UNIT I TESTING BASICS

1.1 Testing as an engineering activity

This is an exciting time to be a software developer. Software systems are becoming more challenging to build. They are playing an increasingly important role in society. People with software development skills are in demand. New methods, techniques, and tools are becoming available to support development and maintenance tasks. Because software now has such an important role in our lives both economically and socially, there is pressure for software professionals to focus on quality issues. Poor quality software that can cause loss of life or property is no longer acceptable to society. Failures can result in catastrophic losses. Conditions demand software development staffs with interest and training in the areas of software product and process quality. Highly qualified staff ensure that software products are built on time, within budget, and are of the highest quality with respect to attributes such as reliability, correctness, usability, and the ability to meet all user requirements.

Using an engineering approach to software development implies that:

- the development process is well understood;
- projects are planned;
- life cycle models are defined and adhered to;
- standards are in place for product and process;
- measurements are employed to evaluate product and process quality;
- components are reused;
- validation and verification processes play a key role in quality determination;
- engineers have proper education, training, and certification.

1.2 Role of process in software quality

The need for software products of high quality has pressured those in the profession to identify and quantify quality factors such as usability, testability, maintainability, and reliability, and to identify engineering practices that support the production of quality products having these favorable attributes. Among the practices identified that contribute to the development of high-quality software are project planning, requirements management, development of formal specifications, structured design with use of information hiding and encapsulation, design and code reuse,inspections and reviews, product and process measures, education and training of software professionals, development and application of CASE tools, use of effective testing techniques, and integration of testing activities into the entire life cycle. In addition to identifying these individual best technical and managerial practices, software researchers realized that it was important to integrate them within the context of a high-quality software development process.

Process, in the software engineering domain, is the set of methods, practices, standards, documents, activities, policies, and procedures that software engineers use to develop and maintain a software system and its associated artifacts, such as project and test plans, design documents, code, and manuals.

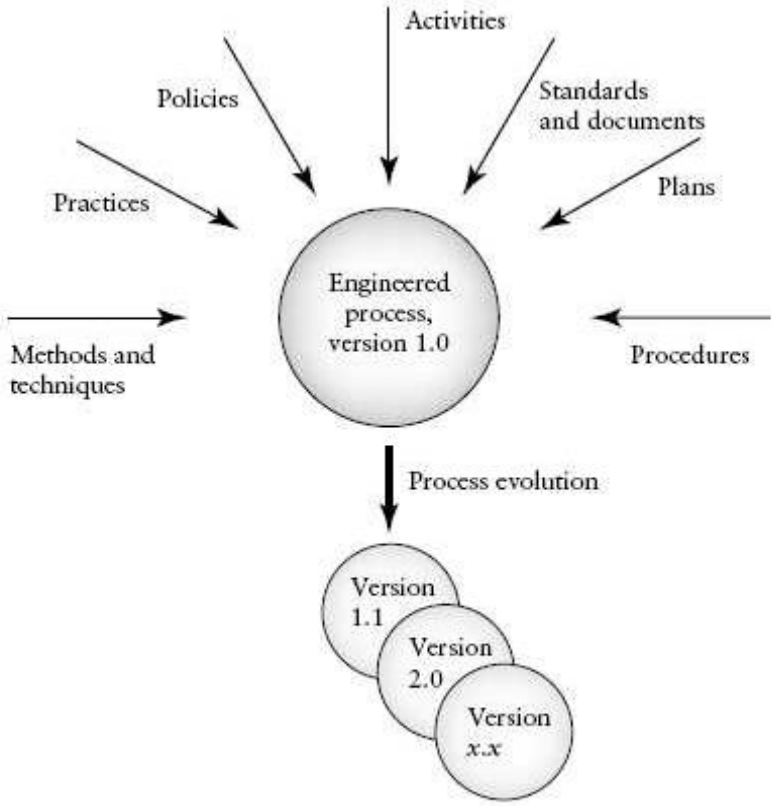


FIG. 1.2

Components of an engineered process.

It also was clear that adding individual practices to an existing software development process in an ad hoc way was not satisfactory. The software development process, like most engineering artifacts, must be engineered. That is, it must be designed, implemented, evaluated, and maintained. As in other engineering disciplines, a software development process must evolve in a consistent and predictable manner, and the best technical and managerial practices must be integrated in a systematic way. These models allow an organization to evaluate its current software process and to capture an understanding of its state. Strong support for incremental process improvement is provided by the models, consistent with historical process evolution and the application of quality principles. The models have received much attention from industry, and resources have been invested in process improvement efforts with many successes recorded.

All the software process improvement models that have had wide acceptance in industry are high-level models, in the sense that they focus on the software process as a whole and do not offer adequate support to evaluate and improve specific software development sub processes such as design and testing. Most software engineers would agree that testing is a vital component of a quality software process, and is one of the most challenging and costly activities carried out during software development and maintenance.

1.3 Testing as a process

The software development process has been described as a series of phases, procedures, and steps that result in the production of a software product. Embedded within the software development process are several other processes including testing. Some of these are shown in Figure 1.3. Testing itself is related to two other processes called verification and validation as shown in Figure 1.3.

Validation is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements.

Validation is usually associated with traditional execution-based testing, that is, exercising the code with test cases.

Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [11].

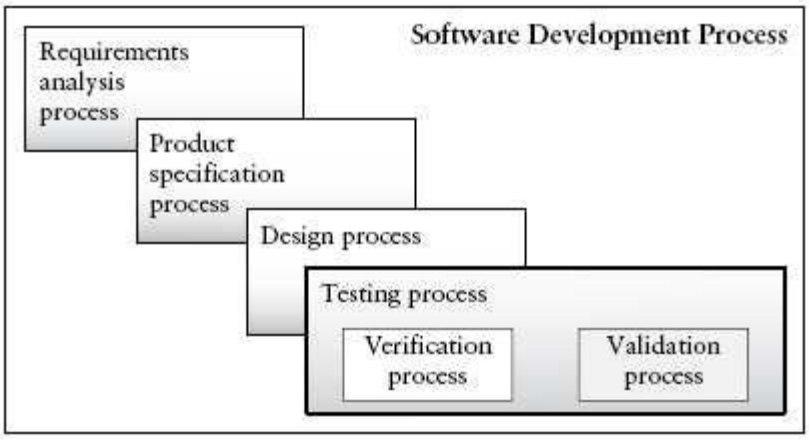


FIG. 1.3
Example processes embedded in the software development process.

Verification is usually associated with activities such as inspections and reviews of software deliverables. Testing itself has been defined in several ways. Two definitions are shown below.

Testing is generally described as a group of procedures carried out to evaluate some aspect of a piece of software.

Testing can be described as a process used for revealing defects in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes.

Note that these definitions of testing are general in nature. They cover both validation and verification activities, and include in the testing domain all of the following: technical reviews, test planning, test tracking, test case design, unit test, integration test, system test, acceptance test, and usability test. The definitions also describe testing as a dual-purpose process—one that reveals defects, as well as one that is used to evaluate quality attributes of the software such as reliability, security, usability, and correctness.

Also note that testing and debugging, or fault localization, are two very different activities. The debugging process begins *after* testing has been carried out and the tester has noted that the software is not behaving as specified.

Debugging, or fault localization is the process of (1) locating the fault or defect, (2) repairing the code, and (3) retesting the code.

Testing as a process has economic, technical and managerial aspects. Economic aspects are related to the reality that resources and time are available to the testing group on a limited basis. In fact, complete testing is in many cases not practical because of these economic constraints. An organization must structure its testing process so that it can deliver software on time and within budget, and also satisfy the client's requirements. The technical aspects of testing relate to the techniques, methods, measurements, and tools used to insure that the software under test is as defect-free and reliable as possible for the conditions and constraints under which it must operate. Testing is a process, and as a process it must managed. Minimally that means that an organizational policy for testing must be defined and documented. Testing procedures and steps must be defined and documented. Testing must be planned, testers should be trained, the process should have associated quantifiable goals that can be measured and monitored. Testing as a process should be able to evolve to a level where there are mechanisms in place for making continuous improvements.

1.4 Basic definitions

Errors

An error is a mistake, misconception, or misunderstanding on the part of a software developer.
In the category of developer we include software engineers, programmers, analysts, and testers. For example, a developer may misunderstand a design notation, or a programmer might type a variable name incorrectly.

Faults (Defects)

A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification.

Faults or defects are sometimes called "bugs." Use of the latter term trivializes the impact faults have on software quality. Use of the term "defect" is also associated with software artifacts such as requirements and design documents. Defects occurring in these artifacts are also caused by errors and are usually detected in the review process.

Failures

A failure is the inability of a software system or component to perform its required functions within specified performance requirements **.**

During execution of a software component or system, a tester, developer, or user observes that it does not produce the expected results. In some cases a particular type of misbehavior indicates a certain type of fault is

Test case

A test case in a practical sense is a test-related item which contains the following information:
1. *A set of test inputs.* These are data items received from an external source by the code under test. The external source can be hardware, software, or human.
2. *Execution conditions.* These are conditions required for running the test, for example, a certain state of a database, or a configuration of a hardware device.
3. *Expected outputs.* These are the specified results to be produced by the code under test.

Test

A test is a group of related test cases, or a group of related test cases and test procedures**.**

Test Oracle

A test oracle is a document, or piece of software that allows testers to determine whether a test has been passed or failed.

A program, or a document that produces or specifies the expected outcome of a test, can serve as an oracle.Examples include a specification (especially one that contains pre- and post conditions), a design document, and a set of requirements. Other sources are regression test suites. The suites usually contain components with correct results for previous versions of the software. If some of the functionality in the new version overlaps the old version, the appropriate oracle information can be extracted. A working trusted program can serve as its own oracle in a situation where it is being ported to a new environment. In this case its intended behavior should not change in the new environment.

Test Bed

A test bed is an environment that contains all the hardware and software needed to test a software component or a software system.This includes the entire testing environment, for example, simulators, emulators,memory checkers, hardware probes, software tools, and all other items needed to support execution of the tests.

Software Quality

1. Quality relates to the degree to which a system, system component, or process meets specified requirements.
2. Quality relates to the degree to which a system, system component, or process meets customer or user needs, or expectations.

In order to determine whether a system, system component, or process is of high quality we use what are called quality attributes. the degree to which they possess a given quality attribute with quality metrics.

Quality metric

A metric is a quantitative measure of the degree to which a system, system component,or process possesses a given attribute.

There are product and process metrics. A very commonly used example of a software product metric is software size, usually measured in lines of code (LOC). Two examples of commonly used process metrics are costs and time required for a given task. Quality metrics are a special kind of metric.

A quality metric is a quantitative measurement of the degree to which an item possesses a given quality attribute.

Some examples of quality attributes with brief explanations are the following:

correctness—the degree to which the system performs its intended function
reliability—the degree to which the software is expected to perform its required functions under stated conditions for a stated period of time
usability—relates to the degree of effort needed to learn, operate, prepare input, and interpret output of the software
integrity—relates to the system's ability to withstand both intentional and accidental attacks
portability—relates to the ability of the software to be transferred from one environment to another
maintainability—the effort needed to make changes in the software
interoperability—the effort needed to link or couple one system to another.

Another quality attribute that should be mentioned here is testability.
1. the amount of effort needed to test the software to ensure it performs according to specified requirements (relates to number of test cases needed),
2. the ability of the software to reveal defects under testing conditions (some software is designed in such a way that defects are well hidden during ordinary testing conditions).
Testers must work with analysts, designers and, developers throughout the software life system to ensure that testability issues are addressed.

Software Quality Assurance Group

The software quality assurance (SQA) group in an organization has ties to quality issues. The group serves as the customers' representative and advocate. Their responsibility is to look after the customers' interests.

The software quality assurance (SQA) group is a team of people with the necessary training and skills to ensure that all necessary actions are taken during the development process so that the resulting software conforms to established technical requirements.

Review

A review is a group meeting whose purpose is to evaluate a software artifact or a set of software artifacts.

The composition of a review group may consist of managers, clients,developers, testers and other personnel depending on the type of artifact under review. A special type of review called an audit is usually conducted by a Software Quality Assurance group for the purpose of assessing compliance with specifications, and/or standards, and/or contractual agreements.

1.5 Software testing principles

Principles play an important role in all engineering disciplines and are usually introduced as part of an educational background in each branch of engineering. Figure 1.1 shows the role of basic principles in various engineering disciplines. Testing principles are important to test specialists/ engineers because they provide the foundation for developing testing knowledge and acquiring testing skills. They also provide guidance for defining testing activities as performed in the practice of a test specialist.A principle can be defined as:

1. a general or fundamental, law, doctrine, or assumption;
2. a rule or code of conduct;
3. the laws or facts of nature underlying the working of an artificial device.

Extending these three definitions to the software engineering domain we can say that software engineering principles refer to laws, rules, or doctrines that relate to software systems, how to build them, and how they behave. In the software domain, principles may also refer to rules or codes of conduct relating to professionals who design, develop, test, and maintain software systems. Testing as a component of the software engineering discipline also has a specific set of principles that serve as guidelines for the tester. They guide testers in defining how to test software systems, and provide rules of conduct for testers as professionals. Glenford Myers has outlined such a set of *execution-based* testing principles in his pioneering book, *The Art of Software Testing* [9]. Some of these principles are described below. Principles 1-8, and 11 are derived directly from Myers' original set. The author has reworded these principles, and also has made modifications to the original set to reflect the evolution of testing from an art, to a quality-related process within the context of an engineering discipline. Note that the principles as stated below only relate to execution-based testing. Principles relating to reviews, proof of correctness, and certification as testing activities are not covered.

Principle 1. Testing is the process of exercising a software component using a selected set of test cases, with the intent of (i) revealing defects, and (ii) evaluating quality.

Software engineers have made great progress in developing methods to prevent and eliminate defects. However, defects do occur, and they have a negative impact on software quality. Testers need to detect these defects before the software becomes operational. This principle supports testing as an execution-based activity to detect defects. It also supports the separation of testing from debugging since the intent of the latter is to locate defects and repair the software. The term "software component" is used in this context to represent any unit of software ranging in size and complexity from an individual procedure or method, to an entire software system. The term "defects" as used in this and in subsequent principles represents any deviations in the software that have a negative impact on its functionality, performance, reliability, security, and/or any other of its specified quality attributes.

Principle 2. When the test objective is to detect defects, then a good test case is one that has a high probability of revealing a yetundetected defect(s).

Principle 2 supports careful test design and provides a criterion with which to evaluate test case design and the effectiveness of the testing effort when the objective is to detect defects. It requires the tester to consider the goal for each test case, that is, which specific type of defect is to be detected by the test case. In this way the tester approaches testing in the same way a scientist approaches an experiment. In the case of the scientist there is a hypothesis involved that he/she wants to prove or disprove by means of the experiment. In the case of the tester, the hypothesis is related to the suspected occurrence of specific types of defects. The goal for the test is to prove/disprove the hypothesis, that is, determine if the specific defect is present/absent. Based on the hypothesis, test inputs are selected, correct outputs are determined, and the test is run. Results are analyzed to prove/disprove the hypothesis. The reader should realize that many resources are invested in a test, resources for designing the test cases, running the tests, and recording and analyzing results. A tester can justify the expenditure of the resources by careful test design so that principle 2 is supported.

Principle 3. Test results should be inspected meticulously.

Testers need to carefully inspect and interpret test results. Several erroneous and costly scenarios may occur if care is not taken. For example: A failure may be overlooked, and the test may be granted a "pass" status when in reality the software has failed the test. Testing may continue based on erroneous test results. The defect may be revealed at some later stage of testing, but in that case it may be more costly and difficult to locate and repair.

- A failure may be suspected when in reality none exists. In this case the test may be granted a "fail" status. Much time and effort may be spent on trying to find the defect that does not exist. A careful reexamination of the test results could finally indicate that no failure has occurred.
- The outcome of a quality test may be misunderstood, resulting in unnecessary rework, or oversight of a critical problem.

Principle 4. A test case must contain the expected output or result.

It is often obvious to the novice tester that test inputs must be part of a test case. However, the test case is of no value unless there is an explicit statement of the expected outputs or results, for example, a specific variable value must be observed or a certain panel button that must light up. Expected outputs allow the tester to determine (i) whether a defect has been revealed, and (ii) pass/fail status for the test. It is very important to have a correct statement of the output so that needless time is not spent due to misconceptions about the outcome of a test. The specification of test inputs and outputs should be part of test design activities. In the case of testing for quality evaluation, it is useful for quality goals to be expressed in quantitative terms in the requirements document if possible, so that testers are able to compare actual software attributes as determined by the tests with what was specified.

Principle 5. Test cases should be developed for both valid and invalid input conditions.

A tester must not assume that the software under test will always be provided with valid inputs. Inputs may be incorrect for several reasons. For example, software users may have misunderstandings, or lack information about the nature of the inputs. They often make typographical errors even when complete/correct information is available. Devices may also provide invalid inputs due to erroneous conditions and malfunctions. Use of test cases that are based on invalid inputs is very useful for revealing defects since they may exercise the code in unexpected ways and identify unexpected software behavior. Invalid inputs also help developers and testers evaluate the robustness of the software, that is, its ability to recover when unexpected events occur (in this case an erroneous input). Principle 5 supports the need for the independent test group called for in Principle 7 for the following reason. The developer of a software component may be biased in the selection of test inputs for the component and specify only valid inputs in the test cases to demonstrate that the software works correctly. An independent tester is more apt to select invalid inputs as well.

Principle 6. The probability of the existence of additional defects in a software component is proportional to the number of defects already detected in that component.

What this principle says is that the higher the number of defects already detected in a component, the more likely it is to have additional defects when it undergoes further testing. For example, if there are two components A and B, and testers have found 20 defects in A and 3 defects in B, then the probability of the existence of additional defects in A is higher than B. This empirical observation may be due to several causes. Defects often occur in clusters and often in code that has a high degree of complexity and is poorly designed. In the case of such components developers and testers need to decide whether to disregard the current version of the component and work on a redesign, or plan to expend additional testing resources on this component to insure it meets its requirements. This issue is especially important for components that implement mission or safety critical functions.

Principle 7. Testing should be carried out by a group that is independent of the development group.

This principle holds true for psychological as well as practical reasons. It is difficult for a developer to admit or conceive that software he/she has created and developed can be faulty. Testers must realize that (i) developers have a great deal of pride in their work, and (ii) on a practical level it may be difficult for them to conceptualize where defects could be found. Even when tests fail, developers often have difficulty in locating the defects since their mental model of the code may overshadow their view of code as it exists in actuality. They may also have misconceptions or misunderstandings concerning the requirements and specifications relating to the software. The requirement for an independent testing group can be interpreted by an organization in several ways. The testing group could be implemented as a completely separate functional entity in the organization. Alternatively, testers could be members of a Software Quality Assurance Group, or even be a specialized part of the development group, but in the latter case especially, they need the capability to be objective. Reporting management that is separate from development can support their objectivity and independence. As a member of any of these groups, the principal duties and training of the testers should lie in testing rather than in development. Finally, independence of the testing group does not call for an adversarial relationship between developers and testers. The testers should not play "gotcha" games with developers. The groups need to cooperate so that software of the highest quality is released to the customer.

Principle 8. Tests must be repeatable and reusable.

Principle 2 calls for a tester to view his/her work as similar to that of an experimental scientist. Principle 8 calls for experiments in the testing domain to require recording of the exact conditions of the test, any special events that occurred, equipment used, and a careful accounting of the results. This information is invaluable to the developers when the code is returned for debugging so that they can duplicate test conditions. It is also useful for tests that need to be repeated after defect repair. The repetition and reuse of tests is also necessary during regression test (the retesting of software that has been modified) in the case of a new release of the software. Scientists expect experiments to be repeatable by others, and testers should expect the same!

Principle 9. Testing should be planned.

Test plans should be developed for each level of testing, and objectives for each level should be described in the associated plan. The objectives should be stated as quantitatively as possible. Plans, with their precisely specified objectives, are necessary to ensure that adequate time and resources are allocated for testing tasks, and that testing can be monitored and managed. Test planning activities should be carried out throughout the software life cycle (Principle 10). Test planning must be coordinated with project planning. The test manager and project manager must work together to coordinate activities. Testers cannot plan to test a component on a given date unless the developers have it available on that date. Test risks must be evaluated. For example, how probable are delays in delivery of software components, which components are likely to be

complex and difficult to test, do the testers need extra training with new tools? A test plan template must be available to the test manager to guide development of the plan according to organizational policies and standards. Careful test planning avoids wasteful "throwaway" tests and unproductive and unplanned "test-patch-retest" cycles that often lead to poor-quality software and the inability to deliver software on time and within budget.

Principle 10. Testing activities should be integrated into the software life cycle.

It is no longer feasible to postpone testing activities until after the code has been written. Test planning activities as supported by Principle 10, should be integrated into the software life cycle starting as early as in the requirements analysis phase, and continue on throughout the software life cycle in parallel with development activities. In addition to test planning, some other types of testing activities such as usability testing can also be carried out early in the life cycle by using prototypes. These activities can continue on until the software is delivered to the users. Organizations can use process models like the V-model or any others that support the integration of test activities into the software life cycle [11].

Principle 11. Testing is a creative and challenging task [12].

Difficulties and challenges for the tester include the following:
• A tester needs to have comprehensive knowledge of the software engineering discipline.
• A tester needs to have knowledge from both experience and education as to how software is specified, designed, and developed.
• A tester needs to be able to manage many details.
• A tester needs to have knowledge of fault types and where faults of a certain type might occur in code constructs.
• A tester needs to reason like a scientist and propose hypotheses that relate to presence of specific types of defects.
• A tester needs to have a good grasp of the problem domain of the software that he/she is testing. Familiarly with a domain may come from educational, training, and work-related experiences.
• A tester needs to create and document test cases. To design the test cases the tester must select inputs often from a very wide domain.

1.6 The tester's role in a software development organization

Testing is sometimes erroneously viewed as a destructive activity. The tester's job is to reveal defects, find weak points, inconsistent behavior, and circumstances where the software does not work as expected. As a tester you need to be comfortable with this role. Given the nature of the tester's tasks, you can see that it is difficult for developers to effectively test their own code (Principles 3 and 8). Developers view their own code as their creation, their "baby," and they think that nothing could possibly be wrong with it! This is not to say that testers and developers are adversaries. In fact, to be most effective as a tester requires extensive programming experience in order to understand how code is constructed, and where, and what kind of, defects are likely to occur. Your goal as a tester is to work with the developers to produce high-quality

software that meets the customers' requirements. Teams of testers and developers are very common in industry, and projects should have an appropriate developer/tester ratio. The ratio will vary depending on available resources, type of project, and TMM level. For example, an embedded realtime system needs to have a lower developer/tester ratio (for example, 2/1) than a simple data base application (4/1 may be suitable). At higher TMM levels where there is a well-defined testing group, the developer/ tester ratio would tend to be on the lower end (for example 2/1 versus 4/1) because of the availability of tester resources. Even in this case, the nature of the project and project scheduling issues would impact on the ratio. In addition to cooperating with code developers, testers also need to work along side with requirements engineers to ensure that requirements are testable, and to plan for system and acceptance test (clients are also involved in the latter). Testers also need to work with designers to plan for integration and unit test. In addition, test managers will need to cooperate with project managers in order to develop reasonable test plans, and with upper management to provide input for the development and maintenance of organizational testing standards, polices, and goals. Finally, testers also need to cooperate with software quality assurance staff and software engineering process group members. In view of these requirements for multiple working relationships, communication and team working skills are necessary for a successful career as a tester. and marketing staff need to realize that testers add value to a software product in that they detect defects and evaluate quality as early as possible in the software life cycle. This ensures that developers release code with few or no defects, and that marketers can deliver software that satisfies the customers' requirements, and is reliable, usable, and correct. Low-defect software also has the benefit of reducing costs such as support calls, repairs to operational software, and ill will which may escalate into legal action due to customer dissatisfaction. In view of their essential role, testers need to have a positive view of their work. Management must support them in their efforts and recognize their contributions to the organization.

1.7 Origins of defects

The term *defect* and its relationship to the terms *error* and *failure* in the context of the software development domain has been discussed in Chapter 2. Defects have detrimental affects on software users, and software engineers work very hard to produce high-quality software with a low number of defects. But even under the best of development circumstances errors are made, resulting in defects being injected in the software during the phases of the software life cycle. Defects as shown in Figure 3.1 stem from the following sources [1,2]:

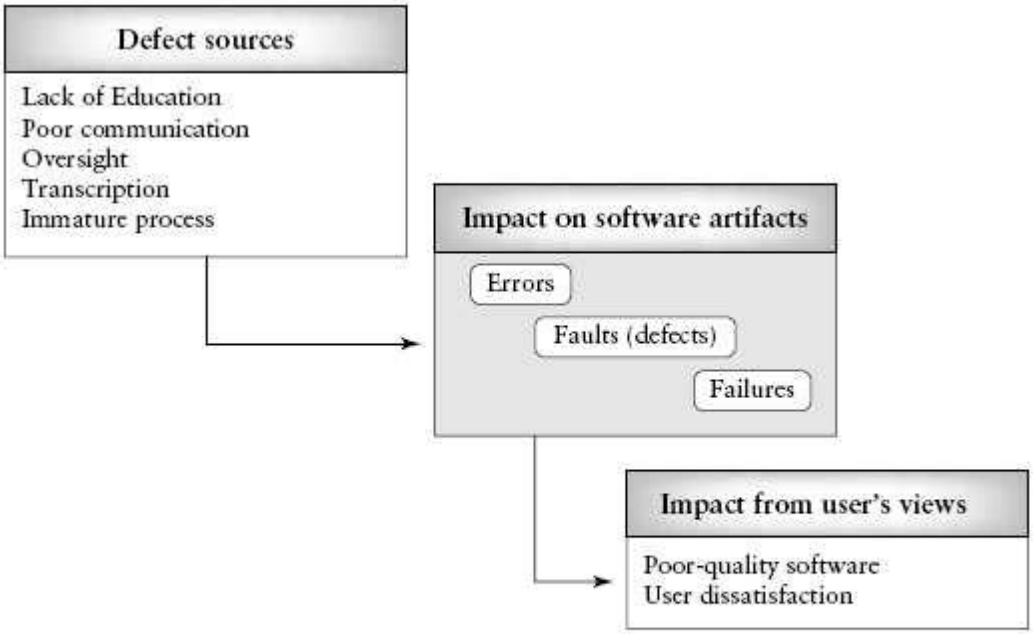


FIG. 3.1

Origins of defects.

1. *Education:* The software engineer did not have the proper educational background to prepare the software artifact. She did not understand how to do something. For example, a software engineer who did not understand the precedence order of operators in a particular programming language could inject a defect in an equation that uses the operators for a calculation.
2. *Communication:* The software engineer was not informed about something by a colleague. For example, if engineer 1 and engineer 2 are working on interfacing modules, and engineer 1 does not inform engineer 2 that a no error checking code will appear in the interfacing module he is developing, engineer 2 might make an incorrect assumption relating to the presence/absence of an error check, and a defect will result.

3. *Oversight:* The software engineer omitted to do something. For example, a software engineer might omit an initialization statement.

4. *Transcription:* The software engineer knows what to do, but makes a mistake in doing it. A simple example is a variable name being misspelled when entering the code.

5. *Process:* The process used by the software engineer misdirected her actions. For example, a development process that did not allow sufficient time for a detailed specification to be developed and reviewed could lead to specification defects.

When defects are present due to one or more of these circumstances, the software may fail, and the impact on the user ranges from a minor inconvenience to rendering the software unfit for use. Our goal as testers is to discover these defects preferably before the software is in operation.One of the ways we do this is by designing test cases that have a high probability of revealing defects. How do we develop these test cases? One approach is to think of software testing as an experimental activity. The results of the test experiment are analyzed to determine whether the software has behaved correctly. In this experimental scenario a tester develops hypotheses about possible defects (see Principles 2 and 9). Test cases are then designed based on the hypotheses. The tests are run and results analyzed to prove, or disprove, the hypotheses.

Myers has a similar approach to testing. He describes the successful test as one that reveals the presence of a (hypothesized) defect. He compares the role of a tester to that of a doctor who is in the process of constructing a diagnosis for an ill patient. The doctor develops hypotheses about possible illnesses using her knowledge of possible diseases, and the patients' symptoms. Tests are made in order to make the correct diagnosis. A successful test will reveal the problem and the doctor can begin treatment. Completing the analogy of doctor and ill patient, one could view defective software as the ill patient. Testers as doctors need to have knowledge about possible defects (illnesses) in order to develop defect hypotheses. They use the hypotheses to:

• design test cases;
• design test procedures;
• assemble test sets;
• select the testing levels (unit, integration, etc.)appropriate for the tests;
• evaluate the results of the tests.

A successful testing experiment will prove the hypothesis is true—that is, the hypothesized defect was present. Then the software can be repaired (treated).A very useful concept related to this discussion of defects, testing, and diagnosis is that of a fault model.

A fault (defect) model can be described as a link between the error made (e.g., a missing requirement, a misunderstood design element, a typographical error), and the fault/defect in the software.

Digital system engineers describe similar models that link physical defects in digital components to electrical (logic) effects in the resulting digital system [4,5]. Physical defects in the digital world may be due to manufacturing errors, component wear-out, and/or environmental effects.

The fault models are often used to generate a fault list or dictionary. From that dictionary faults can be selected, and test inputs developed for digital components. The effectiveness of a test can be evaluated in the context of the fault model, and is related to the number of faults as expressed in the model, and those actually revealed by the test. This view of test effectiveness (success) is similar to the view expressed by Myers stated above.

Although software engineers are not concerned with physical defects, and the relationships between software failures, software defects, and their origins are not easily mapped, we often use the fault model concept and fault lists accumulated in memory from years of experience to

design tests and for diagnosis tasks during fault localization (debugging) activities. A simple example of a fault model a software engineer might have in memory is "an incorrect value for a variable was observed because the precedence order for the arithmetic operators used to calculate its value was incorrect." This could be called "an incorrect operator precedence order" fault. An error was made on the part of the programmer who did not understand the order in which the arithmetic operators would execute their operations. Some incorrect assumptions about the order were made.

The defect (fault) surfaced in the incorrect value of the variable. The probable cause is a lack of education on the part of the programmer. Repairs include changing the order of the operators or proper use of parentheses. The tester with access to this fault model and the frequency of occurrence of this type of fault could use this information as the basis for generating fault hypotheses and test cases. This would ensure that adequate tests were performed to uncover such faults.

In the past, fault models and fault lists have often been used by developers/ testers in an informal manner, since many organizations did not save or catalog defect-related information in an easily accessible form. To increase the effectiveness of their testing and debugging processes, software organizations need to initiate the creation of a defect database, or defect repository. The defect repository concept supports storage and retrieval of defect data from all projects in a centrally accessible location. A defect classification scheme is a necessary first step for developing the repository. The defect repository can be organized by projects and for all projects defects of each class are logged, along their frequency of occurrence, impact on operation, and any other useful comments. Defects found both during reviews and execution-based testing should be cataloged.

1.8 Defect classes, the defect repository and test design

Defects can be classified in many ways. It is important for an organization to adapt a single classification scheme and apply it to all projects. No matter which classification scheme is selected, some defects will fit into more than one class or category. Because of this problem, developers,testers, and SQA staff should try to be as consistent as possible when recording defect data. The defect types and frequency of occurrence should be used to guide test planning, and test design. Execution-based testing strategies should be selected that have the strongest possibility of detecting particular types of defects. It is important that tests for new and modified software be designed to detect the most frequently occurring defects. The reader should keep in mind that execution-based testing will detect a large number of the defects that will be described; however, software reviews as described in Chapter 10 are also an excellent testing tool for detection of many of the defect types that will be discussed in the following sections.

Defects, as described in this text, are assigned to four major classes reflecting their point of origin in the software life cycle—the development phase in which they were injected. These classes are: requirements/ specifications, design, code, and testing defects as summarized in Figure 3.2. It should be noted that these defect classes and associated subclasses focus on defects that are the major focus of attention to execution-based testers. The list does not include other defects types that are best found in software reviews, for example, those defects related to conformance to styles and standards. The review checklists in Chapter 10 focus on many of these types of defects.

2 .1.1 Requirements and Specification Defects

The beginning of the software life cycle is critical for ensuring high quality in the software being developed. Defects injected in early phases can persist and be very difficult to remove in later phases. Since many requirements documents are written using a natural language representation, there are very often occurrences of ambiguous, contradictory, unclear, redundant, and imprecise requirements. Specifications in many organizations are also developed using natural language representations, and these too are subject to the same types of problems as mentioned above.

However, over the past several years many organizations have introduced the use of formal specification languages that, when accompanied by tools, help to prevent incorrect descriptions of system behavior. Some specific requirements/specification defects are:

1 . Functional Description Defects
The overall description of what the product does, and how it should behave (inputs/outputs), is incorrect, ambiguous, and/or incomplete.

2 . Feature Defects

Features may be described as distinguishing characteristics of a software component or system.
Features refer to functional aspects of the software that map to functional requirements as described by the users and clients. Features also map to quality requirements such as performance and reliability. Feature defects are due to feature descriptions that are missing, incorrect, incomplete, or superfluous.

3 . Feature Interaction Defects

These are due to an incorrect description of how the features should interact. For example, suppose one feature of a software system supports adding a new customer to a customer database. This feature interacts with another feature that categorizes the new customer. The classification feature impacts on where the storage algorithm places the new customer in the database, and also affects another feature that periodically supports sending advertising information to customers in a specific category. When testing we certainly want to focus on the interactions between these features.

4 . Interface Description Defects

These are defects that occur in the description of how the target software is to interface with external software, hardware, and users. For detecting many functional description defects, black box testing techniques, which are based on functional specifications of the software, offer the best approach. In Chapter 4 the reader will be introduced to several black box testing techniques

such as equivalence class partitioning, boundary value analysis, state transition testing, and cause-and-effect graphing, which are useful for detecting functional types of detects. Random testing and error guessing are also useful for detecting these types of defects. The reader should note that many of these types of defects can be detected early in the life cycle by software reviews. Black box-based tests can be planned at the unit, integration, system, and acceptance levels to detect requirements/specification defects. Many feature interaction and interfaces description defects are detected using black box-based test designs at the integration and system levels.

D e s i g n D e f e c t s

Design defects occur when system components, interactions between system components, interactions between the components and outside soft ware/hardware, or users are incorrectly designed. This covers defects in the design of algorithms, control, logic, data elements, module interface descriptions, and external software/hardware/user interface descriptions.

When describing these defects we assume that the detailed design description for the software modules is at the pseudo code level with processing steps, data structures, input/output parameters, and major control structures defined. If module design is not described in such detail then many of the defects types described here may be moved into the coding defects class.

1 . Algorithmic and Processing Defects

These occur when the processing steps in the algorithm as described by the pseudo code are incorrect. For example, the pseudo code may contain a calculation that is incorrectly specified, or the processing steps in the algorithm written in the pseudo code language may not be in the correct order. In the latter case a step may be missing or a step may be duplicated.

Another example of a defect in this subclass is the omission of error condition checks such as division by zero. In the case of algorithm reuse, a designer may have selected an inappropriate algorithm for this problem (it may not work for all cases).

2 . Control, Logic, and Sequence Defects

Control defects occur when logic flow in the pseudo code is not correct. For example, branching to soon, branching to late, or use of an incorrect branching condition. Other examples in this subclass are unreachable pseudo code elements, improper nesting, improper procedure or function calls. Logic defects usually relate to incorrect use of logic operators, such as less than (_), greater than (_), etc. These may be used incorrectly in a Boolean expression controlling a branching instruction.

3 . Data Defects

These are associated with incorrect design of data structures. For example, a record may be lacking a field, an incorrect type is assigned to a variable or a field in a record, an array may not have the proper number of elements assigned, or storage space may be allocated incorrectly.

Software reviews and use of a data dictionary work well to reveal these types of defects.

4 . Module Interface Description Defects
These are defects derived from, for example, using incorrect, and/or inconsistent parameter types, an incorrect number of parameters, or an incorrect ordering of parameters.

5 . Functional Description Defects
The defects in this category include incorrect, missing, and/or unclear design elements. For example, the design may not properly describe the correct functionality of a module. These defects are best detected during a design review.

6 . External Interface Description Defects
These are derived from incorrect design descriptions for interfaces with COTS components, external software systems, databases, and hardware devices (e.g., I/O devices). Other examples are user interface description defects where there are missing or improper commands, improper sequences of commands, lack of proper messages, and/or lack of feedback messages for the user.

C o d i n g D e f e c t s

Coding defects are derived from errors in implementing the code. Coding defects classes are closely related to design defect classes especially if pseudo code has been used for detailed design. Some coding defects come from a failure to understand programming language constructs, and miscommunication with the designers. Others may have transcription or omission origins. At times it may be difficult to classify a defect as a design or as a coding defect. It is best to make a choice and be consistent when the same defect arises again.

1 . Algorithmic and Processing Defects
Adding levels of programming detail to design, code-related algorithmic and processing defects would now include unchecked overflow and underflow conditions, comparing inappropriate data types, converting one data type to another, incorrect ordering of arithmetic operators (perhaps due to misunderstanding of the precedence of operators), misuse or omission of parentheses, precision loss, and incorrect use of signs.

2 . Control, Logic and Sequence Defects
On the coding level these would include incorrect expression of case statements, incorrect iteration of loops (loop boundary problems), and missing paths.

3 . Typographical Defects
These are principally syntax errors, for example, incorrect spelling of a variable name, that are usually detected by a compiler, self-reviews, or peer reviews.

4 . I n i t i a l i z a t i o n Defects
These occur when initialization statements are omitted or are incorrect. This may occur because of misunderstandings or lack of communication between programmers, and/or programmers and designers, carelessness, or misunderstanding of the programming environment.

5 . Data-Flow Defects
There are certain reasonable operational sequences that data should flow through. For example, a variable should be initialized, before it is usedin a calculation or a condition. It should not be initialized twice before there is an intermediate use. A variable should not be disregarded before it is used. Occurrences of these suspicious variable uses in the code may, or may not, cause anomalous behavior. Therefore, in the strictest sense of the definition for the term "defect," they may not be considered as true instances of defects. However, their presence indicates an error has occurred and a problem exists that needs to be addressed.

6 . Data Defects
These are indicated by incorrect implementation of data structures. For example, the programmer may omit a field in a record, an incorrect type or access is assigned to a file, an array may not be allocated the proper number of elements. Other data defects include flags, indices, and constants set incorrectly.

7 . Module Interface Defects
As in the case of module design elements, interface defects in the code may be due to using incorrect or inconsistent parameter types, an incorrect number of parameters, or improper ordering of the parameters. In addition to defects due to improper design, and improper implementation of design, programmers may implement an incorrect sequence of calls or calls to nonexistent modules.

8 . Code Documentation Defects
When the code documentation does not reflect what the program actually does, or is incomplete or ambiguous, this is called a code documentation defect. Incomplete, unclear, incorrect, and outof-date code documentation affects testing efforts. Testers may be misled by documentation defects and thus reuse improper tests or design new tests that are not appropriate for the code. Code reviews are the best tools to detect these types of defects.

9 . External Hardware, Software Interfaces Defects
These defects arise from problems related to system calls, links to databases, input/output sequences, memory usage, resource usage, interrupts and exception handling, data exchanges with hardware, protocols, formats, interfaces with build files, and timing sequences (race conditions may result).

Many initialization, data flow, control, and logic defects that occur in design and code are best addressed by white box testing techniques applied at the unit (single-module) level. For example, data flow testing is useful for revealing data flow defects, branch testing is useful for detecting control defects, and loop testing helps to reveal loop-related defects. White box testing approaches are dependent on knowledge of the internal structure of the software, in contrast to black box approaches, which are only dependent on behavioral specifications. The reader will
be introduced to several white box-based techniques in Chapter 5. Many design and coding defects are also detected by using black box testing techniques. For example, application of decision tables is very useful for detecting errors in Boolean expressions. Black box tests as

described in Chapter 4 applied at the integration and system levels help to reveal external hardware and software interface defects. The author will stress repeatedly throughout the text that a combination of both of these approaches is needed to reveal the many types of defects that are likely to be found in software.

T e s t i n g D e f e c t s

Defects are not confined to code and its related artifacts. Test plans, test cases, test harnesses, and test procedures can also contain defects. Defects in test plans are best detected using review techniques.

1 . Test Harness Defects

In order to test software, especially at the unit and integration levels, auxiliary code must be developed. This is called the test harness or scaffolding code. Chapter 6 has a more detailed discussion of the need for this code. The test harness code should be carefully designed, implemented, and tested since it a work product and much of this code can be reused when new releases of the software are developed. Test harnesses are subject to the same types of code and design defects that can be found in all other types of software.

2 . Test Case Design and Test Procedure Defects

These would encompass incorrect, incomplete, missing, inappropriate test cases, and test procedures. These defects are again best detected in test plan reviews as described in Chapter 10. Sometimes the defects are revealed during the testing process itself by means of a careful analysis of test conditions and test results. Repairs will then have to be made.

1.10 Defect Examples: The Coin Problem

The following examples illustrate some instances of the defect classes that were discussed in the previous sections. A simple specification, a detailed design description, and the resulting code are shown, and defects in each are described. Note that these defects could be injected via one or more of the five defect sources discussed at the beginning of this chapter. Also note that there may be more than one category that fits a given defect. Figure 3.3 shown a sample informal specification for a simple program that calculates the total monetary value of a set of coins. The program could be a component of an interactive cash register system to support retail store clerks. This simple example shows requirements/ specification defects, functional description defects, and interface description defects.

The functional description defects arise because the functional description is ambiguous and incomplete. It does not state that the input, number_of_coins, and the output, number_of_dollars and number _of_cents, should all have values of zero or greater. The number_of_coins cannot be negative, and the values in dollars and cents cannot be negative in the real-world domain. As a consequence of these ambiguities and specification incompleteness, a checking routine may be omitted from the design, allowing the final program to accept negative values for the input

number_of_coins for each of the denominations, and consequently it may calculate an invalid value for the results. A more formally stated set of preconditions and postconditions would be helpful here, and would address some of the problems with the specification. These are also useful for designing black box tests.

A precondition is a condition that must be true in order for a software component to operate properly.

In this case a useful precondition would be one that states for example:number_of_coins __0
A postcondition is a condition that must be true when a software component completes its operation properly.
A useful postcondition would be:
number_of_dollars, number_of_cents __ 0.

Specification for Program `calculate_coin_value`

This program calculates the total dollars and cents value for a set of coins. The user inputs the amount of pennies, nickels, dimes, quarters, half-dollars, and dollar coins held. There are six different denominations of coins. The program outputs the total dollar and cent values of the coins to the user.

Inputs: `number_of_coins` is an integer
Outputs: `number_of_dollars` is an integer
`number_of_cents` is an integer

FIG. 3.3
A sample specification with defects.

In addition, the functional description is unclear about the largest number of coins of each denomination allowed, and the largest number of dollars and cents allowed as output values. Interface description defects relate to the ambiguous and incomplete description of user-software interaction. It is not clear from the specification how the user interacts with the program to provide input, and how the output is to be reported. Because of ambiguities in the user interaction description the software may be difficult to use. Likely origins for these types of specification defects lie in the nature of the development process, and lack of proper education and training. A poor-quality development process may not be allocating the proper time and resources to specification development and review. In addition, software engineers may not have the proper education and training to develop a quality specification. All of these specification defects, if not detected and repaired, will propagate to the design and coding phases. Black box testing techniques, which we will study in Chapter 4, will help to reveal many of these functional weaknesses. Figure 3.4 shows the specification transformed in to a design description. There are numerous design defects, some due to the ambiguous and incomplete nature of the specification; others are newly introduced. Design defects include the following:

Control, logic, and sequencing defects. The defect in this subclass arises from an incorrect "while" loop condition (should be less than or equal to six)

Algorithmic, and processing defects. These arise from the lack of error checks for incorrect and/or invalid inputs, lack of a path where users can correct erroneous inputs, lack of a path for recovery from input errors. The lack of an error check could also be counted as a functional design defect since the design does not adequately describe the proper functionality for the program.

```
Design Description for Program calculate_coin_values

Program calculate_coin_values
number_of_coins is  integer
total_coin_value is integer
number_of_dollars is integer
number_of_cents is integer
coin_values is array of  six integers representing
each coin value in cents
initialized to: 1,5,10,25,25,100
begin

initialize total_coin_value to zero
initialize loop_counter to one
while loop_counter is less then six
begin
       output "enter number of coins"
       read (number_of_coins )
       total_coin_value = total_coin_value +
       number_of_coins * coin_value[loop_counter]
       increment loop_counter
end
number_dollars =  total_coin_value/100
number_of_cents = total_coin_value - 100 * number_of_dollars
output (number_of_dollars, number_of_cents)
end
```

FIG. 3.4
A sample design specification with defects.

Data defects. This defect relates to an incorrect value for one of the elements of the integer array, coin_values, which should read 1,5,10,25,50,100.

External interface description defects. These are defects arising from the absence of input messages or prompts that introduce the program to the user and request inputs. The user has no way of knowing in which order the number of coins for each denomination must be input, and when to stop inputting values. There is an absence of help messages, and feedback for user if he wishes to change an input or learn the correct format and order for inputting the number of coins. The output description and output formatting is incomplete. There is no description of what the outputs means in terms of the problem domain. The user will note that two values are output, but has no clue as to their meaning. The control and logic design defects are best addressed by white

box- based tests, (condition/branch testing, loop testing). These other design defects will need a combination of white and black box testing techniques for detection.
Figure 3.5 shows the code for the coin problem in a "C-like" programming language. Without effective reviews the specification and design defects could propagate to the code. Here additional defects have been introduced in the coding phase.

Control, logic, and sequence defects. These include the loop variable increment step which is out of the scope of the loop. Note that incorrect loop condition (i _ 6) is carried over from design and should be counted as a design defect.

Algorithmic and processin g defects. The division operator may cause problems if negative values are divided, although this problem could be eliminated with an input check.

Data Flow defects. The variable total_coin_value is not initialized. It is used before it is defined.
(This might also be considered a data defect.)

Data Defects. The error in initializing the array coin_values is carried over from design and should be counted as a design defect.

External Hardware, Software Interface Defects. The call to the external func tion "scanf" is incorrect. The address of the variable must be provided (&number_of_coins).

Code Documentation Defects. The documentation that accompanies this code is incomplete and ambiguous. It reflects the deficiencies in the external interface description and other defects that occurred during speci fication and design. Vital information is missing for anyone who will need to repair, maintain or reuse this code.

```
/*************************************************************
program calculate_coin_values  calculates the dollar and cents
value of a set of coins of different dominations input by the user
denominations are pennies, nickels, dimes, quarters, half dollars,
and dollars
*************************************************************/
main ()
{
int total_coin_value;
int number_of_coins = 0;
int number_of_dollars = 0;
int number_of-cents = 0;
int coin_values = {1,5,10,25,25,100};
{
int  i = 1;
while ( i < 6)
{
       printf("input number of coins\n");
       scanf ("%d", number_of_coins);
       total_coin_value = total_coin_value +
        (number_of_coins * coin_value{i]);
}
i = i + 1;
number_of_dollars =  total_coin_value/100;
number_of_cents = total_coin_value - (100 * number_of_dollars);
printf("%d\n", number_of_dollars);
printf("%d\n", number_of-cents);
}

/*************************************************************/
```

FIG. 3.5

A code example with defects.

The control, logic, and sequence, data flow defects found in this example could be detected by using a combination of white and black box testing techniques. Black box tests may work well to reveal the algorithmic and data defects. The code documentation defects require a code review for detection. The external software interface defect would probably be caught by a good compiler.

The poor quality of this small program is due to defects injected during several of the life cycle phases with probable causes ranging from lack of education, a poor process, to oversight on the part of the designers and developers. Even though it implements a simple function the program is unusable because of the nature of the defects it contains. Such software is not acceptable to users; as testers we must make use of all our static and dynamic testing tools as described in subsequent chapters to ensure that such poor-quality software is not delivered to our user/client group. We must work with analysts, designers and code developers to ensure that quality issues are addressed early the software life cycle. We must also catalog defects and try to eliminate them by improving education, training, communication, and process.

1.11 Developer/Tester Support for Developing a Defect Repository

The focus of this chapter is to show with examples some of the most common types of defects that occur during software development. It is important if you are a member of a test organization to illustrate to management and your colleagues the benefits of developing a defect repository to store defect information. As software engineers and test specialists we should follow the examples of engineers in other disciplines who have realized the usefulness of defect data. A requirement for repository development should be a part of testing and/or debugging policy statements. You begin with development of a defect classification scheme and then initiate the collection defect data from organizational projects. Forms and templates will need to be designed to collect the data. Examples are the test incident reports as described in Chapter 7, and defect fix reports as described in Chapter 4. You will need to be conscientious about recording each defect after testing, and also recording the frequency of occurrence for each of the defect types. Defect monitoring should continue for each on-going project. The distribution of defects will change as you make changes in your processes. The defect data is useful for test planning, a TMM level 2 maturity goal. It helps you to select applicable testing techniques,
design (and reuse) the test cases you need, and allocate the amount of resources you will need to devote to detecting and removing these defects. This in turn will allow you to estimate testing schedules and costs.

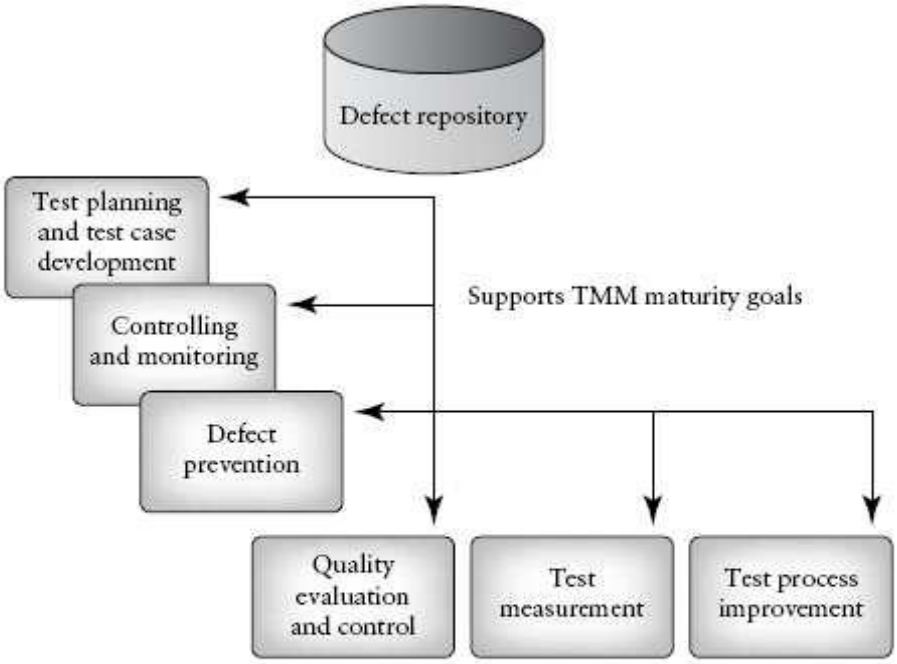


FIG. 3.6

The defect repository, and support for TMM maturity goals.

The defect data can support debugging activities as well. In fact, as Figure 3.6 shows, a defect repository can help to support achievement and continuous implementation of several TMM maturity goals including controlling and monitoring of test, software quality evaluation and control, test measurement, and test process improvement. Chapter 13 will illustrate the application of this data to defect prevention activities and process improvement. Other chapters will describe the role of defect data in various testing activities.

CS1016 - SOFTWARE TESTING

IMPORTANT QUESTIONS

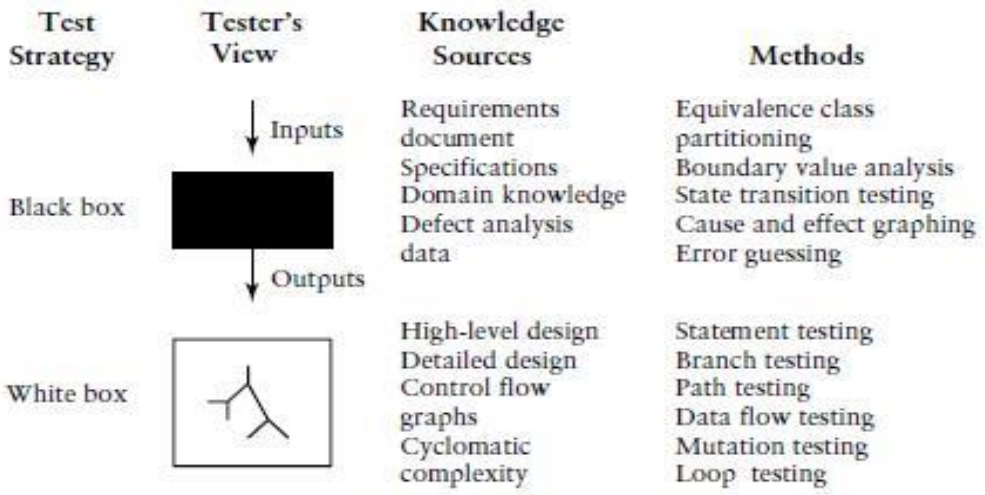
Unit I
Part-A Questions

1. Compare Validation and Verification.
2. Define Software quality.
3. Define:Process
4. Define:Testing and debugging
5. Compare:Errors,faults and failures
6. Define:metrics
7. Define the role of SQA Group.
8. Define: Defect repository

Part-B Questions

1. Explain the Software testing principles.
2. Describe the defect classes in detail with example.
3. Explain defect repository.

2.1 Test Case Design Strategies



The two basic testing strategies

1. Using the Black Box Approach to Test Case Design
2. Using the White Box Approach to Test Case Design

1. Using the Black Box Approach to Test Case Design

- Using the black box approach, a tester considers the software-under test to be an opaque box. There is no knowledge of its inner structure (i.e., how it works).
- The tester only has knowledge of what it does.
- The size of the software-under-test using this approach can vary from a simple module, member function, or object cluster to a subsystem or a complete Software system.
- The description of behavior or functionality for the software-under-test may come from a formal specification, an Input/Process/Output Diagram (IPO), or a well-defined set of pre and post conditions.
- Another source for information is a requirements **specification document** that usually describes the functionality of the software-under-test and its inputs and expected outputs.
- The tester provides the specified inputs to the software-under-test, runs the test and then determines if the outputs produced are equivalent to those in the specification.
- Because the black box approach only considers software behavior and functionality, it is often called functional or specification- based testing.
- This approach is especially useful for revealing **requirements and specification defects**.

Example

LOCK AND KEY: Should not know the levers in the lock, but we know the set of inputs and expected output (Lock and unlock)

Functionality

1. Features of a Lock:

Made up of metal
It has a Hole Provision to lock
Facility to insert key
Ability to turn clockwise and anticlockwise direction

2. Features of a Key

It is made of metal
It Fit into a particular lock's keyhole

3. Action performed

Key inserted and turned clockwise to lock
Key inserted and turned anticlockwise unlock

4. States

Locked and unlocked

5. Inputs

Key turned clockwise or anticlockwise

6. Expected outcome

Locking and unlocking

Note:

- Black box test strategy- only inputs and outputs are considered as a basis for designing test cases
- Selection of set of inputs from the set of all possible valid and invalid inputs is an important, because exhaustive testing is not possible.

Different types of black box testing:

Requirement based testing, Random Testing Requirements based testing, Boundary Value Analysis , Equivalence Class Partitioning, State-based testing , Cause-effect graphing, Compatibility testing, user documentation testing, domain testing

2.2. Requirement based Testing

Requirement based testing is validating the requirements given in the SRS of the software system and also validating explicitly stated and implied requirements.

- Precondition for requirements testing is Review of the requirements specification.
- Review ensures the requirements are consistent, correct, complete and testable.
- During review process – Implied requirements converted into explicit requirements.

Example :
Sample requirements specification for lock and key system

The following table shows the requirements for lock and key system and its priority also fixed.

Requirement Identifier	Description	Priority
BR-01	Inserting the key numbered KEY09 and turning it CW should facilitate locking	High
BR-02	Inserting the key numbered KEY09 and turning it ACW should facilitate unlocking	High
BR-03	Only key no. KEY09 should be used for lock and unlock	High
BR-04	No other object can be used for lock	Medium
BR-05	No other object can be used for unlock	Medium
BR-06	Lock should not open even with a heavy object	Medium
BR-07	Should be made of metal and weight should be 150 grams	Low
BR-08	Lock and unlock directions should be changeable for usability of left-handers	Low

For the above requirements, test cases are framed with precondition and it is also listed in the next table.

Req Id	Description	Priority	Test conditions	Test case Ids	Phase of Testing
BR-01	Inserting the key numbered KEY09 and turning it CW should facilitate locking	High	Use Key KEY09	TC1	Unit, Component
BR-02	Inserting the key numbered KEY09 and turning it ACW should facilitate unlocking	High	Use Key KEY09	TC2	Unit, Component
BR-	Only key no. KEY09 should be	High	Use Key	TC3	Component

03	used for lock and unlock		KEY09 to lock Use Key KEY09 to unlock	TC4	
BR-04	No other object can be used for lock	Medium	Use Key KEY09 Use hairpin Use toothpick	TC5 TC6 TC7	Integration
BR-05	No other object can be used for unlock	Medium	Use Key KEY08 Use hairpin Use toothpick	TC8 TC9 TC10	Integration
BR-06	Lock should not open even with a heavy object	Medium	Use stone to break the lock	TC11	System
BR-07	Should be made of metal and weight should be 150 grams	Low	Use Weighing machine	TC11	System
BR-08	Lock and unlock directions should be changeable for usability of left-handers	Low	----------	--------	Not implemented

In the above table, test cases are identified based on the requirements specified by the customer. From this table, we can construct Requirement traceability matrix.

Requirement Traceability Matrix

One to one - For each requirement there is only one TC.

Example: BR01

One to many – For each requirement many Test cases are needed to check whether requirements are satisfied or not.

Example: BR03

Many to one- only one Test case are enough to check the multiple requirements:

Example: Not available

Many to Many- Many Requirements are tested by executing many test cases

One to None – The set of requirements can have no TC.

Example: Requirement has not been implemented or it has the lowest priority.

Example: BR08

Advantages of RTM

- RTM provides a tool to track the testing status of each requirement without missing any requirements
- Identifies defects in the high priority area by prioritization
- Time limit – Omit low priority TCs.

After execution of Test cases, the test results can be used to collect metrics such as,

- Total No. Of TCs passed
- Total No. Of TCs failed
- Total number of defects- in requirements
- Number of requirements completed
- Number of requirements pending

Sample test execution data

Rq.ID	Priority	TC	Total TCs	No. of TC Passed	No. of TC Failed	% Pass	No. of defects
BR-01	High	TC1	1	1	-	100	1
BR-02	High	TC2	1	1	-	100	1
BR-03	High	TC3 TC4	2	1	1	50	3
BR-04	Medium	TC5 TC6 TC7	3	2	1	67	5
BR-05	Medium	TC8 TC9 TC10	3	3	-	100	1
BR-06	Medium	TC11	1	1	-	100	1
BR-07	Low	TC11	1	1	-	100	0
BR-08	Low	--------	------	-----	-------	--------	1

Observation from the table

83% of passed TCs correspond to 71 % of requirements being met (five out of seven requirements met, one requirement is not implemented)

*****

2.3. Boundary Value Analysis

- "Boundary value analysis is useful to generate test cases when the input data is made up of clearly identifiable boundaries or ranges".
- Mostly in s/w defects occur due to boundaries and conditions.

Reason

- Confusion to use the <= operator or < operator.
- Confusion caused by the availability of multiple ways to implement loops and condition checking.(for, while repeat loop – each of these having different terminating conditions).
- Sometimes requirement may not be clearly understood especially around the boundaries.

Example:

Consider the following table which consists of price and No. of units. The price has been fixed based on the number of units purchased. Framing test cases on its boundary condition is very much important. For the given table, Generated test cases are listed in the next table.

Number of units bought	Price per unit (rs)
First 10 units (1-10)	5
Next ten units (11-20)	4.75
Next ten units (21-30)	4.50
More than 30 units	4.00

Table 1: Example for BVA

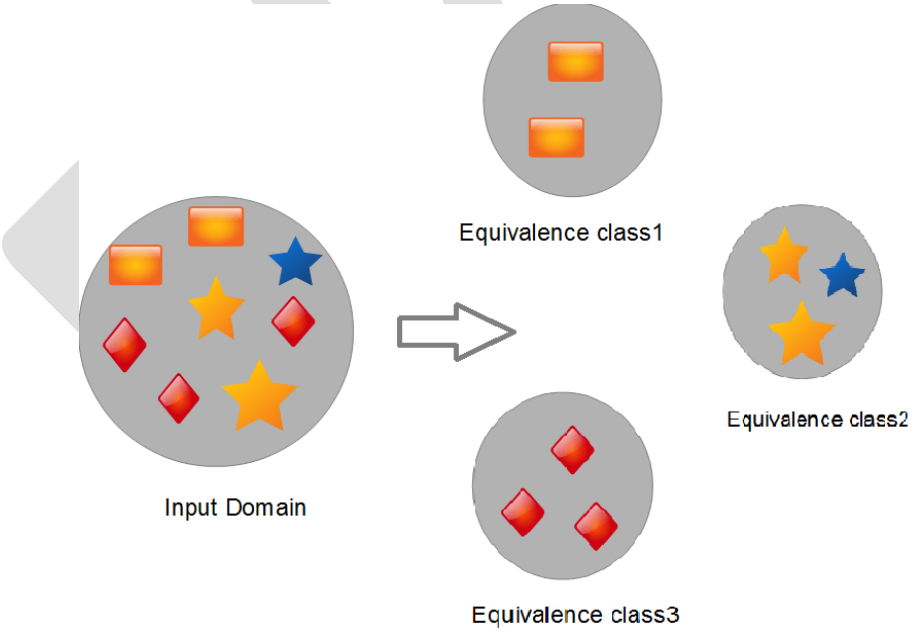
Values to be tested	Why this value should be tested	Expected value of the o/p
1	Beginning of the first slab	5
5	Value in the first slab, removed from the boundaries	25
9	End of the first slab (or) just below the second slab	45
10	Limit for the II slab	50
16	Value in the II slab, removed from the boundaries	76
21	Beginning of the III slab	94.5
27	Value in the III slab	121.5
31	Beginning of the IV slab	124

Table: Sample test cases

*****

2.4. Equivalence Class Partitioning

- The set of input values that generate one single expected output is called a "Partition".
- When the behaviour of the software is the same for a set of values then the set is termed as an "Equivalence class" or a "Partition".
- One sample from each partition is picked up for testing.



Equivalence Testing

- Identify all partitions for the complete set of input, output values for a product and
- Picking up one member value from each partition for testing to maximize complete coverage

Steps for testing

1. Choose criteria for doing the equivalence partitioning (range , list of values)
2. Identify all equivalence classes
3. Select a sample data from the partition
4. Write expected result based on requirements
5. Identify special values
6. Check the expected result
7. If result is not clear for particular test case, take corrective action.

Advantages

- Good coverage with a small number of test cases.
- Redundancy of tests is minimized.

Example

- A life insurance company has base premium of Rs.100 for all ages. Additional monthly premium has to be paid based on the age group. Conditions given below

Age Group	Additional Premium
Under 35	Rs. 200
35-59	Rs. 400
60 +	Rs.1000

Based on the partition - Inputs are,

- Below 35 years of age (valid input)
- Below 35 and 59 years of age (valid input)
- Above 60 years of age (valid input)
- Negative age (invalid input)
- Age as 0 (invalid input)
- Age as any three-digit number (valid input)

The valid and invalid inputs are listed in the following table.

TC No.	Equivalence partitions	Type of input	Test data	Expected results
1	Age below 35	Valid	26,12	MP=100+200
2	35-59	Valid	37	100+400
3	Age above 60	Valid	67,90	100+1000
4	Negative age	Invalid	-23	Warning message “invalid input”
5	Age as 0	Invalid	0	Warning message “invalid input”

Equivalence partitioning is useful to **minimize** the number of test cases when the input data can be divided into distinct sets, where the behaviour or outcome of the product within **each member of the set is the same.**

*****

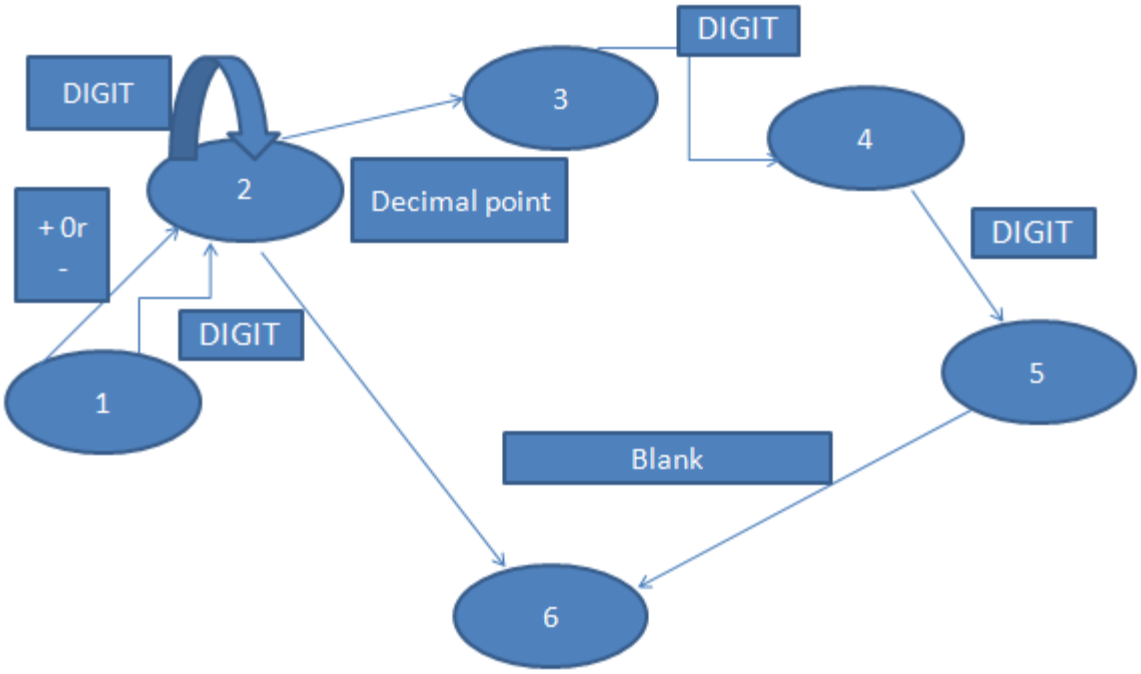
2.5. State-based testing

Graph based testing methods are applicable to generate test cases for state machines such as language translators, workflows, transaction flows and data flows.

Example

An application validate a number according to the following simple rules

1. A number can start with an optional sign.
2. The optional sign can be followed by any number of digits.
3. The digit can be optionally followed by a decimal point, represented by a period.
4. If there is decimal point, then there should be two digits after the decimal
5. No – whether or not it has a decimal point, should be terminated by a blank.



State transition table can be used to derive test cases to test valid and invalid numbers.

1. Start from the start state.
2. Choose a path that leads to the next state
3. Invalid input - generate an error condition test case.
4. Repeat the process until reach the final state.

The test cases to check the above are listed in the following table.

Current state	Input	Next state
1	Digit	2
1	+	2
1	-	2

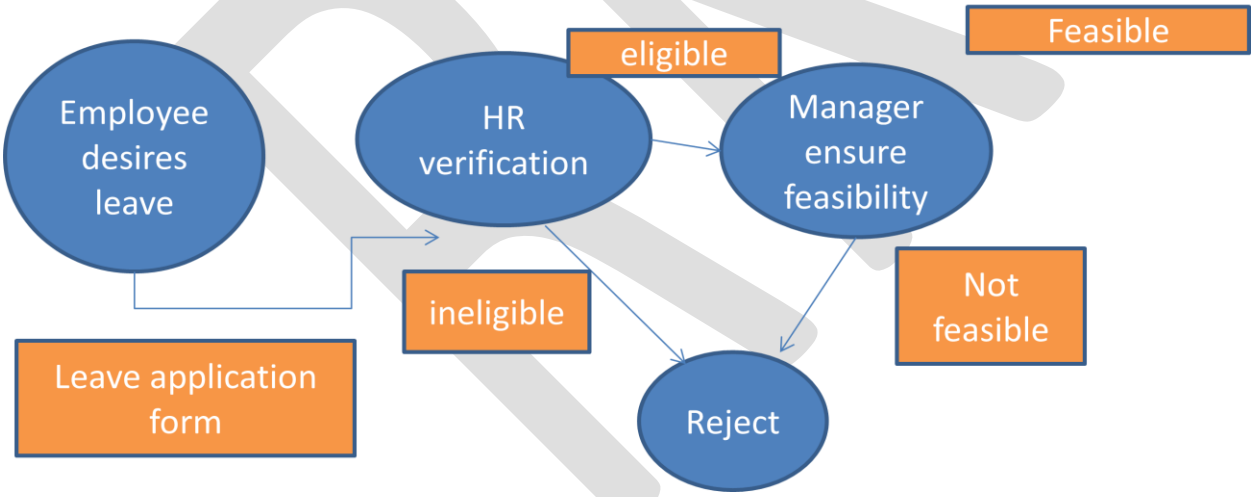
2	Digit	2
2	Blank	6
2	Decimal point	3
3	Digit	4
4	Digit	5
5	Blank	6

Graph based testing

- It is useful to represent a transaction or workflows.

Example : Employee leave application

1. He fills up a leave application, by providing the details like ID no, starting and ending date.
2. Automation system validate whether the employee is eligible for the requisite number of days of leave.
3. Verification by manager (any deadlines during that period)
4. Final approval/rejection.



*****

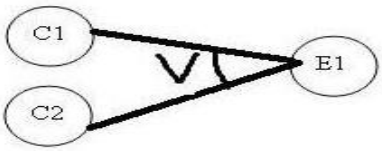
2.6. Cause-effect graphing

- Cause-Effect Graphing is a technique which starts with set of requirements and determines the minimum possible test cases for maximum test coverage which reduces test execution time and ultimately cost.
- The goal is to reduce the total number of test cases still achieving the desired application quality by covering the necessary test cases for maximum coverage.
- The Cause-Effect graph technique restates the requirements specification in terms of logical relationship between the input and output conditions.
- Since it is logical, it is obvious to use Boolean operators like AND, OR and NOT.

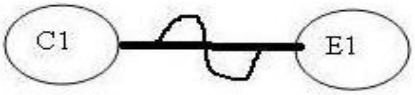
Boolean Operators

AND – For effect E1 to be true, both the causes C1 and C2 should be true

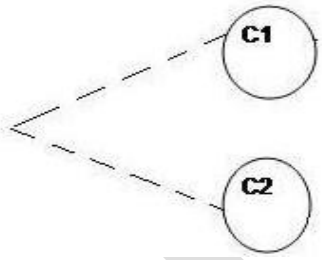
OR – For effect E1 to be true, either of causes C1 OR C2 should be true



NOT – For Effect E1 to be True, Cause C1 should be false



MUTUALLY EXCLUSIVE – When only one of the causes will hold true.



1. Draw a cause and effect graph based on a requirement/situation
2. Cause and Effect graph is given, draw a decision table based on it to draw the test case.

- The first character must be an "A" or a "B".
- The second character must be a digit.
- If the first character is an "A" or "B" and the second character is a digit, the file must be updated.
- If the first character is incorrect (not an "A" or "B"), the message X must be printed.
- If the second character is incorrect (not a digit), the message Y must be printed.

The causes for this situation are:

C1 – First character is A

C2 – First character is B

C3 – Second character is a digit

The effects (results) for this situation are

E1 – Update the file

E2 – Print message "X"

E3 – Print message "Y"

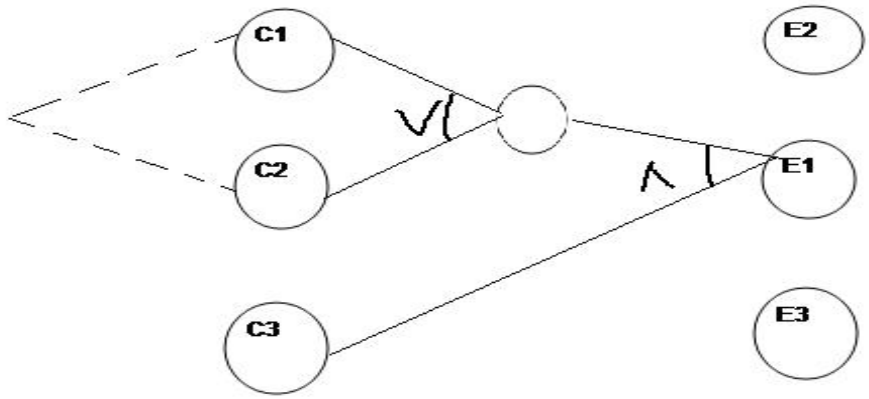
Effect E1 is to update the file. The file is updated when

- First character is "A" and second character is a digit
- First character is "B" and second character is a digit
- First character can either be "A" or "B" and cannot be both.

For E1 to be true – following are the causes:

- C1 and C3 should be true
- C2 and C3 should be true
- C1 and C2 cannot be true together. This means C1 and C2 ae mutually exclusive.

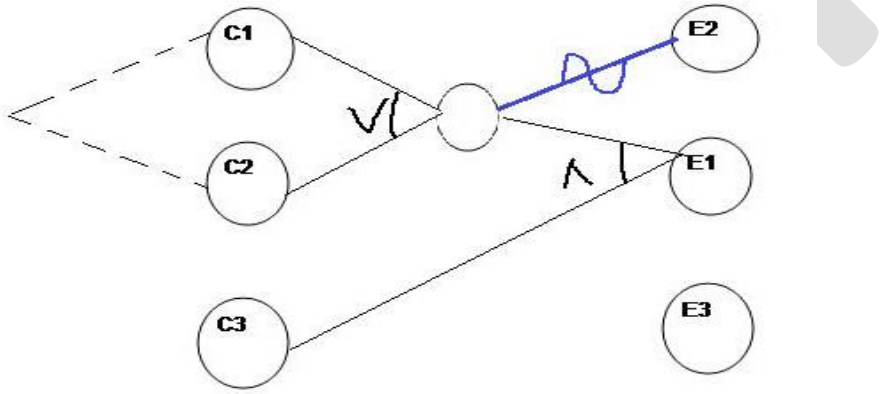
(C1 u C2) ∩ C3



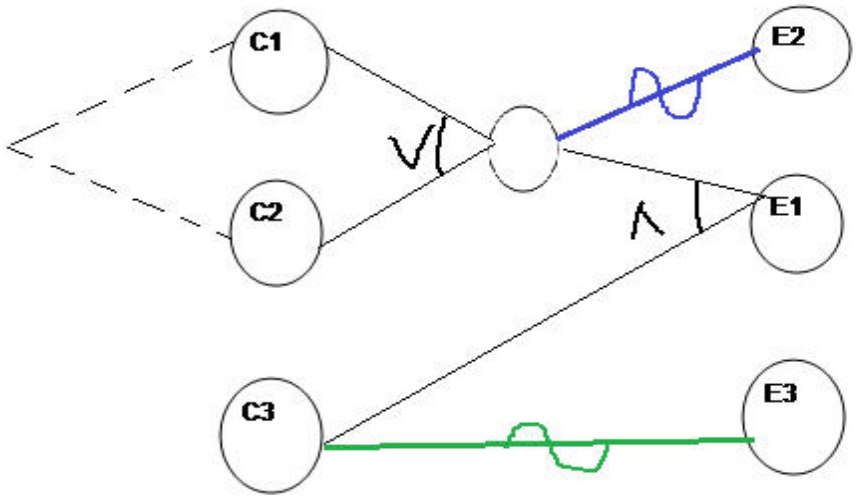
E2 states to print message "X". Message X will be printed when First character is neither A nor B.

Which means Effect E2 will hold true when either C1 OR C2 is invalid.

So the graph for Effect E2 is shown as (In blue line)



E3 states to print message "Y". Message Y will be printed when Second character is incorrect.
Which means Effect E3 will hold true when C3 is invalid. So the graph for Effect E3 is shown as (In Green line)



Event 1

Now for E1 to be "1" (true), we have the below two conditions –

C1 AND C3 will be true

C2 AND C3 will be true

Actions		
C1	1	
C2		1
C3	1	1
E1	1	1
E2		
E3		

Event 2

For E2 to be True, either C1 or C2 has to be false

Actions				
C1	1		0	
C2		1		0
C3	1	1	0	1
E1	1	1		
E2			1	1
E3				

Event 3

For E3 to be true, C3 should be false.

Actions						
C1	1		0		1	
C2		1		0		1
C3	1	1	0	1		
E1	1	1				
E2			1	1		
E3					1	1

Test cases

Actions	TC1	TC2	TC3	TC4	TC5	TC6
C1	1	0	0	0	1	0
C2	0	1	0	0	0	1
C3	1	1	0	1	0	0
E1	1	1	0	0	0	0
E2	0	0	1	1	0	0
E3	0	0	0	0	1	1

2.7. Compatibility testing

- Testing done to ensure that the product features work consistently with different infrastructure components is called compatibility testing.
- Actually Test case results not only depend on the product for proper functioning, they depend equally on the infrastructure for delivering functionality.

- When infrastructure parameters are changed, the product is expected to still behave correctly and produce the desired or expected result.
- The infrastructure parameter may be H/W, S/W or any other component.

The parameters that generally affect the compatibility of the product are

1. Processor (P-III, P-IV, XEON) and the No. Of processors.
2. Architecture and characteristics of the m/c (32 bit, 64 bit).
3. Recourse availability of the product (RAM, disk space, network card).
4. Operating system (windows, linux)
5. Middle tier infrastructure components (web server, application server, network server)
6. Backend components such as database server (oracle, sybase).
7. Services that require special h/w and s/w solutions.
8. Various technological components

Compatibility Matrix

C.M has its columns various parameters the combinations of which have to be tested.
Each row represents a unique combination of a specific set of values of the parameters.
Sample C.M for a mail application

SERVER	**APPLICATION SERVER**	**WEB SERVER**	**Browser**	**MS Office**
Windows 2000 advanced server	Windows 2000 advanced server With .Net framework	IIS 5.0	IE 6.0 and IE 5.5 SP2	Office 2K and XP
Windows 2000 advanced server	Windows 2000 advanced server With .Net framework	IIS 5.0	Netscape 7.1, and Mozila	Office 2K and XP

Backward compatibility testing

The testing that ensures the current version of the product continues to work with the older versions of the same product is called backward compatibility testing.

Forward compatibility testing

Provisions for the product to work with later versions of the product and other infrastructure components, keeping future requirements in mind.

*****

2.8. User Documentation Testing

- User documentation covers all the manuals, user guides, installation guides, setup guides, read me file, software release notes and online help that are provided along with the software to help the end user to understand the software system.
- U.D is done to ensure the Documentation matches the product.

Why user documentation testing?

- To check if what is stated in the document is available in the product.
- To check if what is there in the product is explained correctly in the document.

- When Product is upgraded, the corresponding product documentation should also get updated as necessary to reflect any changes that may affect a user.
- User documentation testing focuses on ensuring what the document exactly matches the product behaviour, by sitting in front of the system and verifying screen by screen, transaction by transaction and report by report.
- U.D- Also check spelling and grammar.
- A badly written installation document can put off a user and bias him against the product, even the product offers high functionality.
- It highlights the problems during reviews.
- Customer Satisfaction - Customer should get correct result after following the instructions. Otherwise they need supporting staff for that.
- If new programmers and testers joined in the group, U.D will be useful to know the functionality.
- Customer needs less training- U.D is user friendly.

*****

2.9. Domain Testing

- White box testing – Checking code
- Black Box testing – Checking functionality without coding knowledge. (Checking will be done by looking the specifications)
- Domain testing – Do not look the specification- Purely based on domain knowledge.
- It requires critical understanding of day-to-day business activities.
- Depth in business domain is a prerequisite for this testing.
- Give training for testing by taking people from the domain area (banking, insurance) – it reduces time and increases the testing effectiveness.
- DT- Need not to test several steps in design logic
- It‘s BBT- Check the denomination also.
- Generally DT is done after the BBT.
- DT ensures the software is written with the intelligence needed for domain.
- It examines the realistic business scenarios.
-

Example Banking : atm

- Go to the ATM
- PUT ATM card inside.
- Enter correct pin
- Choose cash withdrawal.
- Enter the amount
- Take the cash
- Exit and retrieve the card

Test Cases For One Rupee Coin Telephone Box

TC1: Pick up the Handset
Expected: Should display the message “Insert one rupee coin"

TC2: Insert the coin
Expected: Should display the message “Dial the Number“

TC3: When you get a busy tone, hang-up the receiver
Expected: The inserted one rupee coin comes out of the exit door.

TC4: Finish off the conversation and hang-up the receiver
Expected: The inserted coin should not come out.

TC5: During the conversation, in case of a local call, (assume the duration is of 60 sec), when 45s are completed
Expected: It should prompt you to insert another coin to continue by giving beeps.

TC6: In the above scenario, if another coin is inserted
Expected: 60 sec will be added to the counter.

TC7: In the TC5 scenario, if you don't insert one more coin.
Expected: The call gets ended.

TC8: Pick up the receiver. Insert appropriate one rupee coin; Dial the number after hearing the ring tone. Assume it got connected and you are getting the ring tone. Immediately you end up the call.
Expected: The inserted one rupee coin comes out of the exit door.

*****

2.10. Using white box testing /Test adequacy criteria

- WBT is checking if all the logical and data elements in the software unit are functioning properly. But, BBT is used for testing both small and large s/w components.
- But WBT is used for testing the small s/w component. Since the level of detail required for test design is very high, and the granularity of the items consider for testing is very small.
- WBT is to ensure that the internal components of a program are working properly.
- Its focus mainly on structural components like statements and branches.
- Testers need a framework – to decide the structural element, to select appropriate test cases. Framework exists in the form of **test adequacy criteria**.
- Test adequacy criterion is a stopping rule – since rules can be used to determine whether the sufficient testing has been carried out or not.

Application of TAC includes,

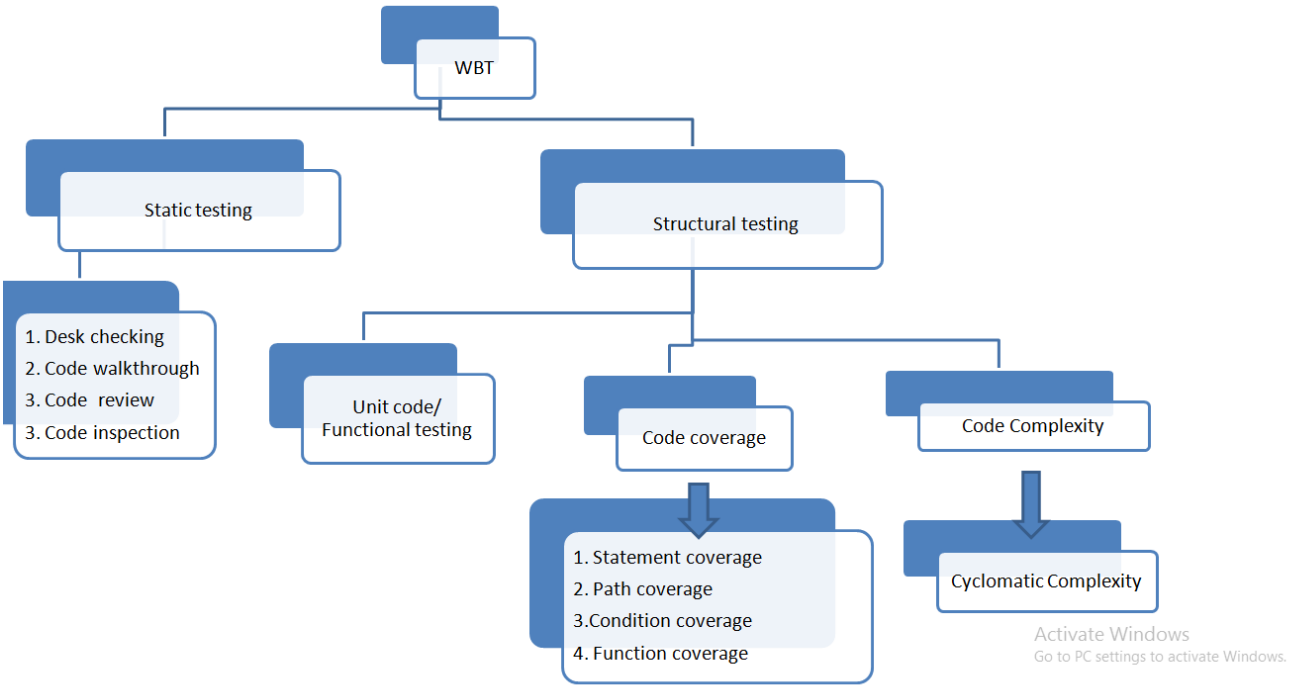
- Helps to select properties of a program.
- Helps to select test case set for a program based on the property.
- Indicate the testers whether or not testing can be stopped for the program

TYPES - TAC

- Program based adequacy criteria – If TAC focuses on the structural criterion is called PBTAC.
- Specification based adequacy criteria – If TAC focuses on program specification is called SBTAC.
  Example: statement coverage, functional coverage and branch coverage

*****

2.11. Types of White Box Testing / Static Testing



STATIC TESTING

- Requires only source code product, not the binaries or executable
- Static testing does not involve executing the programs on computers but involves select people going through the code to find

1. Whether cod works according to functional requirements.
2. The code has been written in accordance with the design developed earlier in the project life cycle.
3. Any functionality has been missed out
4. Code handles errors properly.

Static testing by Humans

1. When there are two variables with similar names and the programmer used a ‘wrong “ variable by mistake in an expression- Computer did not detect the error but execute the program and product incorrect results. But human being can spot such an error.
2. By making multiple humans read and evaluate the program, we can get multiple perspectives.
3. A human evaluation of the code can compare it against the specification or design and thus ensure that it does what is intended to do.
4. Human can identify the root cause of the problem.
5. By making humans test the code, before execution, computer resources can be saved.
6. Minimizes the delay in identification of the problem – cost is less

7. Finding defects at an earlier stage of testing- Reduce the pressure on programmer – less effort in coding

Static testing- Testing by human before the code is compiled and executed- so it is process oriented process or prevention oriented or quality oriented.

Types

1. Desk checking
2. Code Walkthrough
3. Code review
4. Code inspection

1. Desk checking

- Done manually by author of the code
- Verification of the code is done for its correctness
- Verification by comparing the code with specification
- Process over before the compilation and execution of the code

Advantages

1. Programmer understand the code easily (own code)

Disadvantages

1. Finding fault in own code is not effective method.
2. Normally they start to write new code instead of testing.
3. It is a person dependent and informal

2. Code Walkthrough

- Walkthrough are less formal than inspection
- Group oriented method
- Set of people look at the program code, and Raise the questions for the author
- Author should answer the questions.

3. Code Review

The code is checked on various factors.

Data item declaration related

- Are the name of the variable meaninfgul?
- Different names – confusing use of lower case and upper case letters.
- Are variables initialized?
- Are there similar sounding names (singular and plural names)
- All the common structures, constants and flags to be used, defined in a header file rather than in each file separately?

- Values of right data types being assigned to the variables?
- Is the access of data from any standard files, repositories or database s done through publicly supported interfaces?
- Pointers are properly initialized?
- Usage of Similar operators checked? (= and == & and&&)

Control Flow Related

- Are all the conditional paths reachable?
- Are all the individual conditions in a complex condition separately evaluated?
- I f there is a nested IF statement, are the THEN and ELSE parts appropriately delimited?

4. Code Inspection

It detects all faults, violations and other side effects. It is started after desk checking and walkthrough

Four roles in Inspections

1. Author of the code
2. Moderator – Formally run the inspection according to the process
3. Inspectors – Provide review comments on the code
4. Scribe – takes detailed notes during inspection meeting and circulates them to the inspection team after meeting

STATIC ANALYSIS TOOLS

Static analysis tools are Inspection, review and Manual works

Static analysis tool- Reduce the manual work and perform analysis on the code to find errors.

Reason for using SA tools:

1. Whether there are unreachable codes (GOTO statement – sometimes creates this situation)
2. Variables declared but not used
3. Mismatch in definition and assignment of values to variables
4. Calculation of cyclomatic complexity
5. Static analysis tool are considered as extension of compilers, Since they use same concepts and implementation to locate errors
6. Good compiler is also a static analysis tool. Most compilers provide different level of code checking

Ex: POSIX – Check consistency in coding (naming conventions, allowed data types, permissible programming constructs)

*****

2.12. Structural Testing

Structural testing is based on Code , Code structure and Internal design.
Structural testing runs by computer on the built product.
Types

1. Unit/Code functional

- Developer performs testing, giving various i/p values and check the actual test result with expected result. (Initial level)
- Repeating tests for multiple values – Increase the confidence level.
- Complex logic or condition – developer built a "debug version "of the product by putting intermediate print statement and then performing the test.
- After fixing of defects the intermediate print statements are removed.
- Run the product under a debugger or an IDE (Integrated Development Environment)
- (single stepping of instructions)

2. Code coverage testing

- Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by testing.
- Instrumentation of code: percentage of code covered by a test .
- Specialized tools are available to achieve instrumentation.
- Tools monitor the codes and keep an audit of what portions of code are covered (also maintain code that are covered frequently)

Types

1. Statement coverage
2. Path coverage
3. Condition coverage
4. Function coverage

1. Statement coverage

- Statement coverage - identifies the statements executed and where the code is not executed because of blockage.
- Each and every line of code needs to be checked and executed

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} \times 100\%$$

Example 1:
Consider the following program

Read X
Read y
If X > Y then Z=0

To achieve 100% statement coverage of this code segment, just one test case is required, one which ensures that variable X contains a value that is greater than the value of variable Y
Value of variables are X = 12 and Y=10.

Example 2:
Consider the program
1. Read X
2. Read Y
3. Z=X+2*Y
4. IF Z>50 then
5. Print large Z
6. Endif

Test Cases:
1. X = 2 , Y = 3
2. X = 0, Y = 25
3. X = 47 , Y = 1

For the first test case the value of Z = 8, it covers line 1, 2, 3, 4 and 6.
For the second test case value of Z=50, it covers line 1, 2, 3, 4 and 6.
For the third test case value of Z= 49, it also covers line 1, 2, 3, 4 and 6.

The statement coverage is 5/6 = 83%

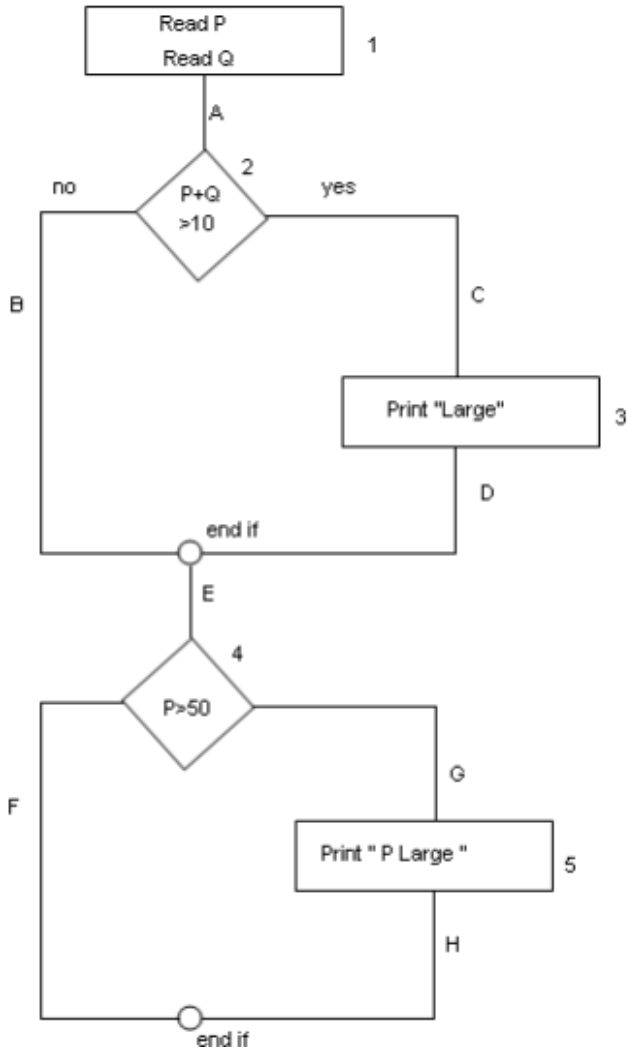
Suppose the test case consist of X = 20 and Y = 25, then value of Z = 70, and it will exercise all the six statements. **Now the statement coverage is 100%.**

2. Path Coverage

- In this the test case is executed in such a way that every path is executed at least once.

Example:
Read P
Read Q
IF P+Q > 100 THEN
Print “Large”

ENDIF
If P > 50 THEN
Print “P Large”
ENDIF
Consider the flowchart for the above program.



- Path Coverage ensures covering of all the paths from start to end.

All possible paths are-
1A-2B-E-4F
1A-2B-E-4G-5H
1A-2C-3D-E-4G-5H
1A-2C-3D-E-4F
So path coverage is 4. All the paths should be checked while testing the code.

3. Condition / Branch Coverage

$$\text{Condition coverage} = \frac{\text{Total decisions exercised}}{\text{Total number of decisions in program}}$$

Sometime all the conditions are not evaluated, even though the right path is chosen. Consider the example if path is selected and it is found to be true. So the second part will not be evaluated.

So path testing may not be sufficient, it is necessary to have test cases that exercise all the conditions.

4. Function Coverage

- Testing, to identify how many program functions are covered by test cases.
- The requirements of a product are mapped into functions during the design phase and each of the function form a logical unit.

Example: Database software

Inserting a row into the database- Function

Example: Payroll application

- Calculate tax - Function

Advantages

1. It is easier to identify the function in a program – it is easier to write test cases.
2. Function has a more logical mapping to requirements – Test coverage using traceability matrix.
3. Prioritizing the functions based on the importance of requirements.

*****

2.13. White Box testing- Coverage and control flow graph – Covering code logic

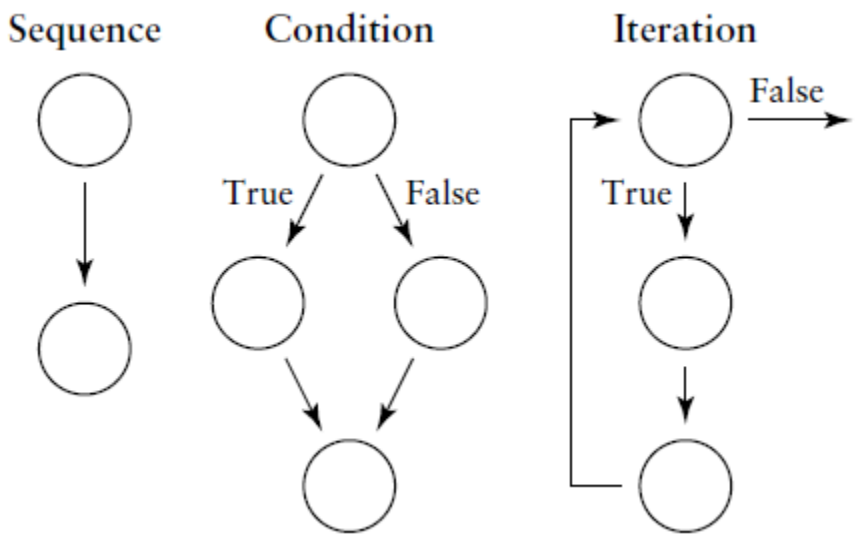
- Testing the Internal logic structure of the software under test.
- The tester's goal is to determine if all the logical and data elements in the software unit are functioning properly. This is called the white box, or glass box, approach to test case design.
- The knowledge needed for the white box test design approach often becomes available to the tester in the later phases of the software life cycle, specifically during the detailed design phase of development.
- This is in contrast to the earlier availability of the knowledge necessary for black box test design.

Coverage and control flow graph

- Coverage analysis is typically associated with the use of control and data flow models to represent program structural elements and data.

- The logic elements most commonly considered for coverage are based on the flow of control in a unit of code.
- For example,
  (i) Program statements;
  (ii) Decisions/branches (these influence the program flow of control)
  (iii) Conditions (expressions that evaluate to true/false, and do not contain any other true/false-valued expressions.
  (iv) Combinations of decisions and conditions;
  (v) Paths (node sequences in flow graphs)
- These logical elements are rooted in the concept of a program prime.
- **A program prime is an atomic programming unit. All structured programs can be built from three basic primes-sequential (e.g., assignment statements), decision (e.g., if/then/else statements), and iterative (e.g., while, for loops).**
- Graphical representations for these three primes are shown in figure.

Figure : Representation of program primes



- Using the concept of a prime and the ability to use combinations of primes to develop structured code, a (control) flow diagram for the software unit under test can be developed.
- The flow graph can be used by the tester to evaluate the code with respect to its testability, as well as to develop white box test cases.

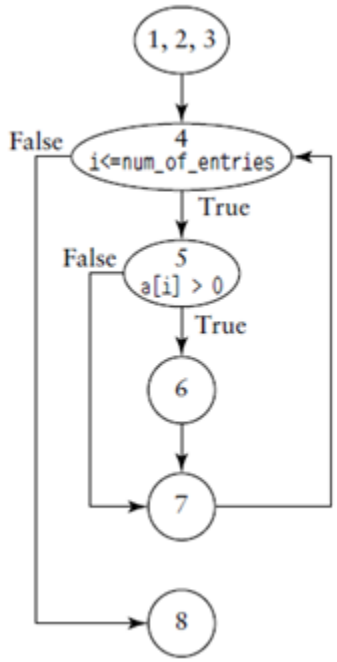
Figure: Code sample with branch and loop

```
/* pos_sum finds the sum of all positive numbers (greater than zero) stored in an integer
array a. Input parameters are num_of_entries, an integer, and a, an array of integers with
num_of_entries elements. The output parameter is the integer sume */

1.  pos_sum(a, num_of_entries, sum)
2.      sum = 0
3.      inti = 1
4.      while (i <= num_of_entries)
5.          if a[i] > 0
6.              sum = sum + a[i]
            endif
7.          i = i + 1
        end while
8.  end pos_sum
```

The following figure is the control flow graph of the above code.

Figure: Control flow graph



Flow graph

- Nodes represent sequential statements, as well as decision and looping predicates.
- Sequential statements are often omitted or combined as a block that indicates that if the first statement in the block is executed, so are all the following statements in the block.
- Edges in the graph represent transfer of control.
- The direction of the transfer depends on the outcome of the condition in the predicate (true or false).

- Commercial tools are available that will generate control flow graphs from code and in some cases from pseudo code.
- The tester can use tool support for developing control flow graphs especially for complex pieces of code.

Covering Code Logic

Testers are using tools and concepts to decide on the target logic elements (properties or features of the code) and the degree of coverage.

If the goal is to satisfy the statement adequacy/coverage criterion, then the tester should develop a set of test cases so that when the module is executed, *all* (100%) of the statements in the module are executed at least once.

STATEMENTS / TEST CASES	T1	T2	T3
1	X		
2		X	X
3		X	
4			X

- For example, to achieve complete (100%) decision (branch) coverage test cases must be designed so that each decision element in the code (if-then, case, loop) executes with all possible outcomes at least once.
- In terms of the control flow model, this requires that all the edges in the corresponding flow graph must be exercised at least once.
- Complete decision coverage is considered to be a stronger coverage goal than statement coverage since its satisfaction results in satisfying statement coverage as well (covering all the edges in a flow graph will ensure coverage of the nodes).
- The statement coverage goal is so weak that it is not considered to be very useful for revealing defects.
- Decision (branch) coverage for the code example -Figure requires test cases to check the two decision statements.
- Input values must ensure execution the true/false possibilities for the decisions in line 4 (while loop) and line 5 (if statement)
- The “if” statement has a “null else” component, that is, there is no “else” part. However, we include a test that covers both the true and false conditions for the statement.

Possible test case that satisfies 100% decision coverage is shown in Table

Decision or branch	Value of variable i	Value of predicate	Test case: Value of a, num_of _entries
			a=1, -45, 3 Num_of_entries = 3
While	1	True	
	4	False	
If	1	True	
	2	False	

Consider the below example:

if(age <65 and married true)
do X
do Y .....
else
do Z

Condition 1: Age less than 65
Condition 2: Married is true

Test cases for simple decision coverage

Value for age	Value for married	Decision outcome Compound predicate as a whole	Test case ID
30	True	True	1
75	True	False	2

The above test cases would not exercise the possible outcome for married as false. A defect in the logical operator for condition 2, for example, may not be detected. All possible outcomes for the decision as a whole are not exercised so it would not satisfy decision/condition coverage criteria.

Value for age	Value for married	Condition 1 outcome	Condition 2 outcome	Decision outcome Compound predicate as a whole	Test case ID
30	**True**	**True**	**True**	**True**	**1**
75	**True**	**False**	**True**	**False**	**2**

30	**False**	**True**	**False**	**False**	**3**

- The criteria described above do not require the coverage of all the possible combinations of conditions.
- This is represented by yet another criterion called multiple condition coverage where all possible combinations of condition outcomes in each decision must occur at least once when the test cases are executed.
- That means the tester needs to satisfy the following combinations for the example decision statement:

Condition 1	**Condition 2**
True	False
True	True
False	True
False	False

- In most cases the stronger the coverage criterion, the larger the number of test cases that must be developed to insure complete coverage.

*****

2.14. Path - White box testing

- Tools are available to generate control flow graphs. These tools typically calculate a value for a software attribute called McCabe's Cyclomatic Complexity *V*(*G*) from a flow graph.
- The cyclomatic complexity attribute is very useful to a tester.
- The complexity value is usually calculated from the control flow graph (*G*) by the formula

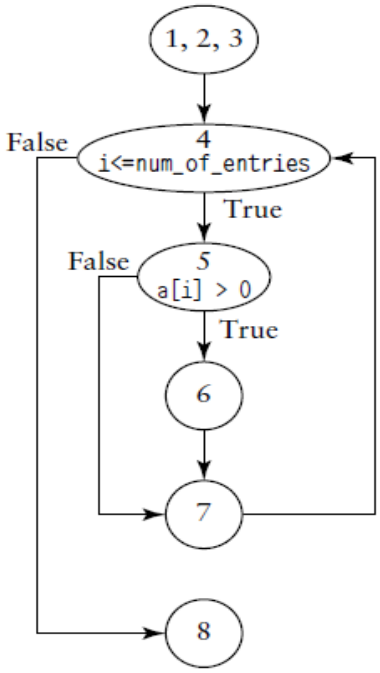
V(G) = E-N+2

E - number of edges in the control flow graph
N - number of nodes.
Here, E=7, N=6
V (G)= 3

Cyclomatic complexity value indicates the number of test cases needed for branch coverage in a module of structured code**.**

- A path is a sequence of control flow nodes usually beginning from the entry node of a graph through to the exit node.
- A path may go through a given segment of the control flow graph one or more times.

- For example, Path ---- 1-2-3-4-8 where the dashes represent edges between two nodes.
- For example, the sequence "4-8" represents the edge between nodes 4 and 8.
- Cyclomatic complexity "independent paths" in the graph.
- The independent paths are defined as any new path through the graph that introduces a new edge that has not be traversed before the path is defined.



A set of independent paths for a graph is sometimes called a basis set. For the given flow graph
(i) 1-2-3-4-8
(ii) 1-2-3-4-5-6-7-4-8
(iii) 1-2-3-4-5-7-4-8

Path Coverage

- It is the strongest program-based testing criterion, and it calls for complete path coverage; that is, every path (as distinguished from independent paths) in a module must be exercised by the test set at least once.
- This may not be a practical goal for a tester.
- For example, even in a small and simple unit of code there may be many paths between the entry and exit nodes.
- Adding even a few simple decision statements increases the number of paths.
- Every loop multiplies the number of paths based on the number of possible iterations of the loop since each iteration constitutes a different path through the code.

- Thus, complete path coverage for even a simple module may not be practical, and for large and complex modules it is not feasible.
- . Under these circumstances coverage goals are best expressed in terms of the number of feasible or achievable paths, branches, or statements respectively.
- The basis set is a special set of paths and does not represent all the paths in a module; it serves as a tool to aid the tester in achieving decision coverage.

*****

2.15 Error Guessing

- Error guessing is a test case design technique where the tester has to guess what faults might occur and to design the tests to represent them.
- Previous testing knowledge – Helpful for current testing

Example:

Test cases for a pen

- To check the pen type
- To check the pen cap is present or not
- To check the pen ink is filled or not
- To check the pen writing or not
- To check the ink color i.e black or blue
- To check the pen color
- To check whether the pen is used to write all types of papers or not
- To check the ink capacity of the pen
- To check the pen product by fiber or plastic

*****

UNIT-III – LEVELS OF TESTING

The need for Levers of Testing – Unit Test – Unit Test Planning – Designing the Unit Tests – The Test Harness – Running the Unit tests and Recording results – Integration tests – Designing Integration Tests – Integration Test Planning – Scenario testing – Defect bash elimination System Testing – Acceptance testing – Performance testing – Regression Testing – Internationalization testing – Ad- hoc testing – Alpha, Beta Tests – Testing OO systems – Usability and Accessibility testing – Configuration testing – Compatibility testing – Testing the documentation – Website testing.

NEED FOR LEVELS OF TESTING

- Execution based software testing, especially large systems, is usually carried out at different levels mostly 3-4 levels.
- Major Phases of testing:
    - **Unit Testing**
    - **Integration Testing**
    - **System Testing**

✓ A Principal goal is to detect functional and structural defects in the unit.

✓ At the integration level several components are tested as group, and tester investigates component interaction.

✓ At the system level the system as a whole is tested and a principal goal is to evaluate attribute such as ability, reliability and performance

Level of Testing and Software Development Paradigms

- The approach used to design and develop a software system has an impact on how testers plan and design suitable tests.
- The TWO major approaches to system development are **Bottom up and Top down approaches.**

- These approaches are supported by two major types of programming languages- **procedure Oriented and Object Oriented.**

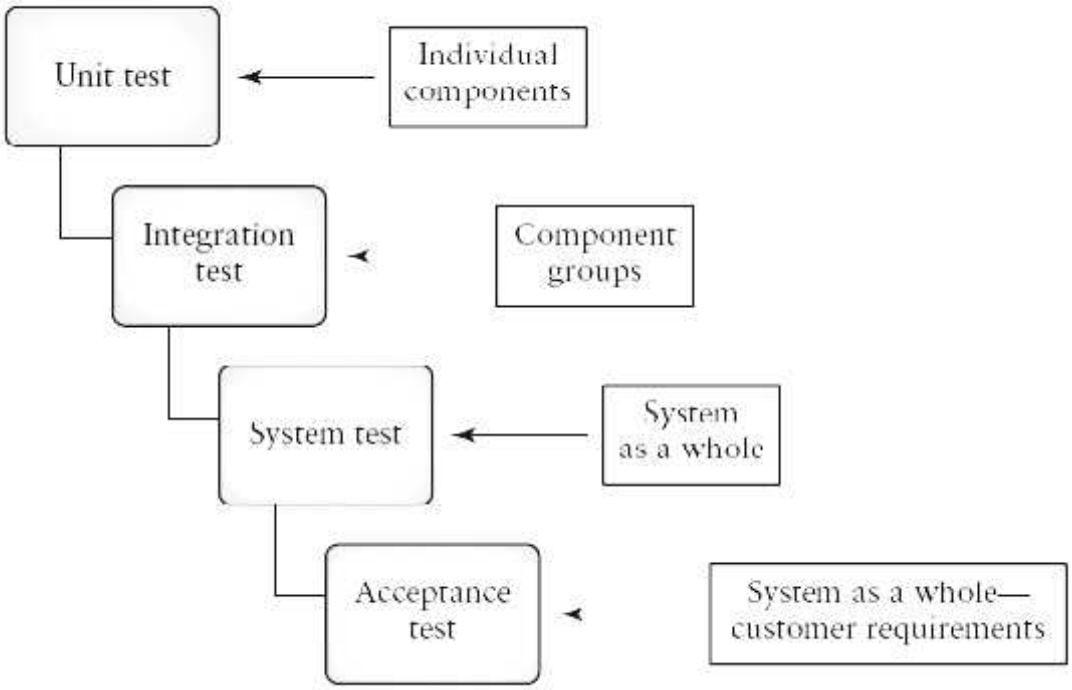


FIG 3.1 LEVELS OF TESTING

- The different nature of the code produced requires testers to use different strategies to identify and test components and component groups.
- Systems developed with procedural languages are generally viewed as being composed of passive data and active procedures
- When test cases are developed the focus is on generating input data to pass to the procedures (or functions) in order to reveal defects.
- Object oriented systems are viewed as being composed of active data along with allowed operations on that data, all encapsulated within a unit similar to abstract data type.

UNIT TESTING

Unit test: Functions, Procedures, Classes, and Method as Unit

- **A workable definition for a software unit is as follows**
  - A Unit is the smallest possible testable software component
    - It can be characterized in several ways. For example a unit in a typical procedure oriented software system:
      - perform a single cohesive function
      - can be compiled separately
      - is a task in a work breakdown structure (from the manager's point of view)
      - contains code that can fit on a single page or screen.
- A unit is traditionally viewed as a function or procedure implemented in a procedural (imperative) programming language.
- A unit may also be a small sized COTS component purchased from an outside vendor that is undergoing evaluation by the purchaser, or simple module retrieved from an in-house reuse library

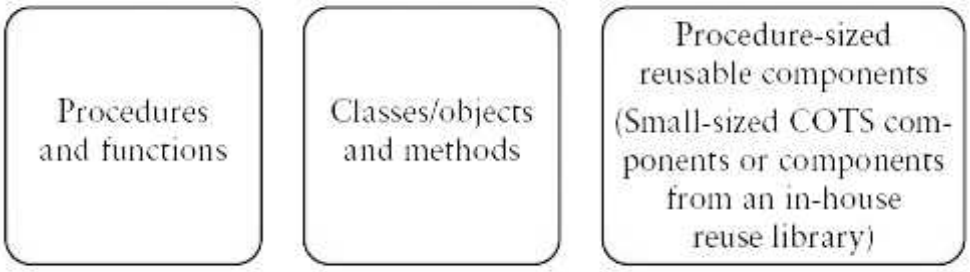


FIG. 3.1 SOME COMPONENTS SUITABLE FOR UNIT TEST

Unit Test- The Need for Preparation

- The principal goal for unit testing is insure that each individual software is functioning according to its specification
- Good testing practice calls for unit tests that are planned and public.

- Planning includes designing test to reveal defects such as functional description defects, algorithmic defects, data defects, and control logic and sequence defects.
- Resources should be allocated and test cases should be developed, using both white and black box test design strategies.
- The unit should be tested by an independent tester (other than testers) and the test results and defects found should be recorded as apart of the unit history (made public).
- Each unit should also be reviewed by a team of reviewers, preferably before the unit test.
- Unit test in many cases is performed informally by the unit developer soon after the module is completed, and it compiles cleanly.
- Some developers also perform an informal review of the unit.
- To prepare for unit test the developers/ testers must perform several tasks. These are:-

    1. **Plan the general approach to unit testing**
    2. **Design the test cases, and test procedures**
    3. **Define relationships between the test**
    4. **Prepare the auxiliary code necessary for unit test.**

Unit test Planning

- A general unit test plan should be prepared.
- It may be prepared as a component of the master test plan or a stand alone plan.
- It should be developed in conjunction with the master plan and the project plan for each project
- **Phase 1 : Describe Unit test Approach and Risks**

In this phase of unit test planning the general approach to unit test planning is outlined: The test planner:

- identifies test risks
- describes techniques to be used for designing the test cases for units
- describes techniques to be used for data validation and recording of test results
- describes the requirement for test harness and other software that interfaces with unit to be tested eg:- any special software needed for testing object oriented units

- During this phase the planner also identifies completeness requirements ie what will be covered by the unit test and to what degree (state, functionality, control, and data flow patterns)
- Planner also identifies termination condition for unit test.
- They include coverage requirement and special cases
- Special cases may result in abnormal termination of unit test
- Planner estimate the resources needed for unit test such as hardware, software and staff and develop tentative schedule under constraints identified at that time

Phase 2:- Identify Unit Features to be Tested

- This phase requires information from the unit specification and detailed design description
- The planner determines which features of each unit will be tested, for example functions, performance requirement, state and state transition, control structures, messages and data flow patterns
- Some features will be covered by the tests, they should be mentioned and risks of not testing them be assessed.
- Input and output of each test unit should be identified.

Phase 3: Add levels of Detail to the Plan

- In this phase the planner refines the plan as produced in the previous two phases.

- The planner adds new details to the approach, resource and scheduling portions of the unit test plan.
- **Eg:-** Existing test cases that can be reused for this project can be identified in the phase.
- Unit availability and integration scheduling information should be included in the revised version of the test plan.
- Planner must be sure to include a description of how test results will be recorded.
- Test related documents that will be required for this task eg. test logs, test incidents report should be described.

Designing the Unit test

Part of the preparation work for unit test involves unit test design. It is important to specify

1. The test cases (including I/O and expected output for each test cases) and
2. The test procedures (steps required run the test)

- As a part of the unit test design process, developers / tester should also describe the relationship between the tests. Test suites can be defined that binds related tests together as a group. All of this test design information is attached to the unit test plan. Test cases, test procedures and test suites may be reused from the past projects if the organization has been careful to store them so that they can be easily retrievable and reusable
- Test case design at unit level can be based on the use of black and white box design strategies. Both of these approaches are useful for designing test cases for functions and procedures.

The Test Harness

- ***The auxiliary code developed to support testing of units and components is called as test harness***

- ***The harness consist of drivers that call the target code and stubs that represent modules it calls.***
- The development of drivers and stubs requires testing resources. The drivers and stubs must be tested themselves to insure they are working properly and that they are reusable for subsequent releases of the software
- Drivers and stubs can be developed at several levels of functionality
- **Eg:-** a driver could have the following options and combinations of options:

  i. Call the target unit
  ii. Do 1, and pad pass input parameters from the table
  iii. Do 1,2, and display parameters
  iv. Do 1,2,3 and display result (output parameters)

The stub should also exhibit different levels of functionality. For example a stub could

  i. Display a message that it has been called the target unit
  ii. Do1, and display any input parameters passes from the target units
  iii. Do1,2, and pass back result from a table
  iv. Do1,2,3 and display result from table

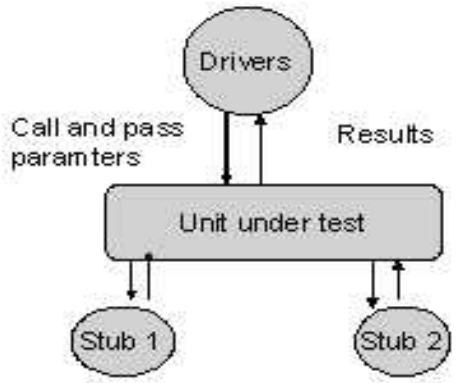


FIG. 3.5 THE TEST HARNESS

Running a Unit tests and Recording Results

Unit test can begin when

- the unit become available from the developers
- the test cases have been designed and reviewed
- the test harness and any other supplemental to supporting tools

The status of the test efforts for a unit, and a summary of test results must be recorded in a unit test worksheet.

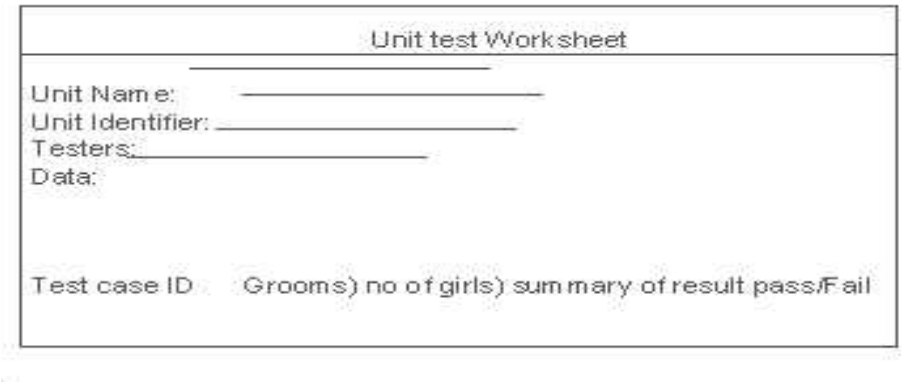


FIG. 3.6 SUMMARY WORK SHEET FOR UNIT TEST RESULTS

- The tester must determine from the results whether the unit has passed or failed the test
- If the test is failed, the nature of the problem should be recorded in what is sometimes called the test incident report.
- Differences from expected behavior should be described in detail.
- When a unit fails a test there may be several reasons for the failure. The most likely reason for the failure is a fault in the unit implementation

i. A fault in the test case
ii. A fault in test procedures execution
iii. A fault in the test environment
iv. A fault in the unit design

- When a unit has been completely tested and finally passes all of the required tests it is ready for integration.

INTEGRATION TESTING

Testing the interaction between the modules and interaction with other systems externally.

Integration Test-Goals

- Integration test for procedural code has two major goals:
  i. To detect defects that occur on the interfaces of units
  ii. To assemble the individual unit into working subsystem and finally a complete system that is ready for system test.
- In unit test the tester attempts to detect defects that are related to the functionality and structure of the unit.
- Integration test should only be performed on unit that have been reviewed and have successfully passed unit testing.
- Integration testing works best as an iterative process procedural oriented system.
- One unit at a time integrated into a set of previously integrated modules which have passed a set of integration tests.
- The interface and functionality of the new unit is combination with the previously integrated units is tested
- When a subsystem is built from units integrated in the stepwise manner, then performance, security and stress test can be performed in this subsystem.

Designing Integration tests

- Integration test for procedural software can be designed using a black or white box approach.
- Some unit test can be reused.

- Since many error occur at module interfaces, test designers need to focus on exercising all input/output parameter pairs and all calling relationships
- The tester needs to insure the parameters are of the correct type and in the correct order.

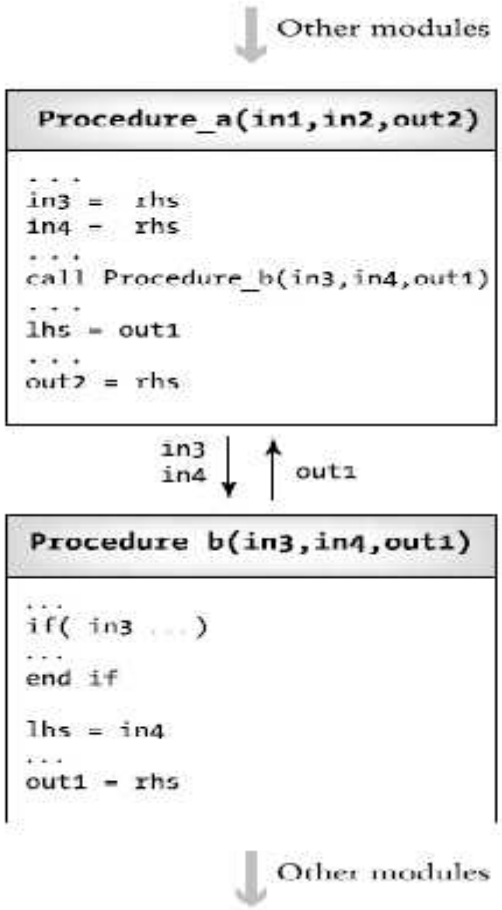


FIG.3.7 EXAMPLE INTEGRATION OF TWO PROCEDURES

- In the example above, Procedure_b has two input parameters int3,int4.
- Procedure_b uses those parameters and then returns a value for the output parameter out1.
- The terms such as *rhs* and *lhs* could be any variable or expression.
- The parameter could be involved in a number of *def* and/or *use* data flow patterns
- The actual usage patterns of the parameters must be checked at integration time.

- Some black box test used for module integration may be reusable from unit testing.
- When units are integrated and subsystems are to be tested as a whole, new tests will a have to be designed to cover the functionality tests at the integration level are the requirements document and the user manual.
- Tester need to work with requirement analyst to insure that the requirements are testable, accurate and complete.
- Black Box tests should be developed to insure proper functionally and ability to handle subsystem stress.
- Integration Testing of clusters of classes also involves building test harness which in this case are special classes of objects built for testing
- In class testing we evaluated intra class method interaction, and at the cluster level we test inter class method interaction as well
- We want to insure that message are being passed properly to interfacing objects, object state transition are correct when specific events occur , and that the cluster are performing their required functions.

Integration test Planning

- Integration test must be planned.
- Planning can begin when high level design is complete so that the system architecture is defined.
- Documents relevant to integration test planning are the requirement document, the user manual and usage scenarios.
- These documents contain structure charts, the state charts and data dictionary , cross reference table, module interface description
- The strategy for integration of the unit must be defined .
- The detailed description is given :-
    - Cluster this cluster is dependent on

- A natural language description of the functionality of the cluster to be retested.
- List a classes in the cluster
- A set of cluster test cases

What is Integration Testing:-

- ✓ **Integration testing is both a type of testing and phase of testing**.
- ✓ Integration is defined to be a set of interactions, all defined interaction among the components need to be tested.

Integration Testing As a Type of Testing :-

- ✓ Integration testing means testing of interfaces.
- ✓ They are ***Internal*** Interfaces and ***Exported*** or ***External Interfaces***
- ✓ Internal Interfaces are those that provide communication across two modules within a project or product, internal to the product, and not exposed to the customer or external developers.
- ✓ Exported interfaces are those that are visible outside the product to third party developers and solution providers.

"Integration Testing Type Focuses on testing interfaces that are "Implicit and Explicit" and "Internal and External"

Methodologies for deciding the order of integration testing

There are several methodologies available, to in decide the order for integration testing.

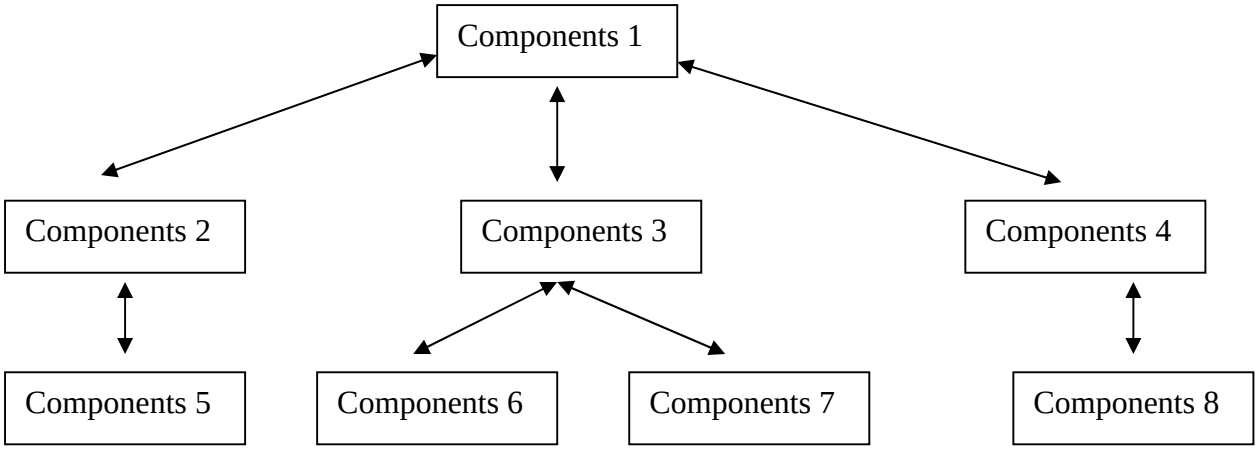
These are as follows:-

1. **Top Down Integration**
2. **Bottom up Integration**
3. **Bi-Directional Integration**
4. **System Integration**

Top-Down Integration:

Integration Testing involves testing the topmost component interface with other components in same order as you navigate from top to bottom, till we cover all the components.

To understand this methodology, we will assume that a new product/ software development where components become available one after another in the order of component number specified .The integration starts with testing the interface between Component 1 and Component 2 .To complete the integration testing all interfaces mentioned covering all the arrows, have to be tested together. The order in which the interfaces are to be tested is depicted in the table below. In an incremental product development, where one or two components gets added to the product in each increment, the integration testing methodology pertains to only those new interfaces that are added .



Order of testing Interfaces

Steps	***Interfaces Tested***
1	***1-2***
2	***1-3***
3	***1-4***
4	***1-2-5***
5	***1-3-6***

6	***1-3-6-(3-7)***
7	***(1-2-5)-(1-3-6-(3-7))***
8	***1-4-8***
9	***(1-2-5)-(1-3-6-(3-7))-(1-4-8)***

Bottom-Up Integration:-

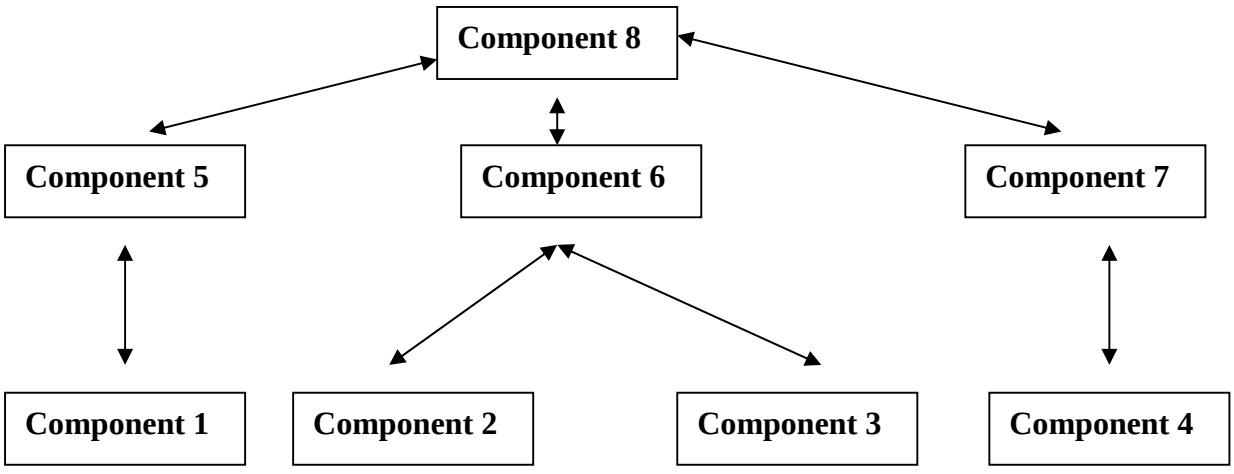
Bottom-up integration is just the opposite of top-down integration, where the components for a new product development become available in reverse order, starting from the bottom.

Testing takes place from the bottom of the control flow upwards.

Components or systems are substituted by drivers.

Logic Flow is from top to bottom and integration path is from bottom to top. Navigation in bottom-up integration starts from component 1 converting all sub systems , till components 8 is reached. The order is listed below. The number of steps in the bottom up can be optimized into four steps.

By combining step2 and step3 and by combining step 5-8 in the previous table.



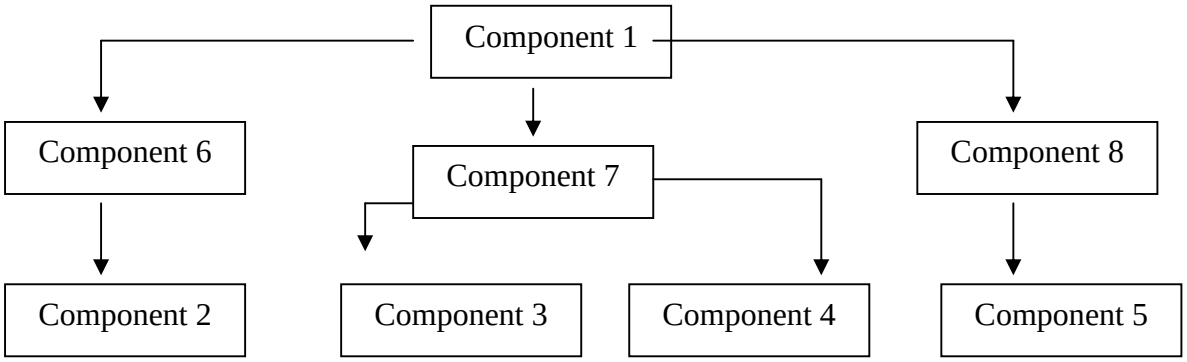
Order of Interface tested using Bottom Up Integration

Steps	*Interfaces Tested*
1	*1-5*
2	*2-6,3-6*
3	*2-6-(3-6)*
4	*4-7*
5	*1-5-8*
6	*2-6-(3-6)-8*
7	*4-7-8*
8	*(1-5-8)-(2-6-(3-6)-8)-(4-7-8)*

Integration:

Bi directional integration is a combination of the top-down and bottom –up integration approaches used together to derive integration steps.

This approach is also called as "*Sandwich Integration*".



Steps for Integration Using Sandwich Testing :-

Steps	*Integration Tested*
1	*6-2*

2	***7-3-4***
3	***8-5***
4	***(1-6-2)-(1-7-3-4)-(1-8-5)***

System Integration:

System Integration means that all the components of the system are integrated and tested as a single unit.

Integration testing, which is testing of interface, can be divided into two types:-

- ☞ **Components or Sub-System Integration**
- ☞ **Final Integration testing or system Integration**

Big bang Integration is deal for a product where the interfaces are stable with less number of defects.

Choosing Integration Methods:-

Sno	***Factors***	***Suggested Integration Methods***
1	**Clear Requirement and Design**	***Top Down***
2	**Dynamically, Changing Requirements, Design, Architecture**	***Bottom-Up***
3	**Changing Architecture, Stable Design**	***Bi-Directional***
4	**Limited Changes to existing Architecture with less Impact**	***Big Bang***
5	**Combination of all the above**	***Select one of the above after careful analysis***

Integration Testing As a Phase of testing :-

"All testing activities that are conducted from the point where two components are integrated to the point where all system components work together are considered a part of the integration testing phase."

The Integration testing phases focuses on finding defects which predominantly arise because of combining various components for testing, and should not be focused on for component or few components .Integration testing as a type focuses on testing the interfaces. This is a subnet of the integration testing phase.

SCENARIO TESTING

Scenario testing is defined as a "set of realistic user activities that are used for evaluating the products" .It is also defined as testing involving customer scenarios.

There are two methods to evolve scenario

1. **System Scenario**
2. **Use case Scenario/ Role Based Scenarios.**

System Scenario:-

System Scenario is a method where by the set of activities used for scenario testing covers several components in the system.

The **following approaches** can be used to develop system scenarios.

Story-line:

Develop a story-line that combines various activities of the product

Life-cycle / state transitions:

Consider an object, derive the different transitions / modification that happen to the object and derive scenarios to cover them

Deployment / implementation details from customer:

Develop a scenario from a known customer Deployment / implementation details and create a set of activities by various users in that implementation

Business verticals:

Visualizing how a product / software will be applied to different business verticals and create a set of activities as scenarios (e.g., insurance, life sciences)

Battle-ground scenarios:

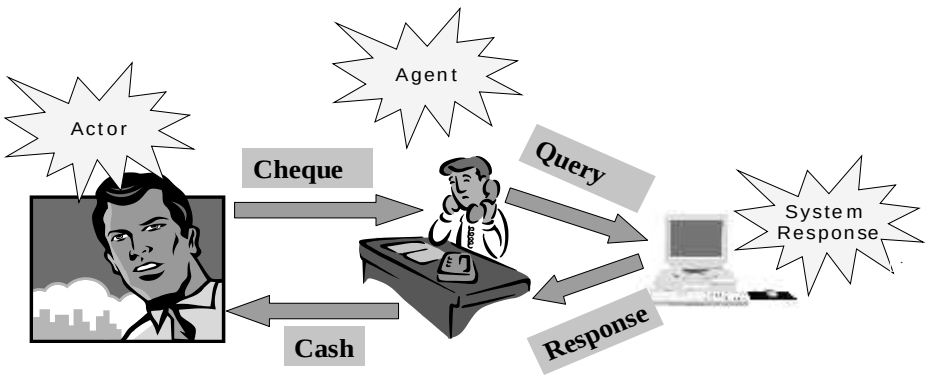
Create some scenarios to justify "the product works" and some scenarios to "try and break the system" to justify "the product doesn't work."

Use Case Scenarios:-

- ✓ ***A use case Scenario is a stepwise procedure on how a user intends to use a system, with different user roles and associated parameters.***
- ✓ A use case scenario can include stories, pictures and deployment details.
- ✓ Use cases are useful for explaining customer problems and how the software can solve those problems without any ambiguity

Example:-

The scenario below is an example of withdrawing a cash from a bank. A customer fills up a cheque and gives it to an official in the bank. The official verifies the balance in the account from the computer and gives the required cash to the customer .The customer in this example is a actor, the clerk the agent , and the response given by the computer which gives the balance in the account , is called the system response.



Actor	System Response
User would like to withdraw cash and inserts the card in ATM machine	Request for Password or PIN (Personal identification number)
User Fill-in the Password or PIN	
	Validate the password or PIN
	Give a list containing types of accounts
User selects an account type	
	Ask the user for amount
User Fill-in the amount of cash required	
	Check availability of funds Update account balance Prepare receipt Dispense cash
Retrieve cash from ATM	
	Print receipt

Actor and System Response in Use Case for ATM cash withdrawal

DEFECT BASH

- **Defect bash is an *ad hoc* testing, done by people performing different roles to bring out all types of defects.**
- It is very popular among application development companies, where the products can be used by people who perform different roles.
- The testing by all the participants during the defect bash is not based on written test cases.
- Defect bash brings together plenty of **good practices** that are popular in testing industry. They are as follows :-
    1. Enabling people to "***cross boundaries and test beyond assigned area***"
    2. Bringing different people performing different roles together in the organization for testing - "***Testing isn't for testers alone***"
    3. Let everyone in organization use the product before delivery - "***Eat your own dog food***"

4. Bringing fresh pairs of eyes to uncover new defects – "***Fresh eyes have less bias***"
5. Bringing in people who have different levels of product understanding, to test the product together randomly – "***Users of software are not the same***"
6. Testing doesn't wait for the time taken for documentation – "***Does testing wait till all documentation is done?***"
7. Enabling people to say the "system works" as well as enabling them to "break the system" – "***Testing isn't to conclude that the system works or doesn't work***"

Even though it is said that defect bash is an ad hoc testing, not all activities of defects bash are unplanned. All the activities in the defect bash are planned activities, except for what to be tested .

It involves **several steps:-**

1. **Choosing the frequency and duration of defect bash.**
2. **Selecting the right product build.**
3. **Communicating the objectives of each defect bash to everyone**
4. **Setting up and monitoring the lab for defect bash.**
5. **Taking action and fixing issues.**
6. **Optimizing the effort involved in defect bash.**

1. **Choosing frequency and duration**
    - Too frequent or too few rounds may not meet objective
    - Optimize duration involved
2. **Selecting right product build**
    - Good-quality product
    - Regression tested build
    - Too many defects spoil confidence
3. **Communication objective of defect bash**
    - Purpose & objective has to be clear

- Areas of focus to be communicated
- Defects that can be found easily by test team shouldn't be objective

4. **Setting up and monitoring lab**
    - Right configuration and resources
    - Easy install & set-up help
    - Optimized for both functional & non-functional defects
    - Monitor all resources (RAM, disk, CPU, network)
5. **Taking actions and fixing issues**
    - Duplicate defects
    - Not possible to look at each defect alone due to volume
    - Code reviews and inspections
    - Communication to all users on defects and their resolution

SYSTEM TESTING

System Test- The Different Types

- When integration tests are completed, a software system has been assembled and its major subsystems have been tested.
- System test planning should begin at the requirement based (black box) test.
- System test planning is a complicated task. There are many components of the plan that need to be prepared such as test approaches, costs, schedules, test cases and test procedures.
- System testing itself requires large amount of resources.
- **The goal is to ensure that the system performs according to its requirements.**
- **System test evaluates both functional behavior and quality requirement such as reliability, usability, performance and security.**

- **The phase of testing is especially useful for detecting external hardware and software interface defects.** Eg:- those causing race conditions, deadlocks, problems with interrupts and exception handling.

- There are several types of system tests
    - Functional testing
    - Performance Testing
    - Stress testing
    - Configuration testing
    - Security Testing
    - Recovery testing
    - Two other types of system testing called reliability and usability testing.

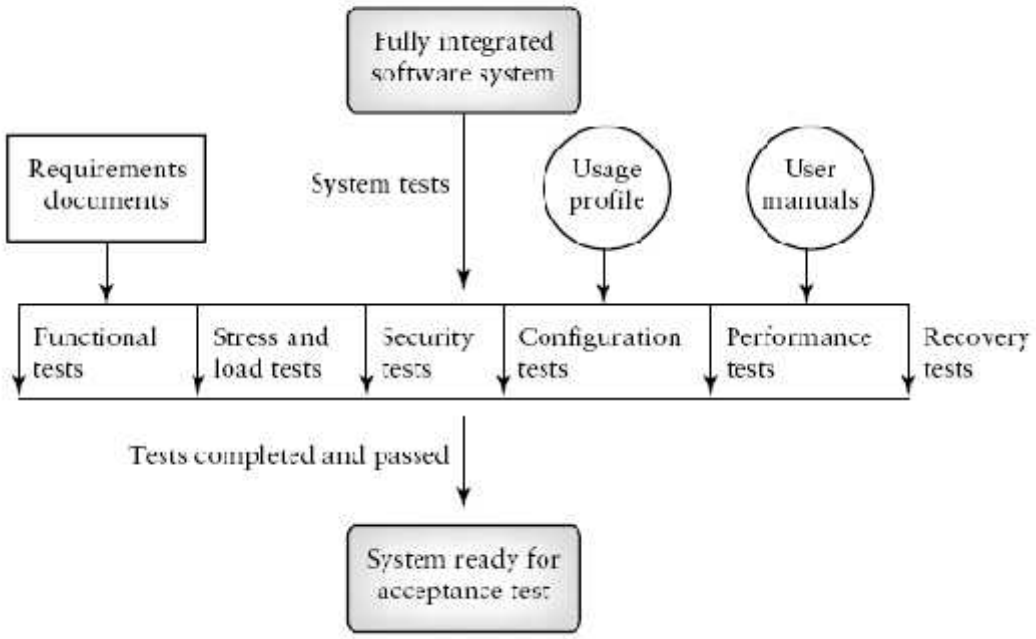


FIG 3.8 TYPES OF SYSTEM TESTS

An important tool for implementing system tests is a load generator. A load generator is essential for testing quality requirements such as performance and stress.

- A load is a serious of input that stimulated a group of transaction
- A transaction is a unit of work seen from the system user's view. A transaction consist of set of operations that may be performed by a person , software system or a device that is outside the system.
- A use case can be used to describe a transaction. If you were system testing a telecomm system you would need a load that simulated a series of phone calls (transactions) of particular types and lengths arriving from different locations
- A load can be a real load, which is we can put the system under test to real usage by having actual telephone users connected to it.
- Loads can also produced by tools called load generators; they will generate test input data from system test. Load generators can be simple tools that outputs a fixed set of predetermined transaction

1.Performance Testing

- To evaluate the time taken by the system to perform its required function in comparison with different versions of the same product

2.Scalability testing

- To find out the maximum capability of the system parameters

3.Reliability testing

- To evaluate the ability of the system to perform its required functions repeatedly for a specified period of time

4.Stress testing

- Evaluating a system beyond the limits of the specified requirements to ensure the system does not break down unexpectedly

4.Interoperability testing

- To ensure that two or more products can exchange information, use the information and work closely

5.Localization testing

- Testing conducted to verify that the localized product works in different languages

Why system testing done?

- Helps in identifying as many defects as possible before the customer finds them in the deployment
- Last chance for the testing team to find any remaining defects before the product is handed over to the customer
- Objective to find product level defects
- Test both functional & non functional aspects of the product
- Build confidence in the product
- Test the product behavior in a complete and realistic environment
- Ensure all the requirements are met and ready the product for acceptance testing

FUNCTIONAL TESTING

- **Functional tests at system level are used to ensure that the behavior of the system adheres to the requirements specification.**
- All functional requirements for the system must be achievable by the system.
- For example , if a personal finance system is required to allow users to set up account, add, modify and delete entries in the accounts, and print reports, the function based system and acceptance test must ensure that the system can perform these tasks
- **Functional test are black box in nature**
- **The focus is on the inputs and proper output for each function**
- **Improper and illegal inputs must also be handled by the system**
- System behavior under the latter circumstances.

- **<u>The test should focus on the following goals</u>**
  - ✓ All types or classes of legal input must be accepted by the software
  - ✓ All classes of illegal inputs must be rejected
  - ✓ All possible classes of system output must exercised and examined
  - ✓ All effective system states and state transition must be exercised and examined
  - ✓ All functions must be exercised

PERFORMANCE TESTING

- ℘ **The goal of system performance tests is to see if the software meets performance requirements.**
- ℘ **Confirms whether there are any hardware or software factors that impact on the systems requirement.**
- ℘ **Resources for the performance testing must be allocated in the system test plan**
- There are two major requirements:

- **Functional Requirement**: Users describe what function the software should perform. We test for compliance of these requirements at the system level with the functional based system test.
- **Quality Requirement**:- There are nonfunctional in nature but describes quality levels expected for the software. One example of a quality requirement is performance level, the users may have objectives for the software system in terms of memory use, response time, throughput and delays
  - Performance testing allows the testers to tune the system, ie to optimize the allocation of system resource.
  - Performance objectives must be articulated clearly by the users/client in the requirement documents , and stated clearly in the system test plans

- Objective must be quantified
- Example of resources are given below in a diagram

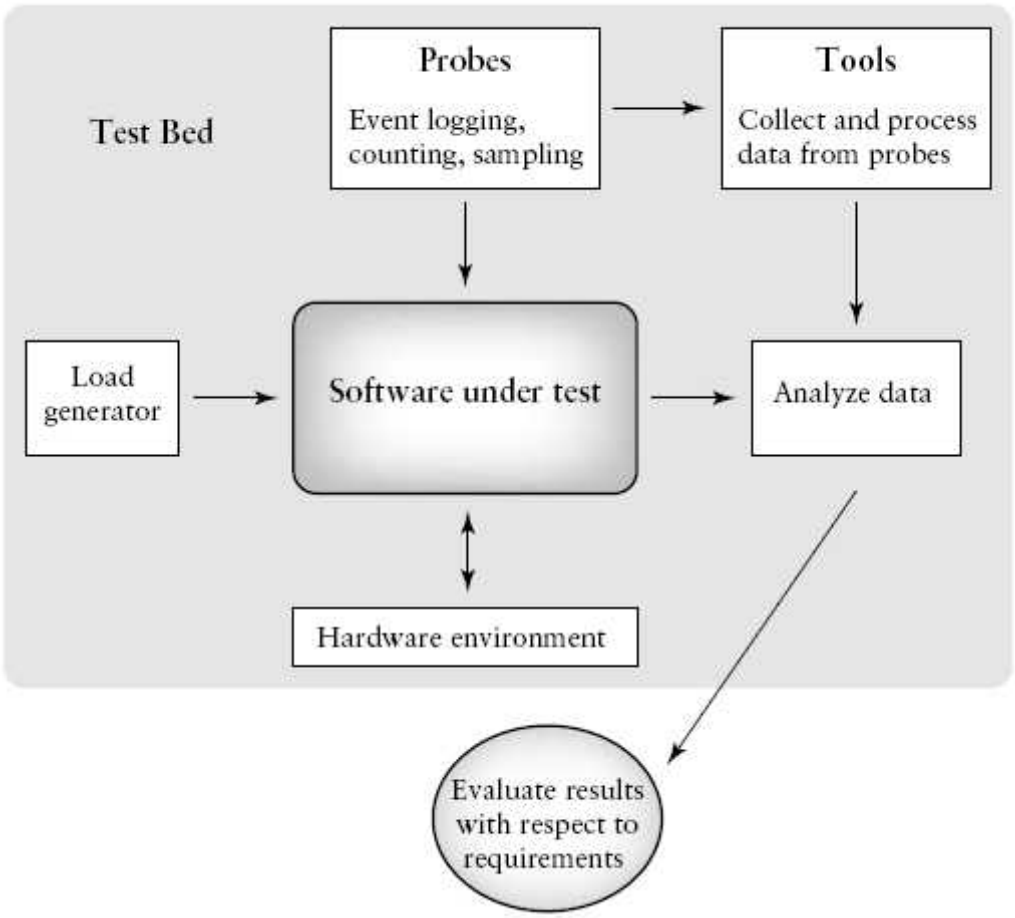


FIG. 3.9 EXAMPLES OF SPECIAL RESOURCES NEEDED FOR A PERFORMANCE TEST

- A source of transaction to drive the experiments. For example if you were performance testing an operating system you need a stream of data that represent typical user interactions.
- Typically the source of transaction for many systems is load generator.
- An experimental test bed that includes hardware and software the system under test interacts with. The test bed requirement sometimes includes special laboratory equipment and space that must be reserved for the tests.

- Instruments or probes that help to collect the performance data, probes may be hardware or software in nature.
- Some probe tasks are event counting and event duration measurement.
- Eg:- if you are investigating memory requirements for your software you could use a hardware probe that collected information on memory usage as the system executes
- The tester must keep in mind that the probes themselves may have an impact on system performance
- A set of tools to collect, store, process and interpret the data.

Very often, large volume of data are collected, and without tools the testers may have difficulty in processing and analyzing the data in order to evaluate true performance levels.

STRESS TESTING

- **The goal of stress test is to try to break the system; find the circumstance which it will crash, this is sometimes called "*breaking the system*".**
- **Stress testing often uncovers race conditions, deadlocks**
- **The goal of stress test is to try to break the system; find the circumstance which it will crash, this is sometimes called "*breaking the system*".**
- **Stress testing is important because it can reveal defects in real time** and other types of systems, as well as weak areas where poor design could cause unavailability of services.
- **Stress testing often uncovers race conditions, deadlocks, depletion of resource in unusual or un planned patterns, and upset in normal operation of the software system.**
- System limits and threshold values are exercised

- Stress testing is important from the user/client point of view. When system operate correctly under conditions of stress then client have confidence that the software can perform as required.

CONFIGURATION TESTING

- **Allows testers to evaluate system performance and availability when hardware exchanges and reconfigurations occur.**
- To test the software using configuration testing, many resources such as multiple hardware devices are needed
- Software Systems interact with hardware devices such as disc drivers, tape drivers and printers. Many Software system also interact with multiple CPU some of which are redundant
- Eg:- a printer of type X should be substitutable for a printer of type Y, CPU A should be removable from a system composed of several other CPUs
- Sensor A should be replaced with Sensor B
- **Configuration testing has the following objectives**
    - **i. Show that all configuration changing commands and menus work properly**
    - **ii. Show that all interchangeable and that they each enter the proper states for the specified conditions**
    - **iii. Shows that the system performance level is maintained when devices are interchanged, or when they fail**
- Several types of operations should be performed during configuration test, some sample operations for tester are:-
    a. Rotate and Permutate the position of devices to ensure physiological/logical device permutations work for each device

b. Induce malfunctions in each devices , to see if the system properly handles the malfunction

c. Induce multiple device malfunctions to see how the system reacts

- These operation will help to reveal problems (defects) relating to hardware and software when hardware exchange, and the reconfiguration occur.

SECURITY TESTING

- **Security testing is a process used to reveal defects in the security mechanisms of a software system**
- Computer Software and data can be compromised by

i. **Criminals, intent on doing damages, stealing data and information, causing denial of service , invading privacy**

ii. **Errors on the part of honest developers/ maintainers who modify, destroy or compromise data because of misinformation , misunderstanding , and/or lack of knowledge**

- Attacks can be random or systematic.
- Damage can be done through various means such as:-

a. Viruses

b. Trojan Horses

c. Trap Doors

d. Illicit channels

- The effect of security breaches could be extensive and can cause
  - Loss of information
  - Corruption of information
  - Privacy violations
  - Denial of service

Key areas of security testing

1. Password Checking:-

Test the password checker to insure that users will select a password that meets the condition descried in the password checker specification. Equivalence class partitioning and boundary value analysis based on the rules and conditions that specify a valid password can be used to design the tests.

2. **Legal and Illegal Entry with password**:-

Test for legal and illegal system/data access via legal and illegal passwords.

3. Password Expiration:-

If it is decided that password will expire after certain time period, tests should be designed to insure the expiration period is properly supported and that users can enter a new and appropriate password.

4. Encryption:-

Design test cases to evaluate the correctness of both encryption and decryption algorithms for systems where data/message are encoded

5. Browsing:-

Evaluate browsing privileges to insure that unauthorized browsing doesn't occur. Tester should attempt to browse illegally and observe system responses. They should determine what types of private information can be inferred by both legal and illegal browsing

6. Trap Doors:-

Identify any unprotected entries into the system that may allow access through unexpected channel (trap doors) .Design test cases that attempt to gain illegal entry and observe results. tester will need to support of designer and developers for this task

7. Viruses:-

Design test to insure that system virus checkers prevent or curtail entry of viruses into the system. Tester may attempt to infect the system with various viruses and observer the system response.

RECOVERY TESTING

- **Recovery testing is a type of nonfunctional testing technique performed in order to determine how quickly the system can recover after it has gone through system crash or hardware failure**
- **Recovery testing is especially important for transaction system**

  Eg:- on line banking software
- Beizer advises that tester focus on the following areas during recovery testing
  - ☞ ***Restart:-*** The current system state and transaction state are discarded The most recent checkpoint record retrieved an d the system initialized to the state in the checkpoint record. Tester must insure that all transaction have been reconstructed correctly and that all devices are in proper state. The system should then be able to begin to process new transaction
  - ☞ ***Switchover:-*** The ability of the system to switch to a anew processor must be tested .Switch over is the result of a command or detection of faulty processor by a monitor
- All transaction and processes must be carefully examined to detect:-
  - Loss of transaction
  - Merging of transaction
  - Incorrect Transactions
  - An unnecessary duplication of transaction

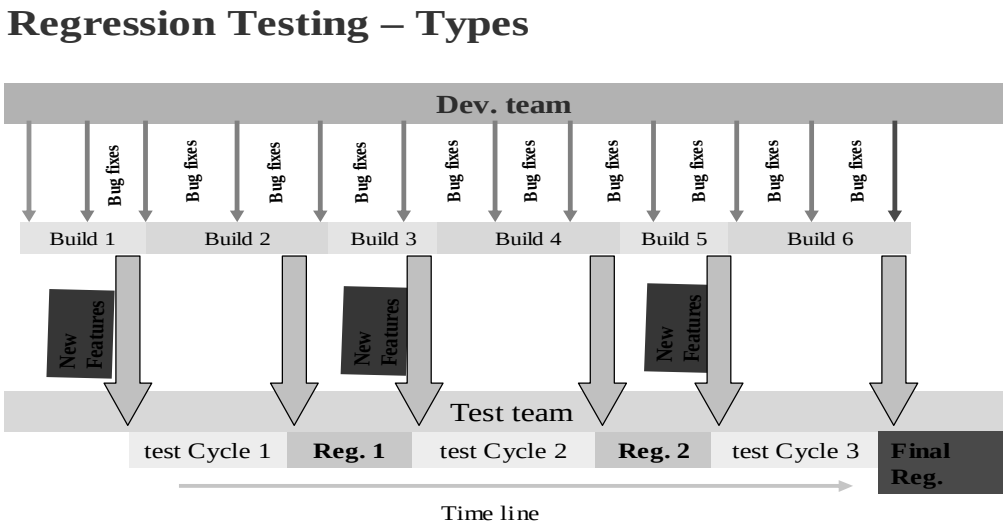
REGRESSION TESTING

- **Regression testing is retesting of software that occur when changes are made to ensure that new version of the software has retained the capability of the old version and no new defects has been introduced due to the changes.**

- Regression Testing can occur at any level of test.
- Ex:- When unit tests are run the unit may pass a number of these tests until one of the test does reveal a defect. The unit is repaired and then retested with all the old test cases to ensure that the changes have not affected its functionality

Doctor: Congratulations! The stomach ulcer that was bothering you and preventing digestion is now completely cured!	*Patient:* That is fine doctor, but I have got such a bad mouth ulcer that I can't eat anything and hence there is nothing to digest!

- Regression testing is **selective re-testing** of the system with an objective to ensure that the bug fixes work and those bug fixes have not caused any un-intended effects in the system
- This testing is done to ensure that:
  - The bug-fixes work
  - The bug-fixes do not create any side-effects



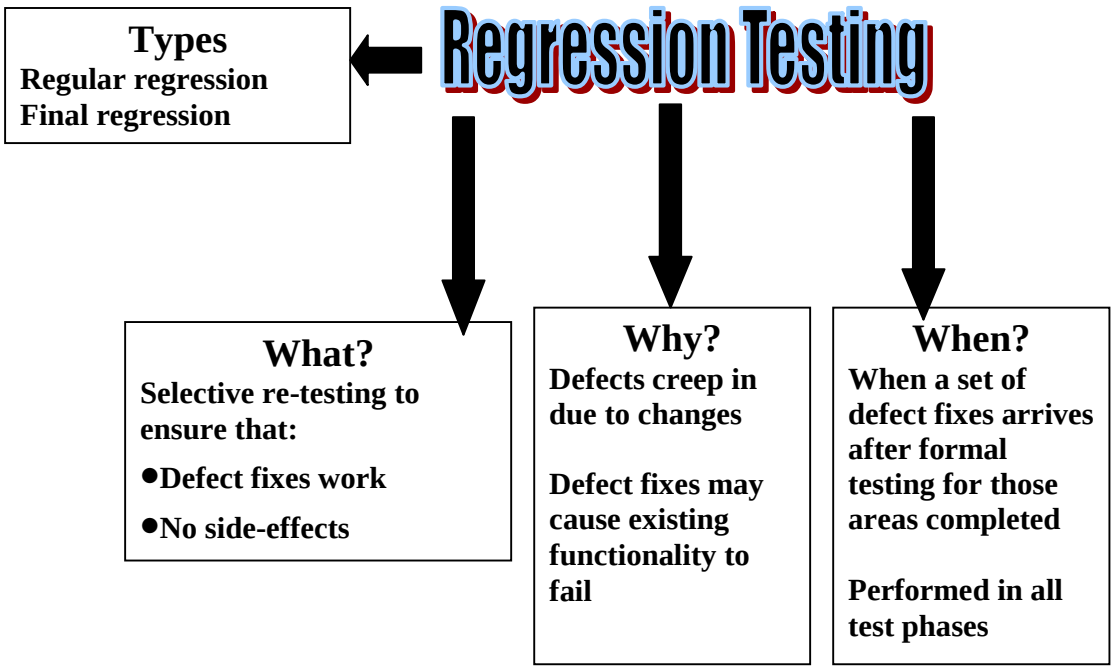
Regression Testing – Types

I. Final regression testing

- To ensure that "**the same build of the product that was tested reaches the customer**"
- Used to get a comfort feeling on the product prior to release

II. Regression testing

- To validate the product builds between test cycles
- Unchanged build is recommended but not mandatory
- Used to get a comfort feeling on the bug fixes, and to carry on with next cycle of testing



ACCEPTANCE TESTING

- After the software has passed all the system tests and defect repairs have been made, the users take a more active role in the testing process.
- Developers/testers must keep in mind that the software is being developed to satisfy the users requirements, and no matter how elegant its design it will not be accepted by the users unless it helps them to achieve their goals as specified in the requirements.
- Alpha, beta, and acceptance tests allow users to evaluate the software in terms of their expectations and goals.
- **When software is being developed for a specific client, acceptance tests are carried out after system testing.**
- The acceptance tests must be planned carefully with input from the client/users. Acceptance test cases are based on requirements.
- The user manual is an additional source for test cases. System test cases may be reused. The software must run under real-world conditions on operational hardware and software. The software-under-test should be stressed.

- Acceptance tests are a very important milestone for the developers. At this time the clients will determine if the software meets their requirements. Contractual obligations can be satisfied if the client is satisfied with the software. Development organizations will often receive their final payment when acceptance tests have been passed.
- Acceptance tests must be prepared by the developers/testers.
- Clients should be provided with documents and other material to help them participate in the acceptance testing process, and to evaluate the results.
- **After acceptance testing the client will point out to the developers which requirement have/have not been satisfied.**
- **If the client is satisfied that the software is usable and reliable, and they give their approval, then the next step is to install the system at the client's site.**
- **If the client's site conditions are different from that of the developers, the developers must set up the system so that it can interface with client software and hardware. Retesting may have to be done to insure that the software works as required in the client's environment. This is called installation test.**

Acceptance tests are run to verify both functional and non functional requirements.

Acceptance Criteria:

i. **Acceptance Criteria – Product Acceptance**: During the requirement phase, each requirement is associated with acceptance criteria. It is possible that one or more requirements may be mapped to form acceptance criteria. Whenever there are changes to requirements, the acceptance criteria are accordingly modified and maintained.

ii. **Acceptance Criteria – procedure acceptance:** Acceptance criteria can be defined based on the procedures followed for delivery. Some examples of acceptance criteria are as follows:

a. User, administration and troubleshooting documentation should be part of the release.

b. A minimum of 20 employees are trained on the product usage prior to deployment.

iii. **Acceptance Criteria – Service level agreements:** Service level agreements can become part of acceptance criteria. Service level agreements are generally part of a contract signed by the customer and product organization.

Selecting Test Cases for Acceptance Testing:

The test cases for acceptance testing are selected from the existing set of test cases from different phases of testing.

Some guidelines on what test cases can be included for acceptance testing are given below:

1. **End-to-end functionality verification**: Test cases that include the end-to-end functionality of the product are taken up for acceptance testing. This ensures that all the business transactions are tested as a whole.
2. **Domain Tests**: Since acceptance tests focus on business scenarios the product domain tests are included. Test cases that reflect business domain knowledge are included.
3. **User Scenario Tests**: Acceptance tests reflect the real-life user scenario verification.
4. **Basic Sanity tests**: Tests that verify the basic existing behavior of the product are included. These tests ensure that the system performs the basic operations that it was intended to do.
5. **New Functionality**: When the product undergoes modifications or changes, the acceptance test cases focus on verifying the new features.

Executing acceptance tests:

- ✓ An acceptance test team comprises members who are involved in the day-to-day activities of the product usage or are familiar with such scenarios.
- ✓ The product management, support, and consulting team, who have good knowledge of the customers contribute to the acceptance testing.
- ✓ Test team members help the acceptance members to get the required test data, select and identify test cases, and analyze the acceptance test results.
- ✓ During test execution, the acceptance test team reports its progress regularly.
- ✓ The defect reports are generated on a periodic basis.
- ✓ Defect reported during acceptance tests could be of different priorities. Test teams help acceptance test team report defects.
- ✓ When the defect fixes point to scope or requirement changes, then it may either result in the extension of the release due to include the feature in the current release or get postponed to subsequent releases.

INTERNATIONALIZATION TESTING

Internationalization ($I_{18}n$) is used as an umbrella term to mean all activities that are required to make the software available for international market.

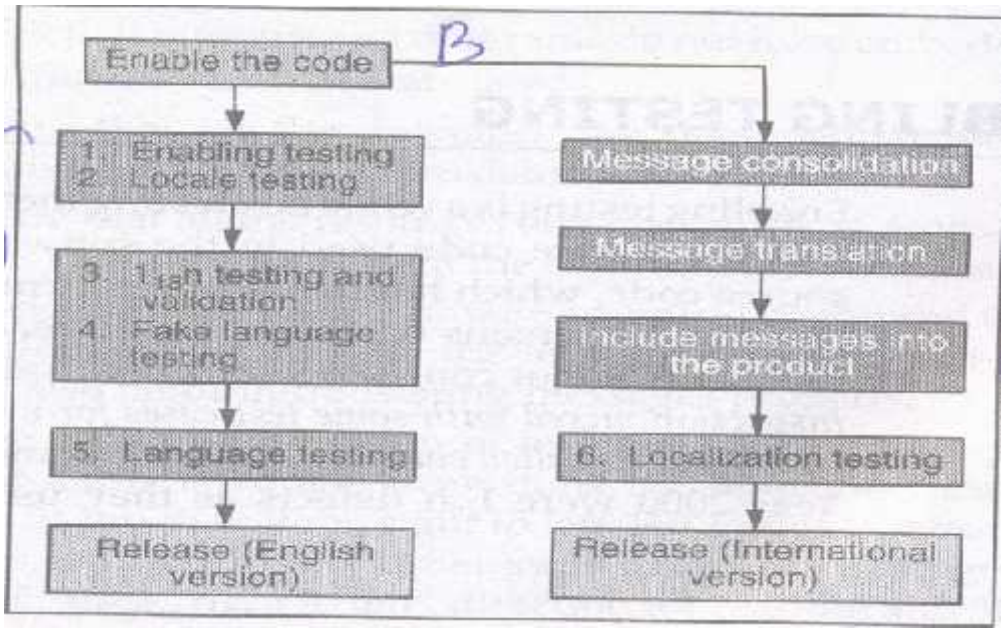
(18 is used to mean that there are 18 characters between “I” and the last “n” in the word Internationalization.

The testing that is done in various phases to ensure that all those activities are done right is called internationalization testing or $I_{18}n$ testing.

Test Phases for Internationalization Testing:

Testing for internationalization requires a clear understanding of all activities involved and their sequence.

The figure below depicts the various major activities involved in internationalization testing.



The testing for internationalization is done in multiple phases in the project life cycle.

Some important aspects of internationalization testing are:

a. Testing the code for how it handles input, strings, and sorting items
b. Display of messages for various languages and
c. Processing of messages for various languages and conventions

1. Enabling Testing:

Enabling testing is a white box testing methodology, which is done to ensure that the source code used in the software allows internationalization.

An activity of code review or code inspection mixed with some test cases for unit testing, with an objective to catch $I_{18}n$ defects is called enabling testing.

Enabling testing uses a checklist. Some items to be kept in the review checklist for enabling testing are as follows:

- Check the code for APIs/function calls that are not part of the $I_{18}n$.

- Check the code for hard-coded date, currency formats, ASCII code, or character constants.
- Ensure that adequate size is provided for buffers and variables to contain translated messages.

2. **Locale Testing**:

Once the code is verified for I18n and the enabling test is completed, the next step is to validate the effects of locale change in the product.

A local change affects date, currency format, and the display of items on screen, in dialog boxes and text.

Changing the different locales using the system settings or environment variables, and testing the software functionality, number, date, time and currency format is called locale testing.

Some of the items to be checked in locale testing are as follows:

1. All features that are applicable to I_{18}n are tested with different locales of the software for which they are intended.
2. Function keys and help screens are tested with different applicable locales.
3. Date and time format are in line with the defined locale of the language.
4. Time zone information and daylight saving time calculations are consistent and correct.

3. Internationalization Validation:

I_{18}n validation is different from I_{18}n testing. I_{18}n testing is the superset of all types of testing.

I_{18}n validation is performed with the following objectives.

a. The software is tested for functionality with ASCII and European Characters.
b. The software handles string operations, sorting, sequence operations as per the language and characters selected.
c. The software display is consistent with characters which are non-ASCII in GUI and menus.

d. The software messages are handled properly.

A Checklist for the I18n validation includes the following:

- The functionality in all languages and locales are the same.
- Sorting and sequencing the items to be as per the conventions of language and locale.
- The software functions correctly with different languages.
- The documentation contains consistent documentation style, punctuations, and all language/locale conventions are followed for every target audience.

4. Fake Language Testing:

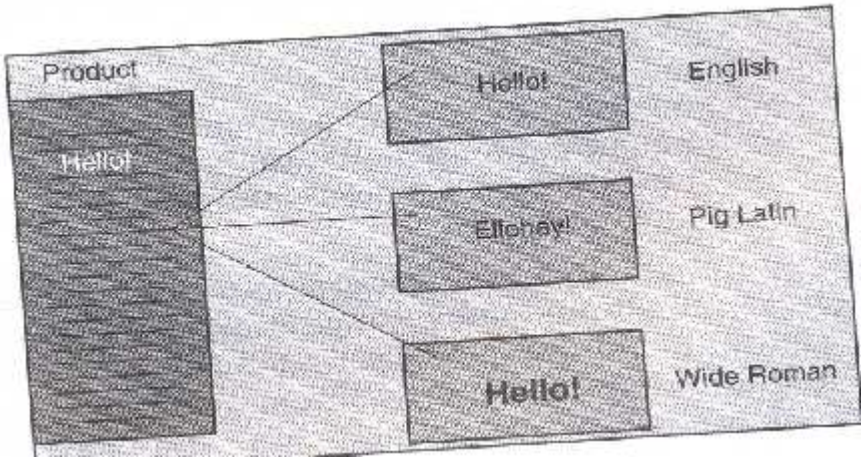
Fake language testing uses software translators to catch the translation and localization issues early.

This also ensures that switching between languages work properly and correct messages are picked from proper directories that have the translated messages.

Fake language testing helps in identifying the issues proactively before the product is localized.

The fake language translators use English-like target language, which are easy to understand and test.

This type of testing helps English testers to find the defects that may otherwise be found only by language experts during localization testing.



The following items in the checklist can be used for fake language testing.

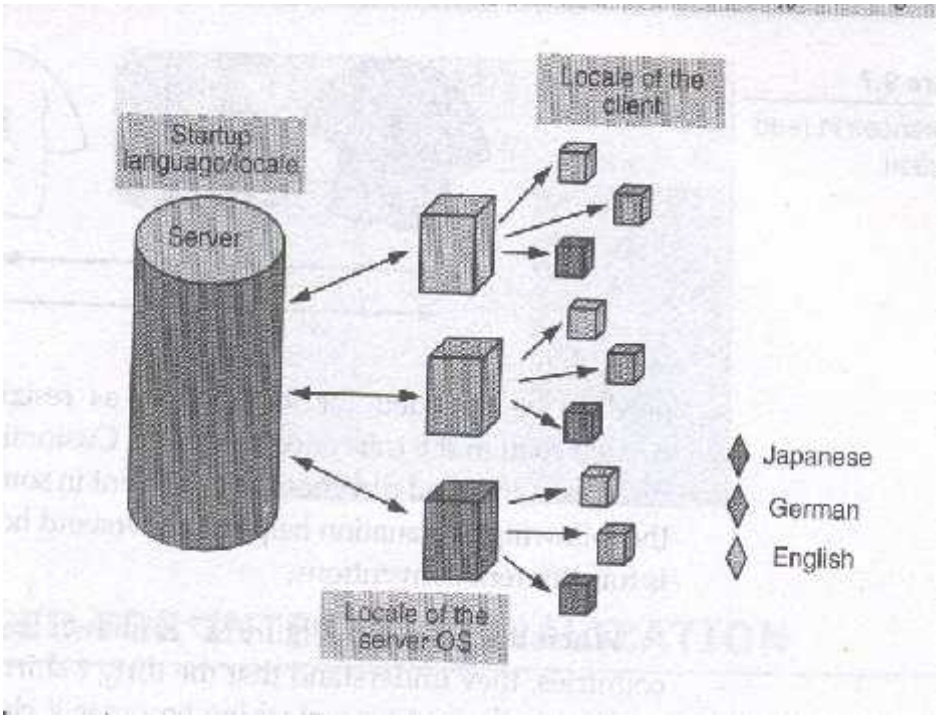
a. Ensure software functionality is tested for atleast one of the European single-byte fake language.(Latin)
b. Ensure software functionality is tested for atleast one double-byte language (Roman)

5. Language Testing:

Language testing focuses on testing the English product with a global environment of products and services functioning in non English.

It is the short form of “language compatibility testing”.

This ensure that software created in English can work with platforms and environments that are English and Non-English.



The figure above illustrates the language testing and various combinations of locales that have to be tested in client-server architecture.

While testing, it is important to look for locale-related issues, as some of the defects that escaped from locale testing may show up in this testing.

Language testing should have the following checklists:

a. Check the functionality on English, one non-English and one double-byte language platform combination.
b. Check the performance of key functionality on different language platforms and across different machines connected in the network.

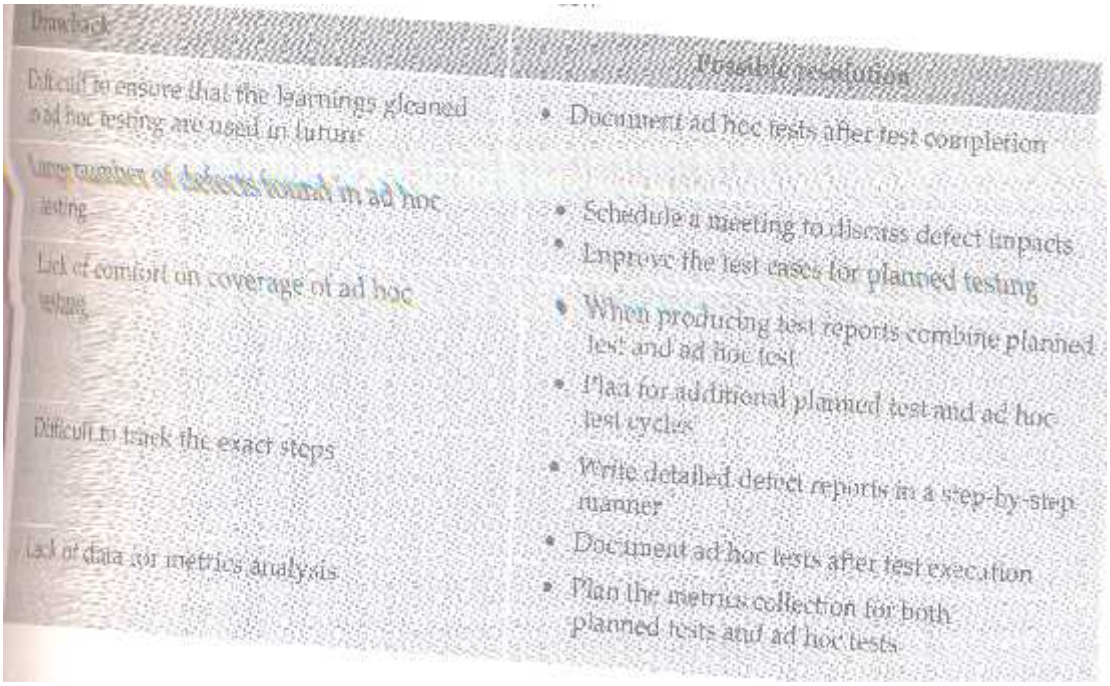
6. <u>Localization Testing:</u>

- ✓ When the software is approaching the release date, messages are consolidated into a separate file and sent to multilingual experts for translation.
- ✓ A set of build tools consolidates all the messages and other resources automatically and puts them in separate files.
- ✓ The multilingual experts may not be aware of the software or their target customers.
- ✓ Localization is a very expensive and labour-intensive process.
- ✓ While translating, not only the messages but also resources such as GUI screens, dialog boxes, icons and bitmaps need to be included for localization.
- ✓ Customization is important as scroll directions and read directions are different in some languages.
- ✓ The following checklist may help in doing localization testing.
    - All the messages, documents, pictures, screens are localized to reflect the native users and the conventions of the country, locale and language.
    - Sorting and case conversions are right as per language convention.
    - Font sizes and hot keys are working correctly in the translated messages, documents and screens.
    - Filtering and searching capabilities of the software work as per the language and locale conventions.

ADHOC TESTING

Testing done without using any formal testing technique is called adhoc testing.

- Adhoc testing is done to explore the undiscovered areas in the product by using intuition, previous experience in working with the product, expert knowledge of the platform or technology and experience of testing a similar product.
- Adhoc testing does not make use of any of the test case design techniques.



There are different types of adhoc testing. They are:

1. **Buddy Testing:**

A developer and tester working as buddies to help each other on testing and in understanding the specialization is called buddy testing.

- Two team members are identified as buddies.
- The buddies mutually help each other, with a common goal of identifying defects early and correcting them.
- A developer and tester become buddies.
- Buddies should not feel mutually threatened or get a feeling of insecurity during buddy testing.
- They stay close together to be able to follow the agreed plan.
- Buddy testing is normally done at the unit test phase, where there are both coding and testing activities.

2. Pair Testing:

Pair Testing is testing done by two testers working simultaneously on the same machine to find defects in the product.

- Two testers pair up to test a product's feature on the same machine.
- The objective of this testing is to maximize the exchange of ideas between the two testers.
- When one person is executing the tests, the other person takes notes.
- The other person suggests an idea or helps in providing additional perspectives.
- Pair testing can be done during any phase of testing.
- It encourages idea generation right from the requirements analysis phase, taking it forward to the design, coding and testing phases.
- Testers can pair together during the coding phase to generate various ideas to test the code and various components.

3. Exploratory Testing:

- Exploratory testing is a technique to find defects by exploring the product, covering more depth and breadth.
- It can be done during any phase of testing.
- Exploratory testers may execute their tests based on their past experiences in testing a similar product.
- Since there is large creative element to exploratory testing, similar test cases may result in different kinds of defects when done by two different individuals.
- **Exploratory test techniques are:**
  a. Guesses
  b. Architecture diagram, use cases
  c. Past defects
  d. Error Handling
  e. Discussions
  f. Questions and checklists.

4. Iterative Testing:

- In an iterative model, the requirements keep coming and the product is developed iteratively for each requirement. The testing associated with this process is called iterative testing.
- Regression tests may be repeated at least every alternative iteration so that the current functionality is preserved.
- Since iterative testing involves repetitive test execution of tests that were run for the previous iterations, it becomes a tiresome exercise for the testers.

5. Agile and Extreme Testing:

- Agile and extreme models take the processes to the extreme to ensure that customer requirements are met in timely manner.

- In this model, customers partner with the project teams to go step by step in bringing the project to completion in a phased manner.
- The customer becomes part of the project team so as to clarify any doubts/questions.
- Agile and extreme methodology emphasizes the involvement of the entire team, and their interaction with each other, to produce a workable software that can satisfy a given set of features.

USABILITY AND ACCESSIBILITY TESTING

USABILITY TESTING:

The testing that validates the ease of use, speed, and aesthetics of the product from the users' point of view is called usability testing.

A right approach for usability is to test every artifact that impacts users – such as product binaries, documentation, messages, media-covering usage patterns through both graphical and command user interfaces as applicable.

Usability testing can be done is two phases:

1. Design validation Phase
2. Component and integration testing phase.

Usability design is verified through several means. Some of them are as follows:

a. Style sheets
b. Screen prototypes
c. Paper designs
d. Layout Design.

12.1 Development and testing of client applications and web application.

Client application	Web application
Step 1: Design for functionality	Step 1: Design for user interface
Step 2: Perform coding for functionality	Step 2: Perform coding for user interface (prototype)
Step 3: Design for user interface	Step 3: Test user interface (Phase 1)
Step 4: Perform coding for user interface	Step 4: Design for functionality
Step 5: Integrate user interface with functionality	Step 5: Perform coding for functionality
Step 6: Test the user interface along with functionality (Phase 1 and 2)	Step 6: Integrate user interface with functionality
	Step 7: Test user interface along with functionality (Phase 2)

ACCESSIBILITY TESTING:

Verifying the product usability for physically challenged users is called accessibility testing.

Accessibility to the product can be provided by two means:

a. Making use of accessibility features provided by the underlying infrastructure called basic accessibility and
b. Providing accessibility in the product through standards and guidelines, called product accessibility.

Basic Accessibility:

- It is provided by the hardware and operating system. All the input and output devices of the computer and their accessibility options are categorized under basic accessibility.
- Some of the basic accessibility options are:

i. Keyboard accessibility

ii. Screen accessibility

Product Accessibility:

- A good understanding of the basic accessibility features is needed while providing accessibility to the product.
- A product should do everything possible to ensure that the basic accessibility features are utilized by it.
- **Sample Requirements are:**

  i. Providing detailed text equivalent for multimedia files ensures the captions feature is utilized by the product.

  ii. Documents and fields should be organized so that they can be read without requiring a particular resolution of the screen, and templates.

  iii. User interfaces should be designed so that all information conveyed with color is also available without color.

  iv. Reduce flicker rate, speed of moving text; avoid flashes and blinking text.

  v. Reduce physical movement requirements for the users when designing the interface and allow adequate time for user responses.

  The sample requirements given above are some examples to improve accessibility.

ALPHA AND BETA TESTING

ALPHA TESTING

If the software has been developed for the mass market, then testing it for individual clients/users is not practical in most cases. Very often this type of software undergoes two stages of acceptance test.

The first is called alpha test. This test takes place at the developer's site. A cross-section of potential users and members of the developer's organization are invited to use the software. Developers observe the users and note problems.

- Alpha testing is testing of an application when development is about to complete. Minor design changes can still be made as a result of alpha testing.
- Alpha testing is typically performed by a group that is independent of the design team, but still within the company, e.g. in-house software test engineers, or software QA engineers.
- Alpha testing is final testing before the software is released to the general public. It has two phases:
- In the **first phase** of alpha testing, the software is tested by in-house developers. They use either debugger software, or hardware-assisted debuggers. The goal is to catch bugs quickly.
- In the **second phase** of alpha testing, the software is handed over to the software QA staff, for additional testing in an environment that is similar to the intended use.

BETA TESTING

Beta test sends the software to a cross-section of users who install it and use it under real world working conditions. The users send records of problems with the software to the development organization where the defects are repaired sometimes in time for the current release.

The goal of beta testing is to place your application in the hands of real users outside of your own engineering team to discover any flaws or issues from the user's perspective that you would not want to have in your final, released version of the application.

Advantages of beta testing

- You have the opportunity to get your application into the hands of users prior to releasing it to the general public.
- Users can install, test your application, and send feedback to you during this beta testing period.
- Your beta testers can discover issues with your application that you may have not noticed, such as confusing application flow, and even crashes.
- Using the feedback you get from these users, you can fix problems before it is released to the general public.
- The more issues you fix that solve real user problems, the higher the quality of your application when you release it to the general public.
- Having a higher-quality application when you release to the general public will increase customer satisfaction.
- These users, who are early adopters of your application, will generate excitement about your application.

WEBSITE TESTING

Website Testing is checking your web application for potential bugs before its made live or before code is moved into the production environment.

Some or all of the following testing types may be performed depending on your web testing requirements.

1.Functionality Testing:

- This is used to check if your product is as per the specifications you intended for it as well as the functional requirements you charted out for it in your developmental documentation.
- Testing Activities Included:

Test all **links** in your webpages are working correctly and make sure there are no broken links.

It can be **carried out by testers** like you **or a small focus group** similar to the target audience of the web application.

Test the site **Navigation**:

Menus, buttons or Links to different pages on your site should be easily visible and consistent on all webpages

Test the **Content**:

Content should be legible with no spelling or grammatical errors.

Images if present should contain an "alt" text

2.Interface Testing:

Three areas to be tested here are - Application, Web and Database Server

Application: Test requests are sent correctly to the Database and output at the client side is displayed correctly. Errors if any must be caught by the application and must be only shown to the administrator and not the end user.

Web Server: Test Web server is handling all application requests without any service denial.

Database Server: Make sure queries sent to the database give expected results.

3.Database Testing:

Database is one critical component of your web application and stress must be laid to test it thoroughly. Testing activities will include-

- Test if any errors are shown while executing queries
- Data Integrity is maintained while creating, updating or deleting data in database.
- Check response time of queries and fine tune them if necessary.
- Test data retrieved from your database is shown accurately in your web application

4.Compatibility testing.

Compatibility tests ensures that your web application displays correctly across different devices. This would include-

Browser Compatibility Test: Same website in different browsers will display differently. You need to test if your web application is being displayed correctly across browsers, JavaScript, AJAX and authentication is working fine. You may also check for Mobile Browser Compatibility.

5.Performance Testing:

This will ensure your site works under all loads.

6. Security testing:

Security Testing is vital for e-commerce website that store sensitive customer information like credit cards. Testing Activities will include-

- Test unauthorized access to secure pages should not be permitted
- Restricted files should not be downloadable without appropriate access

TESTING OO SYSTEMS

Testing is a continuous activity during software development. In object-oriented systems, testing encompasses three levels, namely, unit testing, subsystem testing, and system testing.

Unit Testing

In unit testing, the individual classes are tested.

Smallest testable unit is the encapsulated class

It is seen whether the class attributes are implemented as per design and whether the methods and the interfaces are error-free. Unit testing is the responsibility of the application engineer who implements the structure.

Subsystem Testing

This involves testing a particular module or a subsystem and is the responsibility of the subsystem lead. It involves testing the associations within

the subsystem as well as the interaction of the subsystem with the outside. Subsystem tests can be used as regression tests for each newly released version of the subsystem.

System Testing

System testing involves testing the system as a whole and is the responsibility of the quality-assurance team. The team often uses system tests as regression tests when assembling new releases.

Object-Oriented Testing Techniques

Grey Box Testing

The different types of test cases that can be designed for testing object-oriented programs are called grey box test cases. Some of the important types of grey box testing are:

State model based testing : This encompasses state coverage, state transition coverage, and state transition path coverage.

Use case based testing : Each scenario in each use case is tested.

Class diagram based testing : Each class, derived class, associations, and aggregations are tested.

Sequence diagram based testing : The methods in the messages in the sequence diagrams are tested.

Techniques for Subsystem Testing

The two main approaches of subsystem testing are:

Thread based testing : All classes that are needed to realize a single use case in a subsystem are integrated and tested.

Use based testing : The interfaces and services of the modules at each level of hierarchy are tested. Testing starts from the individual classes to the small modules comprising of classes, gradually to larger modules, and finally all the major subsystems.

Categories of System Testing

Alpha testing : This is carried out by the testing team within the organization that develops software.

Beta testing : This is carried out by select group of co-operating customers.

Acceptance testing : This is carried out by the customer before accepting the deliverables.

TESTING THE DOCUMENTATION

Documentation testing is a non-functional type of **software testing**.

- It is a type of **non-functional testing**.
- Any written or pictorial information describing, defining, specifying, reporting, or certifying activities, requirements, procedures, or results'. Documentation is as important to a product's success as the product itself. If the documentation is poor, non-existent, or wrong, it reflects on the quality of the product and the vendor.
- As per the IEEE Documentation describing plans for, or results of, the testing of a system or component, Types include test case specification, test incident report, test log, test plan, test procedure, test report. Hence the testing of all the above mentioned documents is known as documentation testing.
- This is one of the most cost effective approaches to testing. If the documentation is not right: there will be major and costly problems. The documentation can be tested in a number of different ways to many different degrees of complexity. These range from running the documents through a spelling and grammar checking device, to manually reviewing the documentation to remove any ambiguity or inconsistency.
- Documentation testing can start at the very beginning of the software process and hence save large amounts of money, since the earlier a **defect** is found the less it will cost to be fixed.

JEPPIAAR INSTITUTE OF TECHNOLOGY

"Self-Belief | Self Discipline | Self Respect"

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

IT8076 – SOFTWARE TESTING

(Regulation 2017)

Year/Semester: III/VI CSE

2020 – 2021

Prepared by

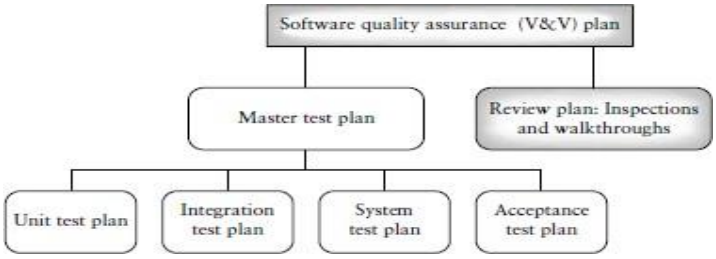
Ms. R. Revathi

Assistant Professor/CSE

Unit -IV

Test Management

- A plan is a document that provides a framework or approach for achieving a set of goals.
- Milestones are tangible events that are expected to occur at a certain time in the project's lifetime. Managers use them to determine project status
- Test plans for software projects are very complex and detailed documents. The planner usually includes the following essential high-level items.
- Overall test objectives
    - What to test (scope of the tests).
    - Who will test.
    - How to test.
    - When to test.
    - When to stop testing



Test Plan Components

Test Plan Components
1. Test plan identifier
2. Introduction
3. Items to be tested
4. Features to be tested
5. Approach
6. Pass/fail criteria
7. Suspension and resumption criteria
8. Test deliverables
9. Testing Tasks
10. Test environment
11. Responsibilities
12. Staffing and training needs
13. Scheduling
14. Risks and contingencies
15. Testing costs
16. Approvals

Components of a test plan

4.0 Test Planning	
4.1	Meet with project manager. Discuss test requirements.
4.2	Meet with SQA group, client group. Discuss quality goals and plans.
4.3	Identify constraints and risks of testing.
4.4	Develop goals and objectives for testing. Define scope.
4.5	Select test team.
4.6	Decide on training required.
4.7	Meet with test team to discuss test strategies, test approach, test monitoring, and controlling mechanisms.
4.8	Develop the test plan document.
4.9	Develop test plan attachments (test cases, test procedures, test scripts).
4.10	Assign roles and responsibilities.
4.11	Meet with SQA, project manager, test team, and clients to review test plan.

A breakdown of testing planning element

Test Plan Attachments

Requirement identifier	Requirement description	Priority (scale 1–10)	Review status	Test ID
SR-25-13.5	Displays opening screens	8	Yes	TC-25-2 TC-25-5
SR-25-52.2	Checks the validity of user password	9	Yes	TC-25-18 TC-25-23

Example of entries in a requirements traceability matrix

Test Design Specification

- Test Design Specification Identifier
- Features to Be Tested
- Approach Refinements
- Test Case Identification
- Pass/Fail Criteria

Test Case specification

- Test Case Specification Identifier
- Test Items
- Input Specifications
- Output Specifications
- Special Environmental Needs
- Special Procedural Requirements
- Intercase Dependencies

Test procedure specification

- A procedure in general is a sequence of steps required to carry out a specific task.
- Test Procedure Specification Identifier
- Purpose
- Specific Requirements
- Procedure Steps

i) setup: to prepare for execution of the procedure;

(ii) start: to begin execution of the procedure;

proceed: to continue the execution of the procedure;

(iv) measure: to describe how test measurements related to outputs will

be made;

(v) shut down: to describe actions needed to suspend the test when unexpected

events occur;

(vi) restart: to describe restart points and actions needed to restart the

procedure from these points;

(vii) stop: to describe actions needed to bring the procedure to an orderly

halt;

(viii) wrap up: to describe actions necessary to restore the environment;

(ix) contingencies: plans for handling anomalous events if they occur

during execution of this procedure.

Locating Test Items: The Test Item Transmittal Report

(i) version/revision number of the item;

(ii) location of the item;

(iii) persons responsible for the item (e.g., the developer);

(iv) references to item documentation and the test plan it is related to;

(v) status of the item;

(vi) approvals—space for signatures of staff who approve the transmittal.

Reporting Test Results

- **Test Log**
  - Test Log Identifier
  - Description
  - Activity and Event Entries
    - Execution description
    - Procedure results
    - Environmental information
    - Anomalous events
    - Incident report identifiers
- **Test Incident Report**

1. Test Incident Report identifier: to uniquely identify this report.

2. Summary: to identify the test items involved, the test procedures, test cases, and test log associated with this report.

3. Incident description: this should describe time and date, testers, observers,

environment, inputs, expected outputs, actual outputs, anomalies, procedure step, environment, and attempts to repeat.

4. **Impact: what impact will this incident have on the testing effort,** the test plans, the test procedures, and the test cases

- **Test Summary Report**

1. Test Summary Report identifier: to uniquely identify this report.

2. Variances: these are descriptions of any variances of the test items from their original design.

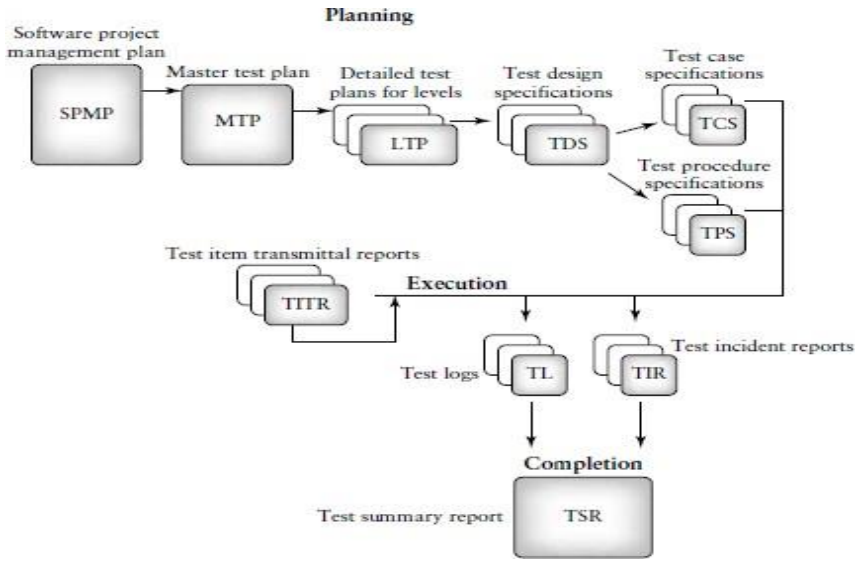
3. Comprehensiveness assessment: the document author discusses the comprehensiveness of the test effort as compared to test objectives

Summary of results: the document author summarizes the testing results.

5. Evaluation: in this section the author evaluates each test item based on test results.

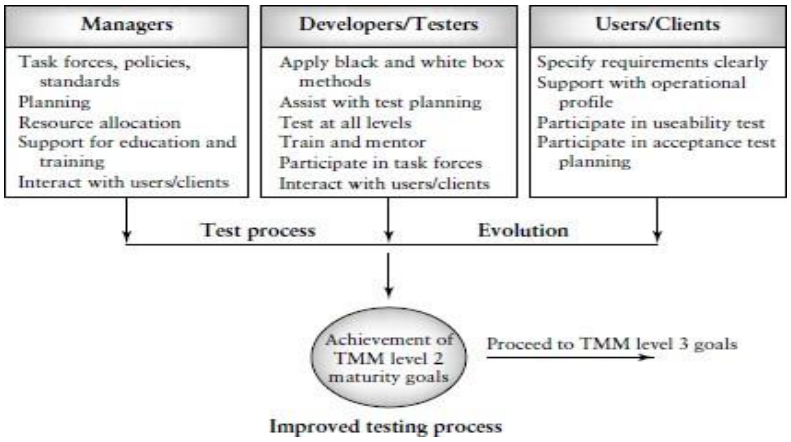
6. Summary of activities: all testing activities and events are summarized.

7. Approvals: the names of all persons who are needed to approve this document are listed with space for signatures and dates.



Test-related documents as recommended by IEEE[5]

The Role of the Three Critical Groups in Testing Planning and Test Policy Development



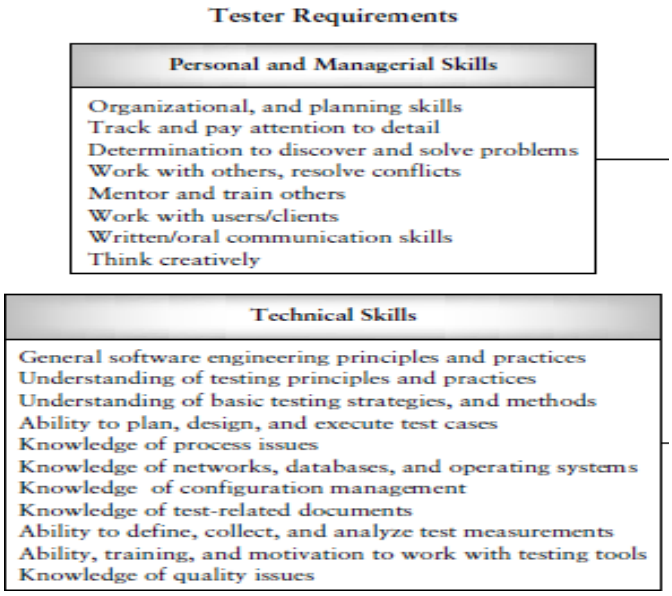
Reaching TMM level 2; summary of critical group roles

Introducing the Test Specialist

- maintenance and application of test policies;
- development and application of test-related standards;
- participating in requirements, design, and code reviews;
- test planning;
- test design;
- test execution;
- test measurement;
- test monitoring (tasks, schedules, and costs);
- defect tracking, and maintaining the defect repository;
- acquisition of test tools and equipment;
- identifying and applying new testing techniques, tools, and methodologies;
- mentoring and training of new test personnel;
- test reporting.

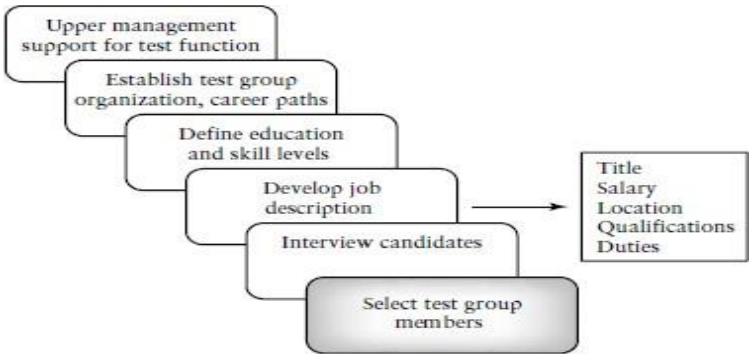
Skills Needed by a Test Specialist

- organizational, and planning skills;
- the ability to keep track of, and pay attention to, details;
- the determination to discover and solve problems;
- the ability to work with others and be able to resolve conflicts;
- the ability to mentor and train others;
- the ability to work with users and clients;
- strong written and oral communication skills;
- the ability to work in a variety of environments;
- the ability to think creatively



Test specialist skills

Building a Testing Group



Test specialist skills

The Structure of the Test Group

• maintain testing policy statements;

• plan the testing efforts;

• monitor and track testing efforts so that they are on time and within budget;

• measure process and product attributes;

• provide management with independent product and process quality information;

• design and execute tests with no duplication of effort;

• automate testing;

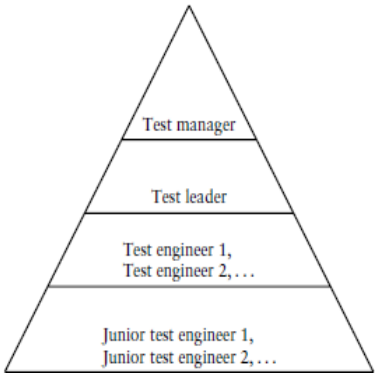
• participate in reviews to insure quality;

• work with analysts, designers, coders, and clients to ensure quality goals are meet;

• maintain a repository of test-related information;

• give greater visibility to quality issues organization wide;

• support process improvement efforts.



The test team hierarchy

The Technical Training Program

• quality issues;

• measurement identification, collection, and analysis;

• testing techniques and methodologies;

• design techniques;

• tool usage (for all life cycle phases);

• configuration management;

• planning;

• process evaluation and improvement;

• policy development;

• technical review skills;

• software acquisition;

• project management skills;

• business skills

• communication skills.

JEPPIAAR INSTITUTE OF TECHNOLOGY

"Self-Belief | Self Discipline | Self Respect"

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

IT8076 – SOFTWARE TESTING

(Regulation 2017)

Year/Semester: III/VI CSE

2020 – 2021

Prepared by

Ms. R. Revathi

Assistant Professor/CSE

UNIT V TEST AUTOMATION

Software test automation – skills needed for automation – scope of automation – design and architecture for automation – requirements for a test tool – challenges in automation – Test metrics and measurements – project, progress and productivity metrics.

Software Test Automation

WHAT IS TEST AUTOMATION?

Developing software to test the software is called test automation.

Test automation can help address several problems.

- **Automation save time as software can execute test cases faster than human do** . The time thus saved can be used effectively for test engineers to

1. develop additional test cases to achieve better coverage;
2. perform some esoteric or specialized tests like ad hoc testing; or
3. Perform some extra manual testing.

The time saved in automation can also be utilized to develop additional test cases, thereby improving the coverage of testing.

- **Test automation can free the test engineers from mundane tasks and make them focus on more creative tasks. -**E.g- Ad hoc testing requires intuition and creativity to test the product for those perspectives that may have been missed out by planned test cases. If there are too many planned test cases that need to be run manually and adequate automation does not exist, then the test team may spend most of its time in test execution.
  Automating the more mundane tasks gives some time to the test engineers for creativity and challenging tasks.
- **Automated tests can be more reliable** -when an engineer executes a particular test case many times manually, there is a chance for human error. As with all machine-oriented activities, automation can be expected to produce more reliable results every time, and eliminates the factors of boredom and fatigue.
- **Automation helps in immediate testing** -automation reduces the time gap between development and testing as scripts can be executed as soon as the product build is ready. Automated testing need not wait for the availability of test engineers.
- **Automation can protect an organization against attrition of test engineers** Automation can also be used as a knowledge transfer tool to train test engineers on the product as it has a repository of different tests for the product.
- **Test automation opens up opportunities for better utilization of global resources** Manual testing requires the presence of test engineers, but automated tests can be run round the clock, twenty- four hours a day and seven days a week.

This will also enable teams in different parts of the words, in different time zones, to monitor and control the tests, thus providing round the- clock coverage.

- **Certain types of testing cannot be executed without automation-**For example, if we want to study the behavior of a system with thousands of users logged in, there is now way one can perform these tests without using automated tolls.
- **Automation means end-to-end, not test execution alone -**Automation should consider all activities such as picking up the right product build, choosing the right configuration, performing installation, running the tests, generating the right test data, analyzing the results, and filling the defects in the defect repository. When talking about automation, this large picture should always be kept in mind.

TERMS USED IN AUTOMATION

A test case is a set of sequential steps to execute a test operating on a set of predefined inputs to produce certain expected outputs. There are two types of test cases – automated and manual. **A manual test case** is executed manually while an **automated test** case is executed using automation.

As we have seen earlier , testing involves several phases and several types of testing. Some test cases are repeated several times during a product release, because the product is built several times. Table describes some test cases for the log in example, on how the login can be tested for different types of testing.

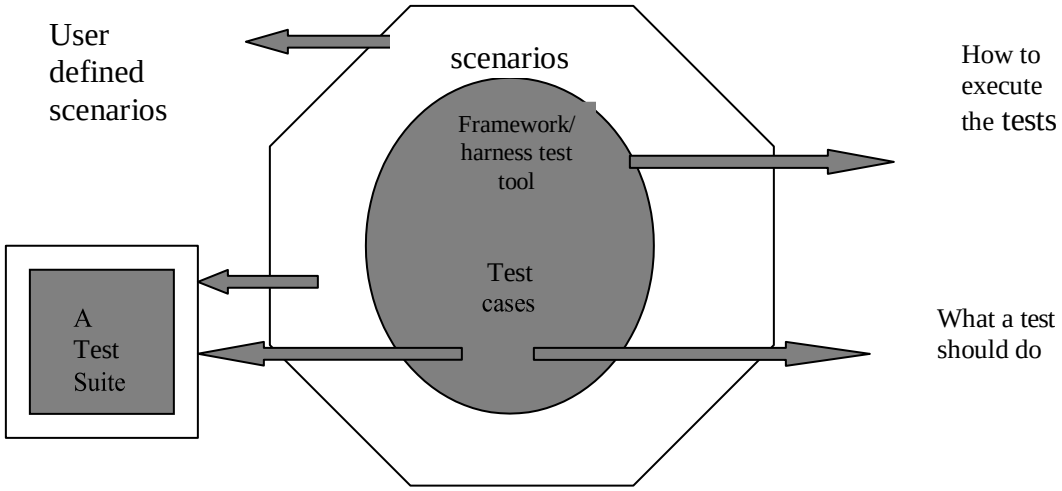
S.No	Test Cases for Testing	Belongs to What type of testing
1.	Check whether login works	Functionality
2.	Repeat Login operation in a loop for 48 hours	Reliability
3.	Perform Login from 10000 clients	Load /Stress Testing
4.	Measure time taken for Login operations in different conditions	Performance
5.	Run log in operation from a machine running Japanese language	Internationalization

From the above table , it is observed that there are 2 important dimensions

1) What operations have to be tested
2) How the operations have to be tested → scenarios

The how portion is called → scenarios.
What an operation has to do → product specific feature
How they are to be run → framework specific requirement
They are the generic requirements for all products that are being tested in an organization.

When a set of test cases is combined and associated with a set of scenarios, they are called "**test suite**". A test suite is nothing but a set of test cases that are automated and scenarios that are associated with the test cases.



Framework for test automation

SKILLS NEEDED FOR AUTOMATION

There are different "Generations of Automation". The skills required for automation depends on what generation automation the company is in or desires to be in the near future.

The automation of testing is broadly classified into three generations.

a. First generation – Record and playback

- Record and playback avoids the repetitive nature of executing tests. Almost all the test tools available in the market have the record and playback feature.
- A test engineer records the sequence of actions by keyboard characters or mouse clicks and those recorded scripts are played back later, in the same order as they were recorded. But this generation of tool has several disadvantages.
- The scripts may contain hard-coded values, thereby making it difficult to perform general types of tests.
- When the application changes, all the scripts have to be re-recorded, thereby increasing the test maintenance costs.

b. Second generation-Data-driven

- This method helps in developing test scripts that generates the set of input conditions and corresponding expected output.
- This enables the tests to be repeated for different input and output conditions. The approach takes as much time and effort as the product.

c. Third generation-Action-driven

- This technique enables a layman to create automated tests. There are no input and expected output conditions required for running the tests.
- All actions that appear on the application are automatically tested, based on a generic set of controls defined for automation.
- The set of actions are represented as objects and those objects are reused. The user needs to specify only the operations and everything else that is needed for those actions are automatically generated.
- Hence, automation in the third generation involves two major aspects- "test case automation" and "framework design".

Classification of skills for automation

Automation-first generation	**Automation- second generation**	**Automation-third generation**	
Skills for test case automation	**Skills for test case automation**	**Skills for test case automation**	**Skills for framework**
Scripting languages	Scripting languages	Scripting languages	Programming languages
Record- playback tools usage	Programming languages	Programming languages	Design and architecture skills for framework creation
	Knowledge of data generation techniques	Design and architecture of the product under test	Generic test requirements for multiple products
	Usage of the product under test	Usage of the framework	

SCOPE OF AUTOMATION

1. Identifying the Types of Testing Amenable to Automation

Certain types of tests automatically lend themselves to automation

a. **Stress, reliability, scalability, and performance testing** these types of testing require the test cases to be run form a large number of different machines for an extended period of time, such as 24 hours, 48 hours, and so on. Test cases belonging to these testing types become the first candidates for automation.

b. **Regression tests** Regression tests are repetitive in nature. These test cases are executed multiple times during the product development phases.

c. **Functional tests** These kinds of tests may require a complex set up and thus require specialized skill, which may not be available on an ongoing basis. Automating these once, using the expert skill sets, can enable using less-skilled people to run these tests on an ongoing basis.

2. Automating Areas Less Prone To Change

Automation should consider those areas where requirements go through lesser or no changes. Normally change in requirements cause scenarios and new features to be impacted, not the basic functionality of the product.

3. Automate Tests That Pertain to Standards

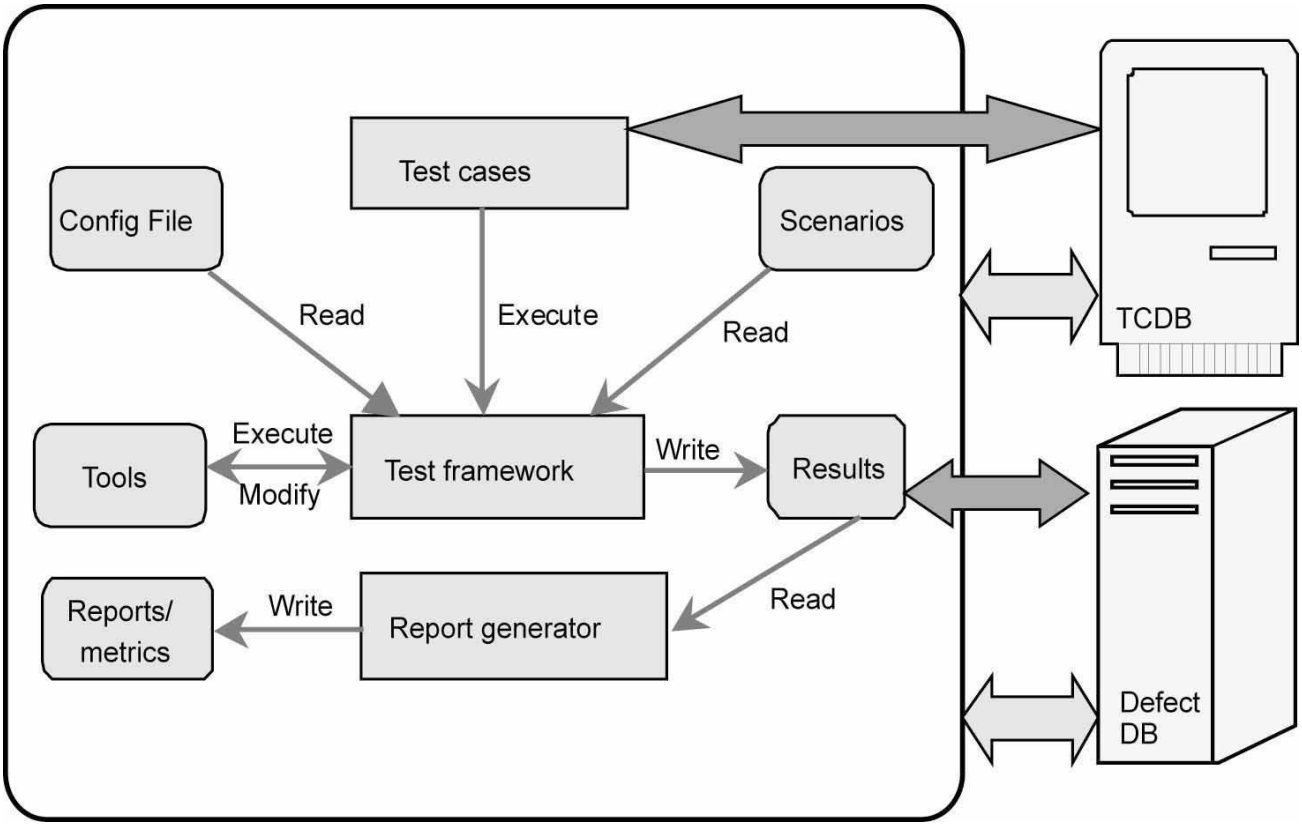
One of the tests that product may have to undergo is compliance to standards. For example, a product providing a JDBC interface should satisfy the standard JDBC tests. Automating for standards provides a dual advantage. Test suites developed for standards are not only used for product testing but can also be sold as test tools for the market.

4. Management Aspects in Automation

Prior to starting automation, adequate effort has to be spent to obtain management commitment. It involves significant effort to develop and maintain automated tools; obtaining management commitment is an important activity. Return on investment is another aspect to be considered seriously.

DESIGN AND ARCHITECTURE FOR AUTOMATION

Design and architecture is an important aspect of automation. As in product development, the design has to represent all requirements in modules and in the interactions between modules and in the interactions between modules.



Integration Testing, both internal interfaces and external interfaces have to be captured by design and architecture. In this figure the thin arrows represent the internal interfaces and the direction of flow and thick arrows show the external interfaces. All the

modules, their purpose, and interactions between them are described in the subsequent sections.

Architecture for test automation involves two major heads: a test infrastructure that covers a test case database and a defect database or defect repository. Using this infrastructure, the test framework provides a backbone that ties the selection and execution of test cases.

1. External Modules

- There are two modules that are external modules to automation-TCDB and defect DB. All the test cases, the steps to execute them, and the history of their execution are stored in the TCDB.
- The test cases in TCDB can be manual or automated. The interface shown by thick arrows represents the interaction between TCDB and the automation framework only for automated test cases.
- Defect DB or defect database or defect repository contains details of all the defects that are found in various products that are tested in a particular organization. It contains defects and all the related information test engineers submit the defects for manual test cases.
- For automated test cases, the framework can automatically submit the defects to the defect DB during execution.

2. Scenario and Configuration File Modules.

Scenarios are nothing but information on "how to execute a particular test case".

A configuration file contains a set of variables that are used in automation. A configuration file is important for running the test cases for various execution conditions and for running the tests for various input and output conditions and states.

3. Test Cases and Test Framework Modules

Test case is an object for execution for other modules in the architecture and does not represent any interaction by itself.

A test framework is a module that combines "what to execute" and "how they have to be executed". It picks up the specific test cases that are automated from TCDB and picks up the scenarios and executes them.

The test framework is considered the core of automation design. It subjects the test cases to different scenarios. The test framework contains the main logic for interacting , initiating, and controlling all modules.

A test framework can be developed by the organization internally or can be bought from the vendor.

4. Tools and Result Modules

- When a test framework performs its operations, there are a set of tools that may be required. For example, when test cases are stored as source code files in TCDB, they need to be extracted and compiled by build tools. In order to run the compiled code, certain runtime tools and utilities may be required.
- For eg , IP Packet Simulators. The result that comes out of the tests run by the test framework should not overwrite the results from the previous test runs. The history of all the previous tests run should be recorded and kept as archives.

5. Report Generator and Reports / Metrics Modules

- Once the results of a test run are available, the next step is to prepare the test reports and metrics. Preparing reports is a complex and time- consuming effort and hence it should be part of the automation design.
- There should be customized reports such as an executive report, which gives very high level status; technical reports, which give a moderate level of detail of the test run; and detailed or debug reports which are generated for developers to debug the failed test cases and the product.
- The module that takes the necessary inputs and prepares a formatted report is called a report generator. Once the results are available, the report generator can generate metrics.

GENERIC REQUIREMENTS FOR TEST TOOL/FRAMEWORK

Requirement 1: No hard coding in the test suite

- The variables for the test suite are called configuration variables. The file in which all variable names and their associated values are kept is called configuration file.
- The variables belonging to the test tool and the test suite need to be separated so that the user of the test suite need not worry about test tool variables.
- Changing test tool variables, without knowing their purpose, may impact the results of the tests.
- Providing inline comment for each of the variables will make the test suite more usable and may avoid improper usage of variables.

Ex: well documented config file

```
#Test Framework Configuration Parameter
TOOL_PATH =/tools
COMMONLIB_PATH =/tools/crm/lib
SUITE_PATH =/tools/crm

#parameter common to all the test cases in the test
VERBOSE_LEVEL =3
MAX_ERRORS=200
USER_PASSWD =hello123

#Test Case1 Parameter
TC1_USR_CREATE =0 # 1=yes 0=no
TC1_USR_PASSWD=hello123
TC1_MAX_USRS =200
```

Requirement 2: Test case/ suite expandability

Points to considered during expansion are

- ❖ Adding a test case should not affect other test cases
- ❖ Adding a test case should not result in retesting the complete test suite
- ❖ Adding a new test suite to the framework should not affect existing test suites

Requirement 3: Reuse of code for different types of testing, test cases

Points to be considered during Reuse of codes are:

1) The test suite should only do what a test is expected to do. The test framework needs to take care of "how" and

2) The test programs need to be modular to encourage reuse of code.

Requirement 4: Automatic setup and cleanup

When test cases expect a particular setup to run the tests, it will be very difficult to remember each one of them and do the setup accordingly in the manual method. Hence, each test program should have a "setup" program that will create the necessary setup before executing the test cases. The test framework should have the intelligence to find out what test cases are executed and call the appropriate setup program.

A setup for one test case may work negatively for another test case. Hence, it is important not only to create.

Requirement 5: Independent test cases

Each test case should be executed alone; there should be no dependency between test cases such as test case-2 to be executed after test case-1 and so on. This requirement enables the test engineer to select and execute any test case at random without worrying about other dependencies.

Requirement 6: Test case dependency

Making a test case dependent on another makes it necessary for a particular test case to be executed before or after a dependent test case is selected for execution

Requirement 7: Insulating test cases during execution

Insulating test cases from the environment is an important requirement for the framework or test tool. At the time of test case execution, there could be some events or interrupts or signals in the system that may affect the execution.

Requirement 8: Coding standards and directory structure

Coding standards and proper directory structures for a test suite may help the new engineers in understanding the test suite fast and help in maintaining the test suite. Incorporating the coding standards improves portability of the code.

Requirement 9: Selective execution of test cases

A Test Framework contains → many Test Suite
A Test Suite contains → many Test Program
A Test Program contains → many Test Cases

The selection of test cases need not be in any order and any combination should be allowed. Allowing test engineers to select test cases reduces the time. These selections are normally done as part of the scenario file. The selection of test cases can be done dynamically just before running the test cases, by editing the **scenario file**.

Example scenario file	**Meaning**
test-pgm-name 2,4,1,7-10	The test cases 2,4,1,7-10 are selected for execution
Tests-pgm-name	Executes all test cases

Requirement 10: Random execution of test cases

Test engineer may sometimes need to select a test case randomly from a list of test cases. Giving a set of test cases and expecting the test tool to select the test case is called random execution of test cases. A test engineer selects a set of test cases from a test suite; selecting a random test case from the given list is done by the test tool.

Ex: **scenario file.**

Random test-pgm-name 2,1,5	test tool select one out of test cases 2,1,5 for execution
Random test-pgm-name1 (2,1,5) test-pgm-name2 test-pgm-name3	Test engineer wants one out of test program 1,2,3 to be randomly executed and if pgm-name1 is selected , then one out of test cases 2,1,5 to be randomly executed, if test program 2,3 are selected , then all TC in those 2 program are executed.

Requirement 11: parallel execution of test cases

In a multi-tasking and multi processing operating systems it is possible to make several instances of the tests and make them run in parallel. Parallel execution simulates the behavior of several machines running the same test and hence is very useful for performance and load testing.

Ex: **scenario file.**

Instance, 5 test-pgm-name1 (3)	5 instances of test case 3 in test-pgm-name1 are executed
Instance, 5 test-pgm-name1 (2,1,5) test-pgm-name2 test-pgm-name3	5 instances of test programs are created , within each of the five instances that are created the test program 1,2,3, are executed in sequence .

Requirement 12: Looping the test cases

Reliability testing requires the test cases to be executed in a loop. There are two types of loops that are available.

1) iteration loop - gives the number of iterations of a particular test case to be executed.
2) timed loop - which keeps executing the test cases in a loop till the specified time duration is reached.

Ex: **scenario file.**

Repeat_loop, 50 test-pgm-name1 (3)	test case 3 in test-pgm-name1 is repeated 50 times.
Time_loop, 5 Hours test-pgm-name1 (2,1,5) test-pgm-name2 test-pgm-name3	TC 2,1,5 from test-pgm-name1 and all test cases from the test program2 and 3 are executed in order, in a loop for 5 hours

Requirement 13: Grouping of test scenarios

The group scenarios allow the selected test cases to be executed in order, random, in a loop all at the same time. The grouping of scenarios allows several tests to be executed in a predetermined combination of scenarios.

Ex: **scenario file.**

Group_scenario1 Parallel, 2 AND repeat,10@scen1 Scen1 test-pgm1 (2,1,5) test-pgm2 test-pgm3	Group scenario was created to execute 2 instances of the individual scenario "scen1" in a loop 10 times

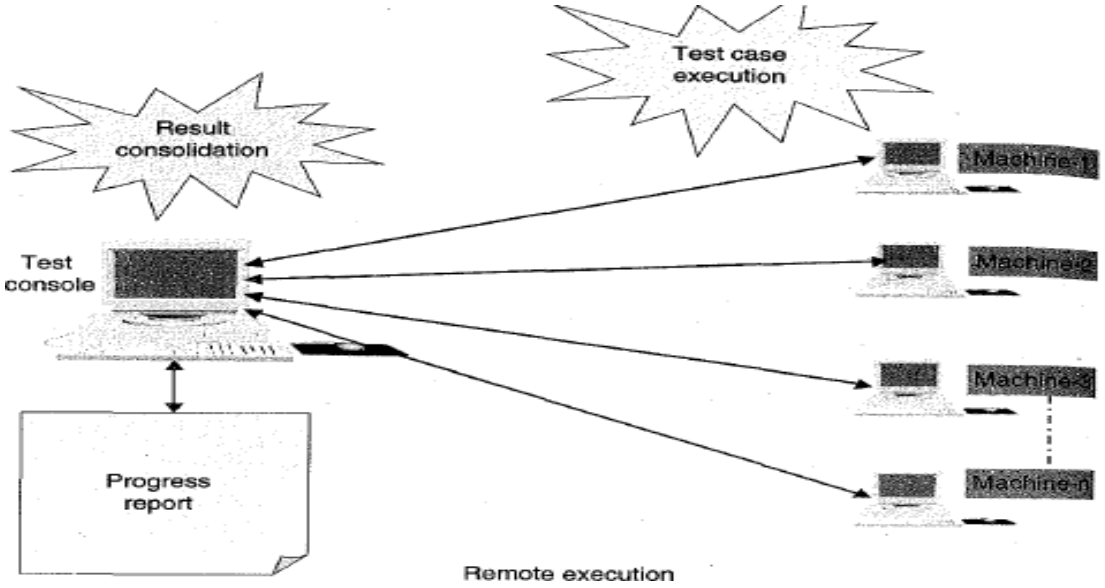
Requirement 14: Test case execution based on previous results

One of the effective practices is to select the test cases that are not executed and test cases that failed in the past and focus more on them. Some of the common scenarios that require test cases to be executed based on the earlier results are

1. Rerun all test cases which were executed previously;
2. Resume the test cases from where they were stopped the previous time;
3. Rerun only failed/not run test cases; and
4. Execute all test cases that were executed previously.

Requirement 15: Remote execution of test cases

The central machine that allocates tests to multiple machines and co-ordinates the execution and result is called test console or test monitor. In the absence of a test console, not only does executing the results from multiple machines become difficult, collecting the results from all those machines also becomes difficult.



Role of test console and multiple execution machine.

Requirement 16: Automatic archival of test data

The test cases have to be repeated the same way as before, with the same scenarios, same configuration variables and values, and so on. This requires that all the related information for the test cases have to be archived. It includes

1) What configuration variables were used
2) What scenario was used
3) What program were executed and from what path

Requirement 17: Reporting scheme

Every test suite needs to have a reporting scheme from where meaningful reports can be extracted. As we have seen in the design and architecture of framework, the report generator should have the capability to look at the results file and generate various reports. Audit logs are very important to analyze the behavior of a test suite and a product. A reporting scheme should include

1. When the framework, scenario, test suite, test program, and each test case were started/ completed;
2. Result of each test case;
3. Log messages;
4. Category of events and log of events; and
5. Audit reports

Requirement 18: Independent of languages

A framework or test tool should provide a choice of languages and scripts that are popular in the software development area.

- A framework should be independent of programming languages and scripts.
- A framework should provide choice of programming languages, scripts, and their combinations.
- A framework or test suite should not force a language/script.
- A framework or test suite should work with different test programs written using different languages, and scripts.
- The internal scripts and options used by the framework should allow the developers of a test suite to migrate to better framework.

Requirement 19: portability to different platforms

With the advent of platform-independent languages and technologies, there are many products in the market that are supported in multiple OS and language platforms.

- The framework and its interfaces should be supported on various platforms.
- Portability to different platforms is a basic requirement for test tool/ test suite.
- The language/script used in the test suite should be selected carefully so that it runs on different platforms.
- The language/ script written for the test suite should not contain platform-specific calls.

CHALLENGES IN AUTOMATION

- Test automation presents some very unique challenges. The most important of these challenges is management commitment.
- Automation should not be viewed as a panacea for all problems nor should it be perceived as a quick-fix solution for all the quality problems in a product.
- The main challenge here is because of the heavy front-loading of costs of test automation, management starts to look for an early payback.
- Successful test automation endeavors are characterized by unflinching management commitment, a clear vision of the goals, and the ability to set realistic short-term goals that track progress with respect to the long-term vision.

TEST METRICS AND MEASUREMENTS

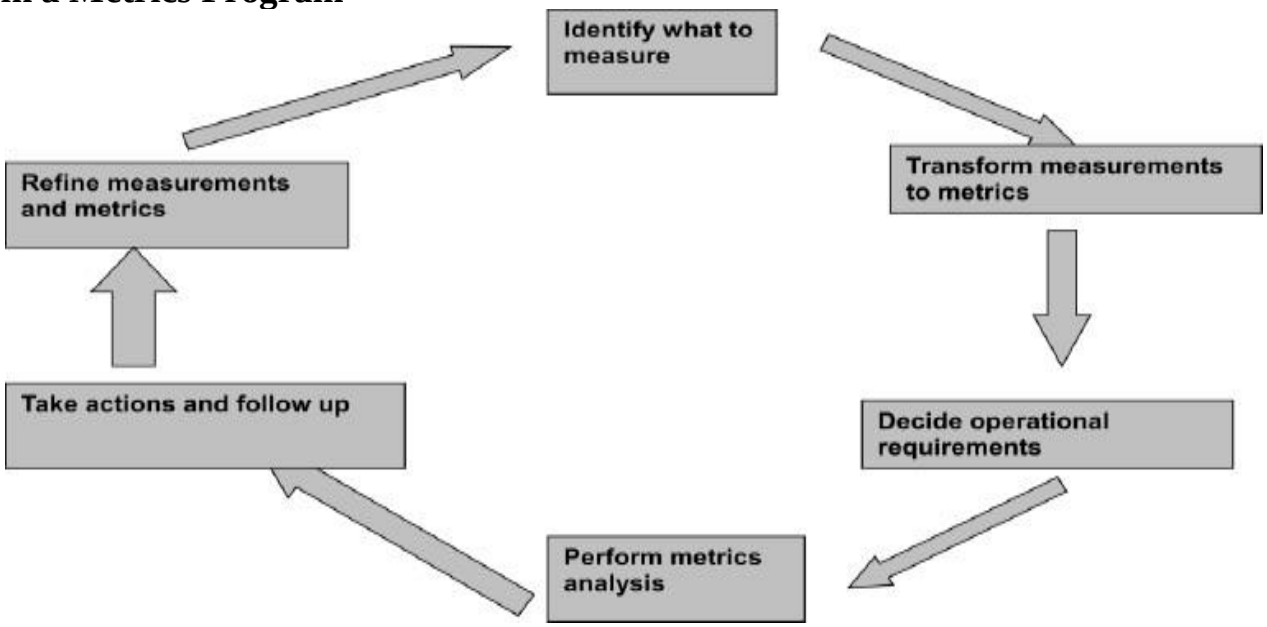
Definition

- Metrics are the source of measurement.
- Metrics derive information from raw data with a view to help in decision making.

Ex: No of defects , No of test cases , effort , schedule
Metrics are needed to know test case execution productivity and to estimate test completion date.

Effort : actual time that is spent on a particular activity or a phase Schedule : Elapsed days for a complete set of activities

Steps in a Metrics Program



Step1: Metrics program is to decide what measurements are important and collect data accordingly. Ex for Measurements: effort spent on testing , no of defects , no of test cases.

Step2: It deals with defining how to combine data points or measurement to provide meaningful metrics. A particular metric can use one or more measurements

Step3: It involves with operational requirement for measurements. It contains
Who should collect measurements?
Who should receive the analysis etc.
This step helps to decide on the appropriate periodicity for the measurements as well as assign operational responsibility for collecting, recording and reporting the measurements.
Daily basis measurements → no of testcases executed, no of defects found, defects fixed..
Weekly measurements → how may testcases produced 40 defects,

Step4: This step analyzes the metrics to identify both positive area and improvement areas on product quality.

Step5: The final step is to take necessary action and follow up on the action.

Step6: To continue with next iteration of metrics programs, measuring a different set of measurements, leading to more refined metrics addressing different issues.

WHY METRICS IN TESTING?

Knowing only how much testing got completed does not answer the question on when the testing will get completed and when the product is ready for release. To Answer these questions , one need to estimate the following

$$\text{Days needed to complete testing} = \frac{\text{Total test cases yet to be executed}}{\text{Total test case execution productivity}}$$

test case execution productivity → testcases executed per person day

$$\text{Total Days needed for defect fixes} = \frac{(\text{outstanding defects yet to fixed} + \text{Defects that can be found in future test cycles})}{\text{Defect fixing capability}}$$

$$\text{Days needed for Release} = \text{Max}(\text{Days needed for testing}, \text{days needed for defect fixes})$$

More accurate estimate with regression testing

$$\text{Days needed for Release} = \text{Max}(\text{Days needed for testing}, (\text{days needed for defect fixes} + \text{Days needed for regressing outstanding defect fixes}))$$

Metrics are needed to know test case execution productivity and to estimate test completion date.

Metrics in testing help in identifying

- When to make the release.
- What to release
- Whether the product is being released with known quality.

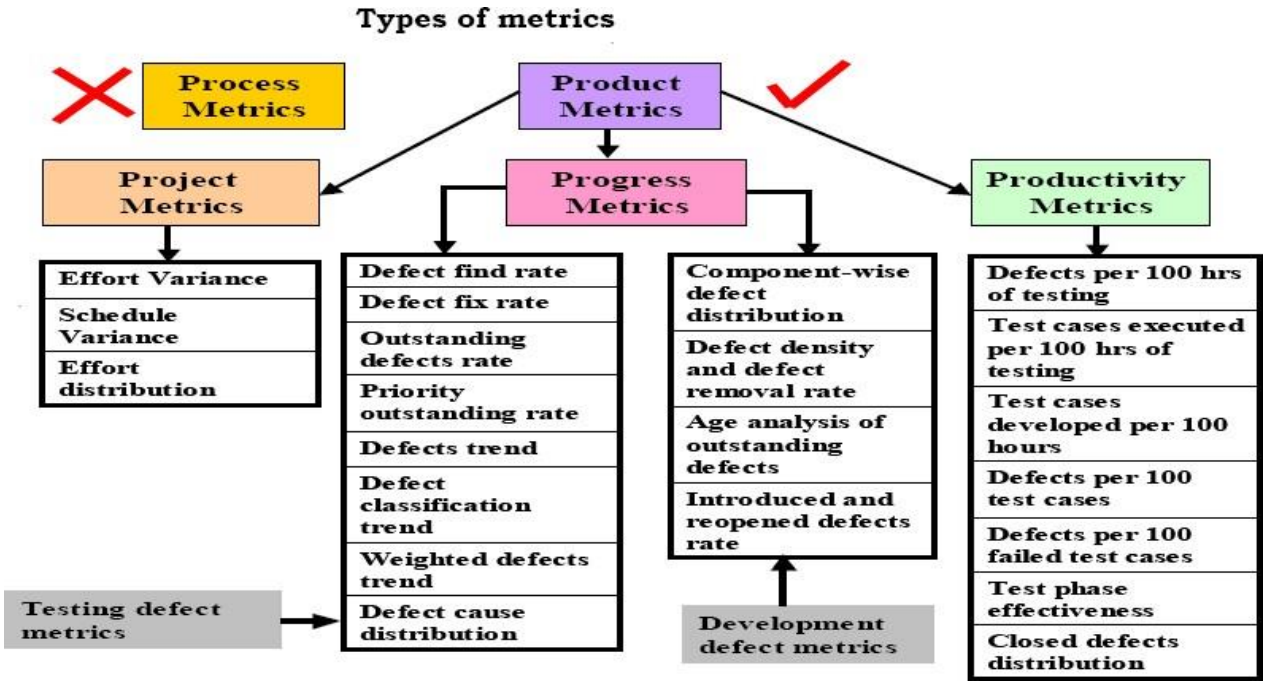
TYPES OF METRICS

Metrics can be classified into different types based on what they measure and what area they focus on. At a very high level, metrics can be classified as product metrics and process metrics.

Product metrics can be further classified as:

1. **Project metrics** A set of metrics that indicates how the project is planned and executed.
2. **Progress metrics** A set of metrics that tracks how the different activities of the project are progressing. The activities include both development activities and testing activities. Progress metrics is monitored during testing phases. Progress metrics helps in finding out the status of test activities and they are also good indicators of product quality. Progress metrics, for convenience, is further classified into
    1) test defect metrics and
    2) development defect metrics.

3. **Productivity metrics** A set of metrics that takes into account various productivity numbers that can be collected and used for planning and tracking testing activities. These metrics help in planning and tracking testing activities. These metrics help in planning and estimating of testing activities.



I) PROJECT METRICS

A typical project starts with requirements gathering and ends with product release. All the phases that fall in between these points need to be planned and tracked. In the planning cycle, the scope of the project is finalized. The project scope gets translated to size estimates, which specify the quantum of work to be done. This size estimate gets translated to effort estimate for each of the phases and activities by using the available productivity data available.

base lined effort →The initial effort

revised effort. → As the project progresses and if the scope of the project changes, then the effort estimates are re-evaluated again and this re-evaluated effort estimate is called revised effort.

Two factors **Effort :** actual time that is spent on a particular activity or a phase **Schedule :** Elapsed days for a complete set of activities

- If the effort is tracked closely & met then schedule can be met.
- If planned effort is equal to actual effort and schedule not met then project is not considered as successful one.
-

The basic measurements are

1. initial baselined effort and schedule
2. The actual effort
3. The revised estimate of effort and schedule

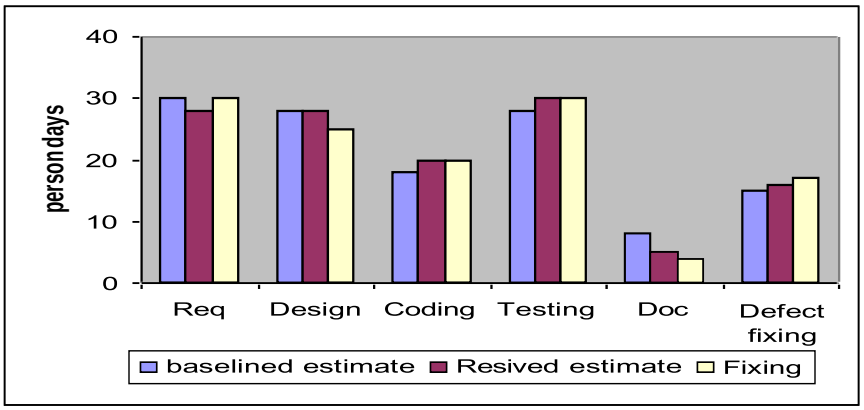
1. Effort Variance (Planned Vs Actual)

When the baselined effort estimates, revised effort estimates, and actual effort are plotted together for all the phases of SDLC, it provides many insights about the estimation process. As different set of people may get involved in different phases, it is a good idea to plot these effort numbers phase-wise. A sample data for each of the phase is plotted in the chart.

If there is a substantial difference between the baselined and revised effort, it points to incorrect initial estimation. Calculating effort variance for each of the phases provides a quantitative measure of the relative difference between the revised and actual efforts.

Calculating effort variance for each of the phases provides a quantitative measure of the relative difference between the revised and actual efforts.

$$\text{Effort variance \%} = \frac{\text{actual effort} - \text{revised estimate}}{\text{Revised estimate}} \times 100$$



Phase wise effort variation

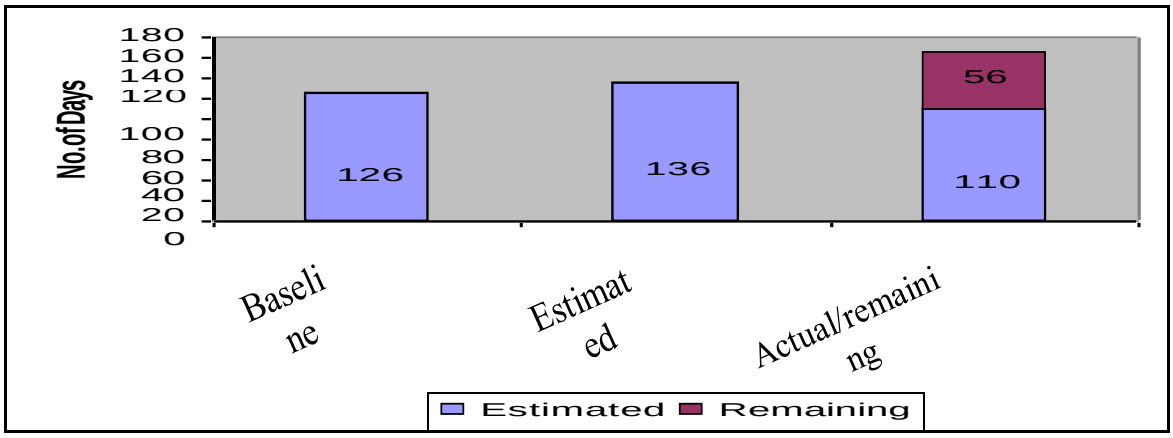
Sample variance percentage by phase.

Effort	Req	Design	Coding	Testing	Doc	Defect Fixing
Variance %	7.1	8.7	5	0	40	15

- All the baseline estimates, revised estimates, and actual effort are plotted together for each of the phases. The variance can be consolidated into as shown in the above table.
- A variance of more than 5% in any of the SDLC phase indicates the scope for improvement in the estimation. The variance is acceptable only for the coding and testing phases.
- The variance can be negative also. A negative variance is an indication of an over estimate.
- The variance is acceptable only for the coding and testing phases.

2. Schedule Variance (Planned vs Actual)

Schedule variance is calculated at the end of every milestone to find out how well the project is doing with respect to the schedule.



To get a real picture on schedule in the middle of project execution, it is important to calculate “remaining days yet to be spent” on the project and plot it along with the “actual

schedule spent" as in the above chart. "Remaining days yet to be spent" can be calculated by adding up all remaining activities. If the remaining days yet to be spent on project is not calculated and plotted, it does not give any value to the chart in the middle of the project, because the deviation cannot be inferred visually from the chare. The remaining days in the schedule becomes zero when the release is met.

Effort and schedule variance have to be analyzed in totality, not in isolation. This is because while effort is a major driver of the cost, schedule determines how best a product can exploit market opportunities, variance can be classified into negative variance, zero variance, acceptable variance, and unacceptable variance.

Interpretation of range of effort and schedule variation

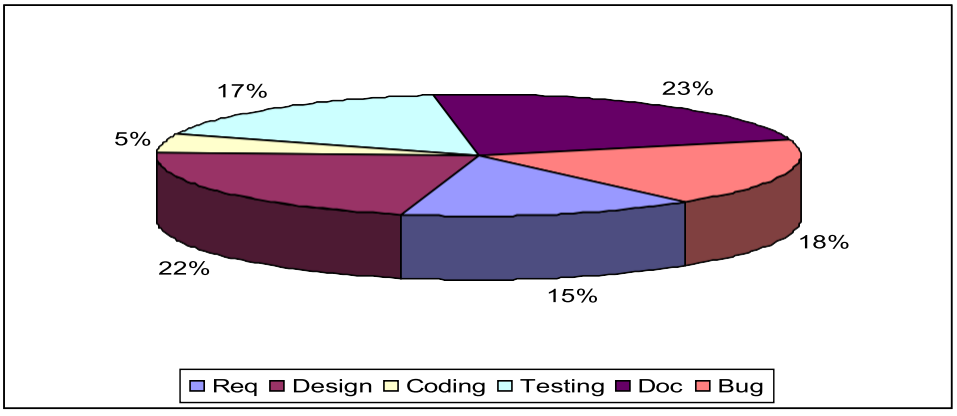
Effort Variance	Schedule Variance	Probable Causes /Result
Zero (or) Acceptable variance (0-5)	Zero variance	A Well executed Project
	Acceptable variance	Need slight improvement
Unacceptable variance (>5)	Zero (or) Acceptable variance	Under estimation , need further analysis
	Unacceptable variance	Under estimation of both effort and schedule
Negative variance	Zero (or) Acceptable variance	Over estimation, need improvement
	Negative variance	Over estimation, need improvement

3. Effort Distribution Across Phases

Adequate and appropriate effort needs to be spent in each of the SDLC phase for a quality product release.

The distribution percentage across the different phases can be estimated at the time of planning and these can be compared with the actual at the time of release for getting a comfort feeling on the release and estimation methods. A sample distribution of effort across phases is given in figure.

Actual Effort Distribution



Effort distribution :

Req > Testing > design > bug fixing > coding > doc

Mature organizations spend at least 10-15 % of the total effort in requirements and approximately the same effort in the design phase. The effort percentage for testing depends on the type of release and amount of change to the existing code base and functionality. Typically, organizations spend about 20 -50 % of their total effort in testing.

II) PROGRESS METRICS

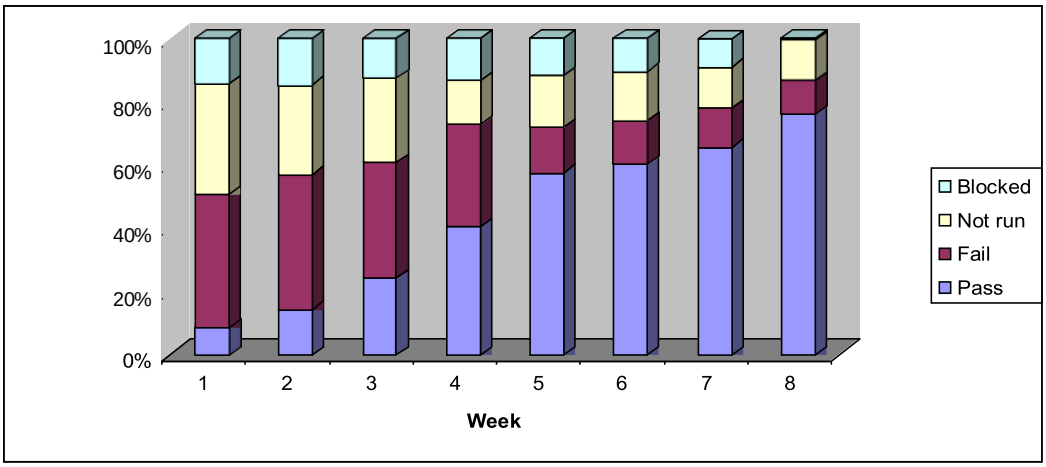
One of the main objectives of testing is to find as many defects as possible before any customer finds them. The number of defects that are found in the product is one of the main indicators of quality.

Defects get detected by the testing team and get fixed by the development team. Defect metrics are further classified in to

1. test defect metrics
2. development defect metrics

The progress chart gives

- pass rate
- fail rate of executed test cases
- pending test cases
- test cases that are waiting for defects to be fixed.



A scenario represented by such a progress chart shows that not only is testing progressing well, but also that the product quality is improving. The chart had shown a trend that as the weeks progress, the “not run” cases are not reducing in number, or “blocked” cases are increasing in number, or “pass” cases are not increasing, then it would clearly point to quality problems in the product that prevent the product from being ready for release.

1. TEST DEFECT METRICS

The next set of metrics helps us understand how the defects that are found can be used to improve testing and product quality.

Some organizations classify effects by assigning a defect priority (for example P1, P2, P3, and so on)Some organizations use defect severity levels (for example, S1, S2, S3, and so

on).The priority of a defect can change dynamically once assigned. Severity is absolute and does not change often as they reflect the state and quality of the product.

Table -Defect priority and defect severity – sample interpretation.

Defect priority is based on defect fixing and defect severity is based on functionality level.

Priority	**What it means**
1	Fix the defect on highest priority; fix it before the next build
2.	Fix the defect on high priority before next test cycle
3	Fix the defect on moderate priority when time permits, before the release
4	Postpone this defect for next release or live with this defect
Severity	**What it means**
1	The basic product functionality failing or product crashes
2	Unexpected error condition or a functionality not working
3	A minor functionality is failing or behaves differently than expected
4	Cosmetic issue and no impact on the users

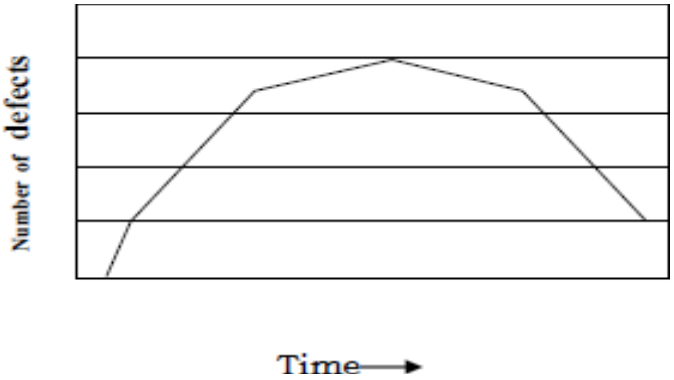
This defect classification is based on priority and severity.

Defect Classification	**What it Means**
Extreme	• Product crashes or unusable • Need to be fixed immediately
Critical	• Basic functionality of the product not working • Needs to be fixed before next test cycle starts
Important	• Extended functionality of the product not working • Does not affect the progress of testing • Fix it before the release
Minor	• Product Behaves differently • No impact on test team or customer • Fix it when time permits
Cosmetic	• Minor Irritant • Need not be fixed for this release

a) Defect Find Rate

The purpose of testing is to find defects early in the test cycle. The idea of testing is to find as many defects as possible early in the cycle. However, this may not be possible for two reasons. First, not all features of a product may become available early; because of scheduling of resources, the features of a product arrive in a particular sequence. Some of the test cases may be blocked because of some show stopper defects.

Once a majority of the modules become available and the defects that are blocking the tests are fixed, the defect arrival rate increases. After a certain period of defect fixing and testing, the arrival of defects tends to slow down and a continuation of that enables product release. This results in a “bell curve” as shown in figure.



b) Defect fix rate

The purpose of development is to fix defects as soon as they arrive. If the goal of testing is to find defects as early as possible, it is natural to expect that the goal of development should be to fix defects as soon as they arrive. There is a reason why defect fixing rate should be same as defect arrival rate. If more defects are fixed later in the cycle, they may not get tested properly for all possible side-effects.

c) Outstanding defects rate

In a well executed project, the number of outstanding defects is very close to zero all the time during the test cycle. The number of defects outstanding in the product is calculated by subtracting the total defects fixed from the total defects found in the product.

The number of defects outstanding = total defects found in the product - total defects fixed

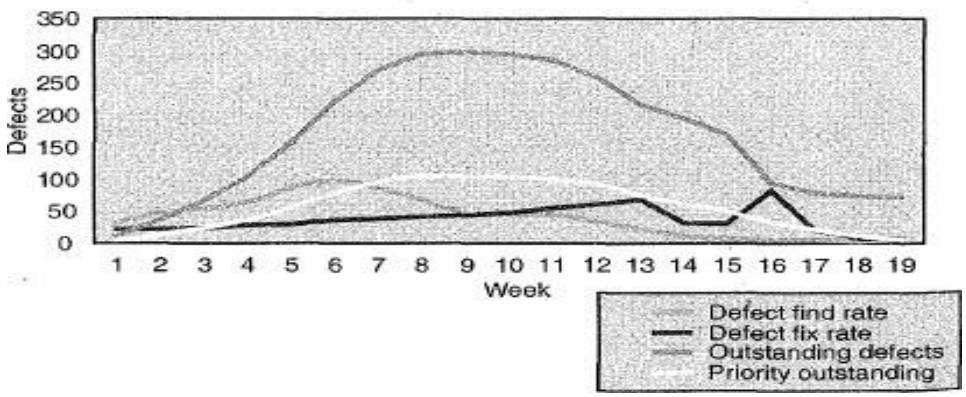
d) Priority outstanding rate

The modification to the outstanding defects rate curve by plotting only the high- priority defects and filtering out the low- priority defects is called priority outstanding defects. This is an important method because closer to product release, the product team would not want to fix the low – priority defects.

Normally only high-priority defects are tracked during the period closer to release. Some high-priority defects may require a change in design or architecture & fixed immediately

e) Defect trend

The effectiveness analysis increases when several perspectives of find rate, fix rate, outstanding, and priority outstanding defects are combined.



Defect trend

The following observations can be made.

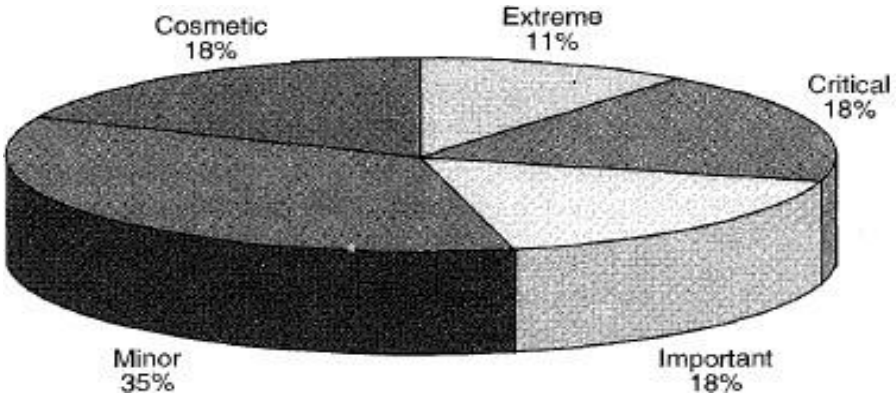
1. The find rate, fix rate, outstanding defects, and priority outstanding follow a bell curve pattern, indicating readiness for release at the end of the 19th week.
2. a sudden downward movement as well as upward spike in defect fixes rate needs analysis (13th to 17th week in the chart above)
3. By looking at the priority outstanding which shows close to zero defects in the 19th week, it can be concluded that all outstanding defects belong to low priority.
4. A smooth priority outstanding rate suggests that priority defects were closely tracked and fixed.

f) Defect Classification trend

Providing the perspective of defect classification in the chart helps in finding out release readiness of the product. When talking about the total number of outstanding defects, some of the questions that can be asked are

- How many of them are extreme defects?
- How many are critical?
- How many are important?

These questions require the charts to be plotted separately based on defect classification. The sum of extreme, critical, important, minor, and cosmetic defects is equal to the total number of defects.



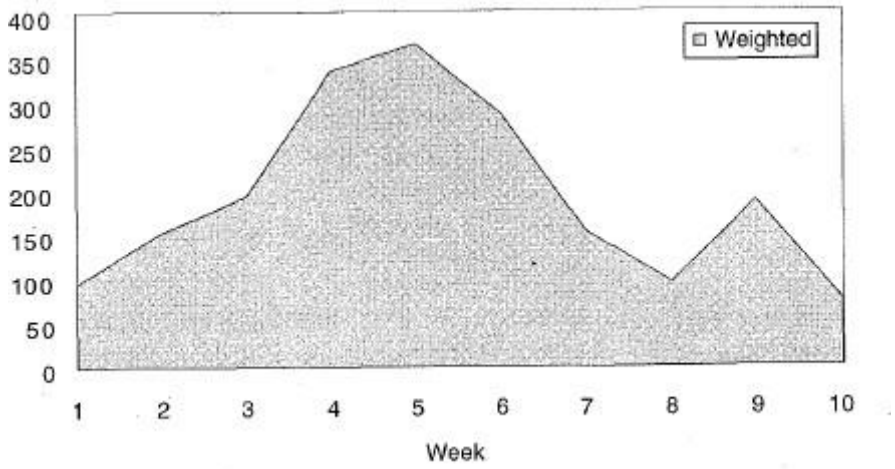
Pie chart of defect distribution

g) Weighted defects trend

Weighted defect helps in quick analysis of defect, instead of worrying about the classification of defects.

Weighted defects= (Extreme * 5 + Critical * 4 + important * 3 + Minor * 2 + Cosmetic)

Both “large defects” and “large number of small defects” affect product release.



Weighted defects trend.

From Figure it can be noted that

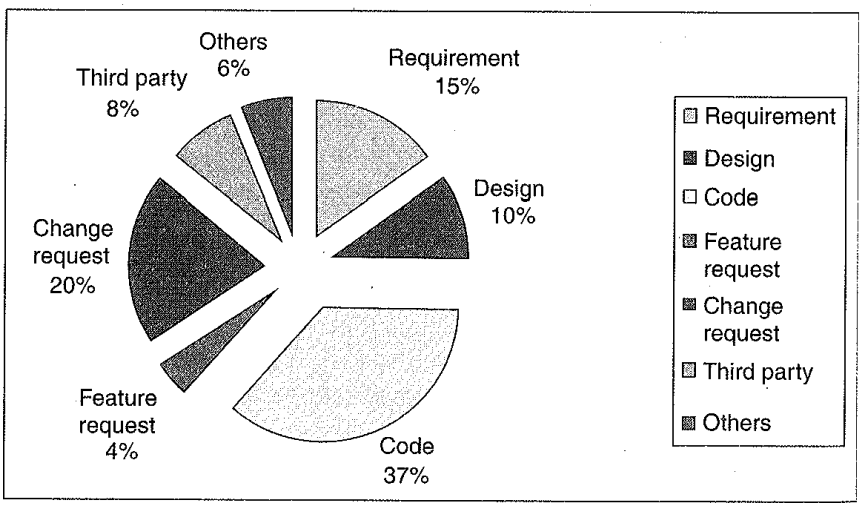
1. The ninth week has more weighted defects, which means existence of "large number of small defects" or "significant numbers of large defects" or a combination of the two. This is consistent with our interpretation of the same data using the stacked area chart.
2. The tenth week has a significant (more than 50) number of weighted defects indicating the product is not ready for release.

h) Defect cause distribution

Logical questions that would arise are:

1. Why are those defects occurring and what are the root causes?
2. What areas must be focused for getting more defects out of testing?

Finding the root causes of the defects help in identifying more defects and sometimes help in even preventing the defects.



Defect cause distribution chart

2. Development Defect Metrics

To map the defects to different components of the product , the parameter is LOC. It has

a) Component wise Defect Distribution
b) Defect Density & defect removal rate
c) Age Analysis of outstanding defect
d) Introduced and reopened defects trend

a) Component wise Defect Distribution

When module wise defect distribution is done , modules like install ,reports , client and database has > 20 defects indicating that more focus and resources are needed for these components.
So knowing the components producing more defects helps in defect fix plan and in deciding what to release.

b) Defect Density & defect removal rate

Defect density maps the defects in the product with the volume of code that is produced for the product.

Defects per KLOC = Total defects found in the product / total Executable line of code in KLOC

Variants to this metrics is to calculate AMD (add , modify , delete code) to find how a release affects product quality .

Defects per KLOC = Total defects found in the product / total Executable AMD line of code in KLOC

Defect removal rate = (defect found by verification activities + defects found in Unit test) / defect found by testing team) * 100

c) Age Analysis of outstanding defect

The time needed to fix a defect may be proportional to its age. It helps in finding out whether the defects are fixed as soon as they arrive and to ensure that long pending defects are given adequate priority.

d) Introduced and reopened defects trend

Introduced defect(ID): when adding new code or modifying the code to provide a defect fix , something that was working earlier may stop working , this is called ID.

reopened defects : fix that is provided in the code may not have fixed the problem completely or some other modification may have reproduced a defect that was fixed earlier. This is called as reopened defects.

Testing is not meant to find the same defects again ; release readiness should consider the quality of defect fixes.

III) PRODUCTIVITY METRICS

Productivity metrics combine several measurements and parameters with effort spend on the product. They help in finding out the capability of the team as well as for other purpose, such as

1. Estimating for the new release.
2. Finding out how well the team is progressing, understanding the reasons for (both positive and negative) variations in results.
3. Estimating the number of defects that can be found
4. Estimating release data and quality
5. Estimating the cost involved in the release.

a) Defects per 100 Hours of Testing

Defects per 100 hours of testing= (Total defects found in the product for a period / Total hours spent to get those defects) * 100

Test Cases Executed per 100 Hours of Testing

Test cases executed per 100 hours of testing = (Total test cases executed for a period / Total hours spent in test execution) * 100

b) Test cases Developed per 100 Hours of Testing

Test cases developed per 100 hours of testing= (Total test cases developed for a period / Total hours spent in test case development) * 100

c) Defects per 100 Test Cases

Defects per 100 test cases = (Total defects found for a period / Total test cases executed for the same period) * 100

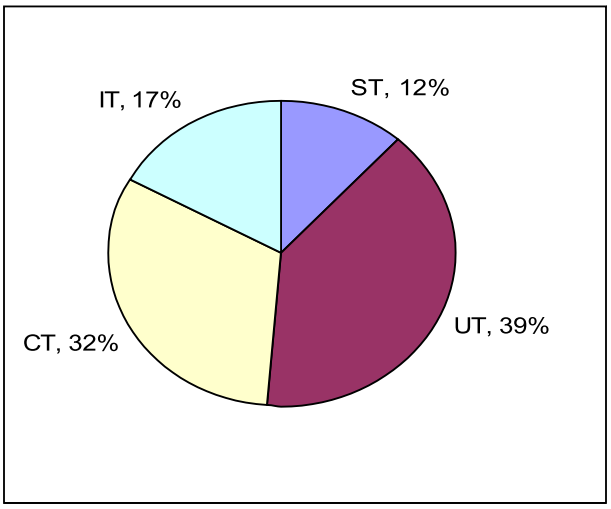
d) Defects per 100 Failed Test Cases

Defects per 100 failed test cases = (Total defects found for a period/Total test cases failed due to those defects) * 100

e) Test Phase Effectiveness

The following few observations can be made

1. A good proportion of defects were found in the early phases of testing (UT and CT).
2. Product quality improved from phase to phase (shown by less percent of defects found in the later test phases – IT and ST)



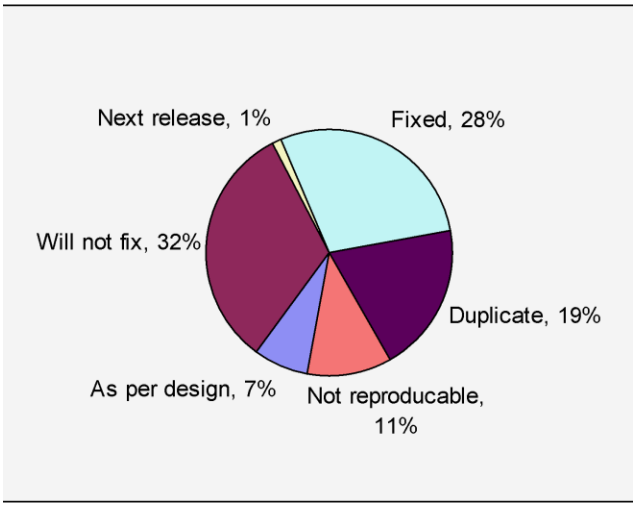
Test phase effectiveness

f) Closed Defect Distribution

The closed defect distribution helps in this analysis as shown in the figure below. From the chart, the following observations can be made.

1. Only 28% of the defects found by test team were fixed in the product. This suggests that product quality needs improvement before release.
2. Of the defects filled 19% were duplicates. It suggests that the test team needs to update itself on existing defects before new defects are filed.
3. Non-reproducible defects amounted to 11%. This means that the product has some random defects or the defects are not provided with reproducible test cases. This area needs further analysis.

4. Close to 40% of defects were not fixed for reasons “as per design,” “will not fix,” and “next release.” These defects may impact the customers. They need further discussion and defect fixes need to be provided to improve release quality.



Closed defect distribution