SATHYABAMA UNIVERSITY

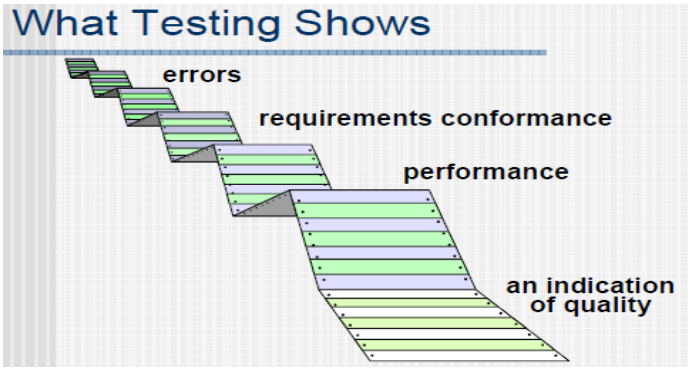
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SCSX1060 SOFTWARE TESTING – ELECTIVE

UNIT 2 – SOFTWARE TESTING TECHNIQUES

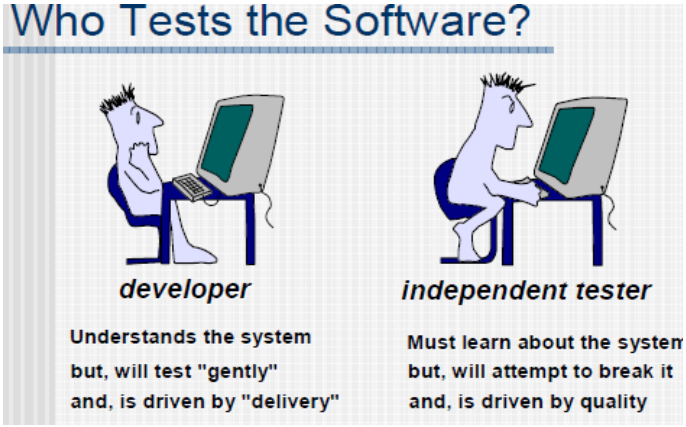
Software Testing Strategies - Overview

Software testing must be planned carefully to avoid wasting development time and resources. Testing begins “in the small” and progresses “to the large”. Initially individual components are tested and debugged. After the individual components have been tested and added to the system, integration testing takes place. Once the full software product is completed, system testing is performed. The Test Specification document should be reviewed like all other software engineering work products.



Strategic Approach to Software Testing

- Many software errors are eliminated before testing begins by conducting effective technical reviews
- Testing begins at the component level and works outward toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- The developer of the software conducts testing and may be assisted by independent test groups for large projects.
- Testing and debugging are different activities.
- Debugging must be accommodated in any testing strategy.



Verification and Validation

- Make a distinction between *verification* (are we building the product right?) and *validation* (are we building the right product?)
- Software testing is only one element of Software Quality Assurance (SQA)
- Quality must be built in to the development process, you can't use testing to add quality after the fact

Establish a Test Policy

The TMMi model, at second maturity level, starts from Process Area named Test Policy and Strategy. This area contains series of Specific Goals. The first is Establish a Test Policy. Establish a Test Policy relies on establish and agree with stakeholders the test policy, which was defined in previous point of this article. The test policy reflects and is complementary with organization quality policy. Which quality statements are included in the test policy is described in point „Define the test policy". Establish the test policy consists of three Specific Practices, such as: define test goals, define the test policy, distribute the test policy to stakeholders, which are presented in next points of this article.

Build a Solid Software Test Strategy

An effective testing strategy includes automated, manual, and exploratory tests to efficiently reduce risk and tighten release cycles. Tests come in several flavors:

- **Unit tests** validate the smallest components of the system, ensuring they handle known input and outputs correctly. Unit test individual classes in your application to verify they work under expected, boundary, and negative cases.
- **Integration tests** exercise an entire subsystem and ensure that a set of components play nicely together.
- **Functional tests** verify end-to-end scenarios that your users will engage in.

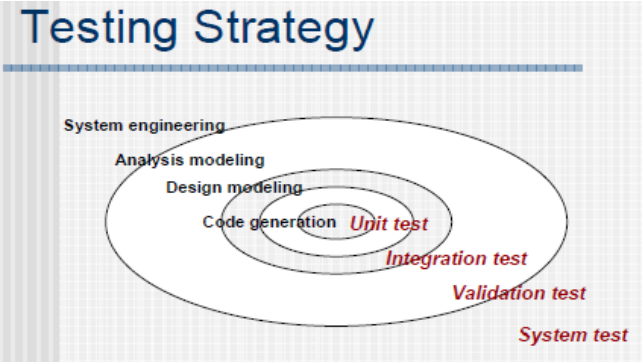
So why bother with unit and integration tests if functional tests hit the whole system? Two reasons: test performance and speed of recovery. Functional tests tend to be slower to run, so use unit tests at compile time as your sanity-check. And when an integration test fails, it pin-points the bug's location better than functional tests, making it faster for developers to diagnose and fix. A healthy strategy requires tests at *all* levels of the technology stack to ensure each part, as well as the system as a whole, works correctly.

Organizing for Software Testing

- The role of the Independent Test Group (ITG) is to remove the conflict of interest inherent when the builder is testing his or her own product.
- Misconceptions regarding the use of independent testing teams
  - The developer should do no testing at all
  - Software is tossed “over the wall” to people to test it mercilessly
  - Testers are not involved with the project until it is time for it to be tested
- The developer and ITGC must work together throughout the software project to ensure that thorough tests will be conducted

Software Testing Strategy

- Unit Testing – makes heavy use of testing techniques that exercise specific control paths to detect errors in each software component individually
- Integration Testing – focuses on issues associated with verification and program construction as components begin interacting with one another
- Validation Testing – provides assurance that the software validation criteria (established during requirements analysis) meets all functional, behavioral, and performance requirements
- System Testing – verifies that all system elements mesh properly and that overall system function and performance has been achieved



Strategic Testing Issues

- Specify product requirements in a quantifiable manner before testing starts.
- Specify testing objectives explicitly.
- Identify categories of users for the software and develop a profile for each.

- Develop a test plan that emphasizes rapid cycle testing.
- Build robust software that is designed to test itself.
- Use effective formal reviews as a filter prior to testing.
- Conduct formal technical reviews to assess the test strategy and test cases.
- Develop a continuous improvement approach for the testing process.

Economics of System Development Life Cycle (SDLC)

Recognition of need

- Feasibility study
- Analysis
- Design
- Implementation
- Post implementation and maintenance

Recognition of need

It is the first stage of information system development cycle. This gives a clearer picture of what actually the existing system is. The preliminary investigation must define the scope of the project and the perceived problems, opportunities and directives that triggered the project

The preliminary investigation include the following tasks:
a. List problems, opportunities and directives.
b. Negotiate preliminary scope.
c. Assess project worth.
d. Plan the project.
e. Present the project and plan.

Feasibility Study

The statement “don’t try to fix it unless you understand it” apply describe the feasibility study stage of system analysis. The goal of feasibility study is to evaluate alternative system and to purpose the most feasible and desirable system for development.

The system investigates the IT proposal. During this step, we must consider all current priorities that would be affected and how they should be handled. Before any system planning is done, a feasibility study should be conducted to determine if creating a new or improved system is a viable solution. This will help to determine the costs, benefits, resource requirements, and specific user needs required for completion. The development process can only continue once management approves of the recommendations from the feasibility study.[

It consists of the following: 1. Statement of the problem, 2. Summary of findings and recommendations, 3. Details of findings, 4. Recommendations and conclusions

There are basically five types of feasibility are addressed in the study.

1. Technical feasibility

2. Economic feasibility
3. Motivational feasibility
4. Schedule feasibility
5. Operational feasibility

Cost /benefit analysis

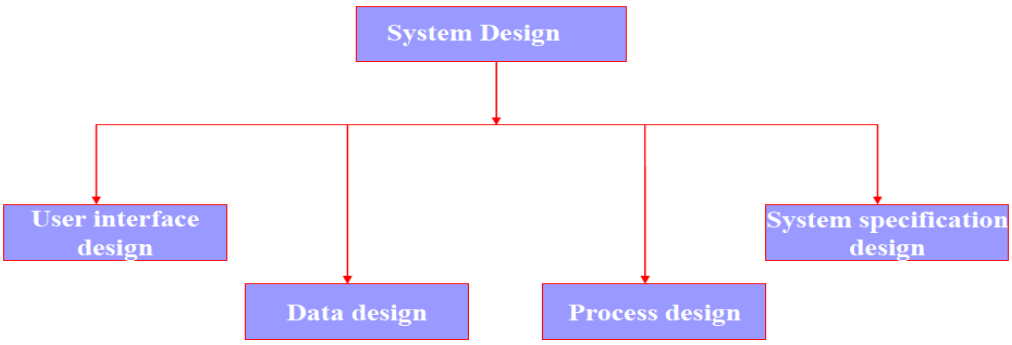
Feasibility studies typically involve cost/benefit analysis. In the process of feasibility study, the cost and benefits are estimated with greater accuracy. If cost and benefit can be quantified, they are tangible; if not, they are called intangible.

System Design

System design can be viewed as the design of user interface, data, process and system specification. In systems design, the design functions and operations are described in detail, including screen layouts, business rules, process diagrams and other documentation. The output of this stage will describe the new system as a collection of modules or subsystems.

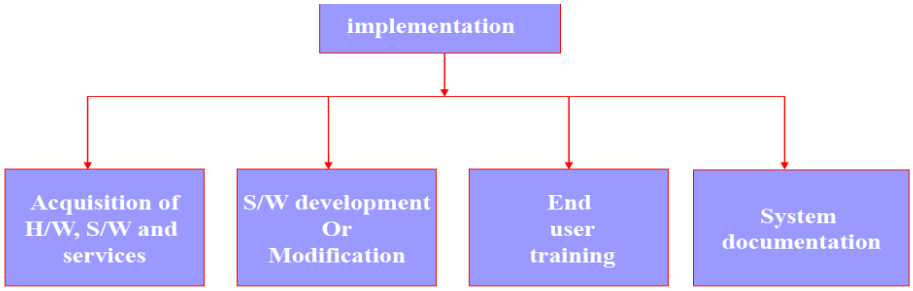
The design stage takes as its initial input the requirements identified in the approved requirements document. For each requirement, a set of one or more design elements will be produced as a result of interviews, workshops, and/or prototype efforts.

Design elements describe the desired system features in detail, and generally include functional hierarchy diagrams, screen layout diagrams, tables of business rules, business process diagrams, pseudo-code, and a complete entity-relationship diagram with a full data dictionary. These design elements are intended to describe the system in sufficient detail, such that skilled developers and engineers may develop and deliver the system with minimal additional input design.



System Implementation

Implementation is the stage where theory is converted into practical. The implementation is a vital step in ensuring the success of new systems. Even a well designed system can fail if it is not properly implemented.



Post Implementation and Maintenance

Once a system is fully implemented and being operated by end user, the maintenance function begins. Systems maintenance is the monitoring, evaluating and modifying of operational information system to make desirable or necessary.

Training and transition

Once a system has been stabilized through adequate testing, the SDLC ensures that proper training on the system is performed or documented before transitioning the system to its support staff and end users. Training usually covers operational training for those people who will be responsible for supporting the system as well as training for those end users who will be using the system after its delivery to a production operating environment.

After training has been successfully completed, systems engineers and developers transition the system to its final production environment, where it is intended to be used by its end users and supported by its support and operations staff.

Operations and maintenance

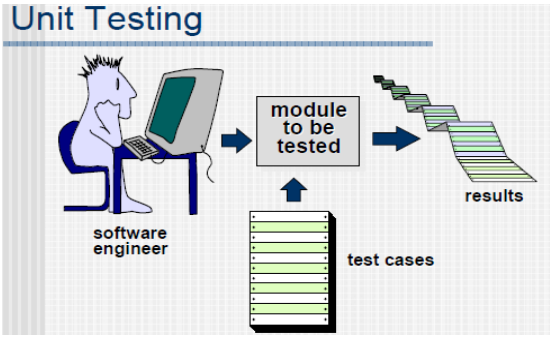
The deployment of the system includes changes and enhancements before the decommissioning or sunset of the system. Maintaining the system is an important aspect of SDLC. As key personnel change positions in the organization, new changes will be implemented. There are two approaches to system development; there is the traditional approach (structured) and object oriented. Information Engineering includes the traditional system approach, which is also called the structured analysis and design technique. The object oriented approach views the information system as a collection of objects that are integrated with each other to make a full and complete information system.

Evaluation

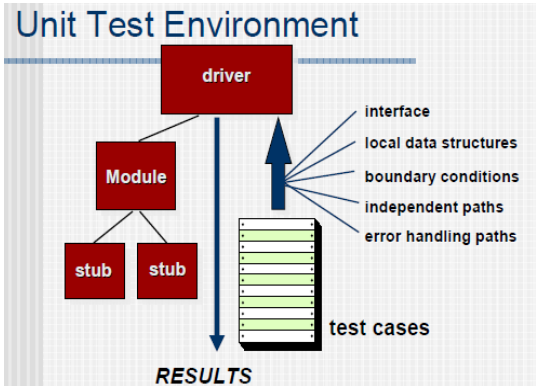
The final phase of the SDLC is to measure the effectiveness of the system and evaluate potential enhancements and improvements.

Different Levels of Testing

Unit Testing



- Module interfaces are tested for proper information flow.
- Local data are examined to ensure that integrity is maintained.
- Boundary conditions are tested.
- Basis (independent) path are tested.
- All error handling paths should be tested.
- Drivers and/or stubs need to be developed to test incomplete software.

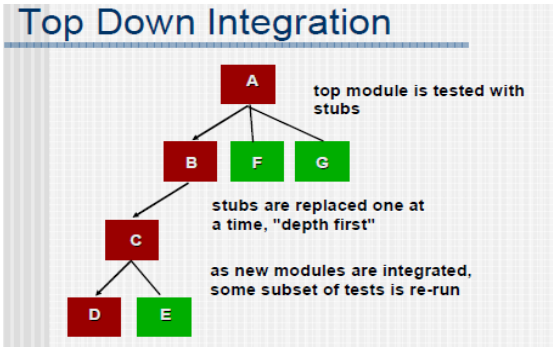


Integration Testing

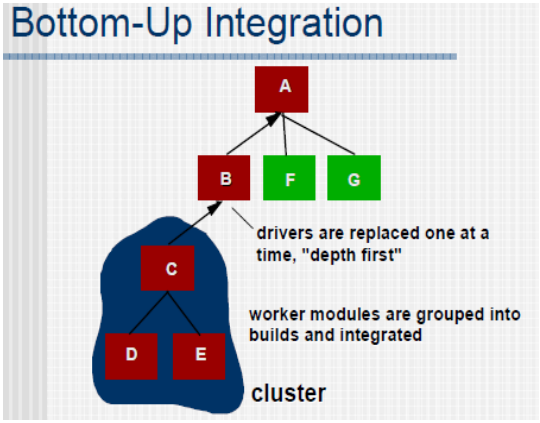
- Sandwich testing uses top-down tests for upper levels of program structure coupled with bottom-up tests for subordinate levels
- Testers should strive to indentify critical modules having the following requirements
- Overall plan for integration of software and the specific tests are documented in a test specification

Integration Testing Strategies

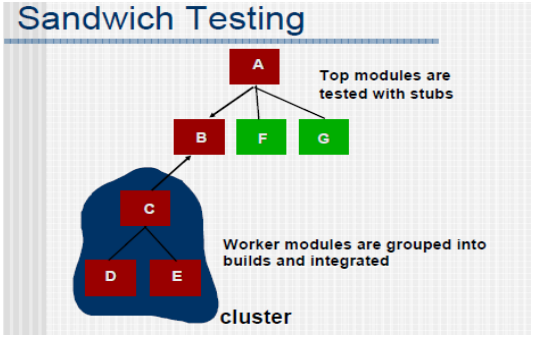
- Top-down integration testing
  1. Main control module used as a test driver and stubs are substitutes for components directly subordinate to it.
  2. Subordinate stubs are replaced one at a time with real components (following the depth-first or breadth-first approach).
  3. Tests are conducted as each component is integrated.
  4. On completion of each set of tests and other stub is replaced with a real component.
  5. Regression testing may be used to ensure that new errors not introduced.



- Bottom-up integration testing
  1. Low level components are combined into clusters that perform a specific software function.
  2. A driver (control program) is written to coordinate test case input and output.
  3. The cluster is tested.
  4. Drivers are removed and clusters are combined moving upward in the program structure.



Both top down and bottom up approach of testing have their pros and cons. Bottom up testing guarantees exhaustive testing in most cases and Top down testing provides effective testing. A balance on the application of both the approaches has to be established for a good testing practice. Sandwich testing applies both the techniques wherever necessary and results in the best test coverage.



- Regression testing – used to check for defects propagated to other modules by changes made to existing program
  1. Representative sample of existing test cases is used to exercise all software functions.

2. Additional test cases focusing software functions likely to be affected by the change.
3. Tests cases that focus on the changed software components.

- Smoke testing
  1. Software components already translated into code are integrated into a build.
  2. A series of tests designed to expose errors that will keep the build from performing its functions are created.
  3. The build is integrated with the other builds and the entire product is smoke tested daily (either top-down or bottom integration may be used).

General Software Test Criteria

- Interface integrity – internal and external module interfaces are tested as each module or cluster is added to the software
- Functional validity – test to uncover functional defects in the software
- Information content – test for errors in local or global data structures
- Performance – verify specified performance bounds are tested

Object-Oriented Test Strategies

- Unit Testing – components being tested are classes not modules
- Integration Testing – as classes are integrated into the architecture regression tests are run to uncover communication and collaboration errors between objects
- Systems Testing – the system as a whole is tested to uncover requirement errors

Object-Oriented Unit Testing

- smallest testable unit is the encapsulated class or object
- similar to system testing of conventional software
- do not test operations in isolation from one another
- driven by class operations and state behavior, not algorithmic detail and data flow across module interface

Object-Oriented Integration Testing

- focuses on groups of classes that collaborate or communicate in some manner
- integration of operations one at a time into classes is often meaningless
- **thread-based testing** – testing all classes required to respond to one system input or event
- **use-based testing** – begins by testing independent classes (classes that use very few server classes) first and the dependent classes that make use of them
- **cluster testing** – groups of collaborating classes are tested for interaction errors
- regression testing is important as each thread, cluster, or subsystem is added to the system

WebApp Testing Strategies

1. WebApp content model is reviewed to uncover errors.
2. Interface model is reviewed to ensure all use-cases are accommodated.
3. Design model for WebApp is reviewed to uncover navigation errors.
4. User interface is tested to uncover presentation errors and/or navigation mechanics problems.
5. Selected functional components are unit tested.
6. Navigation throughout the architecture is tested.
7. WebApp is implemented in a variety of different environmental configurations and the compatibility of WebApp with each is assessed.
8. Security tests are conducted.
9. Performance tests are conducted.
10. WebApp is tested by a controlled and monitored group of end-users (looking for content errors, navigation errors, usability concerns, compatibility issues, reliability, and performance).

Validation Testing

- Focuses on visible user actions and user recognizable outputs from the system
- Validation tests are based on the use-case scenarios, the behavior model, and the event flow diagram created in the analysis model
  - Must ensure that each function or performance characteristic conforms to its specification.
  - Deviations (deficiencies) must be negotiated with the customer to establish a means for resolving the errors.
- Configuration review or audit is used to ensure that all elements of the software configuration have been properly developed, cataloged, and documented to allow its support during its maintenance phase.

Acceptance Testing

- Making sure the software works correctly for intended user in his or her normal work environment.
- Alpha test – version of the complete software is tested by customer under the supervision of the developer at the developer's site
- Beta test – version of the complete software is tested by customer at his or her own site without the developer being present

System Testing

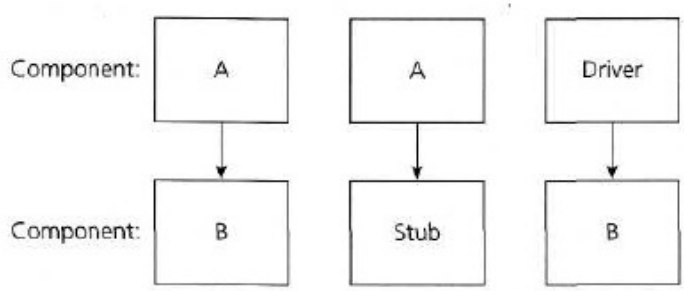
- Series of tests whose purpose is to exercise a computer-based system
- The focus of these system tests cases identify interfacing errors
- Recovery testing – checks the system's ability to recover from failures
- Security testing – verifies that system protection mechanism prevent improper penetration or data alteration
- Stress testing – program is checked to see how well it deals with abnormal resource demands (i.e. quantity, frequency, or volume)

- Performance testing – designed to test the run-time performance of software, especially real-time software
- Deployment (or configuration) testing – exercises the software in each of the environment in which it is to operate

Component Testing

Component testing**,** also known as unit, module and program testing, searches for defects in, and verifies the functioning of software (e.g. modules, programs, objects, classes, etc.) that are separately testable.

Component testing may be done in isolation from the rest of the system depending on the context of the development life cycle and the system. Most often **stubs** and **drivers** are used to replace the missing software and simulate the interface between the software components in a simple manner. A stub is called from the software component to be tested; a driver calls a component to be tested (see Figure). Component testing may include testing of functionality and specific non-functional characteristics such as resource-behavior (e.g. memory leaks), performance or **robustness testing,** as well as structural testing (e.g. decision coverage). Test cases are derived from work products such as the software design or the data model.



Factors for Software Testing

For designing Test Cases the following factors are considered:

1. Correctness
2. Negative
3. User Interface
4. Usability
5. Performance
6. Security
7. Integration
8. Reliability
9. Compatibility

Correctness : Correctness is the minimum requirement of software, the essential purpose of testing. The tester may or may not know the inside details of the software module under test e.g. control flow, data flow etc.

Negative : In this factor we can check what the product it is not supposed to do.

User Interface : In UI testing we check the user interfaces. For example in a web page we may check for a button. In this we check for button size and shape. We can also check the navigation links.

Usability : Usability testing measures the suitability of the software for its users, and is directed at measuring the following factors with which specified users can achieve specified goals in particular environments.

1. **Effectiveness :** The capability of the software product to enable users to achieve specified goals with the accuracy and completeness in a specified context of use.

2. **Efficiency :** The capability of the product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use.

Performance : In software engineering, performance testing is testing that is performed from one perspective to determine how fast some aspect of a system performs under a particular workload.

Performance testing can serve various purposes. It can demonstrate that the system needs performance criteria.

1. **Load Testing:** This is the simplest form of performance testing. A load test is usually conducted to understand the behavior of the application under a specific expected load.

2. **Stress Testing:** Stress testing focuses on the ability of a system to handle loads beyond maximum capacity. System performance should degrade slowly and predictably without failure as stress levels are increased.

3. **Volume Testing:** Volume testing belongs to the group of non-functional values tests. Volume testing refers to testing a software application for a certain data volume. This volume can in generic terms be the database size or it could also be the size of an interface file that is the subject of volume testing.

Security : Process to determine that an Information System protects data and maintains functionality as intended. The basic security concepts that need to be covered by security testing are the following:

1. **Confidentiality :** A security measure which protects against the disclosure of information to parties other than the intended recipient that is by no means the only way of ensuring
2. **Integrity:** A measure intended to allow the receiver to determine that the information which it receives has not been altered in transit other than by the originator of the information.

3. **Authentication:** A measure designed to establish the validity of a transmission, message or originator. Allows a receiver to have confidence that the information it receives originated from a specific known source.

4. **Authorization:** The process of determining that a requester is allowed to receive a service/perform an operation.

Integration : Integration testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested.

Reliability : Reliability testing is to monitor a statistical measure of software maturity over time and compare this to a desired reliability goal.

Compatibility : Compatibility testing of a part of software's non-functional tests. This testing is conducted on the application to evaluate the application's compatibility with the computing environment. Browser compatibility testing can be more appropriately referred to as user experience testing. This requires that the web applications are tested on various web browsers to ensure the following:

1. Users have the same visual experience irrespective of the browsers through which they view the web application.
2. In terms of functionality , the application must behave and respond the same across various browsers.

Different Types of Testing

Test types are introduced as a means of clearly defining the objective of a certain test level for a programme or project. We need to think about differ-ent types of testing because testing the functionality of the component or system may not be sufficient at each level to meet the overall test objectives. Focusing the testing on a specific test objective and, therefore, selecting the appropriate type of test helps making and communicating decisions against test objectives easier.

A **test type** is focused on a particular test objective, which could be the testing of a function to be performed by the component or system; a non-functional quality characteristic, such as reliability or usability; the structure or architecture of the component or system; or related to changes, i.e. con-firming that defects have been fixed (confirmation testing, or re-testing) and looking for unintended changes (regression testing). Depending on its objectives, testing will be organized differently. For example, component testing aimed at performance would be quite different to component testing aimed at achieving decision coverage.

Testing of function (functional testing)

The function of a system (or component) is 'what it does'. This is typically described in a requirements specification, a functional specification, or in use cases. There may be some functions that are 'assumed' to be provided that are not documented that are also part of the requirement for a system, though it is difficult to test against undocumented and implicit requirements. Functional tests are based on these functions, described in documents or understood by the testers and may be performed at all test levels (e.g. test for components may be based on a component specification).

Functional testing considers the specified behavior and is often also referred to as **black-box testing.** This is not entirely true, since black-box testing also includes non-functional testing. **Function** (or functionality) **testing** can, based upon ISO 9126, be done focus-ing on suitability, **interoperability,**

security, accuracy and compliance. Security testing, for example, investigates the functions (e.g. a firewall) relating to detec-tion of threats, such as viruses, from malicious outsiders.

Testing functionality can be done from two perspectives: requirements based or business-process-based.

Requirements-based testing uses a specification of the functional requirements for the system as the basis for designing tests. A good way to start is to use the table of contents of the requirements specification as an initial test inventory or list of items to test (or not to test). We should also prioritize the requirements based on risk criteria (if this is not already done in the specification) and use this to prioritize the tests. This will ensure that the most important and most critical tests are included in the testing effort.

Business-process-based testing uses knowledge of the business processes. Business processes describe the scenarios involved in the day-to-day business use of the system. For example, a personnel and payroll system may have a business process along the lines of: someone joins the company, he or she is paid on a regular basis, and he or she finally leaves the company. Use cases originate from object-oriented development, but are nowadays popular in many development life cycles. They also take the business processes as a starting point, although they start from tasks to be performed by users. Use cases are a very useful basis for test cases from a business perspective.

The techniques used for functional testing are often **specification-based,** but experienced-based techniques can also be used. Test conditions and test cases are derived from the functionality of the component or system. As part of test designing, a model may be developed, such as a process model, state transition model or a plain-language specification.

2.3.2 Testing of software product characteristics (non-functional testing)

A second target for testing is the testing of the quality characteristics, or non-functional attributes of the system (or component or integration group). Here we are interested in how well or how fast something is done. We are testing something that we need to measure on a scale of measurement, for example time to respond.

Non-functional testing, as functional testing, is performed at all test levels. Non-functional testing includes, but is not limited to, **performance testing, load testing, stress testing,** usability testing, maintainability testing, reliability testing and portability testing. It is the testing of 'how well' the system works.

Many have tried to capture software quality in a collection of characteristics and related sub-characteristics. In these models some elementary characteristics keep on reappearing, although their place in the hierarchy can differ. The International Organization for Standardization (ISO) has defined a set of quality characteristics [ISO/IEC 9126, 2001]. The ISO 9126 standard defines six quality characteristics and the subdivision of each quality characteristic into a number of sub-characteristics.
The characteristics and their sub-characteristics are, respectively:

Functionality, consists of five sub-characteristics: suitability, accuracy, security, interoperability and compliance; this characteristic deals with functional testing:

- **Reliability,** which is defined further into the sub-characteristics maturity (robustness), fault-tolerance, recoverability and compliance;
- **Reusability,** which is divided into the sub-characteristics understandability, learnability, operability, attractiveness and compliance;
- **Efficiency,** which is divided into time behaviour (performance), resource utilization and compliance;
- **Maintainability,** which consists of five sub-characteristics: analyzability, changeability, stability, testability and compliance;
- **Portability,** which also consists of five sub-characteristics: adaptability, installability, co-existence, replaceability and compliance.

2.3.3 Testing of software structure/architecture (structural testing)

The third target of testing is the structure of the system or component. If we are talking about the structure of a system, we may call it the system architecture. Structural testing is often referred to as **'white-box'** or 'glass-box' because we are interested in what is happening 'inside the box'.

Structural testing is most often used as a way of measuring the thoroughness of testing through the coverage of a set of structural elements or coverage items. It can occur at any test level, although is it true to say that it tends to be mostly applied at component and integration and generally is less likely at higher test levels, except for business-process testing. At component integration level it may be based on the architecture of the system, such as a calling hierarchy. A system, system integration or acceptance testing test basis could be a business model or menu structure.

At component level, and to a lesser extent at component integration testing, there is good tool support to measure **code coverage.** Coverage measurement tools assess the percentage of executable elements (e.g. statements or decision outcomes) that have been exercised (i.e. covered) by a **test suite.** If coverage is not 100%, then additional tests may need to be written and run to cover those parts that have not yet been exercised. This of course depends on the exit criteria. The techniques used for structural testing are structure-based techniques, also referred to as **white-box techniques.** Control flow models are often used to support structural testing.

Static Testing

Static testing is a very suitable method for improving the quality of software work products. This applies primarily to the assessed products themselves, but it is also important that the quality improvement is not achieved once but has a more structural character. The feedback from the static testing process to the development process allows for process improvement, which supports the avoidance of similar errors being made in the future.

Reviews

Reviews vary from very **informal** to **formal** (i.e. well structured and regulated). Although inspection is perhaps the most documented and formal review technique, it is certainly not the only one. The formality of a review process is related to factors such as the maturity of the development process, any legal or regulatory requirements or the need for an audit trail. In practice the informal review is perhaps the most common type of review. Informal reviews are applied at various times during the early stages in the life cycle of a document. A two-person team can conduct an informal review, as the author can ask a colleague to review a document or code. In later stages these reviews often involve more people and a meeting. This normally involves peers of the author, who try to find defects in the document under review and discuss these defects in a review meeting. The goal is to help the author and to improve the quality of the document. Informal reviews come in various shapes and forms, but all have one characteristic in common - they are not documented.

Phases of a formal review

In contrast to informal reviews, formal reviews follow a formal process. A typical formal review process consists of six main steps:

- Planning
- Kick-off
- Preparation
- Review meeting
- Rework
- Follow-up.

Planning

The review process for a particular review begins with a 'request for review' by the author to the **moderator** (or inspection leader). A moderator is often assigned to take care of the scheduling (dates, time, place and invitation) of the review. On a project level, the project planning needs to allow time for review and rework activities, thus providing engineers with time to thoroughly participate in reviews. For more formal reviews, e.g. inspections, the moderator always performs an entry check and defines at this stage formal exit criteria. The entry check is carried out to ensure that the reviewers' time is not wasted on a document that is not ready for review. A document containing too many obvious mistakes is clearly not ready to enter a formal review process and it could even be very harmful to the review process. It would possibly de-motivate both reviewers and the author. Also, the review is most likely not effective because the numerous obvious and minor defects will conceal the major defects.

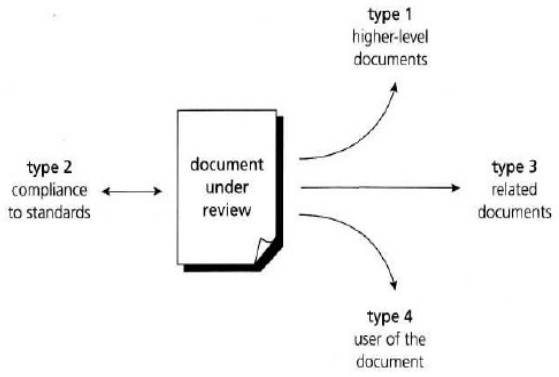
Although more and other **entry criteria** can be applied, the following can be regarded as the minimum set for performing the entry check. A short check of a product sample by the moderator (or expert) does not reveal a large number of major defects. For example, after 30 minutes of checking, no more than 3 major defects are found on a single page or fewer than 10 major defects in total in a set of 5 pages.

- The document to be reviewed is available with line numbers.

- The document has been cleaned up by running any automated checks that apply.
- References needed for the inspection are stable and available.
- The document author is prepared to join the review team and feels confident with the quality of the document.

If the document passes the entry check, the moderator and author decide which part of the document to review. Because the human minds can comprehend a limited set of pages at one time, the number should not be too high. The maximum number of pages depends, among other things, on the objective, review type and document type and should be derived from practical experiences within the organization. For a review, the maximum size is usually between 10 and 20 pages. In formal inspection, only a page or two may be looked at in depth in order to find the most serious defects that are not obvious.

After the document size has been set and the pages to be checked have been selected, the moderator determines, in co-operation with the author, the com-position of the review team. The team normally consists of four to six participants, including moderator and author. To improve the effectiveness of the review, different roles are assigned to each of the participants. These roles help the **reviewers** focus on particular types of defects during checking. This reduces the chance of different reviewers finding the same defects. The moderator assigns the roles to the reviewers. Figure shows the different roles within a review. The roles represent views of the document under review.



Within reviews the following focuses can be identified:

- focus on higher-level documents, e.g. does the design comply to the requirements;
- focus on standards, e.g. internal consistency, clarity, naming conventions, templates;
- focus on related documents at the same level, e.g. interfaces between soft ware functions;
- focus on usage, e.g. for testability or maintainability.

The author may raise additional specific roles and questions that have to be addressed. The moderator has the option to also fulfil a role, alongside the task of being a review leader. Checking the document improves the moderator's ability to lead the meeting, because it ensures better understanding. Furthermore, it improves the review efficiency because the moderator replaces an engineer that would other-wise have to check the document and attend the meeting. It is recommended that the moderator take the role of checking compliance to standards, since this tends to be a highly objective role, which leads to less discussion of the defects found.

Kick-off

An optional step in a review procedure is a kick-off meeting. The goal of this meeting is to get everybody on the same wavelength regarding the document under review and to commit to the time that will be spent on checking. Also the result of the entry check and defined exit criteria are discussed in case of a more formal review. In general a kick-off is highly recommended since there is a strong positive effect of a kick-off meeting on the motivation of reviewers and thus the effectiveness of the review process. At customer sites, we have measured results up to 70% more major defects found per page as a result of per-forming a kick-off.

During the kick-off meeting the reviewers receive a short introduction on the objectives of the review and the documents. The relationships between the document under review and the other documents (sources) are explained, especially if the number of related documents is high. Role assignments, checking rate, the pages to be checked, process changes and possible other questions are also discussed during this meeting. Of course the distribution of the document under review, source documents and other related documentation, can also be done during the kick-off.

Preparation

The participants work individually on the document under review using the related documents, procedures, rules and checklists provided. The individual participants identify defects, questions and comments, according to their understanding of the document and role. All issues are recorded, preferably using a logging form. Spelling mistakes are recorded on the document under review but not mentioned during the meeting. The annotated document will be given to the author at the end of the logging meeting. Using checklists during this phase can make reviews more effective and efficient, for example a specific checklist based on perspectives such as user, maintainer, tester or operations, or a checklist for typical coding problems.

A critical success factor for a thorough preparation is the number of pages checked per hour. This is called the checking rate. The optimum checking rate is the result of a mix of factors, including the type of document, its com-plexity, the number of related documents and the experience of the reviewer. Usually the checking rate is in the range of five to ten pages per hour, but may be much less for formal inspection, e.g. one page per hour. During preparation, participants should not exceed this criterion. By collecting data and measuring the review process, company-specific criteria for checking rate and document size (see planning phase) can be set, preferably specific to a document type.

Review meeting

The meeting typically consists of the following elements (partly depending on the review type): logging phase, discussion phase and decision phase.

During the logging phase the issues, e.g. defects, that have been identified during the preparation are mentioned page by page, reviewer by reviewer and are logged either by the author or by a scribe. A separate person to do the logging (a scribe) is especially useful for formal review types such as an

inspection. To ensure progress and efficiency, no real discussion is allowed during the logging phase. If an issue needs discussion, the item is logged and then handled in the discussion phase. A detailed discussion on whether or not an issue is a defect is not very meaningful, as it is much more efficient to simply log it and proceed to the next one. Furthermore, in spite of the opinion of the team, a discussed and discarded defect may well turn out to be a real one during rework.

Every defect and its severity should be logged. The participant who identifies the defect proposes the severity. Severity classes could be:

Critical: defects will cause downstream damage; the scope and impact of the defect is beyond the document under inspection.

Major, defects could cause a downstream effect (e.g. a fault in a design can result in an error in the implementation).

Minor, defects are not likely to cause downstream damage (e.g. non-compliance with the standards and templates).

During the logging phase the focus is on logging as many defects as possible within a certain timeframe. To ensure this, the moderator tries to keep a good logging rate (number of defects logged per minute). In a well-led and disciplined formal review meeting, the logging rate should be between one and two defects logged per minute.

For a more formal review, the issues classified as discussion items will be handled during this meeting phase. Informal reviews will often not have a separate logging phase and will start immediately with discussion. Participants can take part in the discussion by bringing forward their comments and reasoning. As chairman of the discussion meeting, the moderator takes care of people issues. For example, the moderator prevents discussions from getting too personal, rephrases remarks if necessary and calls for a break to cool down 'heated' discussions and/or participants.

Reviewers who do not need to be in the discussion may leave, or stay as a learning exercise. The moderator also paces this part of the meeting and ensures that all discussed items either have an outcome by the end of the meeting, or are defined as an action point if a discussion cannot be solved during the meeting. The outcome of discussions is documented for future reference.

At the end of the meeting, a decision on the document under review has to be made by the participants, sometimes based on formal **exit criteria.** The most important exit criterion is the average number of critical and/or major defects found per page (e.g. no more than three critical/major defects per page). If the number of defects found per page exceeds a certain level, the document must be reviewed again, after it has been reworked. If the document complies with the exit criteria, the document will be checked during follow-up by the moderator or one or more participants. Subsequently, the document can leave the review process.

If a project is under pressure, the moderator will sometimes be forced to skip re-reviews and exit with a defect-prone document. Setting, and agreeing, quantified exit level criteria help the moderator to make firm decisions at all times. In addition to the number of defects per page, other exit criteria are used that measure the thoroughness of the review process, such as ensuring that all pages have been checked at

the right rate. The average number of defects per page is only a valid quality indicator if these process criteria are met.

Rework

Based on the defects detected, the author will improve the document under review step by step. Not every defect that is found leads to rework. It is the author's responsibility to judge if a defect has to be fixed. If nothing is done about an issue for a certain reason, it should be reported to at least indicate that the author has considered the issue. Changes that are made to the document should be easy to identify during follow-up. Therefore the author has to indicate where changes are made (e.g. using 'Track changes' in word-processing software).

Follow-up

The moderator is responsible for ensuring that satisfactory actions have been taken on all (logged) defects, process improvement suggestions and change requests. Although the moderator checks to make sure that the author has taken action on all known defects, it is not necessary for the moderator to check all the corrections in detail. If it is decided that all participants will check the updated document, the moderator takes care of the distribution and collects the feedback. For more formal review types the moderator checks for compliance to the exit criteria.

In order to control and optimize the review process, a number of measurements are collected by the moderator at each step of the process. Examples of such measurements include number of defects found; number of defects found per page, time spent checking per page, total review effort, etc. It is the responsibility of the moderator to ensure that the information is correct and stored for future analysis.

Test Case Design for OO Software

- Each test case should be uniquely identified and be explicitly associated with a class to be tested
- State the purpose of each test
- List the testing steps for each test including:
- list of states to test for each object involved in the test
- list of messages and operations to exercised as a consequence of the test
- list of exceptions that may occur as the object is tested
- list of external conditions needed to be changed for the test
- supplementary information required to understand or implement the test

Testing Surface Structure and Deep Structure

- Testing surface structure (exercising the structure observable by end-user, this often involves observing and interviewing users as they manipulate system objects)

- Testing deep structure (exercising internal program structure - the dependencies, behaviors, and communications mechanisms established as part of the system and object design)

Testing Methods Applicable at The Class Level

Random testing - requires large numbers data permutations and combinations, and can be inefficient

- Identify operations applicable to a class o Define constraints on their use
- Identify a minimum test sequence
- Generate a variety of random test sequences.

Partition testing - reduces the number of test cases required to test a class

- state-based partitioning - tests designed in way so that operations that cause state changes are tested separately from those that do not.
- attribute-based partitioning - for each class attribute, operations are classified according to those that use the attribute, those that modify the attribute, and
- those that do not use or modify the attribute
- category-based partitioning - operations are categorized according to the function they perform: initialization, computation, query, termination

Fault-based testing

- best reserved for operations and the class level
- uses the inheritance structure
- tester examines the OOA model and hypothesizes a set of plausible defects that may be encountered in operation calls and message connections and builds appropriate test cases
- misses incorrect specification and errors in subsystem interactions

Inter-Class Test Case Design : Test case design becomes more complicated as integration of the OO system begins – testing of collaboration between classes

Multiple class testing

- for each client class use the list of class operators to generate random test sequences that send messages to other server classes
- for each message generated determine the collaborator class and the corresponding server object operator
- for each server class operator (invoked by a client object message) determine the message it transmits

OO Test Design Issues

White-box testing methods can be applied to testing the code used to implement class operations, but not much else. Black-box testing methods are appropriate for testing OO systems. Object-oriented

programming brings additional testing concerns: classes may contain operations that are inherited from super classes; subclasses may contain operations that were redefined rather than inherited.

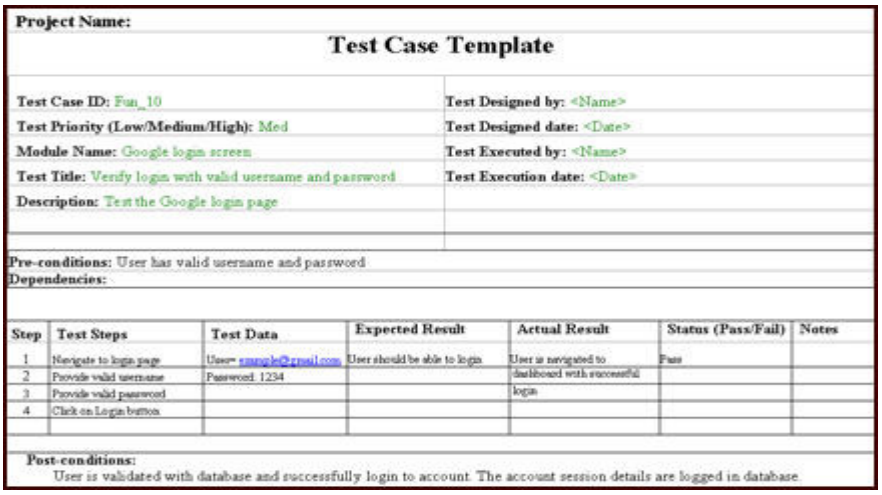
Designing test cases using MS-Excel

Test case formats vary by organization. But using a standard test case format for writing test cases is one step closer to setting up testing process on your project. It also minimizes ad-hoc testing done without proper test case documentation. But even if you use standard templates you need to setup test cases writing, review and approval, test execution and most importantly test report preparation process, all using manual methods. Also there is a process to review test cases by business team then you must format these test cases in a template agreed by both the parties.

Below are the standard fields of sample test case template:

Test case ID: Unique ID for each test case. Follow some convention to indicate types of test. E.g. 'TC_UI_1' indicating 'user interfaces test case #1'.

Test priority (Low/Medium/High): This is useful while test execution. Test priority for business rules and functional test cases can be medium or higher whereas minor user interface cases can be low priority. Test priority should be set by reviewer.
Module Name – Mention name of main module or sub module.
Test Designed By: Name of tester
Test Designed Date: Date when wrote
Test Executed By: Name of tester who executed this test. To be filled after test execution.
Test Execution Date: Date when test executed.
Test Title/Name: Test case title. E.g. verify login page with valid username and password.
Test Summary/Description: Describe test objective in brief.
Pre-condition: Any prerequisite that must be fulfilled before execution of this test case. List all pre-conditions in order to successfully execute this test case.
Dependencies: Mention any dependencies on other test cases or test requirement.
Test Steps: List all test execution steps in detail. Write test steps in the order in which these should be executed. Make sure to provide as much details as you can. Tip – to efficiently manage test case with lesser number of fields use this field to describe test conditions, test data and user roles for running test.
Test Data: Use of test data as an input for this test case. You can provide different data sets with exact values to be used as an input.
Expected Result: What should be the system output after test execution? Describe the expected result in detail including message/error that should be displayed on screen.
Post-condition: What should be the state of the system after executing this test case?
Actual result: Actual test result should be filled after test execution. Describe system behavior after test execution.



Project Name:

Test Case Template

Test Case ID: Fun_10	**Test Designed by:** <Name>
Test Priority (Low/Medium/High): Med	**Test Designed date:** <Date>
Module Name: Google login screen	**Test Executed by:** <Name>
Test Title: Verify login with valid username and password	**Test Execution date:** <Date>
Description: Test the Google login page	

Pre-conditions: User has valid username and password
Dependencies:

Step	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Navigate to login page	User= example@gmail.com	User should be able to login	User is navigated to	Pass	
2	Provide valid username	Password: 1234		dashboard with successful		
3	Provide valid password			login		
4	Click on Login button					

Post-conditions:
User is validated with database and successfully login to account. The account session details are logged in database

Fig. sample testcase template

Status (Pass/Fail): If actual result is not as per the expected result mark this test as failed. Otherwise update as passed.

Notes/Comments/Questions: To support above fields if there are some special conditions which can't be described in any of the above fields or there are questions related to expected or actual results mention those here.

Add following fields if necessary:

Defect ID/Link: If test status is fail, then include the link to defect log or mention the defect number.

Test Type/Keywords: This field can be used to classify tests based on test types. E.g. functional, usability, business rules etc.

Requirements: Requirements for which this test case is being written. Preferably the exact section number of the requirement doc.

Attachments/References: This field is useful for complex test scenarios. To explain test steps or expected result using a visio diagram as a reference. Provide the link or location to the actual path of the diagram or document.

Automation? (Yes/No): Whether this test case is automated or not. Useful to track automation status when test cases are automated.

Your Company LOGO	Project Name:		Test Designed by:						
	Module Name:		Test Designed date:						
	Release Version:		Test Executed by:						
			Test Execution date:						
Pre-condition									
Dependencies:									
Test Priority									
Test Case#	Test Title	Test Summary	Test Steps	Test Data	Expected Result	Post-condition	Actual Result	Status	Notes

Fig. test case template