly, the expected outcome of each test case must be clearly defined.

A 'statement' should be executed completely or not at all. For instance:

IF *x* THEN *y* ENDIF

The above example is considered to be more than one statement because *'y'* may or may not be executed depending upon the condition *'x'*

Code example:

```
Read a
Read b
IF a+b > 100 THEN
    Print "Combined values are large"
ENDIF
IF a > 75 THEN
    Print "Input 'a' is large"
ENDIF
```

It is often useful to work out how many test cases will be required to fully test a program for statement

coverage. Using the above example we can determine that we need only one test case to fully test for 'statement' coverage.

The test would be to set variable 'a' to a value of '76' and variable 'b' to '25' for example. This would then exercise both of the 'IF' statements within the program.

To calculate statement coverage of a program we can use the following formula:

Statement Coverage = Number of executable statements executed
*Total number of executable statements * 100 %*

Branch Decision Testing

This test method uses a model of the source code which identifies individual decisions, and their outcomes. A 'decision' is defined as being an executable statement containing its own logic. This logic may also have the capability to transfer control to another statement. Each test case is designed to exercise the decision outcomes. In order to use this method, the input to each component must be identified. Also, each test case must be able to identify each individual decision. Lastly, the expected outcome of each test case must be clearly defined.

Branch coverage measures the number of executed branches. A branch is an outcome of a decision, so an 'IF' statement, for example, has two branches, which are 'True' and 'False'. Remember though; that code that has 100% branch coverage may still contain errors.

To calculate the decision coverage of a program we can use the following formula:

*Decision Coverage = number of executed decision outcomes * 100%*
total number of decision outcomes

Code example:

```
Read a
Read b
IF a+b > 100 THEN
    Print "Combined values are large"
ENDIF
IF a > 75 THEN
    Print "Input 'a' is large"
ENDIF
```

Using the above example again, we can determine that we need only two test cases to fully test for 'branch' coverage.

The first test again could be to set variable 'a' to a value of '76' and variable 'b' to '25' for example. This would then exercise both of the 'True' outcomes of the 'IF' statements within the program.

A second test would be to set variable 'a' to a value of '70' and variable 'b' to '25' for example. This would then exercise both of the 'False' outcomes of the 'IF' statements within the program.

Branch Condition Testing

Branch Condition Testing uses a model of the source code, and identifies decisions based on *individual* Boolean operands within each decision condition. A 'decision' is defined as being an executable statement containing its own logic. This logic may also have the capability to transfer control to another statement. The decision condition is a Boolean expression which is evaluated to determine the outcome of the decision. An example of a decision would be a 'loop' in a program. In order to use this method, the input to each component must be identified. Also, each test case must

be able to identify each individual Boolean operand along with its value. Lastly, the expected outcome of each test case must be clearly defined.

To calculate the branch condition coverage of a program we can use the following formula:

*Branch Condition Coverage = number of Boolean operand values executed * 100%*
total number of Boolean operand values

Branch Condition Combination Testing

Branch Condition Combination Testing uses a model of the source code, and identifies decisions based on *combinations* of Boolean operands within decision conditions. A 'decision' is defined as being an executable statement containing its own logic. This logic may also have the capability to transfer control to another statement. The decision condition is a Boolean expression which is evaluated to determine the outcome of the decision. An example of a decision would be a 'loop' in a program. In order to use this method, the input to each component must be identified. Also, each test case must be able to identify each individual Boolean operand along with its value. Lastly, the expected outcome of each test case must be clearly defined.

To calculate the branch condition combination coverage of a program we can use the following formula:

Branch Condition Combination Coverage =

*number of Boolean operand value combinations executed * 100%*
total number of Boolean operand value combinations

NON-FUNCTIONAL TESTING TECHNIQUES

While Functional Testing concentrates on what the system does, Non-functional Testing concentrates on how the system works. This form of testing can be performed at all stages of the development and include objectives such as response times for example.

Example:

A software program designed to calculate mortgage repayments.

- Test that the program does not crash
- Test that the program can process results within a set time period
- Test that the program can be upgraded

Useability Testing

This is where consideration is taken into account of how the user will use the product. It is common for considerable resources to be spent on defining exactly what the customer requires and how simple it is to use the program to achieve their aims. For example; test cases could be created based on the Graphical User Interface, to see how easy it would be to use in relation to a typical customer scenario.

Volume Testing

Volume Testing is a form of Systems Testing. Its primary focus is to concentrate on testing the systems while subjected to heavy volumes of data. Testing should be approached from a negative point of view to show that the program/system cannot operate correctly when using the volume of data specified in the requirements.

Performance Testing

A program/system may have requirements to meet certain levels of performance. For a program, this could be the speed of which it can process a given task. For a networking device, it could mean the throughput of network traffic rate. Often, Performance Testing is designed to be negative, i.e. prove that the system does not meet its required level of performance.

Stress Testing

Stress Testing simply means putting the system under stress. The testing is not normally carried out over a long period, as this would effectively be a form of duration testing. Imagine a system was designed to process a maximum of 1000 transactions in an hour. A stress test would be seeing if the systems could actually cope with that many transactions in a given time period. A useful test in this case would be to see how the system copes when asked to process more than 1000.

DYNAMIC ANALYSIS

Dynamic analysis is a testing method that can provide information on the state of software. It can achieve this dynamically i.e. it provides information when the software is actually running. It is commonly used to exercise parts of the program that use memory resources e.g.:

- Memory allocation
- Memory usage
- Memory de-allocation
- Memory leaks
- Unassigned pointers

STATIC ANALYSIS

Static Analysis is a set of methods designed to analyse software code in an effort to establish if it is correct, prior to actually running the software. As we already know, the earlier we find a fault the cheaper it is to fix. So by using ‘Static Analysis’, we can effectively test the program even before it has been written. This would obviously only find a limited number of problems, but at least it is something that can be done very early on in the development lifecycle.

Some advantages of using Static Analysis are:

- Finding defects before any tests are even run
- Early warning of unsatisfactory code design
- Finding dependency issues, such as bad links etc.

The types of errors that can be detected by Static Analysis are:

- Unreachable code
- Uncalled functions
- Undeclared variables
- Programming standard violations
- Syntax errors

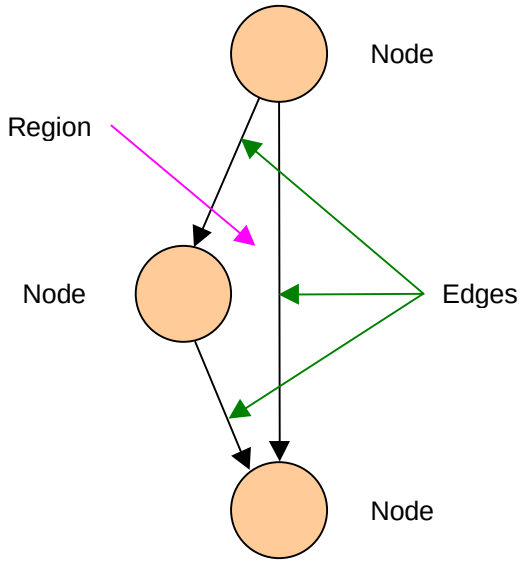
Static Analysis is commonly performed by automatic processes. A ‘Compiler’ for example is said to perform Static Analysis when it detects problems. An example of this would be syntax errors.

Control Flow Graphing

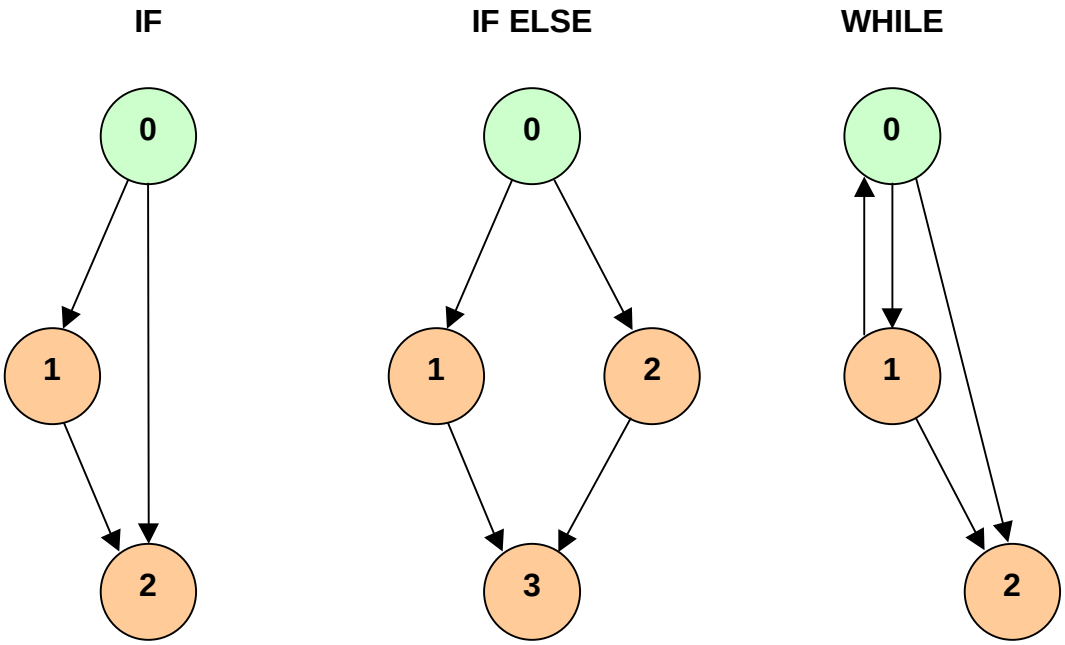
Control flow graphs display the logic structure of software. The flow of logic through the program is charted. It is normally used only by Developers as it is a very low level form testing, often used in Component Testing.

It can be used to determine the number of test cases required to test the programs logic. It can also provide confidence that the detail of the logic in the code has been checked.

Control Flow Graph Display



Here are some examples of some common control structures:



Cyclomatic Complexity

Cyclomatic Complexity is a software metric that is used to measure the complexity of a software program. Once we know how complex the program is, we then know how easy it will be to test.

The actual complexity is computed from a graph describing the control flow of the software program. The formula for calculating the Cyclomatic Complexity is:

C = E – N + P

C = Cyclomatic Complexity
E = number of edges
N = number of nodes
P = number of components

This software metric does have its limitations, for example there is no way of determining the difference between the complexities of the individual decisions within the code.

You will probably come across variations of the above formula, but they will pretty much produce the same end result.

When looking at a piece of code, the Cyclomatic Complexity is the number of paths through it. For example, if there were no decisions in the code (i.e. no 'If statements' or 'loops'). Then the Cyclomatic Complexity would be 'one', because it has a single path. If the piece of code had a single 'If statement' for example, then the Cyclomatic Complexity would be 'two', one for each path. A count of five or less means that the component module is fairly simple. A count of between six and ten means that you should consider simplifying it. It is thought that a component module containing a Cyclomatic Complexity count greater than ten is prone to errors.

For unit testing, the count can be very useful, as it gives you the minimum number of tests you need to fully exercise the code. That means to achieve 100% code coverage!

Lines of Code

The most basic form of a complexity metric is the 'Lines of Code' metric, or 'LOC' metric. Its purpose like other complexity metrics is to estimate the amount of effort that will be required not only to develop such a program, but also assist in estimating how much effort will be required to test it.

In its simplest form we could use the LOC metric by literally counting the number of lines of code in the program. But this would give us a very inaccurate measurement, as this result would also include blank lines, lines containing only comments, lines containing only declarations. It doesn't really give us a feel for how complex the code is, and therefore cannot help us determining the amount of effort required to test it.

So we could further enhance the metric by specifying that lines containing comments should not be counted, and so on, and so on. Eventually we should end up with a more meaningful set of results by tailoring the LOC metric to suit the way in which a program has been written.

Data-flow Analysis

The idea behind Data-flow Analysis is to work-out the dependencies between items of data that are used by a program. When a program is ran, it rarely runs in a sequential order i.e. starting at line 1 and finishing at line 100. What usually happens is that the dependencies of the data within the program will determine the order. Consider the following example lines of code:

```
11    A = Y + Z
12    Y = 10 + 10
13    Z = 20 + 20
```

If we performed a Data-flow analysis on the above example we would see that line '11' could not be run first, because we do not know the values of 'Y' and 'Z' as yet. What would need to happen is lines '12' and '13' would need to be ran first in order for the program to successfully run.

Data-flow Analysis is often used to find 'definitions' that have no intervening 'use'. For example, consider the following sample of code:

```
23    START
24    B = 4 + 3
25    C = 7
26    D = C + 6
27    PRINT D
28    STOP
```

What the example is trying to show is that 'B' is never used. So it is a pointless piece of code.

Data-flow analysis is also used to detect variables that are 'used' after it has effectively been 'killed', for example:

```
102   START LOOP
103   GET M
104   GET P
105   Q = M + P
106   PRINT Q
107   M = 0
108   P = 0
109   Q = 0
110   PRINT P
111   END LOOP
```

The above example shows the 'use' of variable 'P' after it have effectively been initialised for the loops next iteration (i.e. killed). It will always print a zero. The line containing the print command is either not needed or in the wrong place.

NON-SYSTEMATIC TESTING TECHNIQUES

Error Guessing

Why can one Tester find more errors than another Tester in the same piece of software?

More often than not this is down to a technique called 'Error Guessing'. To be successful at Error Guessing, a certain level of knowledge and experience is required. A Tester can then make an educated guess at where potential problems may arise. This could be based on the Testers experience with a previous iteration of the software, or just a level of knowledge in that area of technology.

This test case design technique can be very effective at pin-pointing potential problem areas in software. It is often be used by creating a list of potential problem areas/scenarios, then producing a set of test cases from it. This approach can often find errors that would otherwise be missed by a more structured testing approach.

An example of how to use the 'Error Guessing' method would be to imagine you had a software program that accepted a ten digit customer code. The software was designed to only accept numerical data.

Here are some example test case ideas that could be considered as Error Guessing:

- Input of a blank entry
- Input of greater than ten digits
- Input of mixture of numbers and letters
- Input of identical customer codes

What we are effectively trying to do when designing Error Guessing test cases, is to think about what could have been missed during the software design.

This testing approach should only be used to compliment an existing formal test method, and should not be used on its own, as it cannot be considered a complete form of testing software.

Exploratory Testing

This type of testing is normally governed by time. It consists of using tests based on a test chapter that contains test objectives. It is most effective when there are little or no specifications available. It should only really be used to assist with, or compliment a more formal approach. It can basically ensure that major functionality is working as expected without fully testing it.

Ad-hoc Testing

This type of testing is considered to be the most informal, and by many it is considered to be the least effective. Ad-hoc testing is simply making up the tests as you go along. Often, it is used when there is only a very small amount of time to test something. A common mistake to make with Ad-hoc testing is not documenting the tests performed and the test results. Even if this information is included, more often than not additional information is not logged such as, software versions, dates, test environment details etc.

Ad-hoc testing should only be used as a last resort, but if careful consideration is given to its usage then it can prove to be beneficial. If you have a very small window in which to test something, then the following are points to consider:

- Take some time to think about what you want to achieve
- Prioritise functional areas to test if under a strict amount of testing time
- Allocate time to each functional area when you want to test the whole item
- Log as much detail as possible about the item under test and its environment
- Log as much as possible about the tests and the results

Random Testing

A Tester normally selects test input data from what is termed an 'input domain' in a structured manner. Random Testing is simply when the Tester selects data from the input domain 'randomly'. In order for random testing to be effective, there are some important open questions to be considered:

- Is random data sufficient to prove the module meets its specification when tested?
- Should random data only come from within the 'input domain'?
- How many values should be tested?

As you can tell, there is little structure involved in 'Random Testing'. In order to avoid dealing with the above questions, a more structured Black-box Test Design could be implemented instead. However, using a random approach could save valuable time and resources if used in the right circumstances. There has been much debate over the effectiveness of using random testing techniques over some of the more structured techniques. Most experts agree that using random test data provides little chance of producing an effective test.

There are many tools available today that are capable of selecting random test data from a specified data value range. This approach is especially useful when it comes to tests associated at the system level. You often find in the real world that 'Random Testing' is used in association with other structured techniques to provide a compromise between targeted testing and testing everything.

CHOOSING TEST TECHNIQUES

Careful consideration should be taken when it comes to choosing a test technique, as the wrong decision could result in a meaningless set of results, undiscovered critical faults etc. We can make a judgement on which techniques to use if we have tested a similar product before. That's were the importance of good test documentation comes in, as we could quickly ascertain the right techniques to use based what was the most productive in previous testing projects.

If we are testing something new, then the following list contains points to consider when choosing a technique:

Are there any regulatory standards involved?
Is there a level of risk involved?
What is the test objective?
What documentation is available to us?
What is the Testers level of knowledge?
How much time is available?
Do we have any previous experience testing a similar product?
Are there any customer requirements involved?

The above are a sample of the type of questions you should be asking and are in no particular order of importance. Some of them can be applied only to certain testing situations, while some of them can be applied to every situation.

When we think about standards in relation to testing, then they can be thought of as three separate categories.

Quality Assurance Standards:

A Quality Assurance (QA) standard simply specifies that testing should be performed.

Example: ISO 9000

Industry Specific Standards:

An industry specific standard will detail exactly what level of testing is to be performed.

Examples:

- *Railway Signalling standard*
- *DO-178B*

- *Nuclear Industry standard*
- *MISRA guidelines for motor vehicle software*
- *Pharmaceutical standards*

Testing Standards:

Testing standards will detail how to perform the testing. Ideally, a testing standard should be referenced from a QA or Industry specific standard.

Example: BS7925-1, BS7925-2

Chapter –6: Reviews

INTRODUCTION TO REVIEWS

So what actually is a review?

Review: A process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users or other interested parties for comment or approval. [IEEE]

It is important to remember that any document can be reviewed. For example requirement specifications, design documents, test specifications etc. can, and are reviewed. It is common knowledge that reviews are cost effective. The actual cost of an on-going review process is considered to be approximately 15% of the development budget. This may at first sound quite considerable, but compared to not performing reviews and the associated risk of producing products containing errors, it is obvious of the advantages that a review process can bring. These advantages include development productivity improvements, reduced amounts of product development time, and above all, a reduction in the amount of faults.

Reviews commonly find errors that are not possible to be detected by regular testing. Reviews also provide a form of training, including technical and standards related, for every participant.

From a testing point of view, we can use reviews to allow ourselves to be involved much earlier on in the development lifecycle. Obviously at the beginning of the project there is nothing we can physically test. But what we can do is be involved in the review process of various documents. For example, we could get involved in the review process of the product requirements. From our involvement at this very early stage of the development lifecycle, we would have an idea of what will be required to be tested. This would give us a head-start on thinking about how we might approach testing the requirements too.

Goals:
The goals of a review are primarily to find faults, also to validate and verify the item under review against specifications and standards, with an aim to achieve consensus.

Pitfalls:
The pitfalls of a review could be a lack of training, insufficient documentation, and a lack of support from Management.

The IEEE has a standard for Software Reviews named IEEE 1028-1997.

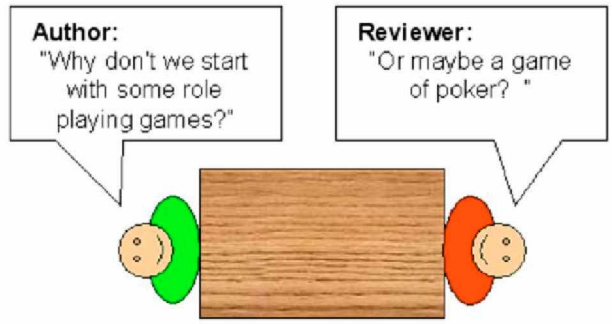
Guidelines for a Successful Review

The following cartoons are obviously light-hearted but emphasise some important points worth remembering:

Have a reasonable objective for the review

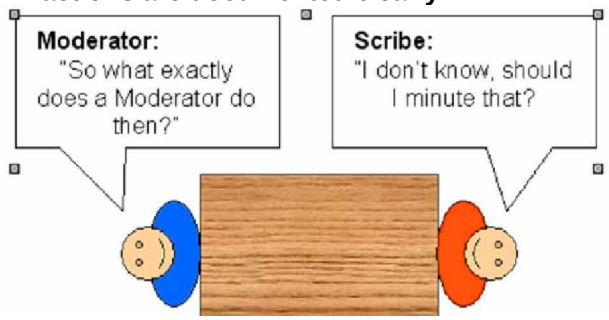
Ensure you invite the right people

Views should be expressed positively to the Author



Apply tried and tested review techniques

All actions are documented clearly



THE PRINCIPLES OF REVIEWS

Ideally, a review should be performed when all of the supporting documentation is available. This can include design documents, requirements documents, standards documents, basically any documentation that has either been influential or is applicable to the document to be reviewed.

The outcome of a review will normally come under one of the following:

- There are no actions to be performed *(very rare!)*
- Some actions to be completed, but no need for additional review
- Major issues with document, actions to be completed before additional review

Review Roles

Let's start by looking at the roles and the responsibilities of the participants of the reviews. Organisations will commonly have different named roles than those listed below, but this will give you an idea of a commonly used set of roles used throughout the world.

Manager:

The Manager will be the person who makes the decision to hold the review. Managing people's time with respect to the review is also a Managers responsibility.

Moderator:

The Moderator effectively has overall control and responsibility of the review. They will schedule the review, control the review, and ensure any actions from the review are carried out successfully. Training may be required in order to carry out the role of Moderator successfully.

Author:

The Author is the person who has created the item to be reviewed. The Author may also be asked questions in the review.

Reviewer:

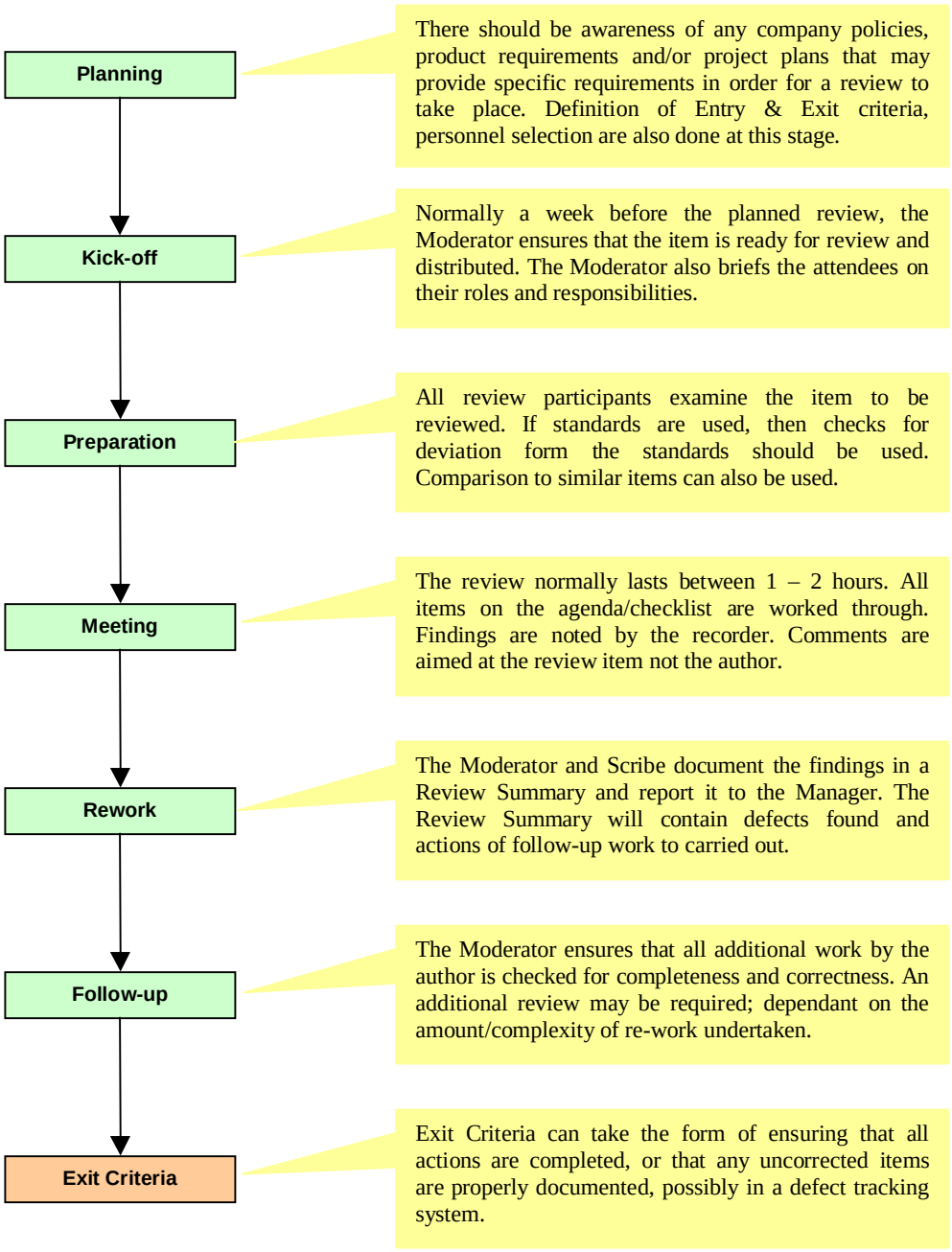
The reviewers are the attendees of the review who are attempting to find errors in the item under review. They should come from different perspectives in order to provide a well balanced review of the item.

Scribe:

The Scribe (or Recorder) is the person who is responsible for documenting issues raised during the process of the review meeting.

Typical Review Process Structure

An example of a typical review process is below. This is probably the most documented review process you will find in the software development world. Similarly to the review roles, this is open to interpretation by different organizations, and serves simply as an example for you.



INFORMAL REVIEW

An informal review is an extremely popular choice early on in the development lifecycle of both software and documentation. Often, pieces of work in software product developments can be lengthy, whether it's a piece of software or a detailed Test Specification. You don't want to present your completed piece of work at a formal review, only to find that you have completely misunderstood the requirements and wasted the last two months work. Think of starting a journey at point 'A' with an aim of arriving at point 'Z' when your piece of work is complete.

This is where informal reviews can be invaluable. Why not have an informal review at point 'B'?

If, for example you are working on creating a detailed test specification which you know will take several weeks to complete. You have just completed the first section, and you are thinking should I continue writing the rest of the specification in the same way? Then, now is the perfect time for an informal review.

You can then ascertain whether or not you are travelling in the right direction. Maybe take on additional suggestions to incorporate in the remaining work. The review is commonly performed by a peer or someone with relevant experience, and should be informal and brief.

Summary:

- Low cost
- No formal process
- No documentation required
- Widely used review

WALKTHROUGH

A walkthrough is a set of procedures and techniques designed for a peer group, lead by the author to review software code. It is considered to be a fairly informal type of review. The walkthrough takes the form a meeting, normally between one and two hours in length. It is recommended that between three to five people attend. The defined roles for the walkthrough attendees would be a Moderator, a Scribe and a Tester. As to who actually attends can vary based upon availability, but a suggested list from who to pick from would be:

- High experienced programmer
- Programming language expert
- Low experienced programmer
- Future maintenance engineer of the software
- Someone external to the project
- Peer programmer from same project

It is important for the participants of the walkthrough to have access to the materials that will be discussed several days prior to the meeting. This gives essential time to read through available documentation in order to digest it and make some notes, basically prepare them for the meeting.

When the walkthrough starts, the person acting as the Tester provides some scenario test cases. These test cases should include a representative set of input data and also the expected output from the program. The test data is then walked through the logic of the program. The test cases themselves simply assist in the generating of dialog between members of the walkthrough. In effect, the test cases should provoke the attendees into raising questions directed towards the program itself. The aim of the walkthrough is not to find fault in the programmer but in the program itself.

Summary:

- Led by the Author
- Attended by a peer group
- Varying level of formality
- Knowledge gathering
- Defect finding

TECHNICAL REVIEW

A Technical Review (also known as a peer review), is considered to be a formal type of review, even though no Managers are expected to attend. It involves a structured encounter, in which a peer/s analyse the work with a view to improve the quality of the original work.

The actual review itself is driven by checklists. These checklists are normally derived from the software requirements and provide a procedure to the review. If the piece of work is software code,

the reviewer will read the code, and may even develop and run some unit tests to check that the code works as advertised.

The documentation from the outcome of the review can provide invaluable information to the author relating to defects. On the other side of the fence, it also provides information to peers on how the development is being implemented. The status of the product may also be obtained by Managers from this type of review.

Summary:

- Ideally led by the Moderator
- Attended by peers / technical experts
- Documentation is required
- No Management presence
- Decision making
- Solving technical problems

INSPECTION

An inspection is a formal type of review. It requires preparation on the part the review team members before the inspection meeting takes place. A person will be in charge of the inspection process, making sure the process is adhered to correctly. This person is called a Moderator. The Moderator is normally a technical person by nature and may have Quality Assurance experience. It is also suggested that the Moderator comes from an unrelated project. This is to ensure an unbiased approach, and prevent a conflict of interests.

The Moderator will be responsible for arranging the inspection meeting and inviting the attendees. An agenda will be sent out by the Moderator to the attendees containing a checklist of items to be dealt with at the inspection meeting.

At the inspection meeting the producer of the item to be reviewed will present it to the meeting attendees. As the item is presented, items on the checklist will be addressed accordingly. Someone will be assigned the task of documenting any findings, known as the ‘Scribe’. Inspection metrics will also play a part during the meeting. These are basically a set of measurements taken from the inspection in order to assist with quality prediction, and preventing defects in the future.

When all checklist items have been addressed, the inspection meeting naturally draws to a close. At this stage a ‘Summary Report’ will be created basically outlining what has happened during the inspection, and what is to be done next. A follow-up stage is also a requirement of the inspection. This ensures that any re-working is carried out correctly. Once any outstanding work has been completed and checked by a re-inspection or just by the Moderator, the inspection will be considered to be complete.

Summary:

- Led by a Moderator
- Attended by specified roles
- Metrics are included
- Formal process
- Entry and Exit Criteria
- Defect finding

Chapter –7: Incident Management

OVERVIEW

We term an incident; any significant, unplanned event that occurs during testing that requires subsequent investigation and/or correction. At first glance it seems very similar to a 'software fault'. But at the time of finding the incident, most of the time we cannot determine whether the incident is a fault or not without further investigation.

The incident should be raised when the actual result differs from the expected result. After the inevitable investigation of the incident, there may be a reason other than a software fault, for example:

- Test environment incorrectly set up
- Incorrect Test Data used
- Incorrect Test Specification

RAISING AN INCIDENT

An incident does not have to be raised only against a piece of software. It can also be raised against an item of documentation.

The incident itself should only be raised by someone other than the author of the product under test. Most companies use some form of software to create and store each incident. The incident management software should be simple to use and training should be provided to all users if required. It should also provide the facility to update each incident with additional information. This is especially useful when a simple way to reproduce the incident has been found, and can then be made available to the person assigned to investigate the incident.

When documenting the incident, it is important to be thoughtful and diplomatic to avoid any conflicts between any involved parties. (i.e. Testers & Developers).

The incident will also need to be graded. This is basically a way of stating how important you think it might be. This can initially be done by the person raising the incident and can be updated at a later time. Most companies have their own idea of grading, some are more complex than others.

Once the incident has been stored, it is important for a Tester to continue with the next task in hand. It is easy to discover an incident and spend too much time trying to work out why it has happened. This can impact the test progress and should be avoided unless authorised to investigate the incident further.

The incidents themselves should be tracked from inception through all stages right through to when it is eventually resolved. It is common practice for regular meetings to occur to discuss the incidents raised. This has the advantage of ascertaining the severity of the problem and assigning appropriate personnel to deal with the incidents in a timely fashion. It is also helpful for management to see the potential impact on the project from the incidents raised.

An example of the type of information to be included when raising an incident:

Software under test ID:

This is normally a code or name assigned to the item under test by the company. It is important to include any version details here too.

Tester's name:

Obviously the Testers given name

Severity:

This can vary from company to company, but a commonly used severity grading method is; Low – Medium – High.

Scope:

The scope should detail in what area the incident was found and which areas may be affected by the incident.

Priority:

The priority will normally be specified by a Manager as it can dictate who will investigate the problem and the arising timescales from that investigation.

Steps to reproduce:

This is probably the most important field of all, as it contains vital information. An ideal incident would be documented with sufficient information clearly explaining what the problem is and simple steps in order for someone else to be able to reproduce it. Make sure that the incident can be understood by someone with limited knowledge of the product. As its not always someone with a high level of technical knowledge assigned to it.

Unfortunately, it is a common occurrence for the incident investigator to require contacting the originator to ask questions on what exactly the problem is, or how to reproduce it. This can waste a lot of time, but can be easily avoided if care is taken when documenting the incident. It is also common for poorly documented incidents to be miss-understood by other parties, which can lead to the wrong action or no action to be taken at all resulting in serious faults 'slipping through the net'.

The 'Steps to reproduce' information is also a good place to detail a 'work-around' if one exists. This can be important as some incidents can effectively block the remainder of any testing to be carried out. However, if a work-around is documented it may assist other testers who come across the problem to avoid the same situation.

IEEE STD. 1044-1993

This standard aims to provide a common approach to classification of anomalies found in software. It includes descriptions of the processes involved in a software life cycle, including details on how anomalies should be recorded and subsequently processed.

Although still in use today, this standard is becoming outdated and will probably be superseded in the near future.

The process defined by the standard is divided into four sequential steps.

Sequential Steps:

- Recognition
- Investigation

- Action
- Disposition

Each of the above steps has three administrative activities.

Administrative Activities:

- Recording
- Classifying
- Identifying impact

Chapter –8: Test Process Improvement

OVERVIEW

When thinking about improvement process, we must remember that this applies not only to software development process, but also to the testing process. In this section we will look at some of the most popular models and methods of software process improvement around today.

Before we look in detail at the maturity models, let us first understand what we mean by the term 'maturity model'. A maturity model is basically a collection of elements that are structured in such a way that they can describe characteristics of processes and their effectiveness.

A maturity model can provide:

- A starting point
- A shared vision
- A structure for organising actions
- Use of previous experience
- Determining real value of improvements

SEI CAPABILITY MATURITY MODEL (CMMI)

The Capability Maturity Model, simply put, is a baseline of practices that should be implemented in order to develop or maintain a product. The product can be completely software, or just partially software. The SW-CMM focuses on the software practices whereas with the CMMI, you may find both software and systems practices.

The CMMI and SW-CMM were published by the SEI. The models are commonly used throughout the world and considered by many to be 'the' standard within this area. In fact, the SEI refers to over 2000 organizations that have taken part in official trial CMMI assessments. Although, there are many other similar models in existence today, none command the respect from the professional community like the CMMI does.

The Capability Maturity Model (or CMM) has been in existence since 1991. More recently the CMM has been referred to as the SW-CMM, the SW obviously stands for 'software'. In 1998 the plan was to replace version 1.1 with version 2.0, but before that happened, the SEI decided to focus on combining of system and software practices. This then became what is known as the CMMI, the 'I' stands for integration.

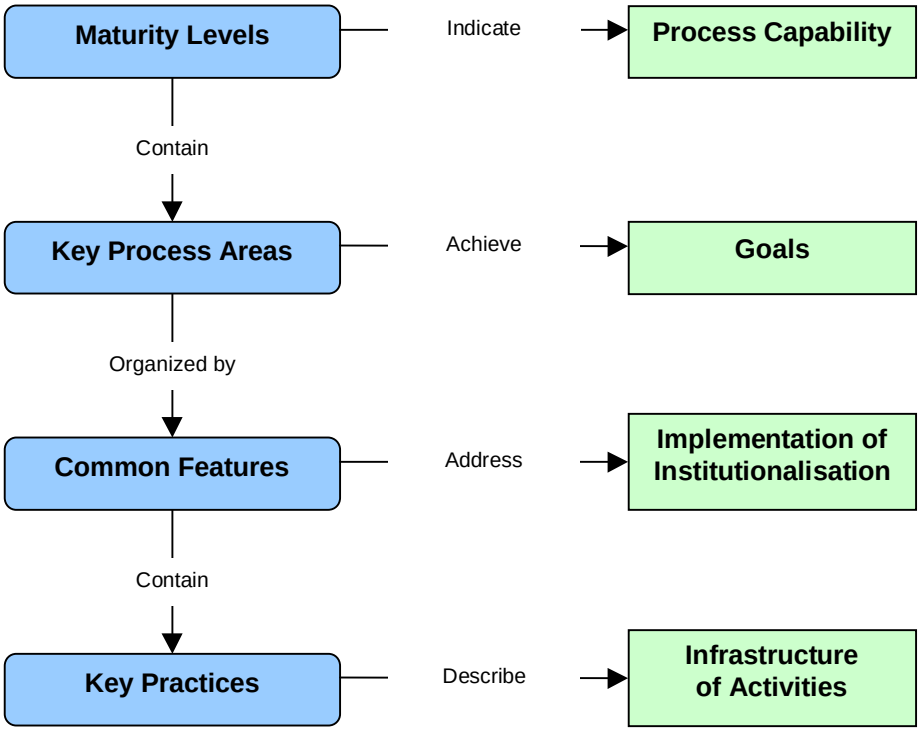
Viewing the SW-CMM Model:
The actual SW-CMM model is made up of two parts, commonly referred to as 'The TR24' and the 'TR25'. The 'TR' stands for Technical Report. The TR24 is basically an overview, the TR25 details the processes, goals and practices.

Viewing the CMMI Model:
The CMMI model is presented in a single volume. The reader can choose between a 'staged' or 'continuous' representation.

The SEI has developed its own training courses for CMMI and SW-CMM. They are commonly a three day introduction course for each model, and provide enough information for most organisations to get started implementing a new process.

CMM Structure

The CMM is made up of five maturity levels. Apart from Level 1, each maturity level is made up of multiple key process areas. Each key process area is separated into five sections, which are called common features. The common features specify the key practices that should accomplish the goals of the key process area.

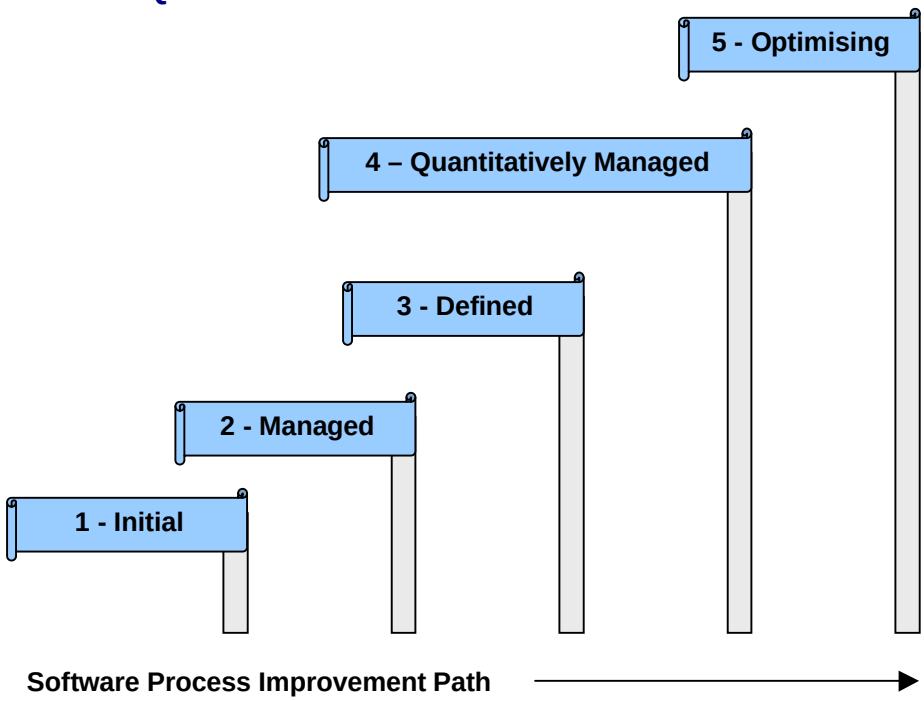


Maturity Levels

A maturity level can be thought of as a defined evolutionary plateau that the software process is aiming to achieve. The CMM defines five maturity levels which form the top-level structure of the CMM itself. Each level is basically a foundation that can be built upon to improve the process in sequence. Starting with basic management practices and progressing through successive proven levels.

When progressing through the path to improving a software process, the path will inevitably lead through each maturity level. The goal should be to achieve success in each maturity level, consolidating what has been learned and implementing this within the software process.

Maturity Levels:



Examples of software process states at each maturity level of organizations are shown below:

1 – Initial

At this level the software development process is often not stable, and often any existing good practices are destroyed by bad planning. Commonly at this level when problems arise in a project, short-cuts are taken, the original plan is abandoned, and with it any formal process structure. This leaves determining the software process capability a difficult task, as most of the defining qualities of the process are unpredictable. Some evidence may exist at this level of a software process through individuals, but probably not through the organisations capability.

2 – Managed

The software processes of a typical 'Level 2' organization would show that there exists some discipline, as planning and tracking activities in a given project have been repeated elsewhere. Project related processes are under effective control by use of a management system.

3 – Defined

This level of organization will show that consistency exists within the software engineering activities and the management activities as the processes used are stable and repeatable. Defined roles and responsibilities exist within the software process, and an organization-wide understanding of the process exists.

4 – Quantitatively Managed

A typical 'level 4' organization will include predictability, as the process is measured and controlled by specified limits. This predictability has the advantage of giving organizations to foresee trends in process and quality within the specified limits. If the process strays out-of-bounds of the limit, then corrective action is taken. Products emerging from these types of organizations are typically of a high quality.

5 – Optimising

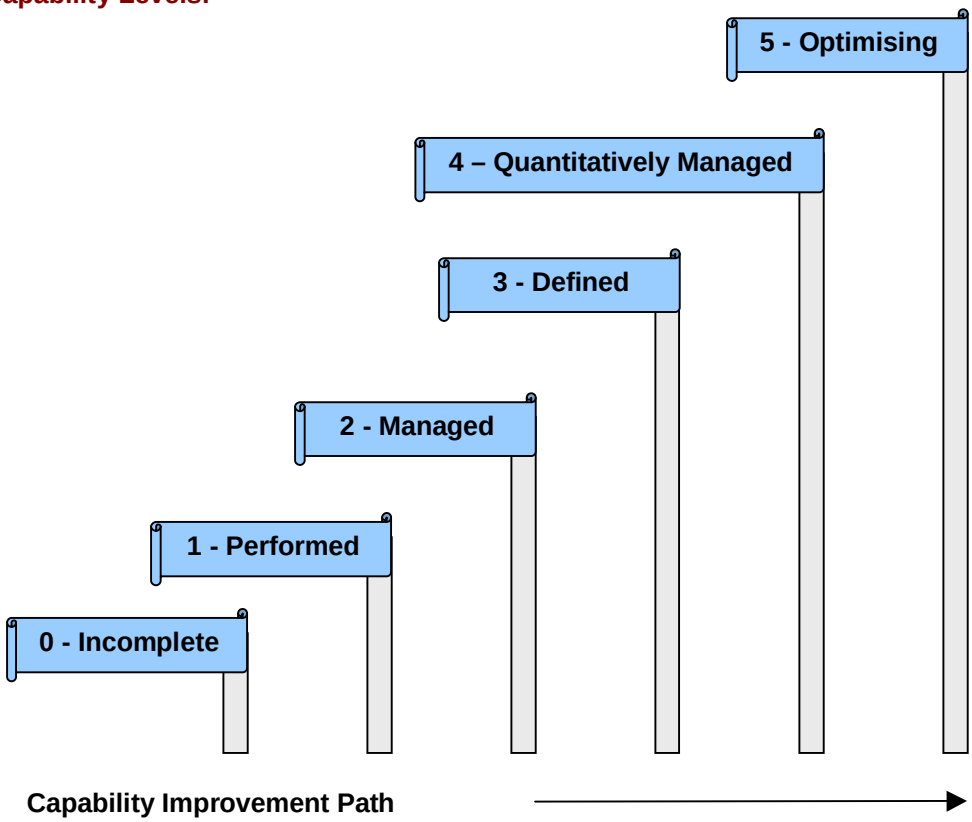
This level of organization will show that continuous improvement is a high priority. Their focus aims to expand the process capability, which has the effect of improving project related process performance.

New technologies and methods are welcomed here, and new processes and improvements to existing processes are their goal.

Process Capability

The software process capability defines what can be achieved by undertaking a specific software process. It achieves this by describing the range of expected results. It can be used to provide an idea of the outcome of future projects that use a specific process. There are six capability levels. Each level provides a foundation that can be built on for continuous process improvement. The levels are cumulative and so a high level would indicate that all lower levels have previously been satisfied.

Capability Levels:



Key Process Areas

Every maturity level is essentially made up of key process areas. The key processes themselves are each made up of a set of activities, which if all are performed correctly can fulfil a given goal.

Goals

The goals effectively form a summary of the key practices of a given key process area. They can be used as a yard-stick to ascertain whether or not the key process area has indeed been implemented correctly.

Common Features

Each key practice is sub-divided into five common feature areas:

- Commitment to Perform

- Ability to Perform
- Activities Performed
- Measurement and Analysis
- Verifying Implementation

The above common features can be used to show whether or not the implementation and institutionalisation of a key process area is effective. The 'Activities Performed' common feature is associated with implementation, while the remaining four common features are associated with institutionalisation.

Key Practices

A key process area can be thought of as individual key practices that assist in achieving the goal of a key process area. The key practices themselves detail the activities that will be the most important to the effectiveness of the implementation and institutionalisation of the key process area.

ISO/IEC 15504 (SPICE)

The ISO/IEC 15504 is also known as SPICE. Spice stands for Software Process Improvement and Capability dEtermination. It is essentially a framework for assessing software processes. It was developed by the Joint Technical Subcommittee between the IEC (International Electrotechnical Committee) and the IEC (International Organization for Standardization).

Rather than concerning itself with specific standards, ISO/ISEC 15504 concerns itself with is the capabilities provided by an organisations structure. These structures include its management structure and its process definition structure.

The ISO/IEC 15504 works by detailing lists of activities which can be used by an organisation throughout a software development lifecycle. The activities also do not have to be carried in a set order. When an ISO/IEC 15504 assessor is present at the organisation they effectively tick-off the performed activities from a checklist, and then use this to determine the organisations software process capability.

The standard is specified in nine individual parts:

Part 1	This part of the ISO/IEC 15504 explains the concepts and provides an overview of the framework.
Part 2	This contains a reference model. The reference model defines a process dimension and a capability dimension.
Part 3	This provides a guide for performing an assessment.
Part 4	This part also provides a guide for performing an assessment.
Part 5	This provides an assessment model, but other models could be used instead, if they meet ISO/IEC 15504 criteria.
Part 6	The competency of assessors is the subject of part 6.
Part 7	Process improvement is the subject of part 7.
Part 8	Supplier process capability determination is the subject of part 8.
Part 9	Part 9 of ISO/IEC 15504 is a vocabulary list.

SEI CMM AND ISO/IEC 15504 RELATIONSHIP

The take-up of ISO/IEC 15504 has been fairly successful as it is recognised internationally. It is also available publicly through national standards bodies.

It currently lies though in second place behind the CMM. This can be put down to the fact that it is expensive in comparison, as the CMM and CMMI are free to download. Also, the CMM is backed by the US Department of Defence which raises its profile above the ISO/IEC 15504 particularly with the American audience. The CMM was also first off the blocks and so established itself before the ISO/IEC 15504 was launched.

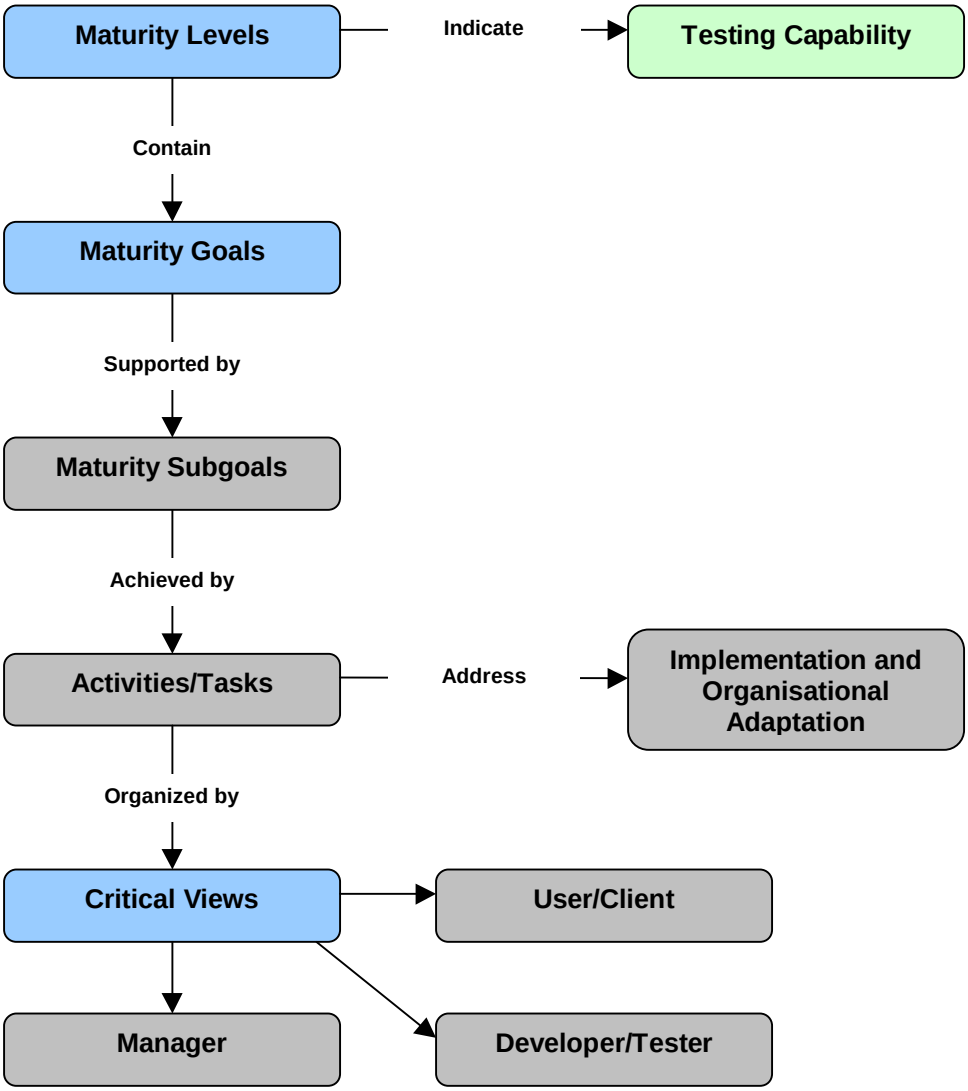
When the CMMI replaced the CMM, it actually incorporated many of the ideas from the ISO/IEC 15504 within it.

THE TESTING MATURITY MODEL (TMM)

Overview

The Illinois Institute of Technology (IIT) developed the Testing Maturity Model (TMM) in 1996. The main reason for developing the TMM was that existing maturity models didn't properly address real testing issues. It was designed to complement the existing CMM. The main purpose of the TMM is to support assessment and improvement drives within an organisation. The model comprises of a Maturity Model and an Assessment Model.

Maturity Model



Maturity Levels

The maturity levels are basically defined levels of maturity that can be achieved by showing that specific practices have been carried out. The TMM has five different achievable levels:

- Initial
- Phase Definition
- Integration
- Management and Measurement
- Optimisation/defect Prevention and Quality Control

To achieve each of the above levels, specific goals must have been proved to have been addressed. The TMM goals are very similar to the 'key processes' of the CMM. Each of these goals also commonly has sub-goals attached to them too. The sub-goals can be achieved through ATRs (Activities, Tasks and Responsibilities). The ATRs are very similar to the 'Activities' part of the CMM.

The ATRs are used to address the implementation and organizational adaptation issues at specific levels. The actions that need to be performed to improve the testing capabilities can be defined as the 'Activities' and 'Tasks'. The people, who ensure that these activities and tasks have been carried out, are the Managers, Developers/testers and User/clients. These three participants are what are known as the 'Critical Views'.

1 - Initial Level

At this first level, testing is does not have a formal process or structure. The testing is often only performed after all development has been completed. The quality of the products developed is questionable. There are also no goals present for improving process maturity.

2 – Phase Definition

This level will normally have organisations that have a defined testing phase for a development lifecycle. Some test planning exists, and a limited amount of repeatability. This level contains the following maturity goals:

- Development of testing and debugging goals
- Creation of a test planning process
- Definition of testing techniques and methods

3 – Integration

Level 3 focuses on the ability to allow quality processes to begin much earlier on in the development lifecycle. Testing activities are well defined and exists throughout the software life cycle phases. Testing is backed by managers here, and also a dedicated test team/group may have been created. Common goals for this phase include:

- Creation of a software test organisation
- Establish a technical training process
- Integrate testing into the development life cycle
- Monitor and control the testing process

4 – Management and Measurement

This level finds the focus placed on broadening the definitions of the testing activities and adequately measuring the results. Examples of this would be; performing staged reviews for testing activities through each phase of the development life cycle. The type of tasks performed at this level are often referred to as; 'verification and validation'. Goals at this level are:

- Creation of review process
- Creation of test measurement program
- Evaluation of software quality

5 – Optimisation/defect Prevention and Quality Control

Testing at this level is aimed at satisfying the software specification, thus ensuring a measured amount of confidence within a product. Faults within the development lifecycle are prevented by specific tests. The overall testing process is well planned, organised and implemented. Goals here are:

- Evaluation of data for defect prevention
- Quality control
- Optimisation of the test process

TMM Assessment Model

The TMM assessment model is designed to allow self-assessment of a given testing process. The assessment model uses the TMM as a reference and consists of three parts:

- The assessment Procedure
- The assessment questionnaire
- Team training and selection

The assessment procedure is simply a consecutive series of steps that allow a self-assessment team to be guided through the task of assessing the process. The assessment questionnaire part is used to gather the information from a self-assessment which can then be studied afterwards to ascertain the level of maturity. The self-assessment team should have had sufficient training in order to carry out and effective self-assessment. The TMM also uses ideas from SPICE in order to effectively select a self-assessment team.

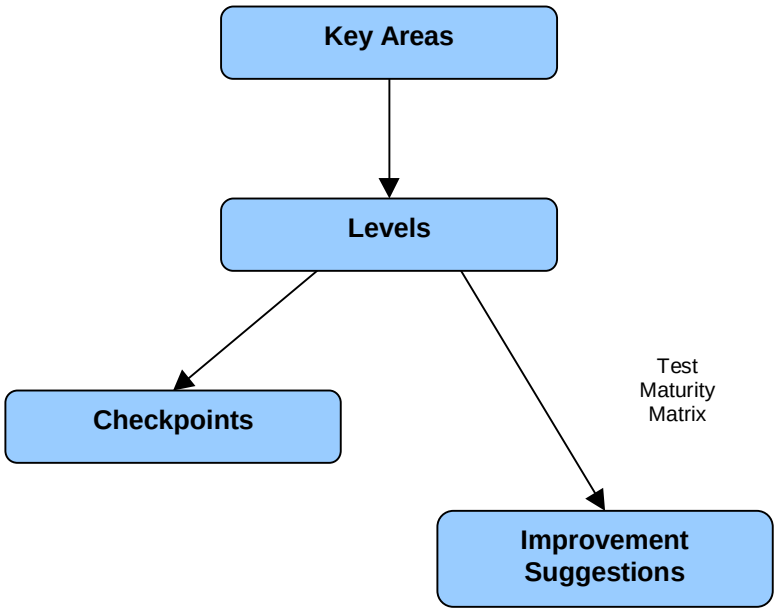
THE TEST PROCESS IMPROVEMENT MODEL (TPI)

Developed by Koomen and Pol in 1997, the Test Process Improvement Model or 'TPI' was created with the goal of simplifying the sometimes over-complicated testing process. The TPI model itself identifies the good and bad parts of a testing process. The maturity of the process can also be assessed by using the TPI.

The TPI consists of the following four components:

- A Maturity Model
- A Test Maturity Matrix
- A Checklist
- Improvement Suggestions

Maturity Model



The TPI model consists of three maturity levels and fourteen scales. The individual levels contain several different scales. The scales themselves provide indication of which key areas require improvement. The scales are displayed in a 'test maturity matrix'.

- Scales 1 to 5 focus on bring the testing process under control

- Scales 6 to 10 focus on establishing test process efficiency

§ Scales 11 to 14 focus on test process optimisation

Key Areas

The key areas of the TPI model mostly relate to improving systems and acceptance testing. Each key area is made up of different levels.

Test strategy:
Prioritisation of defects is an important part of the test strategy in order to reduce development costs. Assignment of tests to specific risks and requirements is also an action performed here.

Life-cycle model:
A process can include separate phases. Within these phases individual activities can be assigned. For each activity goals, input, output, dependencies can be assigned.

Movement of involvement:
The involvement of test personal should be made as early on in the development lifecycle as possible, not just when the code has been completed.

Estimation and planning:
Plans to include who will test, when they will test, and how they will test should be created.

Test specification techniques:
A technique should be established in order to allow a standard method for deriving test cases from given information.

Static test techniques:
Checking and dry running can be executed on documentation and software code early on in the development lifecycle to prevent certain errors.

Metrics:
Metrics can be used to assist with tracking the test process throughout a development life cycle, and can to provide a quantifiable level of quality.

Test tools:
Test tools can be implemented to automate certain test activities to free up time and provide enhanced test coverage.

Test environment:
The test environment consists of facilities, equipment and procedures. Organised correctly, these items can assist with efficient software developments.

Office environment:
Items such as office space, computers and correct furniture can influence the motivation of a development team.

Commitment and motivation:
Motivation can be kept high through the development by the selection of good testers, team leaders and managers. This will result in a better chance of a project running smoothly.

Testing functions and training:
A well balanced test team consisting of multi-disciplined personnel should be used. Training must be provided to cover any short-falls.

Scope of methodology:
Working methodologies should be used to cover all conceivable situations that may arise throughout a project development life cycle.

Communication:

Communication is very important part of any development. This allows the free-flowing of information allowing issues to be resolved more swiftly and any foreseeable problems being highlighted well in advance.

Reporting:
Reporting can exist from the test team to managers for example to provide requested information, or even provided to the customer to fulfil any specific requests that they may have about the product or eve the testing process.

Defect management:
Defect tracking is very important. Any defects should be tracked from conception right up until they have been re-tested. A sense of quality can also be derived from the amount of defects found within a product.

Testware management:
Testware is another name for the products of testing. This should be reusable as much as possible to allow its use on future projects.

Test process management:
Planning, execution, maintaining and adjusting are essential steps in order to control process activities.

Evaluation:
Evaluation can be used to determine items such as requirements and design documents to ensure the product is being developed correctly.

Low level testing:
Low level testing is often termed 'white-box testing as it requires knowledge of the internal workings of the program or system. Unit testing is a common form of low level testing.

Test Maturity Matrix

The TPI takes into account the different aspects of a test process, including design techniques, test tool usage and reporting. Structured evaluation of various key areas has the effect of highlighting the test processes strengths and weaknesses. The state of a key area is determined by assigning a level to it, commonly A to B to C etc. The levels are increased based on time, cost and quality.

Example:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Test strategy		A				B								
Life-cycle model		A		B							C			D
Moment of involvement														
Estimating & planning														
Test Specification			A			B			C					D
Static techniques														
Metrics					A				B			C	D	
Test tools														
Test environment		A			B			C			D			
Office environment														
Commitment & motivation														
Testing functions			A							B				
Scope of methodology														
Communication														
Reporting		A			B							C		
Defect management														
Testware management			A					B		C			D	
Test process management														
Evaluation				A						B				C
Low-level testing			A				B					C		

The above example table shows all of the key areas and can also be used to determine the dependencies between levels and key areas. The matrix also shows that each level is related to a certain scale of test maturity, there being thirteen of them. After filling in the matrix, it makes it is easy to see which areas need to be improved on.

Checklist

To define the level for a key area, the model uses something called ‘checkpoints’ as a measuring tool. The checklist itself contains numerous checkpoints for each level for each key area. In order to assign a level to a key area, all of the checkpoints need to have been satisfied. Also, this system is cumulative and therefore, for example, if an area has been assigned level ‘C’, then all of the checkpoints for levels ‘A’ and ‘B’ must also have been satisfied.

Improvement suggestions

Improvement suggestions are simply hints or tips to assist in improving a key area. They are by no means compulsory. Each level of area key area will have a number of improvement suggestions in order to assist with the improvement process. When combined with the checklist the improvement suggestions can easily be used by management to set certain goals of achievement in order to improve their test process.

Chapter –9: Testing Tools

OVERVIEW

The following section gives a simple overview of popular tools associated with testing available today.

Requirements Testing Tools:

This type of tool is designed to assist with verification and validation of requirements, for example; consistency checking.

Static Analysis Tools:

By examining the code instead of running test cases through the code, this type of tool can provide information on the actual quality of the software. Cyclomatic complexity is one such characteristic that can be obtained by using this type of tool.

Test Design Tools:

This type of tool can generate test cases from specifications, which are normally stored in a CASE tool repository. Some variations of this type of tool can also generate test cases from analysing the code itself.

Test Input Data Preparation Tool:

Data can be selected from existing test specific databases by using this type of tool. Advanced types of this tool can utilise a range of database and file formats.

Test Running Tools:

These are an extremely popular type of tool. They provide capture and replay facilities for WIMP interface based applications. The tools can simulate mouse movement, mouse clicks and keyboard inputs. The tools can even recognize windows and buttons, thus making them extremely versatile. The test procedures are normally written in a specific scripting language. This tool is another popular choice for regression testing.

Test Harnesses:

If the software under test does not have a user interface, then test harnesses and drivers can be used to execute the software. These types of tools can be bought off the shelf, but more commonly they are built for a specific purpose.

Test Script Generators:

Creates actual test scripts based on information held within a test specification. Simulators are commonly used where it is impracticable to use them, for example software to control a space probes trajectory.

Performance Test Tools:

This type of tool comprises of two components; Load Generation and Test Transaction Measurement., Load Generation is commonly performed by running the application using its interface or by using drivers. This has the effect of simulating load on the application. The number of transactions performed this way are then logged. Performance test tools will commonly be able to display reports and graphs of load against response time.

Dynamic Analysis Tools:

Run-time information on the state of the executing software is achieved by using Dynamic Analysis Tools. These tools are ideally suited for monitoring the use and allocation of memory. Faults such as memory leaks, unassigned pointers can be found, which would otherwise be difficult to find manually.

Debugging Tools:

Debugging tools are often used by programmers to try and reproduce code related errors in order to investigate a problem. The debugger allows the program to be run line by line. This

enables halted the program on demand to examine and set program variables.

Comparison Tools:

This type of tool is used to highlight differences between actual results and expected results. Off the shelf Comparison Tools can normally deal with a range of file and database formats. This type of tool often has filter capabilities to allow 'ignoring' of rows or columns of data or even areas on a screen.

Test Management Tools:

Test Management Tools commonly have multiple features. Test Management is mainly concerned with the management, creation and control of test documentation. More advanced tools have additional capabilities such as test management features, for example; result logging and test scheduling.

Coverage Measurement Tools:

This type of tool provides objective measures of structural test coverage when the actual tests are executed. Before the programs are compiled, they are first instrumented. Once this has been completed they can then be tested. The instrumentation process allows the coverage data to be logged whilst the program is running. Once testing is complete, the logs can provide statistics on the details of the tests covered.

Hyperlink Testing Tools:

These tools are simply used to check that no broken hyperlinks exist on a web site.

Monitoring Tools:

These tools are typically used for testing e-commerce and e-business applications. The main purpose of this tool is to check web sites to ensure that they are available to customers and also to produce warnings if problems are detected.

Security Testing Tools:

These tools are commonly used for testing e-commerce and e-business applications, and sometimes web sites. A security testing tool will check for any parts of a web based system that could cause potential security risks if attacked.

Test Oracles:

A Test Oracle is used to automatically generate expected results. They are commonly used in situations where an old system is upgraded with a new system with the same functionality, so the old system can be used as an Oracle.

TOOL SELECTION

In today's modern testing environment there are normally multiple types of testing activities to be performed throughout the project. Luckily for the tester, there are a great deal of tools available that can assist in test automation and other test related tasks.

Imagine that a test department decides to automate all of their tests. If all of the tests are currently performed manually, then you might at first think…..

"Why don't we automate them all?"

Having a fully automated test environment can take an enormous amount of resources to develop. Not only will every test specification have to be converted to an automated script (which will in itself require testing), but automated test tool training will also be required. Once you have your automated test environment in place, it will require constant maintenance. Every test specification change will have to be automated; every software product change will have to be automated. That's just a few things worth bearing in mind!

When considering any new tool, try and use a disciplined approach. Don't instantly choose the tool that has the most features, as it could prove to be an over-complicated tool that may require

additional training. Also, try and consider the tools ability to integrate with your existing environment, for example database connectivity. Another point to consider is the future of your automated environment. A plan of where you expect the automated environment to eventually take your test process may have an impact of type of tool you are considering.

A suggested tool selection and evaluation process is:

- Determine the actual problem or requirement
- Ensure that there are no obvious alternative solutions
- Prepare a business case
- Identify any constraints
- Identify any specific required tool features or characteristics
- Prepare a short-list of possible suitable tools
- Perform a detailed evaluation
- Perform a competitive trial, if needed

TOOL IMPLEMENTATION

The last thing we want is to introduce a tool into the organisation, only to find a few weeks down the line it fails resulting in potentially disastrous scenarios. In order to avoid this situation, once we think we have the right tool, we can implement a pilot project. The benefits of using a pilot project are;

- Gaining experience using the tool
- Identification of any changes in the test process that may be required
- Identifying any shortcomings in the suitability of the tool

An implementation team can be formed to evaluate the tool consisting of:

A Champion:

This person is the driving force behind the day to day implementation of the tool, they would understand all of the issues involved and they need to be a good communicator and team player.

A Change Agent:

Their responsibilities would include planning and managing the implementation of a pilot project and the tool itself. This person can also double-up as the Champion.

A Tool Custodian:

Their responsibilities include technical tool support, providing assistance or consultancy in the use of the tool.

The overall implementation of any tool should have management backing, which will be needed to overcome any ‘red tape’ which is invariably associated with any implementation.

Roll-out of the tool should only occur following a successful pilot project or evaluation period.

Chapter –10: Skills of Personnel

INDIVIDUAL SKILLS

There are no hard and fast rules as to what makes a good Tester. This is because there are so many factors that can affect an individual, whatever there specific purpose in the development lifecycle is. So how can we at least define what an individuals testing capability is then? Well, firstly we should take into consideration the individual's technical background. This can include their previous experience or their training in the area of testing. We can break this down into the following three areas:

Users:

A 'User' is someone who has had experience actually 'using' the software under test, or similar types of software. This knowledge can be useful to determine the type of faults that a typical user may come across, which to most developments would also have the most impact. The 'User' would probably not have sufficient knowledge to test the software to extreme depths though.

Developer:

A Developer's background may have provided them with experience with code, design or requirements analysis. This knowledge can be extremely useful, as the developer would probably have some idea when looking at the software of how it was developed, and so would probably know where to look for weaknesses.

Tester:

Previous knowledge of testing has its obvious advantages. They should be able to analyse a specification, design test cases, execute test cases, and produce results and reports. An individual with previous testing experience would also have the right mindset for testing, as they would already know the reasoning behind why testing is performed.

Additionally to specific technical skills a Tester should be able to communicate faults to the Software Developers in an appropriate manner. They should also be able to take criticism from others in order to self-develop. An important skill that should be acquired by all Testers is 'negotiation', as all Testers will come up against obstacles in their path that need to be tackled in order to progress with their work, and negotiation skills are an important attribute to have under these circumstances.

Lastly, a good Tester is one that enjoys testing. It sounds obvious; but a Tester who is enthusiastic about their work will invariably look harder to find faults, resolve problems and improve the process compared to someone who doesn't care about the quality of the product.

TEST TEAM DYNAMICS

When a test team is performing well, it is often down to individuals clearly understanding their own responsibilities, and also understanding their team member's responsibilities.

An important issue with recruiting new team members is ensuring that the new recruit will compliment the existing team. This should include complimenting the skills and personalities of the existing team members.

A good team should contain a variety of personality types. This allows the different roles and tasks to be allocated to the right team member. Obviously, careful consideration should be taken when assigning roles and responsibilities to the team members though. For example, you would not want to assign a managerial task to someone who is scared of their own shadow and afraid to air an opinion, as they may not be able to put their point across in a crucial meeting. Or you wouldn't want to assign writing a 'user manual' to somebody who struggles using that language.

A team containing a variety of personalities will always give options. This allows future tasks to be allocated to a suitable team member, and will contribute to the flexibility of the team.

Dr. Meredith Belbin has studied the concept of a team role and how it is based on the way a person behaves, contributes and inter-relates when part of a team.

Team Roles: *"a tendency to behave, contribute and interrelate with others in a particular way"* – Dr. M. Belbin

Belbin identified nine team roles which he categorized into three groups (see next page):

- Action Oriented
- People Oriented
- Thought Oriented

Each of the specified team roles can be associated with typical behavioural and interpersonal strength. Belbin also defined weaknesses that often form part of the team role. He referred to the weaknesses of the team roles the "allowable" weaknesses. These weaknesses are behavioural, and therefore have the potential to be improved.

The Belbin Team Roles Model can be of use in a variety of ways. It is of particular use when considering the balance of a team before a project starts, and so highlight any deficiencies within the team. It can also be of use to an individual team member as a guide to develop themselves.

Action Oriented Roles		
Shapers	*Challenging, dynamic, thrives on pressure. The drive and courage to overcome obstacles.*	*Prone to provocation. Offends people's feelings.*
Implementer	*Disciplined, reliable, conservative and efficient. Turns ideas into practical actions.*	*Somewhat inflexible. Slow to respond to new possibilities.*
Completer – Finisher	*Painstaking, conscientious, anxious. Searches out errors and omissions. Delivers on time.*	*Inclined to worry unduly. Reluctant to delegate.*

People Oriented Roles		
Coordinator	*Mature, confident, a good chairperson. Clarifies goals, promotes decision-making, delegates well.*	*Can often be seen as manipulative. Off loads personal work.*
Team Worker	*Co-operative, mild, perceptive and diplomatic. Listens, builds, averts friction.*	*Indecisive in crunch situations.*
Resource Investigator	*Extrovert, enthusiastic, communicative. Explores opportunities. Develops contacts.*	*Over - optimistic. Loses interest once initial enthusiasm has passed.*
Thought Oriented Roles		
Plant	*Creative, imaginative, unorthodox. Solves difficult problems.*	*Ignores incidentals. Too pre-occupied to communicate effectively.*
Monitor – Evaluator	*Sober, strategic and discerning. Sees all options. Judges accurately.*	*Lacks drive and ability to inspire others.*
Specialist	*Single-minded, self-starting, dedicated. Provides knowledge and skills in rare supply.*	*Contributes only on a narrow front. Dwells on technicalities.*

FITTING TESTING WITHIN AN ORGANISATION

It is always a good idea to have a multi-disciplined test team. There will always arise situations during a project where the individual Tester's skills will be called upon. Having multi-skilled Testers brings a level of balance to the team.

Let's take a look at individual roles within the organisation:

The Test Leader
The Test Leader will commonly come from a testing background and have a full understanding of how testing is performed. They will also possess good managerial expertise. They are also responsible for ensuring that test coverage is sufficient and will be required to produce reports. The following list shows some example activities you might expect a Test Leader to perform:

- Coordinate the Test Strategy and Test Plan with managers
- Planning and scheduling of tests
- Monitoring test progress
- Responsible for providing configuration management system

- Create reports on gathered testing information

The Tester

The Tester obviously provides the skills necessary to perform the Testing itself. This role can include test design and test execution. Automated testing skills are also a possible requirement of this role. The following list shows some example activities you might expect a Tester to perform:

- Preparation of test data
- Execute tests and provide results
- Review other peoples tests
- Review Test Plan
- Involvement in automation of tests
- Create Test Specifications

Other roles that exist within a testing organization:

The Client:

The client is effectively the project sponsor, and will provide the budget for the project. The Client can also be the business owner.

The Project Manager:

Management skills are provided by the Project Manager. The Project Manager will be actively involved throughout the project and will provide feedback to the client.

The User:

The User will provide knowledge from the existing system/software and will define requirements for the new system/software.

The Business Analyst:

The Business Analyst will provide knowledge of the business and analysis skills. The Business Analyst will also be responsible for creating User Requirements based on talks with the Users.

The Systems Analyst:

Systems design will be provided by the Systems Analyst. The Systems Analyst will also be responsible for developing the Functional Specification from the User Requirements.

The Technical Designer:

Technical detail and support to the system design is the responsibility of the Technical Designer. This role may include database administration.

The Developer:

A Developer will provide the skills to write the actual software code and perform Unit Testing. They may also be called upon at a later stage to provide bug fixes and technical advice.

A team does not have to have all of the above members, as each testing project will have different requirements. Some of the roles mentioned above may be carried by a single person, while other roles may require several people.

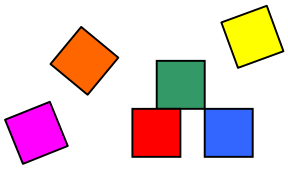
Tester and Developer Communication

One of the primary purposes of Software Tester is to find faults with software. This can often be perceived as destructive to the product development lifecycle. Whereas, the purpose of a Developer is a more creative one. This in some circumstances naturally causes friction between Developers and Testers.

A Developer's perspective

Developer

Tester



A Developer will often spend long hours working on a piece of software, sometimes for many months. They may take great pride in their piece of work. Then a Tester comes along and finds fault with it. You can quickly see where friction might emerge from!

Good communication is essential on any project, whether it is verbal or through documentation. The sooner the problem is understood by all parties, the sooner the problem can be resolved. This is particularly the case when it comes to the relationship between a Developer and a Tester. The way in which a Tester approaches a Developer with a problem is important to the progression of the project. The following example is how NOT to approach a Developer:

Tester: *"Hey, monkey brain's, I found another bug in your software"*

You can imagine the Developers response to that remark. How about the following more tactful approach:

Tester: *"Hi, I seem be to getting some strange results when running my test. Would you mind taking a look at my setup, just in case I have configured it incorrectly?"*

You and I both know that your setup is correct, but at least this way you are implying that the problem could possibly be with your work and not the Developers. When the Developer sees the test fail for himself, he will probably explain what he thinks is going wrong, and you will now be in a better position to work together to resolve the problem.

MOTIVATION

"A happy tester is a good tester" – A. Manager

Recognition and respect are important factors that are often missed from a Tester's work environment. Often Testers are seen as an obstacle in the product development lifecycle and so may not be subjected to praise from outside of the test team. So it is important to make sure the Tester receives recognition and respect from the other test team members and management before their motivation runs out. One way of providing motivation is to offer a Tester a structured career path. Many organisations even today have poor structures to their test departments, so your testing career at such a place will be either 'a Tester' or 'not a Tester'.

By being involved in projects earlier on in the development lifecycle, the Tester may receive recognition and value from other members of the development team. This can help highlight the fact that testing is an important part of not on that development, but also become a more important consideration for future developments.

Best of Luck Friends – For your certification Exam