CHAPTER 1
INTRODUCTION TO SOFTWARE TESTING

1.1 TESTING AS AN ENGINEERING ACTIVITY

Software systems are becoming more challenging to build new methods, techniques and tools are used to develop software and maintenances. The poor quality software that can cause loss of life or property. So, high quality software should be produced.

For that, you should provide necessary training and create interest to the software development team in the areas of software product and process quality. Highly qualified staff ensure that software products are built on time & Software products are built within budget.

Software products with high quality attributes are:

- o Reliability
- o Correctness
- o The ability to meet all user requirements

Elements of the Engineering Design

- Basic principles
- Processes
- Standards
- Measurements
- Tools
- Methods
- Best practices
- Codes of ethics
- Body of knowledge
  - ➢ The profession of software engineering is slowly acted as a formal engineering discipline.
  - ➢ The software development activities are also using the engineering approach as follows:
    - ✓ The development process is well understood
    - ✓ Projects are well planned
    - ✓ Life cycles models are defined and followed
    - ✓ Standards are in place in process and products
    - ✓ Measurements are employed to evaluate product and process quality
    - ✓ Components are reviewed
    - ✓ Verification and validation process play a key role in producing a quality server
    - ✓ Engineers have proper education, training and certification.

- ✓ A testing specialist who is trained as an engineer should have knowledge of test related principles, measurements, standards, plans, tools and methods, and should learn how to apply them to the testing tasks to be performed.

1.2 ROLE OF PROCESS IN SOFTWARE QUALITY

- The software engineering process is the set of methods, practices, standards, documents, activities, policies and procedures that software engineers used to develop and maintain a software system and its associated artifacts such as projects and test plans, design documents, code and manuals.
- Types of manuals namely
    1. user manual
    2. troubleshoot manual
    3. administration manual
    4. Quality factors involved in high quality software products:
        1. usability
        2. reliability
        3. testability
        4. maintainability
- Different models are used in software engineering process such as
    - CMM
    - spice
    - TMM

1.3 TESTING AS A PROCESS

- Software development process has includes five sub process namely
    1. requirement / analysis process
    2. product specification process
    3. Design process
    4. implementation process
    5. testing process : v&v
- The testing process further involves in two processes namely verification and validation.
- The technical aspects of the testing relate to the techniques, method, measurement, and tools used to ensure that the software under test is as defect free and reliable.
- The testing itself is related to two processes called verification, validation.

Validation: It is the process of evaluate a software system during or at the end of the cycle in order to determine whether it satisfies the specified requirements.

Meaning: Validation is associated with traditional execution based technique that means exercising the code with test cases.

Verification: It is process of evaluating a software system to determine whether product of a given development phase satisfy the condition imposed at the start that phase.

Meaning: Verification is usually associated with activities such as inspection and reviews of software deliverables.

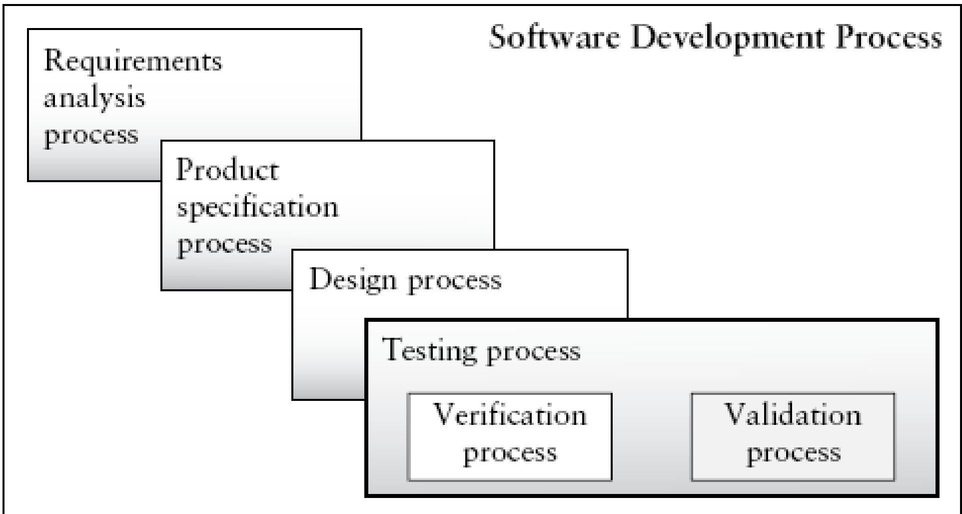


FIG. Example processes embedded in the software development process

- **Software testing:** Testing is generally described as a group of procedures carried out to evaluate source aspects of a piece of software.
  Purpose of testing processes: testing has a dual purpose process namely reveal defects and to evaluate quality attributes of software.
- **Debugging:** it is the process of locating the fault or defect, repairing the code and retesting the code

1.4 BASIC DEFINITIONS

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.

Errors

An error is a mistake, misconception, or misunderstanding on the part of a software developer.

Failures

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

Test Cases

The usual approach to detecting defects in a piece of software is for the tester to select a set of input data and then execute the software with the input data under a particular set of conditions. 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.

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.

Software Quality

Two concise definitions for quality are found in the IEEE Standard Glossary of Software Engineering Terminology:

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. These are characteristics that reflect quality. For software artifacts we can measure the degree to which they possess a given quality attribute with quality metrics.

Quality metrics

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.
Many different quality attributes have been described for software. 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.** This attribute is of more interest to developers/testers than to clients. It can be expressed in the following two ways:
1. the amount of effort needed to test the software to ensure it performs 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.

Reviews
In contrast to dynamic execution-based testing techniques that can be used to detect defects and evaluate software quality, reviews are a type of static testing technique that can be used to evaluate the quality of a software artifact such as a requirements document, a test plan, a design document, a code component. Reviews are also a tool that can be applied to revealing defects in these types of documents.
Definition: A review is a group meeting whose purpose is to evaluate a software artifact or a set of software artifacts.

CHAPTER 2
TESTING FUNDAMENDALS

2.1 SOFTWARE TESTING PRINCIPLES

- ➢ Testing principles are important to test specialists 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;
    - A general or fundamental, law, doctrine, or assumption,
    - A rule or code for conduct,
    - The laws or facts of nature underlying the working of an artificial device.

The principles as stated below only related to execution-based testing.

Principle1: Testing is the process of exercising a software component using a selected set of tests cases, with the internet.

- Revealing defects, and
- 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 a software quality. This principle supports testing as an execution-based activity to detect defects.
- ➢ The term **defect** as used in this and in subsequent principle represents any deviations in the software that have negative impact on its functionality, performance, reliability, security and other of its specified quality attributes.

Principle-2:When the test objectives is to detect defects, then a good test case is one that has a high probability of revealing a yet undetected defects.

- ➢ The goal for the test is to prove / disprove the hypothesis that is, determine if the specific defect is present / absent.
- ➢ A tester can justify the expenditure of the resources by careful test design so that principle two is supported.

Principle-3:Test result should be inspected meticulously.

- ➢ Tester need to carefully inspect and interpret test results. Several erroneous and costly scenarios may occur if care is not taken.

Example:

A failure may be overloaded, 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 result. The defect may be revealed at some later stage of testing, but in that case it may be moke costly and difficult to locate and repair.

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

- The test case is of no value unless there is an explicit statement of the expected outputs or results.

Example:

A specific variable value must be observed or a certain panel button that must light up.

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

- The tester must not assume that the software under test will always be provided with valid inputs.
- Inputs may be incorrect for several reasons.

Example:

Software users may have misunderstandings, or lack information about the nature of the inputs. They often make typographical errors even when compute / correct information are available. Device may also provide invalid inputs due to erroneous conditions and malfunctions.

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

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.

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

- Tester must realize that
    1. Developers have a great deal of pride in their work and
    2. On practical level it may be difficult for them to conceptualize where defects could be found.

Principle-8: Tests must be repeatable and reusable

- This principle calls for experiments in the testing domain to require recording of the exact condition of the test, any special events that occurred, equipment used, and a careful accounting of the results.
- This information invaluable to the developers when the code is returned for debugging so that they can duplicate test conditions.

Principle-9: Testing should be planned.

- Test plan should be developed for each level of testing, and objective for each level should be described in the associated plan.

- The objectives should be stated as quantitatively as possible plan, with their precisely specified objectives.

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 into the software lifecycle starting as early as in the requirements analysis phases, and continue on throughout the software lifecycle in parallel with development activities.

Principle-11: Testing is a creative and challenging task.

Difficult and challenges for the tester includes,

- 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 type and where faults of a certain type night occur in code constructs.
- A tester needs a reason like scientist and propose hypotheses that related to presence of specific type of defects.
- A tester needs to design and record test procedure for running the tests.
- A tester to plan for testing and allocate the proper resources.
- A tester need to execute the tests and is responsible for recording results.
- A tester need to learn to use tools and keeps abreast of the newest test tool advance.

2.2 TESTERS ROLE IN A SOFTWARE DEVELOPMENT ORGANISATION

- The tester job is to reveal defects, find weak points, inconsistent behavior and circumstances where the software doesn't work as expected.
- The main function of a tester is to plan, execute, record and analyze test. They don't debug the software.
- When defects are detected during testing, software should be returned to the developers who locate the defect and repair the code. The tester may cooperate with code developers and also need to work along with the requirements engineers to ensure that requirements are testable and to plan for system and acceptance test.
- Test managers were need to cooperate with project managers in order to develop test plan, and with upper management to provide input for the development and maintenance of organizational testing standards, policies and goals
- Testers also need to cooperate with software quality assurance staff and software engineering group members

CHAPTER 3
DEFECT CLASSES AND DEFECT REPOSITORY

3.1 DEFECTS

Origins of Defects

- Defects have determined effects 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 beings injected in the software during the phase of the software lifecycle.

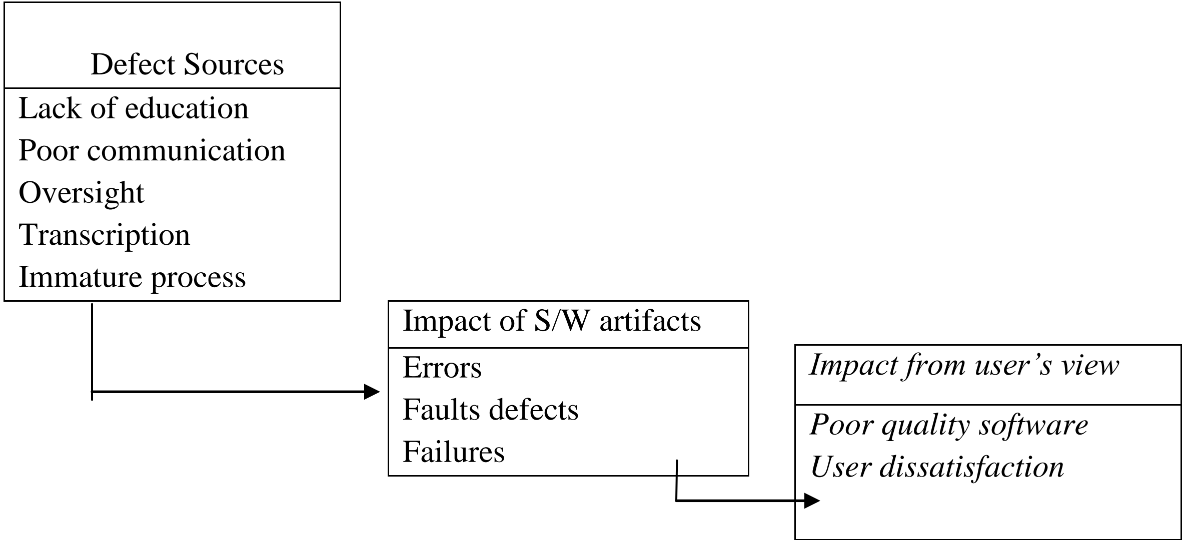


Fig: Origins of Defects

- Tester as doctors need to have knowledge about possible defects in order to develop defect hypotheses, they use the hypotheses to;
  - Design test cases
  - Design test procedure
  - Assemble test sets
  - Select the testing levels appropriate for the tests
  - Evaluate the results of the test.
- A successful testing experiment will prove the hypothesis is true that is, the hypothesized defect was present. Then the software can be repaired.

3.2 DEFECT CLASSES, THE DEFECT REPOSITORY, AND TEST DESIGN

- Defect can be classified in many ways. It is important for an organization to adapt a single classification scheme and apply it to 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.
- Execution-based testing strategies should be selected that have the strongest possibility of defecting particular types of defects.

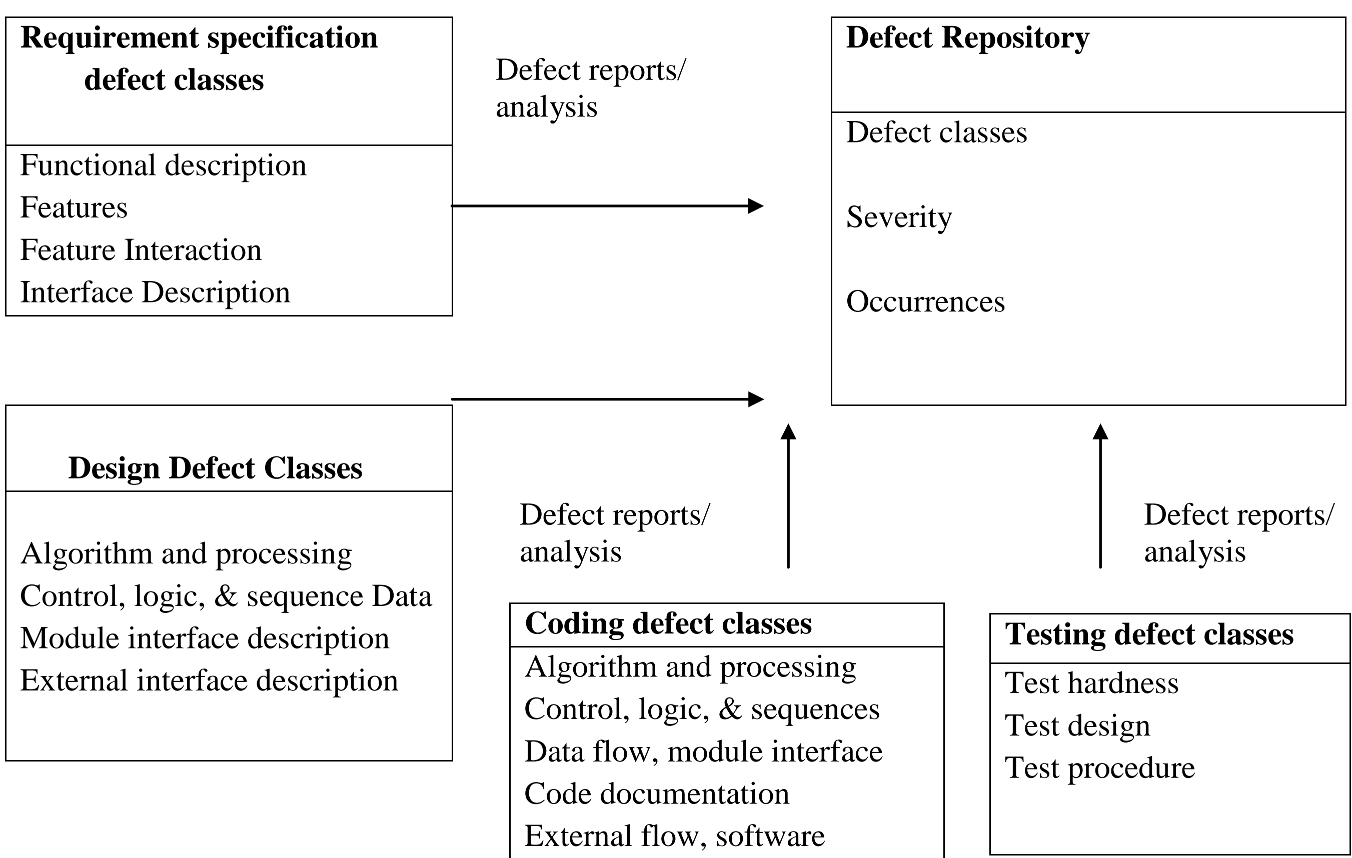


Fig: Defect Classes and Defect Repository

Defect classes

Defect classes are classified into four types namely

1. requirement/specification defect class
2. design defect class
3. coding defect class
4. testing defect class

1. Requirement/Specification Defect Class

Some requirement defects are

1. **functional description defects-** the overall description of what the product does, and how it should behave is incorrect, ambiguous, and/or incomplete
2. **Feature defects-** distinguishing characteristics of a software component or system.
3. Failure interaction defects- these are due to an incorrect description of how the features should interact.
4. **Interface description defects-** these occur in the description of how the target software is to interface with external software, hardware and users.

2. Design defects

Some design defects are

1. **Algorithms and processing defects-** these occur when the processing steps in the algorithm as described by the pseudo code are incorrect.
2. **Control logic and sequence defects-** Control defects occur when logic flow in the pseudo code is not collect.
3. **Data Defects-** These are associated with in collect design of data structures.
4. **Module Interface Description Defects-** This include in correct, missing, and /or inconsistent defects of parameter types.
5. **Functional Description Defects-** This includes incorrect missing, and/ or unclear defects of design elements. These defects are best defected during a design review.
6. **External Interface Description defects-** these are derived four incorrect design description for inter faces with COTS components, external software systems, databases, and hardware devices.

3. Coding Defects

1. **Algorithmic and processing Defects-** Adding levels of programming detail to design, code-related algorithmic and processing defects now include unchecked overflow and underflow conditions , comparing inappropriate data types, converting one data type to another, incorrect ordering of arithmetic operators , misuse or omission of parenthesis , precision loss an incorrect use of signs.
2. **Control logic and sequence Defects-** On the coding level these would include incorrect expression of case statements incorrect iterations of loops.
3. **Typographical Defects-** These are syntax errors.
4. **Initialization 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 of the programming environment.

5. **Data Flow Defects-** These are certain reasonable operational sequence that data should flow through.
6. **Data Defects-** These are indicated by incorrect implementation of data structures.
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 type an incorrect number of parameters.
8. **Code Documentation Defects** – When the documentation does not reflect what the programs actually does, or is in complete or ambiguous, this is called a code documentation defect.
9. **External hardware, software interface defects** – These defects arise from problems related to system called links to database, input/output sequence , memory usage , interrupts and exception handling , data exchange with hardware , protocols , formats, interfaces with build files , and fixing sequences.

4. Testing Defects

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

1. **Test hardness Defects**
2. **Test Case Design and Test Procedure Defects-** These would encompass incorrect, incomplete, missing, inappropriate test cases and test procedures.

3.3 DEVELOPER/TESTER SUPPORT FOR DEVELOPING A DEFECT REPOSITORY

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. The defect data is useful for test planning, a TMM level 2 maturity goals.

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. The defect data can support debugging activities as well.

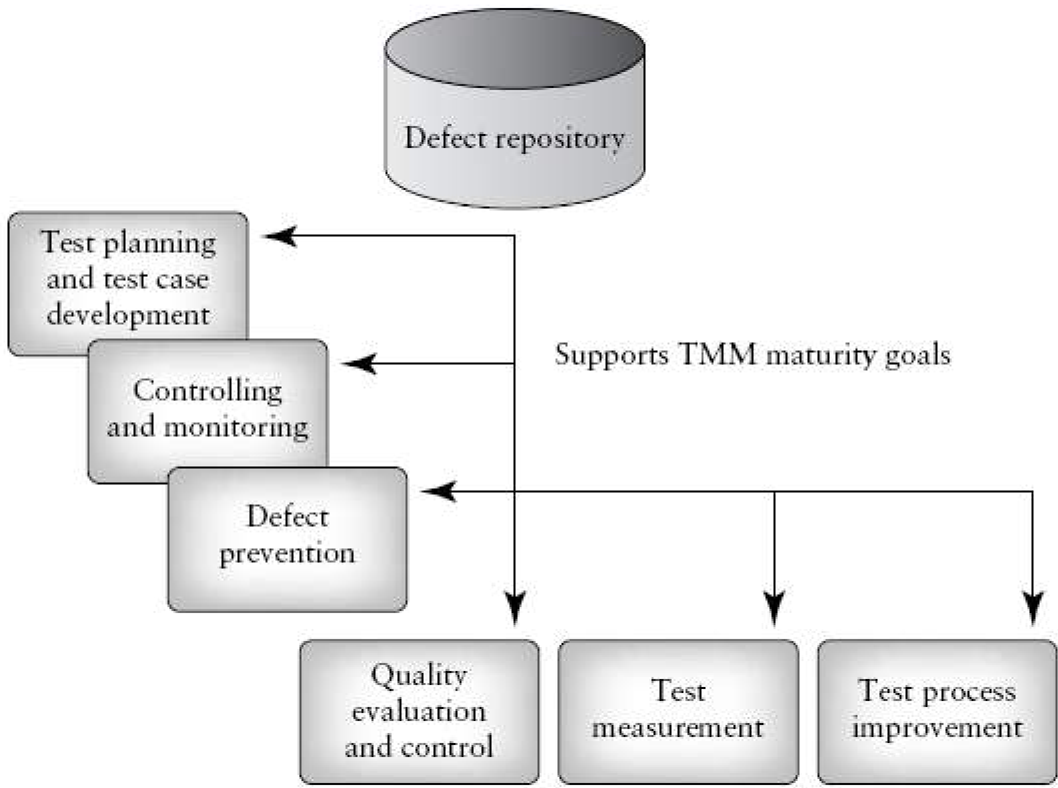


Figure: Defect Repository

A defect repository can help to support achievement and continuous implementation of several TMM maturity goals including con trolling and monitoring of test, software quality evaluation and control, test measurement, and test process improvement.

CHAPTER 4
TEST CASE DESIGN

4.1 INTRODUCTION

- The Testing Maturity Model provides some answers to these questions. It can serve as a learning tool, or framework, to learn about testing. Support for this usage of the TMM lies in its structure. It introduces both the technical and managerial aspects of testing in a manner that allows for a natural evolution of the testing process, both on the personal and organizational levels.

- In this chapter we begin the study of testing concepts using theTMM as a learning framework. We begin the development of testing skills necessary to support achievement of the maturity goals at levels 2–3 of the Testing Maturity Model. TMM level 2 has three maturity goals, two of which are managerial in nature.
- Note that this goal is introduced at a low level of the TMM, indicating its importance as a basic building block upon which additional testing strengths can be built. In order to satisfy this maturity goal test specialists in an organization need to acquire technical knowledge basic to testing and apply it to organizational projects.

4.2 THE SMART TESTER

The smart tester is to understand the functionality, input/output domain and the environment for use of the code being tested. For certain types of testing the user must also understand in detail how the code is constructed.

Novice Tester

Novice testers, taking their responsibility seriously, might try to get test a module or component using all possible inputs and exercise all possible software structures. Using this approach, they reason, will enable them to detect all defects

Roles of a Smart Tester

- Reveal defects
- Can be used to evaluate software performance, usability & reliability.
- Understand the functionality, input/output domain and the environment for use of the code being tested

4.3 TEST CASE DESIGN STRATEGIES AND TECHNIQUES

Test Strategies	Tester's View	Knowledge sources	Techniques / Methods

Black-box testing (not code-based) (sometimes called functional testing)	Inputs Outputs	Requirements document Specifications User manual Models Domain knowledge Defect analysis data Intuition Experience	Equivalence class partitioning Boundary value analysis Cause effect graphing Error guessing Random testing State-transition testing Scenario-based testing
White-box testing (also called code-based or structural testing)		Program code Control flow graphs Data flow graphs Cyclomatic complexity High-level design Detailed design	Control flow testing/coverage: - Statement coverage - Branch (or decision) coverage - Condition coverage - Branch and condition coverage - Modified condition/ decision coverage - Multiple condition coverage - Independent path coverage - Path coverage Data flow testing/ coverage Class testing/coverage Mutation testing

Figure: Two basic Testing Strategies

4.4 USING THE BLACK BOX APPROACH TO TEST CASE DESIGN

- Given the black box test strategy where we are considering only inputs and outputs as a basis for designing test cases. How do we choose a suitable set of inputs from the set of all possible valid and invalid inputs?
- Keep in mind that infinite time and resources are not available to exhaustively test all possible inputs. This is prohibitively expensive even if the target software is a simple software unit. The goal for the smart tester is to effectively use the resources available by developing a set of test cases that gives the maximum yield of defects for the time and effort spent.

- To help achieve this goal using the black box approach we can select from **several methods.** Very often combinations of the methods are used to detect different types of defects. Some methods have greater practicality than others.
    - **Random Testing**
    - **Equivalence Class Partitioning**
    - **Boundary Value Analysis**
    - **Other black box test design approaches**
        - **Cause-and-Effect Graphing**
        - **State Transition Testing**
        - **Error Guessing**

CHAPTER 5
BLACK BOX TEST CASE DESIGN TECHNIQUES

5.1 RANDOM TESTING

Each software module or system has an input domain from which test input data is selected. If a tester randomly selects inputs from the domain, this is called random testing. For example, if the valid input domain for a module is all positive integers between 1 and 100, the tester using this approach would randomly, or unsystematically, select values from within that domain; for example, the values 55, 24, 3 might be chosen.

Issues in Random Testing:

- Are the three values adequate to show that the module meets its specification when the tests are run?
- Should additional or fewer values be used to make the most effective use of resources?
- Are there any input values, other than those selected, more likely to reveal defects? For example, should positive integers at the beginning or end of the domain be specifically selected as inputs?
- Should any values outside the valid domain be used as test inputs? For example, should test data include floating point values, negative values, or integer values greater than 100?

More structured approaches to black box test design address these issues.

Use of random test inputs may save some of the time and effort that more thoughtful test input selection methods require. However, the reader should keep in mind that according to many testing experts, selecting test inputs randomly has very little chance of producing an effective set of test data.

5.2 EQUIVALENCE CLASS PARTITIONING

If a tester is viewing the software-under-test as a black box with well-defined inputs and outputs, a good approach to selecting test inputs is to use a method called equivalence class partitioning. Equivalence class partitioning results in a **partitioning of the input domain of the software-under- test.**

The technique can also be used to partition the output domain, but this is not a common usage. The finite numbers of partitions or equivalence classes that result allow the tester to select a given member of an equivalence class as a representative of that class. Using equivalence class partitioning a test value in a particular class is equivalent to a test value of any other member of that class. Therefore, if one test case in a particular equivalence class reveal a defect, all the other test cases based on that class would be expected to reveal the same defect.

Based on this discussion of equivalence class partitioning we can say that the partitioning of the input domain for the software-under-test using this technique has the following **advantages**:

1. It eliminates the need for exhaustive testing, which is not feasible.
2. It guides a tester in selecting a subset of test inputs with a high probability of detecting a defect.
3. It allows a tester to cover a larger domain of inputs/outputs with a smaller subset selected from an equivalence class. Most equivalence class partitioning takes place for the input domain.

How does the tester identify equivalence classes for the input domain? The tester uses the conditions to partition the input domain into equivalence classes and then develops a set of tests cases to cover (include) all the classes. Given that only the information in an input/output specification is needed, the tester can begin to develop black box tests for software early in the software life cycle in parallel with analysis activities.

There are **several important points** related to equivalence class partitioning:

1. The tester must consider both valid and invalid equivalence classes. Invalid classes represent erroneous or unexpected inputs.
2. Equivalence classes may also be selected for output conditions.
3. The derivation of input or outputs equivalence classes is a heuristic process. The conditions that are described in the following paragraphs only give the tester guidelines for identifying the partitions. There are no hard and fast rules. Given the same set of conditions, individual testers may make different choices of equivalence classes. As a tester gains experience he is more able to select equivalence classes with confidence.
4. In some cases it is difficult for the tester to identify equivalence classes. The conditions/boundaries that help to define classes may be absent, or obscure, or there may seem to be a very large or very small number of equivalence classes for the problem domain. These difficulties may arise from an ambiguous, contradictory, incorrect, or incomplete specification and/or requirements description. It is the duty of the tester to seek out the analysts and meet with them to clarify these documents.

Myers suggests the following conditions as guidelines for selecting input equivalence classes. Test cases, when developed, may cover multiple conditions and multiple variables.

List of conditions

1. ‘‘If an input condition for the software-under-test is specified as a range of values, select one valid equivalence class that covers the allowed range and two invalid equivalence classes, one outside each end of the range.’’
   For example, suppose the specification for a module says that an input, the length of a widget in millimeters, lies in the range 1–499; then select one valid equivalence class that includes all values from 1 to 499. Select a second equivalence class that consists of all values less than 1, and a third equivalence class that consists of all values greater than 499.

2. ‘‘If an input condition for the software-under-test is specified as a number of values, then select one valid equivalence class that includes the allowed number of values and two invalid equivalence classes that are outside each end of the allowed number.’’

3. ‘‘If an input condition for the software-under-test is specified as a set of valid input values, then select one valid equivalence class that contains all the members of the set and one invalid equivalence class for any value outside the set.’’
   For example, if the specification for a paint module states that the colors RED, BLUE, GREEN and YELLOW are allowed as inputs, then select one valid equivalence class that includes the set RED, BLUE, GREEN and YELLOW, and one invalid equivalence class for all other inputs.
4. ‘‘If an input condition for the software-under-test is specified as a “must be” condition, select one valid equivalence class to represent the “must be” condition and one invalid class that does not include the “must be” condition.’’
   For example, if the specification for a module states that the first character of a part identifier must be a letter, then select one valid equivalence class where the first character is a letter, and one invalid class where the first character is not a letter.

5. ‘‘If the input specification or any other information leads to the belief that an element in an equivalence class is not handled in an identical way by the software-under-test, then the class should be further partitioned into smaller equivalence classes.’’

5.3 BOUNDARY VALUE ANALYSIS

The test cases developed based on equivalence class partitioning can be strengthened by use of an another technique called boundary value analysis. With experience, testers soon realize that many defects occur directly on, and above and below, the edges of equivalence classes. Test cases that consider these boundaries on both the input and output spaces as shown in following Figure are often valuable in revealing defects.

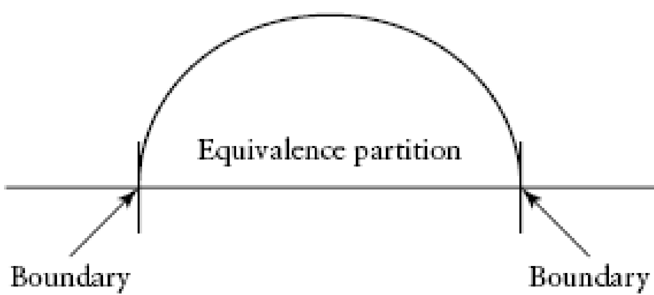


FIG. Boundaries of an equivalence partition

Whereas equivalence class partitioning directs the tester to select test cases from any element of an equivalence class, boundary value analysis requires that the tester select elements close to the edges, so that both the upper and lower edges of an equivalence class are covered by test cases.

The rules-of-thumb described below are useful for getting started with boundary value analysis.

1. If an input condition for the software-under-test is specified as a range of values, develop valid test cases for the ends of the range, and invalid test cases for possibilities just above and below the ends of the range.

For example if a specification states that an input value for a module must lie in the range between -1.0 and +1.0, valid tests that include values for ends of the range, as well as invalid test cases for values just above and below the ends, should be included. This would result in input values of -1.0, -1.1, and 1.0, 1.1.

2. If an input condition for the software-under-test is specified as a number of values, develop valid test cases for the minimum and maximum numbers as well as invalid test cases that include one lesser and one greater than the maximum and minimum.

For example, for the real-estate module mentioned previously that specified a house can have one to four owners, tests that include 0,1 owners and 4,5 owners would be developed.

The following is an example of applying boundary value analysis to output equivalence classes. Suppose a table of 1 to 100 values is to be produced by a module. The tester should select input data to generate an output table of size 0,1, and 100 values, and if possible 101 values.

3. If the input or output of the software-under-test is an ordered set, such as a table or a linear list, develop tests that focus on the first and last elements of the set.

It is important for the tester to keep in mind that equivalence class partitioning and boundary value analysis apply to testing both inputs and outputs of the software-under-test, and, most

importantly, conditions are not combined for equivalence class partitioning or boundary value analysis. Each condition is considered separately, and test cases are developed to insure coverage of all the individual conditions.

5.4 CAUSE - AND – EFFECT GRAPHING

A major weakness with equivalence class partitioning is that it **does not allow testers to combine conditions.** Combinations can be covered in some cases by test cases generated from the classes. Cause-and-effect graphing is a technique that can be used to **combine conditions and derive an effective set of test cases** that may disclose inconsistencies in a specification.

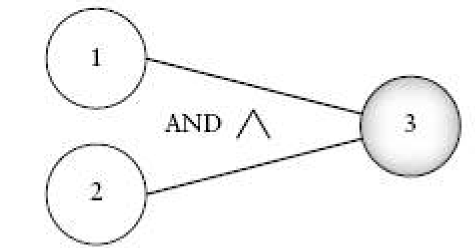
However, the **specification must be transformed into a graph** that resembles a digital logic circuit. The graph must be converted to a **decision table** that the tester uses to **develop test cases**. Tools are available for the latter process and allow the derivation of test cases to be more practical using this approach.

The steps in developing test cases with a cause-and-effect graph are as follows:

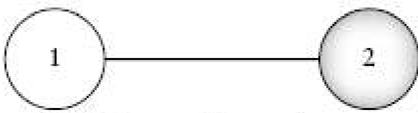
1. The tester must decompose the specification of a complex software component into lower-level units.
2. For each specification unit, the tester needs to identify causes and their effects. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation. Putting together a table of causes and effects helps the tester to record the necessary details. The logical relationships between the causes and effects should be determined. It is useful to express these in the form of a set of rules.
3. From the cause-and-effect information, a Boolean cause-and-effect graph is created. Nodes in the graph are causes and effects. Causes are placed on the left side of the graph and effects on the right. Logical relationships are expressed using standard logical operators such as AND, OR, and NOT, and are associated with arcs. An example of the notation is shown in the following Figure. Myers shows additional examples of graph notations.
4. The graph may be annotated with constraints that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints.
5. The graph is then converted to a decision table.
6. The columns in the decision table are transformed into test cases.

Example

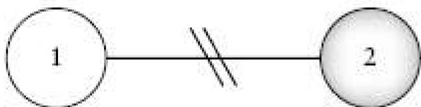
The following example illustrates the application of this technique. Suppose we have a specification for a module that allows a user to perform a search for a character in an existing string. The specification states that the user must input the length of the string and the character to search for.



Effect 3 occurs if both causes 1 and 2 are present.



Effect 2 occurs if cause 1 occurs.



Effect 2 occurs if cause 1 does not occur.

FIG. Samples of cause-and-effect graph notations

If the string length is out-of-range an error message will appear. If the character appears in the string, its position will be reported. If the character is not in the string the message "not found" will be output.

The input conditions, or causes are as follows:
C1: Positive integer from 1 to 80
C2: Character to search for is in string

The output conditions, or effects are:
E1: Integer out of range
E2: Position of character in string
E3: Character not found

The rules or relationships can be described as follows:
If C1 and C2, then E2.
If C1 and not C2, then E3.
If not C1, then E1.

Based on the causes, effects, and their relationships, a cause-and-effect graph to represent this information is shown in the following Figure.

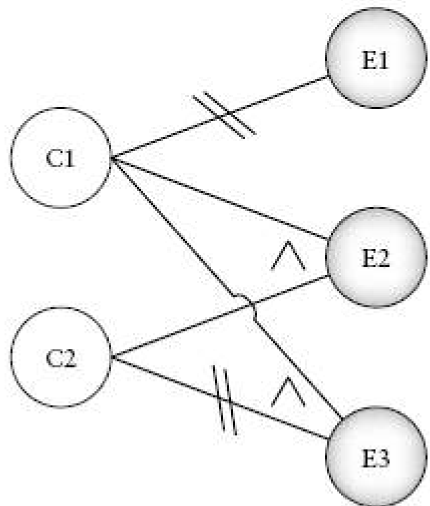


FIG. Cause-and-effect graph for the character search example

The next step is to develop a decision table. The decision table reflects the rules and the graph and shows the effects for all possible combinations of causes. Columns list each combination of causes, and each column represents a test case. Given n causes this could lead to a decision table with 2n entries, thus indicating a possible need for many test cases.

A decision table will have a row for each cause and each effect. The entries are a reflection of the rules and the entities in the cause and effect graph. Entries in the table can be represented by a “1” for a cause or effect that is present, a “0” represents the absence of a cause or effect, and a “—” indicates a “don’t care” value. A decision table for our simple example is shown in Table 4.3 where C1, C2, C3 represent the causes, E1, E2, E3 the effects, and columns T1, T2, T3 the test cases.

The tester can use the decision table to consider combinations of inputs to generate the actual tests. In this example, three test cases are called for. If the existing string is “abcde,” then possible tests are the following:

Inputs	Length	Character to search for	Outputs
T1	5	c	3
T2	5	w	Not found
T3	90		Integer out of range

The major problem is developing a graph and decision table when there are many causes and effects to consider. A possible solution to this is to decompose a complex specification into lower-level, simpler components and develop cause-and-effect graphs and decision tables for these.

TABLE- Decision table for character search example

	T1	T2	T3
C1	1	1	0
C2	1	0	—
E1	0	0	1
E2	1	0	0
E3	0	1	0

5.5 BLACK BOX TESTING AND COMMERCIAL OFF-THE-SHELF (COTS) COMPONENTS

- As software development evolves into an engineering discipline, the reuse of software components will play an increasingly important role. Reuse of components means that developers need not reinvent the wheel; instead they can reuse an existing software component with the required functionality.
- The reusable component may come from a code reuse library within their organization or, as is most likely, from an outside vendor who specializes in the development of specific types of software components.
- Components produced by vendor organizations are known as commercial off-the-shelf, or **COTS**, components. Using COTS components can save time and money. However, the COTS component must be evaluated before becoming a part of a developing system. This means that the **functionality, correctness, and reliability** of the component must be established. In addition, its suitability for the application must be determined, and any unwanted functionality must be identified and addressed by the developers.
- When a COTS component is purchased from a vendor it is basically a **black box**. It can range in size from a few lines of code, for example, a device driver, to thousands of lines of code, as in a telecommunication subsystem. If the COTS component is small in size, and a specification of its inputs/outputs and functionality is available, then equivalence class partitioning and boundary value analysis may be useful for detecting defects and establishing component behavior. The tester should also use this approach for identifying any unwanted or unexpected functionality or side effects.

CHAPTER 6

WHITE BOX TEST CASE DESIGN TECHNIQUES

6.1 DIFFERENT METHODS USED IN THE WHITE BOX TESTING STRATEGIES

- **Control flow testing/coverage:**
  - Statement coverage - Branch (or decision) coverage
  -Condition coverage - Branch and condition coverage
  - Modified condition/decision coverage - Multiple condition coverage
  - Independent path coverage - Path coverage
- **Data flow testing/coverage**
- **Class testing/coverage**
- **Mutation testing**

6.2 TEST ADEQUACY CRITERIA

- The goal for white box testing is to ensure that the internal components of a program are working properly. A common focus is on structural elements such as statements and branches. The tester develops test cases that exercise these structural elements to determine if defects exist in the program structure.
- Testers need a framework for deciding which structural elements to select as the focus of testing, for choosing the appropriate test data, and for deciding when the testing efforts are adequate enough to terminate the process with confidence that the software is working properly. Such a framework exists in the form of **test adequacy criteria**. The application **scope of adequacy criteria** also includes:
  (i) helping testers to select properties of a program to focus on during test;
  (ii) helping testers to select a test data set for a program based on the selected properties;
  (iii) supporting testers with the development of quantitative objectives for testing;
  (iv) indicating to testers whether or not testing can be stopped for that program. For example, an adequacy criterion that focuses on statement/branch properties is expressed as the following:
  A test data set is statement, or branch, adequate if a test set T for program P causes all the statements, or branches, to be executed respectively.

6.3 DECISION (OR) BRANCH COVERAGE

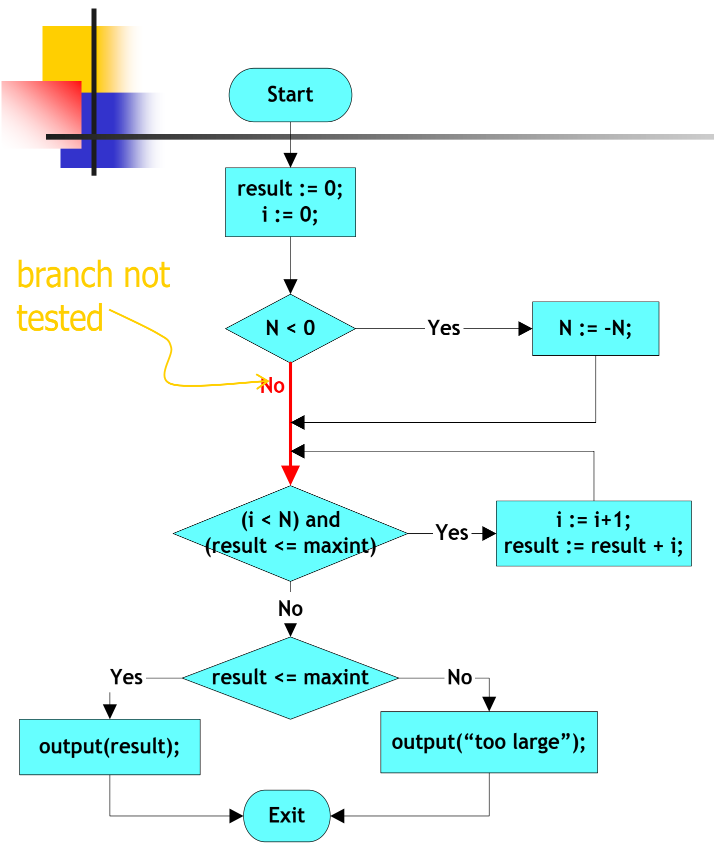
- Execute every branch of a program :
  each possible outcome of each decision occurs at least once
- Example:
  - simple decision: IF b THEN s1 ELSE s2
  - multipledecision:
    CASE x OF
    1 : ….
    2 : ….

3 : ….

- Stronger than statement coverage
  - IF THEN without ELSE – if the condition is always true all the statements are executed, but branch coverage is not achieved

Example: decision (or branch) coverage

Example : decision (or branch) coverage



Tests for complete statement (node) coverage:

inputs		outputs
maxint	N	result
10	-1	1
0	-1	too large

are not sufficient for decision (branch) coverage!

Take:

inputs		outputs
maxint	N	result
10	3	6
0	-1	too large

for complete decision (branch) coverage

16

6.4 DATA FLOW TESTING

- Add data flow information to the control flow graph
  - statements that write variables (a value is assigned or changed)
  - statements that read variables
- Generate test cases that exercise all write-read pairs of statements for each variable
- Several variants of this technique

Data flow testing : Example

1 PROGRAM sum (maxint, N : INT)

write-read pairs for variable result:

write	read
2	5
2	7 (?)
2	

```
2      INT    result := 0 ;   i := 0 ;
3      IF   N < 0
4      THEN   N  :=  - N ;
5      WHILE   ( i < N )  AND  ( result <= maxint )
6      DO i  :=  i + 1 ;
7             result  :=  result + i ;
8      OD;
9      IF   result <= maxint
10     THEN   OUTPUT ( result )
11     ELSE   OUTPUT ( “too large” )
12    END.
```

6.5 CONTROL FLOW TESTING/COVERAGE

Logic Elements Considered For Coverage And Control Flow Graph In White Box Test Design

- Program Statements
- Decision/Branch
- Conditions
- Combination of Decisions & Conditions
- Paths

Control flow analysis

```
1     PROGRAM  sum ( maxint, N : INT )
2       INT    result := 0 ;   i := 0 ;
3       IF   N < 0
4       THEN   N  :=  - N ;
5       WHILE   ( i < N )  AND  ( result <= maxint )
6       DO    i  :=  i + 1 ;
7                result  :=  result + i ;
8       OD;
9       IF   result <= maxint
10      THEN   OUTPUT ( result )
11      ELSE   OUTPUT ( “too large” )
12    END.
```

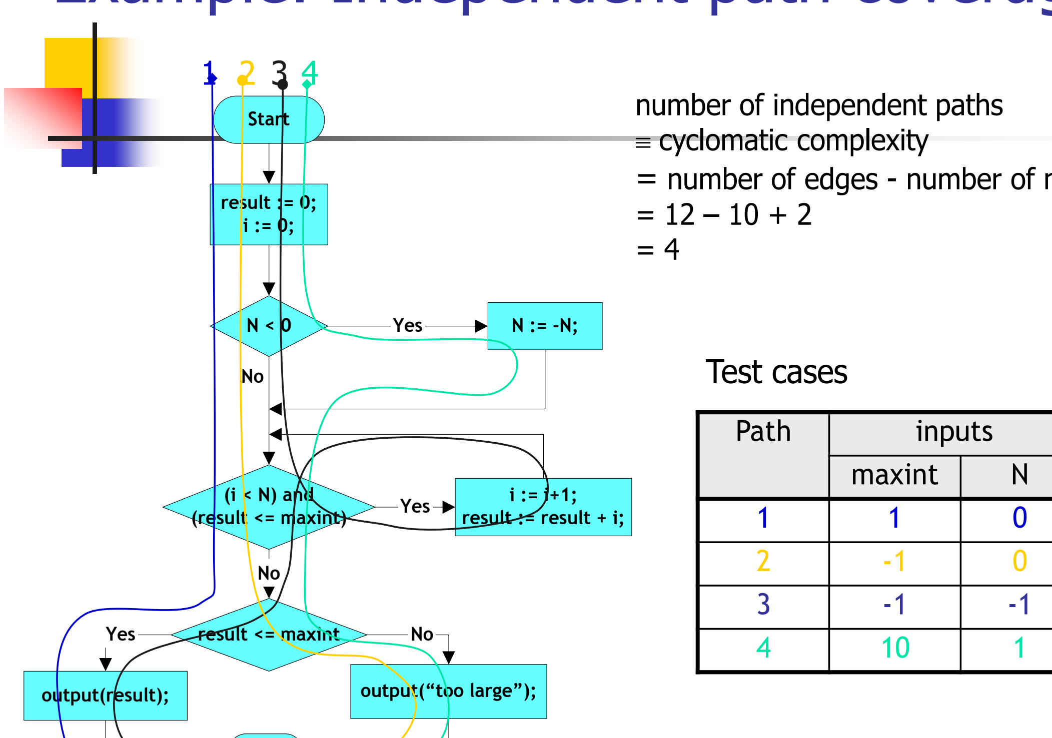
Start
result := 0;
i := 0;
N < 0
Yes
N := -N;
No
(i < N) and
(result <= maxint)
Yes
i := i+1;
result := result + i;
No
result <= maxint
Yes
No
output(result);
output(“too large”);
Exit

6.6 MCCABE’S CYCLOMATIC COMPLEXITY
(OR) INDEPENDENT PATH (OR)BASIS PATH COVERAGE

- Obtain a maximal set of linearly independent paths (also called a basis of independent paths)

  - If each path is represented as a vector with the number of times that each edge of the control flow graph is traversed, the paths are linearly independent if it is not possible to express one of them as a linear combination of the others
- Generate a test case for each independent path
- The number of linearly independent paths is given by the **McCabe's *cyclomatic complexity*** of the program
  - **Number of edges - Number of nodes** + **2** in the control flow graph
  - Measures the structural complexity of the program
- Problem: some paths may be impossible to execute
- Also called structured testing (see McCabe for details)
- McCabe's argument: this approach produces a number of test cases that is proportional to the complexity of the program (as measured by the cyclomatic complexity), which, in turn, is related to the number of defects expected.

Example: Independent path coverage



number of independent paths
≡ cyclomatic complexity
= number of edges - number of nodes + 2
= 12 – 10 + 2
= 4

Test cases

Path	inputs		outputs
	maxint	N	result
1	1	0	0
2	-1	0	too large
3	-1	-1	too large
4	10	1	1

25

Path coverage

- Execute every possible path of a program, i.e., every possible sequence of statements
- Strongest white-box criterion (based on control flow analysis)
- Usually impossible: infinitely many paths (in case of loops)

- So: not a realistic option
- But note : enormous reduction w.r.t. all possible test cases(each sequence of statements executed for only one value) (doesn't mean exhaustive testing)

6.7 MUTATION TESTING

- Starts with a code component and its associated test cases (in a state such that the code passes all test cases)
- The original code component is modified in a simple way (replace operators, constants, etc.) to provide a set of similar components that are called **mutants,** based on typical errors
- The original test cases are run with each mutant
- If a mutant passes all the test cases (the mutant is not "killed"), then either the mutant is equivalent to the original code (and is ignored), or it is not equivalent, in which case additional test cases should be developed
- The rate of mutants "killed" (after removing mutants that are equivalent to the original code) gives an indication of the rate of undetected defects that may exist in the original code.

  To measure the mutation adequacy of a test set T for a program P we can use what is called a

  mutation score (MS), which is calculated as follows:

  MS (P,T) = # of dead mutants/ (# total mutants - # of equivalent mutants)

- Additional test cases can be defined in order to kill all mutants

6.8 EVALUATE TEST SELECTION/ADEQUACY/COVERAGE/STOP CRITERIA

- “A selection criteria can be used for selecting the test cases or for checking whether or nor a selected test suite is adequate, that is, to decide whether or not the testing can be stopped”
- Adequacy criteria - Criteria to decide if a given test suite is adequate, i.e., to give us “enough” confidence that “most” of the defects are revealed
  - In practice, reduced to coverage criteria
- Coverage criteria
  - Requirements/specification coverage
    - At least one test case for each requirement
    - Cover all statements in a formal specification
  - Model coverage
    - State-transition coverage, Use-case and scenario coverage
  - Code coverage
    - Statement coverage, Data flow coverage, ...
  - Fault coverage

Support for evaluating **test adequacy criteria** comes from a theoretical treatment developed by **Weyuker**. She presents a set of axioms that allow testers to formalize properties which should be satisfied by any good program-based test data adequacy criterion.

Testers can use the axioms to

- recognize both strong and weak adequacy criteria; a tester may decide to use a weak criterion, but should be aware of its weakness with respect to the properties described by the axioms; focus attention on the properties that an effective test data adequacy criterion should exhibit;
- select an appropriate criterion for the item under test;
- stimulate thought for the development of new criteria; **the axioms are the framework** with which to evaluate these new criteria.

The axioms are based on the following set of assumptions:

(i) programs are written in a structured programming language;
(ii) programs are SESE (single entry/single exit);
(iii) all input statements appear at the beginning of the program;
(iv) all output statements appear at the end of the program.

The axioms/properties described by Weyuker are the following:

1 . Applicability Property
2 . Nonexhaustive Applicability Property
3 . Monotonicity Property
4 . Inadequate Empty Set
5 . Antiextensionality Property
6 . General Multiple Change Property
7 . Antidecomposition Property
8 . Anticomposition Property
9 . Renaming Property
10. Complexity Property
11. Statement Coverage Property

CHAPTER 7
LEVELS OF TESTING – UNIT TESTING

7.1 INTRODUCTION

Different Levels of Software Testing

- Unit testing
- Integration testing
- System testing
- Acceptance testing

The Need for Level of Software Testing

- Unit test - Individual component.
- Integration test - component groups.
- System test - system as a whole.

- Acceptance test - system as a whole with customer requirements.

The Task Required for Preparing Unit Test by the Developer/Tester
To prepare for unit test by the developer/tester must perform several tasks. They are

- Plan the general approach to unit testing.
- Design the test cases, and test procedures.
- Define the relationship between the tests.
- Prepare the support code necessary for unit test.

The Tasks Required for Planning of a Unit Test

- Describe unit test approach and risks.
- Identify unit features to be tested.
- Add levels of detail to the plan.

The Components Suitable for Conduct the Unit Test

- Procedure and function
- Class/object and manuals.
- Procedure-sized reusable component.

7.2 UNIT TESTING

Functions, Procedures, Classes, and Methods as Units

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:

- performs 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.

Some components suitable for unit test

Procedures and functions

Classes/objects and methods

Procedure-sized reusable components (Small-sized COTS com- ponents or components from an in-house reuse library)

Fig. Some components suitable for unit test.

The principal goal for unit testing

The principal goal for unit testing is insure that each individual software unit is functioning according to its specification. Good testing practice calls for unit tests that are planned and public. Planning includes designing tests to reveal defects such as functional description defects, algorithmic defects, data defects, and control logic and sequence defects. The unit should be tested by an independent tester (someone other than the developer) and the test results and defects found should be recorded as a part of the unit history.

To prepare for unit test the **developer/tester must perform several tasks**. These are:

(i) plan the general approach to unit testing;

(ii) design the test cases, and test procedures (these will be attached to the test plan);

(iii) define relationships between the tests;

(iv) prepare the auxiliary code necessary for unit test.

7.3 UNIT TEST PLANNING

- A general unit test plan should be prepared. It may be prepared as a component of the master test plan or as a stand-alone plan.
- It should be developed in conjunction with the master test plan and the project plan for each project.
- Documents that provide inputs for the unit test plan are the project plan, as well the requirements, specification, and design documents that describe the target units.
- Components of a unit test plan are described in detail the IEEE Standard for Software Unit Testing.

A brief description of a set of development phases for unit test planning is found below. In each phase a set of activities is assigned based on those found in the **IEEE Standard for Software Unit Testing.**

Phase 1: Describe Unit Test Approach and Risks

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

(i) identifies test risks;

(ii) describes techniques to be used for designing the test cases for the units;

(iii) describes techniques to be used for data validation and recording of test results;

(iv) describes the requirements for test harnesses and other software that interfaces with the units to be tested, for example, any special objects needed for testing object-oriented units.

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 requirements, states, and state transitions, control structures, messages, and data flow patterns.

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.

As an example, existing test cases that can be reused for this project can be identified in this phase. Unit availability and integration scheduling information should be included in the revised version of the test plan. The planner must be sure to include a description of how test results will be recorded.

The next steps in unit testing consist of designing the set of test cases, developing the auxiliary code needed for testing, executing the tests, and recording and analyzing the results.

7.4 DESIGNING THE UNIT TESTS

- Part of the preparation work for unit test involves unit test design.
- It is important to specify
  (i) the test cases (including input data, and expected outputs for each test case)
  (ii) the test procedures (steps required run the tests).
- As part of the unit test design process, developers/testers should also describe the relationships between the tests.
- **Test suites** can be defined that bind 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 past projects if the organization has been careful to store them so that they are easily retrievable and reusable.
- We **design test cases for functions and procedures**. They are also useful for designing tests for the individual methods (member functions) contained in a class. This approach gives the tester the opportunity to exercise logic structures and/or data flow sequences, or to use mutation analysis, all with the goal of evaluating the structural integrity of the unit.
- In the case of a **smaller-sized COTS component selected for unit testing**, a black box test design approach may be the only option. It should be mentioned that for units that perform mission/safely/business critical functions, it is often useful and prudent to design stress, security, and performance tests at the unit level if possible.

7.5 UNIT TEST ON CLASS / OBJECTS

Unit testing on object oriented systems

- Testing levels in object oriented systems
  - operations associated with objects

        - usually not tested in isolation because of encapsulation and dimension (too small)
    - classes -> unit testing
    - clusters of cooperating objects -> integration testing
    - the complete OO system -> system testing
- Complete test coverage of a class involves
    - Testing all operations associated with an object
    - Setting and interrogating all object attributes
    - Exercising the object in all possible states
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised

Challenges/issues of Class Testing

If the class is the selected component, testers may need to address **special issues** related to the testing and retesting of these components.

Some of these issues are described follow:

- **Issue 1: Adequately Testing Classes**

  The potentially high costs for testing each individual method in a class have been described. These high costs will be particularly apparent when there are many methods in a class; the numbers can reach as high as 20 to 30. Finally, a tester might use a combination of approaches, testing some of the critical methods on an individual basis as units, and then testing the class as a whole.
- **Issue 2: Observation of Object States and State Changes**

  Methods may not return a specific value to a caller. They may instead change the state of an object. The state of an object is represented by a specific set of values for its attributes or state variables.
- **Issue 3: Encapsulation**
    - Difficult to obtain a snapshot of a class without building extra methods which display the classes' state
- **Issue 4 :Inheritance**
    - Each new context of use (subclass) requires re-testing because a method may be implemented differently (polymorphism).
    - Other unaltered methods within the subclass may use the redefined method and need to be tested
- **Issue 5:White box tests**

  Basis path, condition, data flow and loop tests can all be applied to individual methods within a class.

7.6 THE TEST HARNESS

The **auxiliary code** developed to support testing of units and components is called a test harness. The harness consists of **drivers** that call the **target code** and **stubs** that represent **modules** it calls.

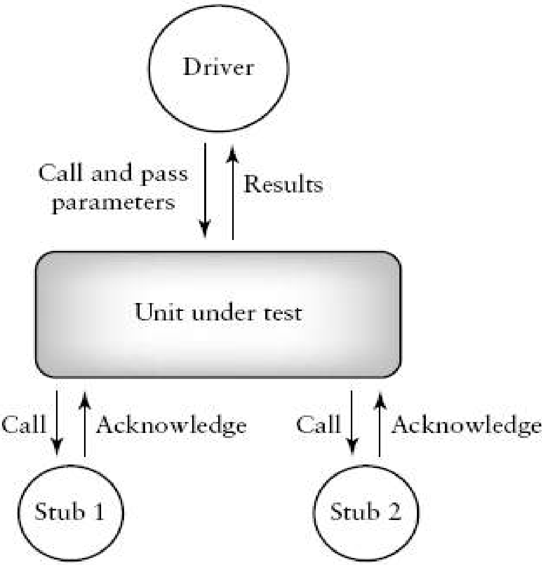


Fig. The test harness

7.7 RUNNING THE UNIT TESTS AND RECORDING RESULTS

Unit tests can begin when

(i) the units becomes available from the developers (an estimation of availability is part of the test plan),

(ii) the test cases have been designed and reviewed, and

(iii) the test harness, and any other supplemental supporting tools, are available.

The testers then proceed to run the tests and record results. Documents called **test logs** that can be used to record the results of specific tests. The status of the test efforts for a unit, and a summary of the test results, could be recorded in a simple format such as shown in Table.

It is very important for the tester at any level of testing to carefully record, review, and check 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 a test incident report Differences from expected behavior should be described in detail. This gives clues to the developers to help them
locate any faults.

TABLE- Summary work sheet for unit test results

Unit Test Worksheet

Unit Name: ____________________

Unit Identifier: ____________________

Tester: ____________________

Date: ____________________

Test case ID	Status (run/not run)	Summary of results	Pass/fail

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 (the code). Other likely causes that need to be carefully investigated by the tester are the following:

- a fault in the test case specification (the input or the output was not specified correctly);
- a fault in test procedure execution (the test should be rerun);
- a fault in the test environment (perhaps a database was not set up properly);
- a fault in the unit design (the code correctly adheres to the design specification, but the latter is incorrect).

The causes of the failure should be recorded in a test summary report, which is a summary of testing activities for all the units covered by the unit test plan.

CHAPTER 8
LEVELS OF TESTING – INTEGRATION TESTING

8.1 INTEGRATION TESTING

The Major Goals of Integration Test

Integration test for procedural code has two major goals

- To detect that occur on the interface of units.
- To assemble the individual units into working subsystems and finally a complete system that is ready for system test.

Cluster Test Plan Used In Integration Testing For OO Systems

- A **cluster** consists of classes that are related, for example, they may work together (cooperate) to support a required functionality for the complete system.
- The **clusters Test Plan** include the following items:
  - A natural languages description of the function of the cluster to be tested;
  - List of classes in the cluster;
  - clusters this cluster is dependent on;
  - A set of cluster test cases.

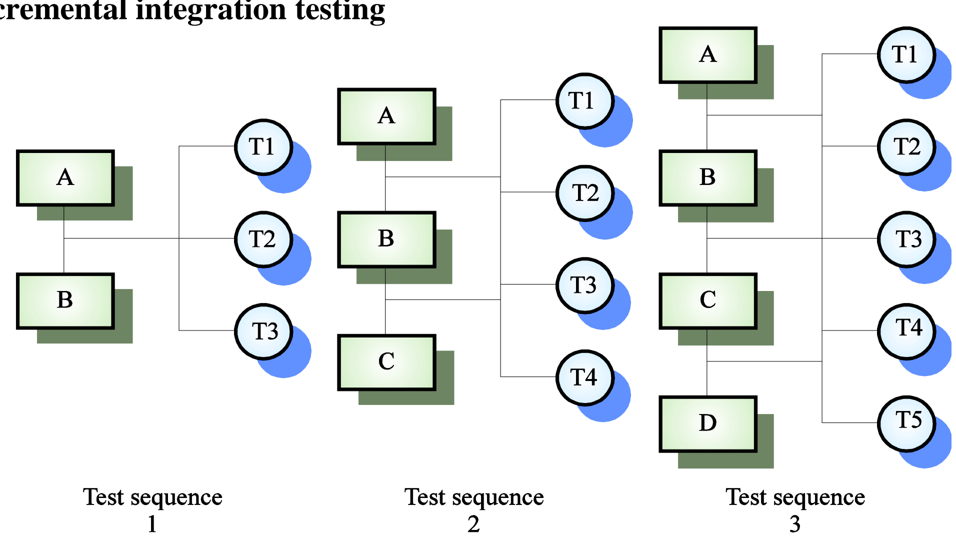
8.2 DESIGN AN INTEGRATION TEST

- Testing of groups of components integrated to create a sub-system
- Usually the responsibility of an independent testing team (except sometimes in small projects)
- Integration testing should be black-box testing with tests derived from the specification
- A principal goal is to detect defects that occur on the interfaces of units
- Main difficulty is localising errors
- ***Incremental integration*** testing (as opposed to ***big-bang integration*** testing) reduces this problem

Test drivers and stubs

- Auxiliary code developed to support testing
- Test drivers
  - call the target code
  - simulate calling units or a user
  - where test procedures and test cases are coded (for automatic test case execution) or a user interface is created (for manual test case execution)
- Test stubs
  - simulate called units
  - simulate modules/units/systems called by the target code

Incremental integration testing

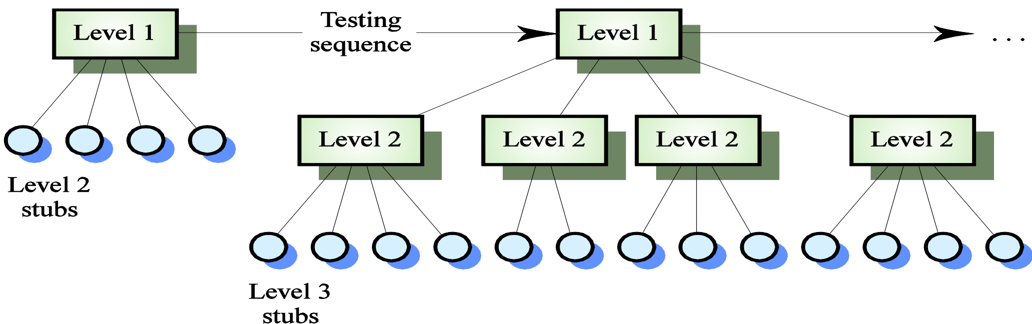


Approaches to integration testing

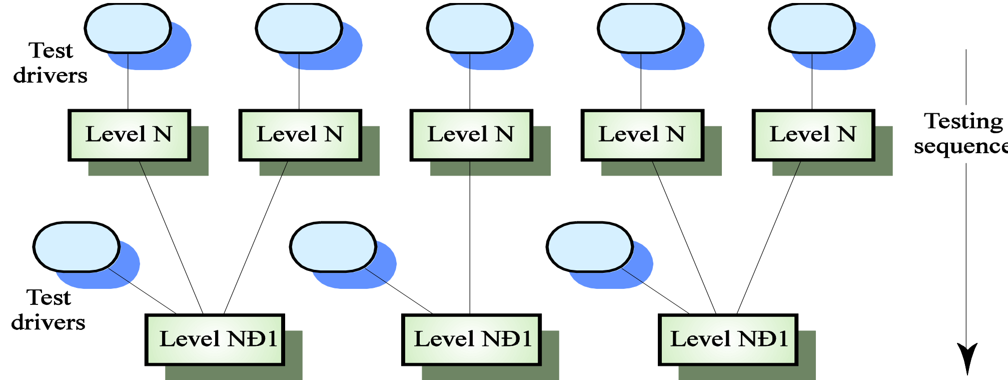
- Top-down testing
  - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- Bottom-up testing
  - Integrate individual components in levels until the complete system is created

- In practice, most integration involves a combination of these strategies
- Appropriate for systems with a hierarchical control structure
  - Usually the case in procedural-oriented systems
  - Object-oriented systems may not have such a hierarchical control structure

Top-down integration testing



Bottom-up integration testing



Advantages and disadvantages

- Architectural validation
  - Top-down integration testing is better at discovering errors in the system architecture
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development
- Test implementation
  - Often easier with bottom-up integration testing
- Test observation

– Problems with both approaches. Extra code may be required to observe tests

8.3 INTEGRATION TEST PLANING

Integration test must be planned. Planning can begin when high-level design is complete so that the system architecture is defined. Other documents relevant to integration test planning are the requirements document, the user manual, and usage scenarios. These documents contain structure charts, state charts, data dictionaries, cross-reference tables, module interface descriptions, data flow descriptions, messages and event descriptions, all necessary to plan integration tests.

For readers integrating **object-oriented systems** Murphy et al. has a detailed description of a **Cluster Test Plan**. The plan includes the following items:
(i) clusters this cluster is dependent on; (ii) a natural language description of the functionality of the cluster to be tested; (iii) list of classes in the cluster; (iv) a set of cluster test cases.

CHAPTER 9
LEVELS OF TESTING – SYSTEM TESTING

9.1 SEVERAL TYPES OF SYSTEM TESTS

- Functional test
- Performance test
- Stress test
- Configuration test
- Security test
- Recovery test

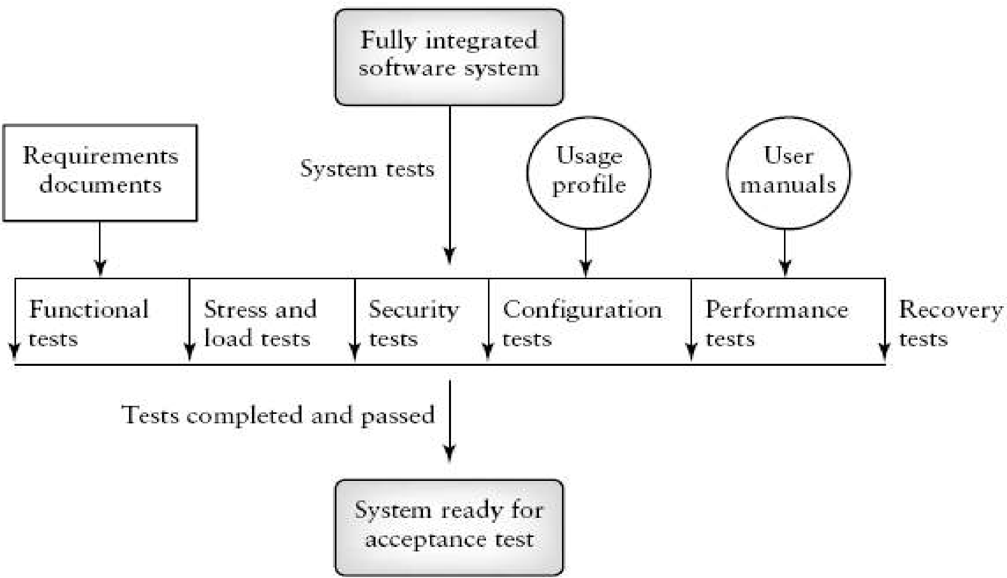


Fig. Types of system tests

9.2 FUNCTIONAL TESTING

- Ensure that the behavior of the system adheres to the requirements specification
- Black-box in nature
- Equivalence class partitioning, boundary-value analysis and state-based testing are valuable techniques
- Document and track test coverage with a (tests to requirements) traceability matrix
- A defined and documented form should be used for recording test results from functional and other system tests
- Failures should be reported in test incident reports
  - Useful for developers (together with test logs)
  - Useful for managers for progress tracking and quality assurance purposes
- The tests should focus on the following goals.
  - All types or classes of legal inputs must be accepted by the software.
  - All classes of illegal inputs must be rejected (however, the system should remain available).
  - All possible classes of system output must exercised and examined.
  - All effective system states and state transitions must be exercised and examined.
  - All functions must be exercised.

9.3 PERFORMANCE TESTING

- Goals:
  - See if the software meets the performance requirements

  - See whether there any hardware or software factors that impact on the system's performance
  - Provide valuable information to tune the system
  - Predict the system's future performance levels
- Results of performance test should be quantified, and the corresponding environmental conditions should be recorded
- Resources usually needed
  - a source of transactions to drive the experiments, typically a load generator
  - an experimental test bed that includes hardware and software the system under test interacts with
  - instrumentation of probes that help to collect the performance data (event logging, counting, sampling, memory allocation counters, etc.)
  - a set of tools to collect, store, process and interpret data from probes

9.4 CONFIGURATION TESTING

- Configuration testing allows developers/testers to evaluate system performance and availability when hardware exchanges and reconfigurations occur.
- Configuration testing also requires many resources including the multiple hardware devices used for the tests. If a system does not have specific requirements for device configuration changes then large-scale configuration testing is not essential.
- Several types of **operations** should be performed during configuration test. Some sample operations for testers are

  (i) rotate and permutate the positions of devices to ensure physical/ logical device permutations work for each device (e.g., if there are two printers A and B, exchange their positions);

  (ii) induce malfunctions in each device, to see if the system properly handles the malfunction;

  (iii) induce multiple device malfunctions to see how the system reacts. These operations will help to reveal problems (defects) relating to hardware/ software interactions when hardware exchanges, and reconfigurations occur.

The Objectives of Configuration Testing

- Show that all the configuration changing commands and menus work properly.
- Show that all the interchangeable devices are really interchangeable, and that they each enter the proper state for the specified conditions.
- Show that the systems’ performance level is maintained when devices are interchanged, or when they fail.

9.5 SECURITY TESTING

- Evaluates system characteristics that relate to the availability, integrity and confidentiality of system data and services
- Computer software and data can be compromised by
  - criminals intent on doing damage, stealing data and information, causing denial of service, invading privacy
  - errors on the part of honest developers/maintainers (and users?) who modify, destroy, or compromise data because of misinformation, misunderstandings, and/or lack of knowledge
- Both can be perpetuated by those inside and outside on an organization
- Attacks can be random or systematic. Damage can be done through various means such as:

  (i) Viruses; (ii) Trojan horses;
  (iii) Trap doors; (iv) illicit channels.
- The effects of security breaches could be extensive and can cause:

  (i) loss of information; (ii) corruption of information;
  (iii) misinformation; (iv) privacy violations;
  (v) denial of service.
- Other Areas to focus on Security Testing: **password checking, legal and illegal entry with passwords, password expiration, encryption, browsing, trap doors, viruses, …**
- Usually the responsibility of a security specialist

9.6 RECOVERY TESTING

- Subject a system to losses of resources in order to determine if it can recover properly from these losses
- Especially important for transaction systems
- Example: loss of a device during a transaction
- Tests would determine if the system could return to a well-known state, and that no transactions have been compromised
  - Systems with automated recovery are designed for this purpose

- Areas to focus [Beizer] on Recovery Testing:
  - **Restart** – the ability of the system to restart properly on the last checkpoint after a loss of a device
  - **Switchover** – the ability of the system to switch to a new processor, as a result of a command or a detection of a faulty processor by a monitor
- In each of these testing situations all transactions and processes must be carefully examined to detect:

  (i) loss of transactions;
  (ii) merging of transactions;
  (iii) incorrect transactions;
  (iv) an unnecessary duplication of a transaction.

A good way to expose such problems is to perform recovery testing under a stressful load. Transaction inaccuracies and system crashes are likely to occur with the result that defects and design flaws will be revealed.

9.7 ACCEPTANCE TEST, ALPHA AND BETA TESTING

- For tailor made software(customized software):
  - **acceptance tests** – performed by users/customers
  - much in common with system test
- For packaged software (market made software):
  - **alpha testing** – on the developers site
  - **beta testing** – on a user site

For more information: Refer Ilene Berstein book.,

CHAPTER 10
TESTING GOALS AND TEST PLANING

10.1 INTRODUCTION

This topic focuses on two fundamental maturity goals at **level 2 of the TMM:**

(i) developing organizational goals/ policies relating to testing and debugging,

(ii) test planning.

These maturity goals are managerial in nature. They are essential to support testing as a **managed process**. According to R. Thayer, a managed process is one that is **planned, monitored, directed, staffed, and organized.**

Goals/polices

Simple examples of the three types of goals mentioned are shown below.

1. ***Business goal:*** to increase market share 10% in the next 2 years in the area of financial software.
2. ***Technical goal****:* to reduce defects by 2% per year over the next 3 years.
3. ***Business/technical goal****:* to reduce hotline calls by 5% over the next 2 years.
4. ***Political goal:*** to increase the number of women and minorities in high management positions by 15% in the next 3 years.

10.2 TESTING AND DEBUGGING GOALS AND POLICIES

- A **goal** can be described as (i) a statement of intent, or (ii) a statement of a accomplishment that an individual or an organization wants to achieve.
- A **policy** can be defined as a high-level statement of principle or course of action that is used to govern a set of activities in an organization.

Testing Policy: Organization X

To ensure that our testing process is effective and that our software products meet the client's requirements we have developed and adopted the following testing policy statement.

1. Delivering software of the highest quality is our company goal. The presence of defects has a negative impact on software quality. Defects affect the correctness, reliability, and usability of a software product, thus rendering it unsatisfactory to the client. We define a testing activity as a set of tasks whose purpose is to reveal functional and quality- related defects in a software deliverable. Testing activities include traditional execution of the developing software, as well as reviews of the software deliverables produced at all stages of the life cycle. The aggregation of all testing activities performed in a systematic manner supported by organizational policies, procedures, and standards constitutes the testing process.
2. A set of testing standards must be available to all interested parties on an intraorganizational web site. The standards contain descriptions of all test-related documents, prescribed templates, and the methods, tools, and procedures to be used for testing. The standards must specify the types of projects that each of these items is to be associated with.
3. In our organization the following apply to all software development/ maintenance projects:
   - Execution-based tests must be performed at several levels such as unit, integration, system, and acceptance tests as appropriate for each software product.
   - Systematic approaches to test design must be employed that include application of both white and black box testing methods.
   - Reviews of all major product deliverables such as requirements and design documents, code, and test plans are required.
   - Testing must be planned for all projects. Plans must be developed for all levels of execution-based testing as well as for reviews of deliverables.

     - Test plan templates must be included in organizational standards documents and implemented online. A test plan for a project must be compatible with the project plan for that project. Test plans must be approved by the project manager and technical staff. Acceptance test plans must also be approved by the client.
   - Testing activities must be monitored using measurements and milestones to ensure that they are proceeding according to plan.
   - Testing activities must be integrated into the software life cycle and carried out in parallel with other development activities. The extended modified V-model as shown in the testing standards document has been adopted to support this goal.
   - Defects uncovered during each test must be classified and recorded.
   - There must be a training program to ensure that the best testing practices are employed by the testing staff.
4. Because testing is an activity that requires special training and an impartial view of the software, it must be carried out by an independent testing group. Communication lines must be established to

support cooperation between testers and developers to ensure that the software is reliable, safe, and meets client requirements.

5. Testing must be supported by tools, and, test-related measurements must be collected and used to evaluate and improve the testing process and the software product.
6. Resources must be provided for continuos test process improvement.
7. Clients/developer/tester communication is important, and clients must be involved in acceptance test planning, operational profile development, and usage testing when applicable to the project. Clients must sign off on the acceptance test plan and give approval for all changes in the acceptance test plan.
8. A permanent committee consisting of managerial and technical staff must be appointed to be responsible for distribution and maintenance of organizational test policy statements.

Debugging Policy: Organization X

Our debugging policy is founded on our quality goal to remove all defects from our software that impact on our customers' ability to use our software effectively, safely, and economically. To achieve this goal we have developed the following debugging policy statement.

1. Testing and debugging are two separate processes. Testing is the process used to detect (reveal) defects. Debugging is the process dedicated to locating the defects, repairing the code, and retesting the software. Defects are anomalies that impact on software functionality as well as on quality attributes such as performance, security, ease of use, correctness, and reliability.
2. Since debugging is a timely activity, all project schedules must allow for adequate time to make repairs, and retest the repaired software.
3. Debugging tools, and the training necessary to use the tools, must be available to developers to support debugging activities and tasks.
4. Developers/testers and SQA staff must define and document a set of defect classes and defect severity levels. These must be must be available to all interested parties on an intraorganizational web site, and applied to all projects.
5. When failures are observed during testing or in operational software they are documented. A problem, or test incident, report is completed by the developer/tester at testing time and by the users when a failure/ problem is observed in operational software. The problem report is forwarded to the development group. Both testers/developers and SQA staff must communicate and work with users to gain an understanding of the problem.
6. A fix report must be completed by the developer when the defect is repaired and code retested. Standard problem and fix report forms must be available to all interested parties on an intraorganizational web site, and applied to all projects.
7. All defects identified for each project must be cataloged according to class and severity level and stored as a part of the project history.
8. Measurement such as total number of defects, total number of defects/ KLOC, and time to repair a defect are saved for each project.

9. A permanent committee consisting of managerial and technical staff must be appointed to be responsible for distribution and maintenance of organizational debugging policy statements.

10.3 DOCUMENT TYPES (The IEEE Standard 829-1998)

TEST PLANNING:

- **Test plan**
  - prescribes the scope, approach, resources, and schedule of the testing activities
  - identifies the items to be tested, the features to be tested, the testing tasks to be performed, the personnel responsible for each task, and the risks associated with the plan

TEST SPECIFICATION/(TEST PLAN ATTACHMENTS):

- **Test design specification**
  - refines the test approach and identifies the features to be covered by the design and its associated tests
  - *identifies* the test cases and test procedures, if any, required to accomplish the testing and specifies the feature pass/fail criteria.
- **Test case specification**
  - documents the actual values used for input along with the anticipated outputs
  - also identifies constraints on the test procedures resulting from use of that specific test case (Test cases are separated from test designs to allow for use in more than one design and to allow for reuse in other situations.)
- **Test procedure specification**
  - identifies all steps required to operate the system and exercise the specified test cases in order to implement the associated test design (Test procedures are separated from test design specifications as they are intended to be followed step by step and should not have extraneous detail.)

TEST REPORTING:

- **Test item transmittal report**
  - identifies the test items being transmitted for testing in the event that separate development and test groups are involved or in the event that a formal beginning of test execution is desired

- **Test log**
  - used by the test team to record what occurred during test execution
- **Test incident report**
  - describes any event that occurs during the test execution which requires further investigation
- **Test summary report**
  - summarizes the testing activities associated with one or more test design specifications

10.4 TEST PLANING

A plan is a document that provides a framework or approach for achieving a set of goals. A **plan** can be defined in the following way.

A plan is a document that provides a framework or approach for achieving a set of goals.

In order to meet a set of goals, a plan describes what specific tasks must be accomplished, who is responsible for each task, what tools, procedures, and techniques must be used, how much time and effort is needed, and what resources are essential.

A plan also contains **milestones**.

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.

Tracking the actual occurrence of the milestone events allows a manager to determine if the project is progressing as planned. Finally, a plan should assess the risks involved in carrying out the project.

Test plans for software projects are very complex and detailed documents. The planner usually includes the following essential high-level items.

1. Overall test objectives. As testers, why are we testing, what is to be achieved by the tests, and what are the risks associated with testing this product?
2. What to test (scope of the tests). What items, features, procedures, functions, objects, clusters, and subsystems will be tested?
3. Who will test. Who are the personnel responsible for the tests?
4. How to test. What strategies, methods, hardware, software tools, and techniques are going to be applied? What test documents and deliverable should be produced?
5. When to test. What are the schedules for tests? What items need to be available?
6. When to stop testing. It is not economically feasible or practical to plan to test until all defects have been revealed.

All of the quality and testing plans should also be coordinated with the overall software project plan. A sample plan hierarchy is shown in the following Figure. At the top of the plan hierarchy there may be a software quality assurance plan. This plan gives an overview of all verification and validation activities for the project, as well as details related to other quality issues such as audits, standards, configuration control, and supplier control.

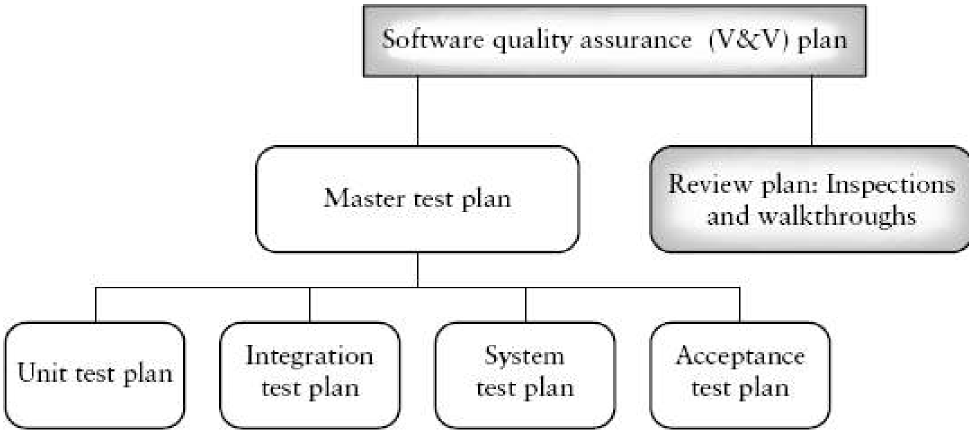


Figure: A Hierarchy of Test plans

Below that in the plan hierarchy there may be a master test plan that includes an overall description of all execution-based testing for the software system. A master verification plan for reviews inspections/walkthroughs would also fit in at this level. The master test plan itself may be a component of the overall project plan or exist as a separatedocument.

10.5 TEST PLAN COMPONENTS/CONTENTS (The IEEE Standard 829-1998)

(Applicable to master test plan and each of the level based test plans (unit, integration, etc.))

1. Test plan identifier
   - Can serve to identify it as a configuration item
2. Introduction *(why)*
   - Overall description of the project, the software system being developed or maintained, and the software items and/or features to be tested
   - Overall description of testing goals (objectives) and the testing approaches to be used
   - References to related or supporting documents
3. Test items *(what)*
   - List the items to be tested: procedures, classes, modules, libraries, components, subsystems, systems, etc.
   - Include references to documents where these items and their behaviors are described (requirements and design documents, user manuals, etc.)
   - List also items that will not be tested

4. Features to be tested *(what)*
   - Features are distinguishing characteristics (functionalities, quality attributes). They are closely related to the way we describe software in terms of its functional and quality requirements
   - Identify all software features and combinations of software features to be tested. Identify the test design specification associated with each feature and each combination of features.

5. Features not to be tested (*what*)
    - Identify all features and significant combinations of features that will not be tested and the reasons.
6. Approach *(how)*
    - Description of test activities, so that major testing tasks and task durations can be identified
    - For each feature or combination of features, the approach that will be taken to ensure that each is adequately tested
    - Tools and techniques
    - Expectations for test completeness (such as degree of code coverage for white box tests)
    - Testing constraints, such as time and budget limitations
    - Stop-test criteria
7. Item pass-fail criteria
    - Given a test item and a test case, the tester must have a set of criteria to decide whether the test has been passed or failed upon execution
    - The test plan should provide a general description of these criteria
    - Failures to a certain severity level may be accepted
8. Suspension criteria and resumption requirements
    - Specify the criteria used to suspend all or a portion of the testing activity on the test items associated with this plan
    - Specify the testing activities that must be repeated, when testing is resumed
    - Testing is done in cycles: test – fix - (resume) test (suspend) – fix - ...
    - Tests may be suspended when a certain number of critical defects has been observed
9. Test deliverables
    - Test documents (possibly a subset of the ones described in the IEEE standard)
    - Test harness (drivers, stubs, tools developed especially for this project, etc.)
10. Testing Tasks
    - Identify all test-related tasks, inter-task dependencies and special skills required
    - Work Breakdown Structure (WBS)
11. Environmental needs
    - Software and hardware needs for the testing effort
12. Responsibilities
    - Roles and responsibilities to be fulfilled
    - Actual staff involved (?)
13. Staffing and training needs
    - Description of staff and skills needed to carry out test-related responsibilities
14. Scheduling
    - Task durations and calendar
    - Milestones
    - Schedules for use of staff and other resources (tools, laboratories, etc.)
15. Risks and contingencies
    - Risks should be (i) identified, (ii) evaluated in terms of their probability of occurrence, (iii) prioritized, and (iv) contingency plans should be developed that can be activated if the risk occurs
    - Example of a risk: some test items not delivered on time to the testers

- Example of a contingency plan: flexibility in resource allocation so that testers and equipment can operate beyond normal working hours (to recover from delivery delays)

16. Testing costs (not included in the IEEE standard)
    - Kinds of costs:
        - costs of planning and designing the tests
        - costs of acquiring the hardware and software necessary
        - costs of executing the tests
        - costs of recording and analyzing test results
        - tear-down costs to restore the environment
    - Cost estimation may be based on:
        - Models (such as COCOMO for project costs) and heuristics (such as 50% of project costs)
        - Test tasks and WBS
        - Developer/tester ratio (such as 1 tester to 2 developers)
        - Test impact items (such as number of procedures) and test cost drivers (or factors, such as KLOC)
        - Expert judgment(Delphi)
17. Approvals
    - Dates and signatures of those that must approve the test plan

10.6 TEST COST ESTIMATION METHODS

COCOMO model

One approach to test cost estimation makes use of the COCOMO model in an indirect way. The test planner can use the COCOMO model to estimate total project costs, and then allocate a fraction of those costs for test. Application of the COCOMO model is based on a group of project constants that depend on the nature of the project and items known as cost drivers. A cost driver can be described as a process or product factor that has an impact on overall project costs. Project constants and cost drivers are available for overall project development efforts. To use the COCOMO model a project manager must first estimate the size of the new project and identify its type. This is facilitated by the availability of historical project data.

The simple COCOMO equation used for an initial estimate is

$$E = a\ (\text{size in KLOC})^b \qquad (1)$$

where E is estimated effort in man-months, and a and b are constants that can be determined from tables provided by Boehm or by the organization itself based on its own historical data. Selection of values from the table depend on project types.

The **intermediate COCOMO model**, used when more project details are known, incorporates project cost drivers and uses a slightly more complex set of calculations. Cost drivers for project the include:

- product attributes such as the required level of reliability;

- hardware attributes such as memory constraints;
- personnel attributes such as experience level;
- project attributes such as use of tools and methods.

The project cost drivers are rated on an ordinate scale and folded into what Boehm calls an effort adjustment factor (EAF). The results from equation (1) can be multiplied by the EAF to give a revised estimate.

Work Breakdown Structure (WBS)

An alternative test cost estimation method uses a **bottom-up, testing task– oriented** approach. This approach will work well if the testing tasks, task durations, and resources (such as hardware and software tools) for similar projects are well defined and documented in the historical database. Testing tasks can be represented in the database as a test-oriented **Work Breakdown Structure (WBS)**, which is a hierarchical representation of all test-related tasks.

High-level components of a test WBS are shown in Table 10.1. These represent high-level testing tasks. Each of these is broken down into lower-level tasks. Table 10.2 shows a breakdown for the higher-level "test planning" task. The historical record should also contain values for the time and manpower needed to perform each task in the testing WBS.

The new project is compared to those in the database in terms of size and complexity. The most similar project provides the best support for the cost estimation. Reuse of existing tests, regression tests and test harnesses should be folded into the estimating process. When tasks, and durations of the tasks have been calculated for the new project, the test planner can use the sum of the time estimated for all the tasks, adjusted to account for differences between the completed and new projects, to estimate total test time and expenses.

TABLE 10.1 Example WBS elements for testing.

1. Project startup
2. Management coordination
3. Tool selection
4. Test planning
5. Test design
6. Test development
7. Test execution
8. Test measurement, and monitoring
9. Test analysis and reporting
10. Test process improvement

TABLE 10.2 A breakdown of testing planning element from table 10.1.

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.

Delphi method

Finally, test planners can use the Delphi method for estimating test costs. This technique, which involves a group of cost estimation experts **lead by a moderator(Chairman)**, is often used to estimate the size/costs of an entire project. It can be applied to estimate test costs as well.

- The group members are given appropriate documentation relating to the project before the estimation meeting.
- The group comes together in the meeting and may have a discussion about the project and its characteristics.
- n the testing domain, test-related issues would be discussed. After the discussion each group member gives an anonymous estimate to the moderator.
- The moderator calculates an average and mean of the estimates and distributes the values to the group.
- Each group member can determine where his/her individual estimate falls with respect to the group, and reestimate based on this information and additional discussion.

- The group may have several cycles of "discussion, estimate, and analysis," until consensus on the estimate is reached.

10.7 TEST PLAN ATTACHEMENTS

Test Design Specifications (The IEEE Standard 829-1998)

One or more documents.

A test design specification describes how a group of features and/or test items is tested by a set of test cases and test procedures.

May include a (test case to) features/requirements traceability matrix

Contents:

- Test Design Specification Identifier
- Features to be tested
    - Test items and features covered by this document
- Approach refinements
    - Test techniques
- Test case identification
- Feature pass/fail criteria

Test Case Specifications (The IEEE Standard 829-1998)

Contents:

- Test case specification identifier
- Test items
    - List of items and features to be tested by this test case
- Input specifications
- Output specifications
- Environmental needs
- Special procedural requirements
- Intercase dependencies

Test Procedure Specifications (The IEEE Standard 829-1998)

A procedure in general is a sequence of steps required to carry out a specific task.

Describe steps required for executing a set of test cases or, more generally, the steps used to analyze a software item in order to evaluate a set of features.

Contents:

- Test procedure specification identifier
- Purpose
- Specific requirements
- Procedure steps
    - Log, set up, proceed, measure, shut down, restart, stop, wrap up, contingencies

10.8 LOCATING TEST ITEMS: TEST ITEM TRANSMITTAL REPORT

(**The IEEE Standard 829-1998)**
Accompanies a set of test items that are delivered for testing.
Contents

- Transmittal report identifier
- Transmitted items
  - version/revision level
  - references to the items documentation and the test plan related to the transmitted items
  - persons responsible for the items
- Location
- Status
  - deviations from documentation, from previous transmissions or from test plan
  - incident reports that are expected to be resolved
  - pending modifications to documentation
- Approvals

CHAPTER 11
TEST REPORTS

11.1 REPORTING TEST RESULTS
(The IEEE Standard 829-1998 for Software Test Documentation)

The following Figure shows the relationships between all the test-related documents we have discussed in this topic as described in the IEEE standards document. In the figure it is assumed that an overall **Master Test Plan (MTP)** is developed at first, and this is followed by more detailed test plans for the different levels of testing, unit, integration, system, acceptance, and so on.

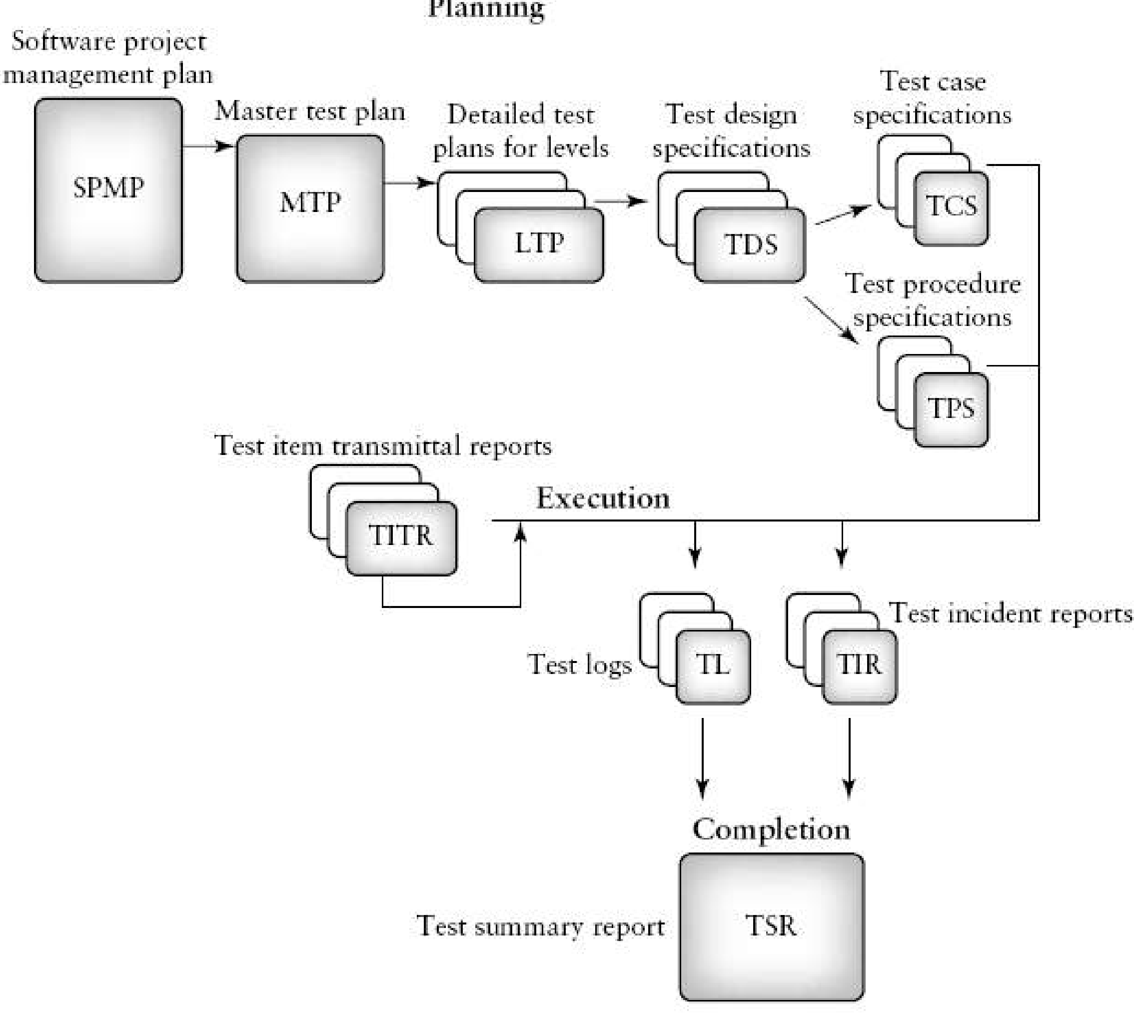


FIG. Test-related documents as recommended by IEEE

The test plan and its attachments are test-related documents that are prepared prior to test execution. There are additional documents related to testing that are prepared during and after execution of the tests.

The **IEEE Standard for Software Test Documentation** describes the following documents:

Test Log (The IEEE Standard 829-1998)

Records detailed results of test execution

Contents

- Test log identifier
- Description
  - Identify the items being tested including their version/revision levels
  - Identify the attributes of the environments in which the testing is conducted
- Activity and event entries
  - Execution description

  - Procedure results
  - Environmental information
  - Anomalous events
  - Incident report identifiers

Test Incident Report

Also called a problem report

Contents:

- Test incident report identifier
- Summary
  - Summarize the incident
  - Identify the test items involved indicating their version/revision level
  - References to the appropriate test procedure specification, test case specification, and test log
- Incident description
  - inputs, expected results, actual results, anomalies, date and time, procedure step, environment, attempts to repeat, testers, observers
  - any information useful for reproducing and repairing
- Impact
  - If known, indicate what impact this incident will have on test plans, test design specifications, test procedure specifications, or test case specifications
  - severity rating (?)

Test Summary Report (The IEEE Standard 829-1998)

Contents

Test summary report identifier

- Summary
  - Summarize the evaluation of the test items
  - Identify the items tested, indicating the environment in which the testing activities took place
- Variances
  - of the test items from their original design specifications
- Comprehensiveness assessment
  - Evaluate the comprehensiveness of the testing process against the comprehensiveness criteria specified in the test plan if the plan exists
  - Identify features or feature combinations that were not sufficiently tested and explain the reasons
- Summary of results
  - Summarize the results of testing
  - Identify all resolved incidents and summarize their resolutions
  - Identify all unresolved incidents.
- Evaluation
  - Provide an overall evaluation of each test item including its limitations
  - This evaluation shall be based upon the test results and the item level pass/fail criteria
  - An estimate of failure risk may be included

- Summary of activities
    - Summarize the major testing activities and events
    - Summarize resource consumption data, e.g., total staffing level, total machine time, and total elapsed time used for each of the major testing activities
- Approvals

11.2 THE ROLE OF THE THREE CRITICAL GROUPS IN TESTING, PLANNING AND TEST POLICY DEVELOPMENT

TMM framework **three groups** were identified as critical players in the testing process. These groups were **managers, developers/testers, and users/clients**. In TMM terminology they are called the three **critical views (CV).**

Each **group views** the testing process from a different perspective that is related to their particular goals, needs, and requirements.

- The **manager's view** involves commitment and support for those activities and tasks related to improving testing process quality.
- The **developer/tester's view** encompasses the technical activities and tasks that when applied, constitute best testing practices.
- The **user/client view** is defined as a **cooperating or supporting view**. The developers/testers work with client/user groups on quality-related activities and tasks that concern user-oriented needs. The focus is on soliciting client/user support, consensus, and participation in activities such as requirements analysis, usability testing, and acceptance test planning.

At each TMM level the three groups play specific roles in support of the maturity goals at that level. Critical group participation for all three TMM level 2 maturity goals is summarized in the following Figure:

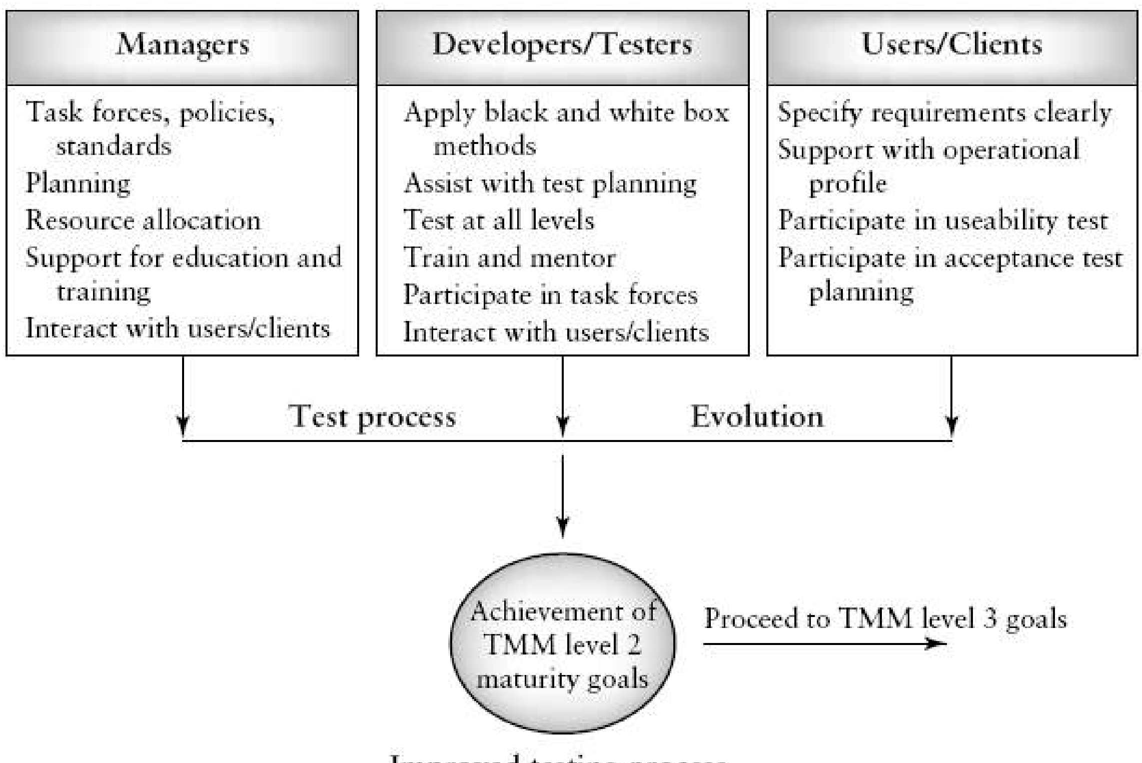


FIG. Reaching TMM level 2: summary of critical group roles

For the TMM maturity goal, "Develop Testing and Debugging Goals," the TMM recommends that **project and upper management**:

- Provide access to existing organizational goal/policy statements and sample testing policies from other sources. These serve as policy models for the testing and debugging domains.
- Provide adequate resources and funding to form the committees (team or task force) on testing and debugging. Committee makeup is managerial, with technical staff serving as co members.
- Support the recommendations and policies of the committee by:
    - distributing testing/debugging goal/policy documents to project managers, developers, and other interested staff,
    - appointing a permanent team to oversee compliance and policy change making.
- Ensure that the necessary training, education, and tools to carry out defined testing/debugging goals is made available.
- Assign responsibilities for testing and debugging.

The activities, tasks, and responsibilities for the **developers/testers** include:

- Working with management to develop testing and debugging policies and goals.
- Participating in the teams that oversee policy compliance and change management.

- Familiarizing themselves with the approved set of testing/debugging goals and policies, keeping up-to-date with revisions, and making suggestions for changes when appropriate.
- When developing test plans, setting testing goals for each project at each level of test that reflect organizational testing goals and policies.
- Carrying out testing activities that are in compliance with organizational policies.

Users and clients play an indirect role in the formation of an organization's testing goals and polices since these goals and policies reflect the organizations efforts to ensure customer/client/user satisfaction. Feedback from these groups and from the marketplace in general has an influence on the nature of organizational testing goals and policies. Successful organizations are sensitive to **customer/client/user needs**.

11.3 PROCESS AND THE ENGINEERING DISCIPLINES

The **Role of the Individual as a Process Facilitator Testing** is such a process.

- If you are a member of a TMM level 1 organization, there is a great opportunity for you become involved in process issues.
- You can initiate the implementation of a defined testing process by working with management and users/clients toward achievement of the technical and managerial-oriented maturity goals at TMM level 2.
- You can also encourage management in your organization to develop testing goals and policies, you can participate in the committees involved, and you can help to develop test planning standards that can be applied organizationwide.
- Finally, you can become proficient in, and consistently apply, black and white box testing techniques, and promote testing at the unit, integration, and system levels.

CHAPTER 12
THE TEST ORGANIZATION

12.1 INTRODUCING THE TEST SPECIALIST

When an organization has reached TMM level 2 it has accomplished a great deal. Fundamental testing maturity goals have been achieved. There are testing and debugging policies in place, which are available for all project personnel to access. There is management support for these policies.

Management ensures they are applied to all projects. Testing for each project is planned. The test plan is prepared in conjunction with the project plan so that project goals can be achieved.

Moving up to TMM level 3 requires further investment of organizational resources in the testing process. One of the maturity goals at TMM level 3 calls for the "Establishment of a test

organization." It implies a commitment to better testing and higher-quality software. This commitment requires that testing specialists be hired, space be given to house the testing group, resources be allocated to the group, and career paths for testers be established.

By supporting a test group an organization acquires leadership in areas that relate to testing and quality issues. For example, there will be staff with the necessary skills and motivation to be responsible for: 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.

The staff members of such a group are called **test specialists or test engineers**. Their primary responsibly is to ensure that testing is effective and productive, and that quality issues are addressed. Testers are not developers, or analysts, although background in these areas is very helpful and necessary. Testers don't repair code. However, they add value to a software product in terms of higher quality and customer satisfaction.

12.2 SKILLS NEEDED BY A TEST SPECIALIST

Given the nature of **technical and managerial** responsibilities assigned to the tester many **managerial and personal skills** are necessary for success in the area of work.

On the **personal and managerial level a test specialist** must have:

- 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

In addition, test specialists must be creative, imaginative, and experiment oriented. They need to be able to visualize the many ways that a software item should be tested, and make hypotheses about the different types of defects that could occur and the different ways the software could fail.

On the **technical level testers** need to have:

- an education that includes an understanding of general software engineering principles, practices, and methodologies;
- strong coding skills and an understanding of code structure and behavior;
- a good understanding of testing principles and practices;
- a good understanding of basic testing strategies, methods, and techniques;
- the ability and experience to plan, design, and execute test cases and test procedures on multiple levels (unit, integration, etc.);
- a knowledge of process issues;
- knowledge of how networks, databases, and operating systems are organized and how they work;
- a knowledge of configuration management;
- a knowledge of test-related documents and the role each documents plays in the testing process;
- the ability to define, collect, and analyze test-related measurements;
- the ability, training, and motivation to work with testing tools and equipment;
- a knowledge of quality issues.

12.3 BUILDING A TESTING GROUP

Establishing a specialized testing group is a major decision for an organization. The steps in the process are summarized in the following Figure. To initiate the process, upper management must support the decision to establish a test group and commit resources to the group. Decisions must be made on how the testing group will be organized, what career paths are available, and how the group fits into the organizational structure (See: The Structure of the Test Group).

When hiring staff to fill test specialist positions, management should have a clear idea of the educational and skill levels required for each testing position and develop formal job descriptions to fill the test group slots.

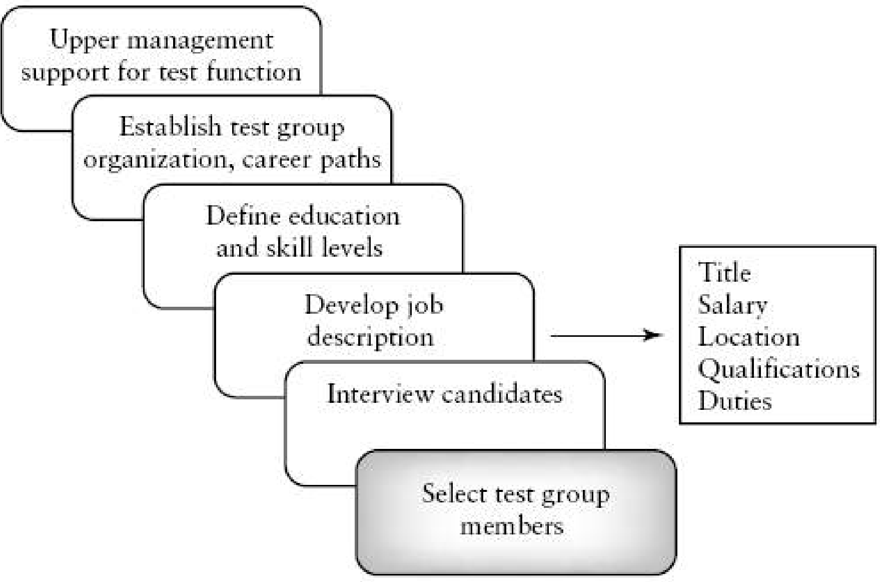


FIG. Steps in forming a test group

12.4 THE STRUCTURE OF THE TEST GROUP

It is important for a software organization to have an **independent testing group.** The group should have a formalized position in the organizational hierarchy. A reporting structure should be established and resources allocated to the group. The group should be staffed by people who have the skills and motivation. They should be dedicated to establishing awareness of, and achieving, existing software quality goals, and also to strengthening quality goals for the future software products. They are **quality leaders—the test and quality policy makers**. They measure quality, and have responsibilities for ensuring the software meets the customers' requirements.

A test organization is expensive, it is a strategic commitment. Given the complex nature of the software being built, and its impact on society, organizations must realize that a test organization is necessary and that it has many benefits. By investing in a test organization a company has access to **a group of specialists who have the responsibilities and motivation to:**

- 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 duties of the team members

The duties of the team members may vary in individual organizations. The following gives a brief description of the duties for each tester that are common to most organizations:

- **The Test Manager**: In most organizations with a testing function, the test manager (or test director) is the central person concerned with all aspects of testing and quality issues.

  The test manager is usually responsible for test policy making, customer interaction, test planning, test documentation, controlling and monitoring of tests, training, test tool acquisition, participation in inspections and walkthroughs, reviewing test work, the test repository, and staffing issues such as hiring, firing, and evaluation of the test team members. He or she is also the liaison with upper management, project management, and the quality assurance and marketing staffs.

- **The Test Lead**: The test lead assists the test manager and works with a team of test engineers on individual projects. He or she may be responsible for duties such as test planning, staff supervision, and status reporting. The test lead also participates in test design, test execution and reporting, technical reviews, customer interaction, and tool training.

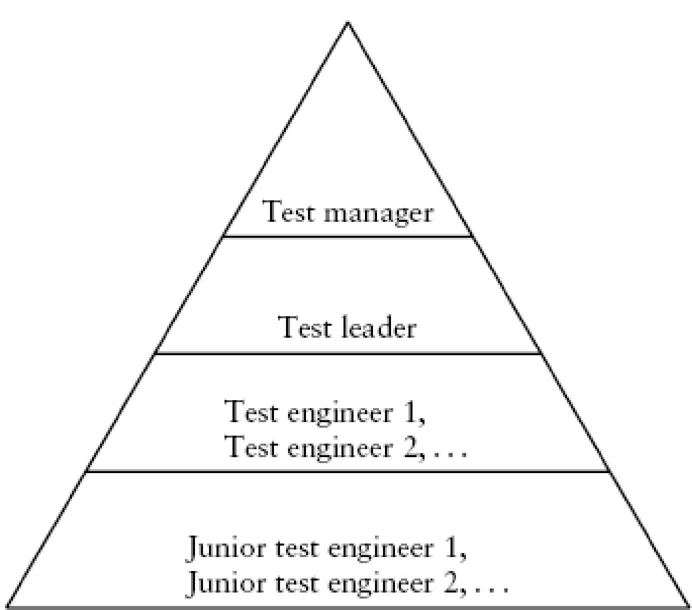


FIG. The test team hierarchy

- **The Test Engineer**: The test engineers design, develop, and execute tests, develop test harnesses, and set up test laboratories and environments. They also give input to test planning and support maintenance of the test and defect repositories.
- **The Junior Test Engineer:** The junior test engineers are usually new hires. They gain experience by participating in test design, test execution, and test harness development. They may also be asked to review user manuals and user help facilities defect and maintain the test and defect repositories.

CHAPTER 13
CONTROLLING AND MONITORING

13.1 INTRODUCTION

Engineers monitor and control the processes that drive each engineering project. Monitoring and controlling are engineering management activities, and should be practiced by software engineers as a part of their professional engineering duties. The TMM supports controlling and monitoring of testing with a maturity goal at level 3. A description of these two activities follows.

Project monitoring (or tracking) refers to the activities and tasks managers engage in to periodically check the status of each project. Reports are prepared that compare the actual work done to the work that was planned.
Monitoring requires a set of tools, forms, techniques, and measures. A precondition for monitoring a project is the existence of a project plan.

Project controlling consists of developing and applying a set of corrective actions to get a project on track when monitoring shows a deviation from what was planned.
If monitoring results show deviations from the plan have occurred, controlling mechanisms must be put into place to direct the project back on its proper track.

Thayer partitions what he calls **"project controlling" into six major tasks.** The following is a **modified description of the tasks** suggested by Thayer.

1. **Develop standards of performance**. These set the stage for defining goals that will be achieved when project tasks are correctly accomplished.
2. **Plan each project**. The plan must contain measurable goals, milestones, deliverables, and well-defined budgets and schedules that take into consideration project types, conditions, and constraints.
3. **Establish a monitoring and reporting system**. In the monitoring and reporting system description the organization must describe the measures to be used, how/when they will be collected, what questions they will answer, who will receive the measurement reports, and how these will be used to control the project. If status meetings are required, then their frequency, attendees, and resulting documents must be described.
4. **Measure and analyze results.** Measurements for monitoring and controlling must be collected, organized, and analyzed. They are then used to compare the actual achievements

with standards, goals, and plans.

5. **Initiate corrective actions for projects that are off track**. These actions may require changes in the project requirements and the project plan.
6. **Reward and discipline**. Reward those staff who have shown themselves to be good performers, and discipline, retrain, relocate those that have consistently performed poorly.
7. **Document the monitoring and controlling mechanisms**. All the methods, forms, measures, and tools that are used in the monitoring and controlling process must be documented in organization standards and be described in policy statements.
8. **Utilize a configuration management system.** A configuration management system is needed to manage versions, releases, and revisions of documents, code, plans, and reports.

It was Thayer's intent that these activities and actions be applied to monitor and control software development projects and l testing efforts as well.

13.2 MEASUREMENTS AND MILESTONES FOR MONITORING AND CONTROLLING

Introduction

All processes should have measurements (metrics) associated with them. The measurements help to answer questions about status and quality of the process, as well as the products that result from its implementation. Measurements in the testing domain can help to track test progress, evaluate the quality of the software product, manage risks, classify and prevent defects, evaluate test effectiveness, and determine when to stop testing. Level 4 of the TMM calls for a formal test measurement program. To begin the collection of meaningful measurements each organization should answer the following questions:

- Which measures should we collect?
- What is their purpose (what kinds of questions can they answer)?
- Who will collect them?
- Which forms and tools will be used to collect the data?
- Who will analyze the data?
- Who to have access to reports?

The following sections describe a collection of measurements that support monitoring of test over time. Each **measurement is shown in italics** to highlight it. It is recommended that measurements followed by an **asterisk (*)** be collected by all organizations, even those at **TMM level 1**. Now we will address the question of how a testing process can be monitored for each project. A test manager needs to start with a test plan. What the manager wants to measure and evaluate is the actual work that was done and compare it to work that was planned. To help support this goal, the test plan must contain **testing milestones**.

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 milestones can be used to monitor the progress of the testing efforts associated with a software project. Each level of testing will have its own specific milestones. Some **examples of testing**

milestones are:

- completion of the master test plan;
- completion of branch coverage for all units (unit test);
- implementation and testing of test harnesses for needed integration of major subsystems;
- execution of all planned system tests;
- completion of the test summary report. At the status meetings, project and test leaders present up-to-date

Measurements, graphs and plots showing the status of testing efforts. Testing milestones met/not met and problems that have occurred are discussed. Test logs, test incident reports, and other test-related documents may be examined as needed. Managers will have questions about the progress of the test effort.

Mostly, they will want to know if testing is proceeding according to schedules and budgets, and if not, what the barriers are. Some of the **typical questions a manager might ask at a status meeting are:**

- Have all the test cases been developed that were planned for this date?
- What percent of the requirements/features have been tested so far?
- How far have we proceeded on achieving coverage goals: Are we ahead or behind what we scheduled?
- How many defects/KLOC have been detected at this time?Howmany repaired? How many are of high severity?
- What is the earned value so far? Is it close to what was planned (see Section 9.1.3)?
- How many available test cases have been executed? How many of these were passed?
- How much of the allocated testing budget has been spent so far? Is it more or less than we estimated?
- How productive is tester X? How many test cases has she developed? How many has she run? Was she over, or under, the planned amount?

The measurement data collected helps to answer these questions.

Goal/Question/Metric Paradigm

The links between measurements and question are described in the **Goals/ Questions/Metrics (GQM) paradigm** reported by Basili.

- In the case of testing, a **major goal** is to monitor and control testing efforts (a maturity goal at TMM level 3).
- An organizational team (developers/testers, SQA staff, project/test managers) constructs a set of likely **questions** that test/project managers are likely to ask in order to monitor and control the testing process. The sample set of questions previously described is a good starting point.
- Finally, the team needs to identify a set of **measurements** that can help to answer these questions.

A sample set of measures is provided in the following sections. Any organizational team can use them as a starting point for selecting measures that help to answer testrelated monitoring and controlling questions.

Four key items are recommended to test managers for monitoring and controlling the test efforts for a project. These are:

(i) testing status;

(ii) tester productivity;

(iii) testing costs;

(iv) errors, faults, and failures.

we will examine the measurements required to track these items.

13.3 MEASUREMENTS FOR MONITORING TESTING STATUS

Monitoring testing status means identifying the current state of the testing process. The manager needs to determine if the testing tasks are being completed on time and within budget. Given the current state of the testing effort some of the questions under consideration by a project or test manager would be the following:

- Which tasks are on time?
- Which have been completed earlier then scheduled, and by how much?
- Which are behind schedule, and by how much?
- Have the scheduled milestones for this date been meet?
- Which milestones are behind schedule, and by how much?

The following set of measures will help to answer these questions. The test status measures are partitioned into **four categories** as shown in Figure.

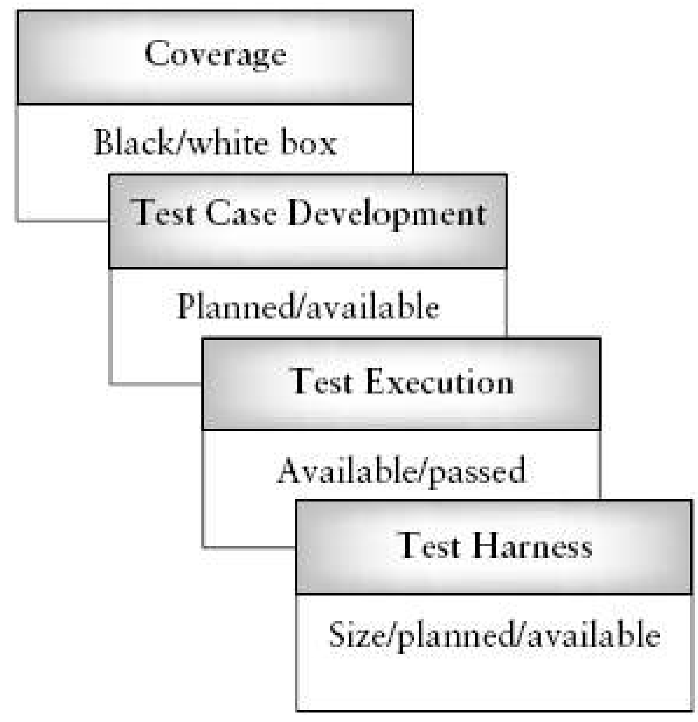


FIG. Types of testing status measurements

1. Coverage Measures

Depending on coverage goals for white box testing, a combination of the following are recommended.

Degree of statement, branch, data flow, basis path, etc., coverage (planned, actual)*

Tools can support the gathering of this data. Testers can also use ratios such as:

Actual degree of coverage/planned degree of coverage to monitor coverage to date.

For black box coverage the following measures can be useful:

Number of requirements or features to be tested*

Number of equivalence classes identified

Number of equivalence classes actually covered

Number or degree of requirements or features actually covered*

Testers can also set up ratios during testing such as:

Number of features actually covered/total number of features*

2. Test Case Development

The following measures are useful to monitor the progress of test case development, and can be applied to all levels of testing.

Number of planned test cases, Number of available test cases, Number of unplanned test cases

3. Test Execution

As testers carry out test executions, the test manager will want to determine if the execution process is going occurring to plan. This next group of measures is appropriate.
Number of available test cases executed*
Number of available tests cases executed and passed*
Number of unplanned test cases executed
Number of unplanned test cases executed and passed.
For a new release where there is going to be regression testing then these are useful:
Number of planned regression tests executed
Number of planned regression tests executed and passed
Testers can also set up ratios to help with monitoring test execution. For example:
Number of available test cases executed/number of available test cases
Number of available test cases executed/number of available test cases executed and passed
These would be derived measures.

4. Test Harness Development
Some useful measurements are:
Lines of Code (LOC) for the test harnesses (planned, available)*
We use lines of code in the measurements listed above as it is the most common size metric and can be easily applied to estimating the size of a test harness. Ratios such as:
Available LOC for the test harness code/planned LOC for the test harnesses
are useful to monitor the test harness development effort over time.

13.4 MEASUREMENTS TO MONITOR TESTER PRODUCTIVITY

Managers have an interest in learning about the productivity of their staff, and how it changes as the project progresses. Measuring productivity in the software development domain is a difficult task since developers are involved in many activities, many of which are complex, and not all are readily measured. For each developer/tester, where relevant, we **measure both planned and actual:**
Time spent in test planning
Time spent in test case design*
Time spent in test execution*
Time spent in test reporting
Number of test cases developed*
Number of test cases executed*
Productivity for a tester could be estimated by a combination of:
Number of test cases developed/unit time*
Number of tests executed/unit time*
Number of LOC test harness developed/unit time*
Number of defects detected in testing/unit time
The last item could be viewed as an indication of testing efficiency. This measure could be partitioned for defects found/hour in each of the testing phases to enable a manager to evaluate the efficiency of defect detection for each tester in each of these activities. For example:
Number of defects detected in unit test/hour
Number of defects detected in integration test/hour, etc.
Marks suggests as a tester productivity measure

Number of test cases produced/week

13.5 MEASUREMENTS FOR MONITORING TESTING COSTS

To calculate planned earned values we need the following measurement data:
Total estimated time or budget for the overall testing effort Estimated time or budget for each testing task
Earned values can be calculated separately for each level of testing. This would facilitate monitoring the budget/resource usage for each individual testing phase (unit, integration, etc.). We want to compare the above measures to:
Actual cost/time for each testing task*
We also want to calculate:
Earned value for testing tasks to date
Finally, the ratio of:
Estimated costs for testing/Actual costs for testing
can be applied to a series of releases or related projects to evaluate and promote more accurate test cost estimation and higher test cost effectiveness through test process improvement.

13.6 MEASUREMENTS FOR MONITORING ERROR, FAULTS, AND FAILURES

Monitoring errors, faults, and failures is very useful for:

- evaluating product quality;
- evaluating testing effectiveness;
- making stop-test decisions;
- defect casual analysis;
- defect prevention;
- test process improvement;
- development process improvement.

Some useful measures for defect tracking are:
Total number of incident reports (for a unit, subsystem, system)*
Number of incident reports resolved/unresolved (for all levels of test)*
Number of defects found of each given type*
Number of defects causing failures of severity level greater than X found (where X is an appropriate integer value)
Number of defects/KLOC (This is called the defect volume. The division by KLOC normalizes the defect count)*
Number of failures*
Number of failures over severity level Y (where Y is an appropriate integer value)
Number of defects repaired*
Estimated number of defects (from historical data)

A sample severity level hierarchy

- **Catastrophic:** a failure that could cause loss of life or property and/or loss of a system.
- **Critical:** a failure that could cause major harm or major injury to life or property and/or cause major damage to a software system.

- **Marginal:** a failure that could cause minor harm or minor injury to life, or cause a software system to perform poorly or reduce its availability.
- **Minor or Annoying:** a failure that does not cause any significant harm or injury to life, property or a software system, but does require repair.

13.7 MONITORING TEST EFFECTIVENESS

To complete the discussion of test controlling and monitoring and the role of test measurements we need to address what is called test effectiveness. Test effectiveness measurements will allow managers to determine if test resources have been used wisely and productively to remove defects and evaluate product quality. Test effectiveness evaluations allow managers to learn which testing activities are or are not productive. We can make such an evaluation in several ways, both before and after release.

1. **Before release.** Compare the numbers of defects found in testing for this software product to the number expected from historical data. The ratio is:

Number of defects found during test/number of defects estimated

This will give some measure of how well we have done in testing the current software as compared to previous similar products.

2. **After release.** Continue to collect defect data after the software has been released in the field. In this case the users will prepare problem reports that can be monitored. Marks suggests we use measures such as "field fault density" as a measure of test effectiveness. This is equal to:

Number of defects found/thousand lines of new and changed code

This measure is applied to new releases of the software. Another measure suggested is a ratio of:

Pre-ship fault density/Post-ship fault density

Other measurements for test effectiveness have been proposed. For example, a measurement suggested by Graham is:

Number of defects detected in a given test phase/total number of defects found in testing.

Another useful measure, called the **"detect removal leverage (DRL)"** described as a review measurement, can be applied to measure the relative effectiveness of: reviews versus test phases, and test phases with respect to one another. The DRL sets up ratios of defects found. The ratio denominator is the base line for comparison. For example, one can compare:

$$\text{DRL (integration/unit test)} = \frac{\text{Number of defects found integration test}}{\text{Number of defects found in unit test}}$$

The costs of each testing phase relative to its defect detecting ability can be expressed as:

$$\frac{\text{Number of defects detected in testing phase X}}{\text{Costs of testing in testing phase X}}$$

The effectiveness metric called the TCE is defined as follows:

$$\text{TCE} = \frac{\text{Number of defects found by the test cases}}{\text{Total number of defects} \times 100}$$

The total number of defects in this equation is the sum of the defects found by the test cases, plus the defects found by what Chernak calls side effects. Side effect are based on so-called “test-escapes.” These are software defects that a test suite does not detect but are found by chance in the testing cycle.

13.8 STATUS MEETINGS, REPORTS, AND CONTROL ISSUES

Status Meetings

Measurement-related data, and other useful test-related information such as test documents and problem reports, should be collected and organized by the testing staff. The test manager can then use these items for presentation and discussion at the periodic meetings used for project monitoring and controlling. These are called project status meetings**.**

Test-specific status meetings can also serve to monitor testing efforts, to report test progress, and to identify any test-related problems. Testers can meet separately and use test measurement data and related documents to specifically discuss test status. Following this meeting they can then participate in the overall project status meeting, or they can attend the project meetings as an integral part of the project team and present and discuss test-oriented status data at that time.

Each organization should decide how to organize and partition the meetings. Some deciding factors may be the size of the test and development teams, the nature of the project, and the scope of the testing effort.

Status meetings usually result in some type of status report published by the project manager that is distributed to upper management. Test managers should produce similar reports to inform management of test progress.

Rakos recommends that the reports be brief and contain the following items

- **Activities and accomplishments during the reporting period.** All tasks that were attended to should be listed, as well as which are complete. The latter can be credited with earned value amounts. Progress made since the last reporting period should also be described.
- **Problems encountered since the last meeting period.** The report should include a discussion of the types of new problems that have occurred, their probable causes, and how they impact on the project. Problem solutions should be described.
- **Problems solved.** At previous reporting periods problems were reported that have now been solved. Those should be listed, as well as the solutions and the impact on the project.
- **Outstanding problems.** These have been reported previously, but have not been solved to date. Report on any progress.
- **Current project (testing) state versus plan.** This is where graphs using process measurement data play an important role. Examples will be described below. These plots show the current state of the project (testing) and trends over time.
- **Expenses versus budget.** Plots and graphs are used to show budgeted versus actual expenses. Earned value charts and plots are especially useful here.

- **Plans for the next time period.** List all the activities planned for the next time period as well as the milestones.

Milestone Meetings

- Another type of project-monitoring meeting is the milestone meeting that occurs when a milestone has been met. A milestone meeting is an important event; it is a mechanism for the project team to communicate with upper management and in some cases user/client groups.
- Major testing milestones should also precipitate such meetings to discuss accomplishments and problems that have occurred in meeting each test milestone, and to review activities for the next milestone phase. Testing staff, project managers, SQA staff, and upper managers should attend.
- In some cases process improvement group and client attendance is also useful. Milestone meetings have a definite order of occurrence; they are held when each milestone is completed.
- Typical test milestone meeting attendees are shown in the following Figure.

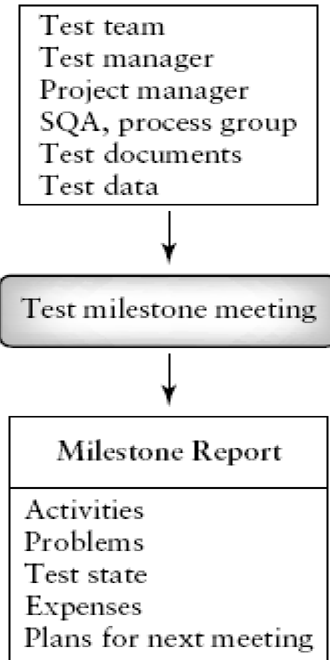


FIG. Test milestone meetings, participants, inputs, and outputs.

- It is important that all test-related information be available at the meeting, for example, measurement data, test designs, test logs, test incident reports, and the test plan itself.

13.9 CRITERIA FOR TEST COMPLETION

Tester managers and staff should do their best to take actions to get the testing effort on track. In any event, whether progress is smooth or bumpy, at some point every project and test manager has

to make the decision on when to stop testing.

Part of the task of monitoring and controlling the testing effort is making this decision about when testing is complete under conditions of uncertainly and risk. Managers should not have to use guesswork to make this critical decision. The test plan should have a set of quantifiable stop-test criteria to support decision making.

The weakest stop test decision criterion is to stop testing when the project runs out of time and resources.

There are **five stop-test criteria** that are based on a more quantitative approach. Managers should use a combination of criteria and cross-checking for better results.

The stop-test criteria are as follows.

1. All the Planned Tests That Were Developed Have Been Executed and Passed.
2. All Specified Coverage Goals Have Been Met.
3. The Detection of a Specific Number of Defects Has Been Accomplished.
4. The Rates of Defect Detection for a Certain Time Period Have Fallen Below a Specified Level.
5. Fault Seeding Ratios Are Favourable.

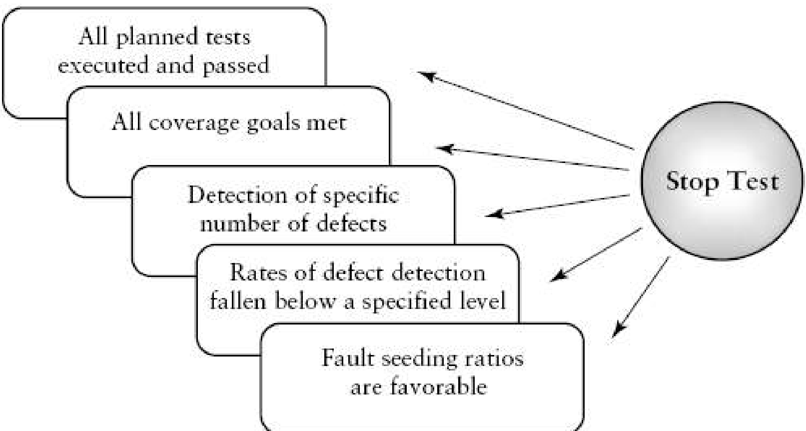


FIG. Some possible stop-test criteria

13.10 SOFTWARE CONFIGURATION MANAGEMENT

- Software systems are constantly undergoing change during development and maintenance. By software systems we include all software artifacts such as requirements and design documents, test plans, user manuals, code, and test cases.
- Different versions, variations, builds, and releases exist for these artifacts.
- Organizations need staff, tools, and techniques to help them track and manage these artifacts and changes to the artifacts that occur during development and maintenance.

- The Capability Maturity Model includes configuration management as a Key Process Area at level 2. This is an indication of its fundamental role in support of repeatable, controlled, and managed processes. To control and monitor the testing process, testers and test mangers also need access to configuration management tools and staff.
- There are **four major activities** associated with **configuration management**.

These are:

1. Identification of the Configuration Items

- The items that will be under configuration control must be selected, and the relationships between them must be formalized. An example relationship is “part-of” which is relevant to composite items. Relationships are often expressed in a module interconnection language (MIL). The following Figure shows four configuration items, a design specification, a test specification, an object code module, and source code module as they could exist in a configuration management system (CMS) repository.
- The arrows indicate links or relationships between them. Note in this example that the configuration management system is aware that these four items are related only to one another and not to other versions of these items in the repository.
- In addition to identification of configuration items, procedures for establishment of **baseline versions** for each item must be in place.

  Baselines are formally reviewed and agreed upon versions of software artifacts, from which all changes are measured. They serve as the basis for further development and can be changed only through formal change procedures.

  Baselines plus approved changes from those baselines constitute the correct configuration identification for the item.

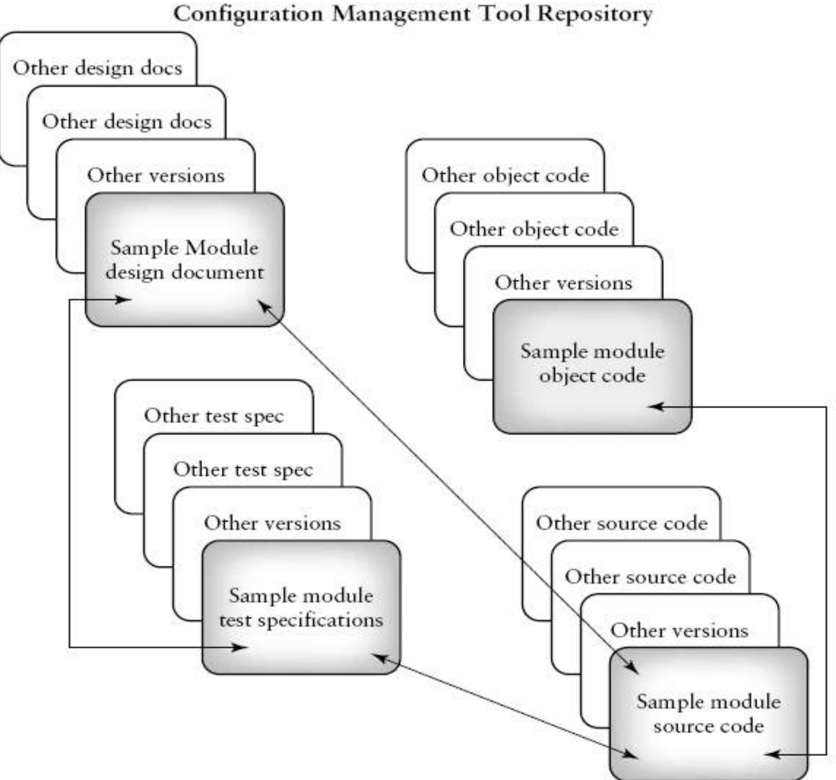


FIG. Sample configuration items

2. Change Control

- There are two aspects of change control—one is tool-based, the other team-based. The team involved is called a configuration control board. This group oversees changes in the software system. The members of the board should be selected from SQA staff, test specialists, developers, and analysts.
- It is this team that oversees, gives approval for, and follows up on changes. They develop change procedures and the formats for change request forms. To make a change, a change request form must be prepared by the requester and submitted to the board.
- It then reviews and approves/ disapproves. Only approved changes can take place. The board also participates in configuration reporting and audits as described further on in this section.

3. Configuration status reporting

These reports help to monitor changes made to configuration items. They contain a history of all the changes and change information for each configuration item. Each time an approved change is made to a configuration item, a configuration status report entry is made. The reports can answer questions such as:

- who made the change;
- what was the reason for the change;
- what is the date of the change;
- what is affected by the change.

Reports for configuration items can be disturbed to project members and discussed at status meetings.

4. Configuration audits

- After changes are made to a configuration item, how do software engineers follow up to ensure the changes have been done properly? One way to do this through a technical review, another through a configuration audit.
- The audit is usually conducted by the SQA group or members of the configuration control board.
- They focuses on issues that are not cov ered in a technical review.
- A checklist of items to cover can serve as the agenda for the audit.
- For each configuration item the audit should cover the following:
  (i) Compliance with software engineering standards. For example, for the source code modules, have the standards for indentation, white space, and comments been followed?
  (ii) The configuration change procedure. Has it been followed correctly?
  (iii) Related configuration items. Have they been updated?
  (iv) Reviews. Has the configuration item been reviewed?

SUMMARY

For review, a summary of the contributions of the three critical groups to TMM level 3 maturity goals is shown in Figure.

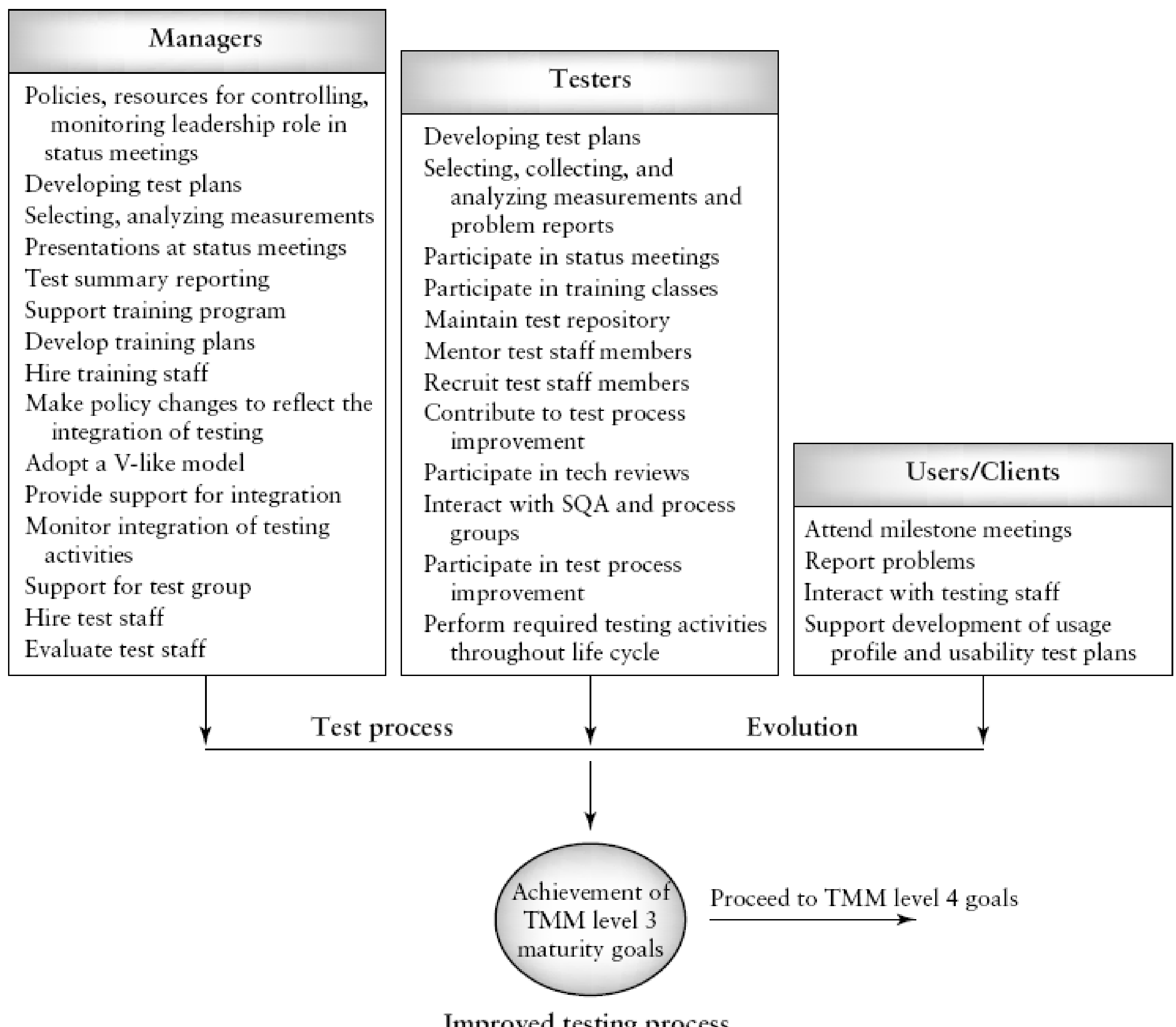


FIG. Contributions of three critical groups to TMM level 3 maturity goals

CHAPTER 14
SOFTWARE REVIEWS

14.1 INTRODUCTION

- One powerful tool that we can use is a manual static testing technique that is generally known as the technical review. Most software deliverables can be tested using review techniques.
- The technical review involves a group of people who meet to evaluate a software-related item. A general definition for a review is given below:
  A review is a group meeting whose purpose is to evaluate a software artifact or a set of software artifacts.

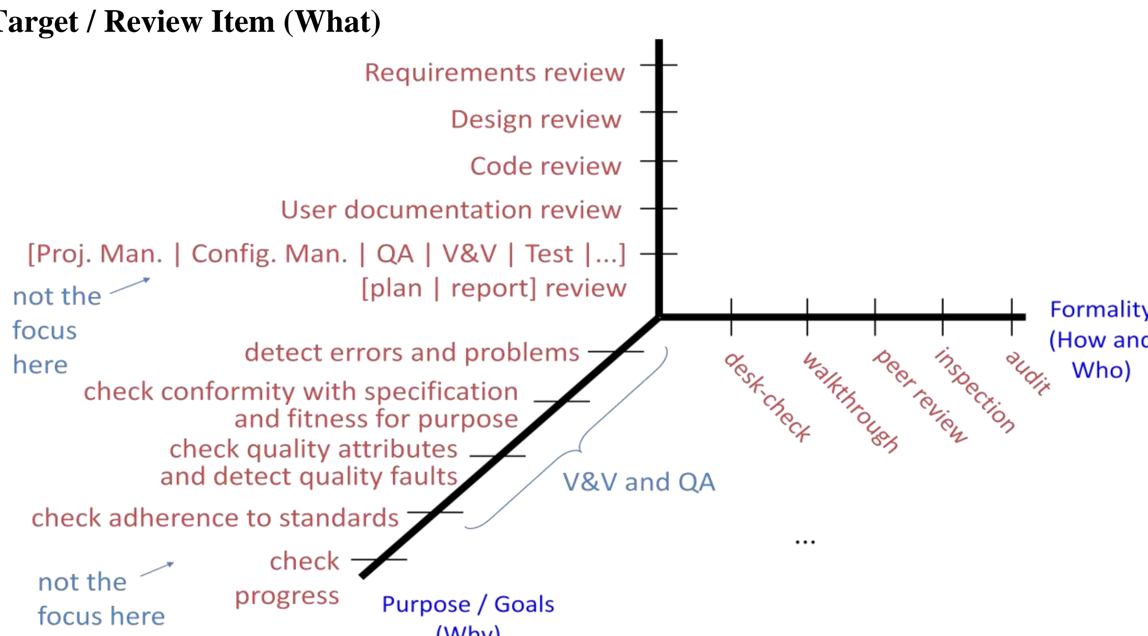
- The **general goals for the reviewers** are to:
  - identify problem components or components in the software artifact that need improvement;
  - identify components of the software artifact that do not need improvement;
  - identify specific errors or defects in the software artifact (defect detection);
  - ensure that the artifact conforms to organizational standards. the many benefits of a review program are:
  - higher-quality software;
  - increased productivity (shorter rework time);
  - closer adherence to project schedules (improved process control);
  - increased awareness of quality issues; • teaching tool for junior staff;
  - opportunity to identify reusable software artifacts;
  - reduced maintenance costs;
  - higher customer satisfaction;
  - more effective test planning;
  - a more professional attitude on the part of the development staff**.**

14.2 TYPES OF REVIEWS

- Reviews can be formal or informal. They can be technical or managerial.
- Managerial reviews usually focus on project management and project status.
- There are two major types of **technical reviews—inspections and walkthroughs**— which are more formal in nature and occur in a meeting-like setting.
- Formal reviews require written reports that summarize findings, and in the case of one type of review called an inspection, a statement of responsibility for the results by the reviewers is also required.
- The two most widely used types of reviews will be described

  1 Inspections as a Type of Technical Review

  2 Walkthroughs as a Type of Technical Review



14.3 TYPES OF REVIEWS ACCORDING TO FORMALITY

- Desk check
- Peer reviews
- Walkthroughs
- Inspections
- Audits

Desk check

- Also called self check
- Informal review performed by the author of the artifact

Peer reviews

- "I show you mine and you show me yours"
- The author of the reviewed item does not participate in the review
- Effective technique that can be applied when there is a team (with two or more persons) for each role (analyst, designer, programmer, technical writer, etc.)
- The peer may be a senior colleague (senior/chief analyst, senior/chief architect, senior/chief programmer, senior/chief technical writer, etc.)

Walkthroughs

- Type of technical review where the producer of the reviewed material serves as the review leader and actually guides the progression of the review (as a review reader)
- Traditionally applied to design and code
- In the case of code walkthrough, test inputs may be selected and review participants then literally walk through the design or code

- Checklist and preparation steps may be eliminated

Inspections

- A **formal evaluation** technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems.

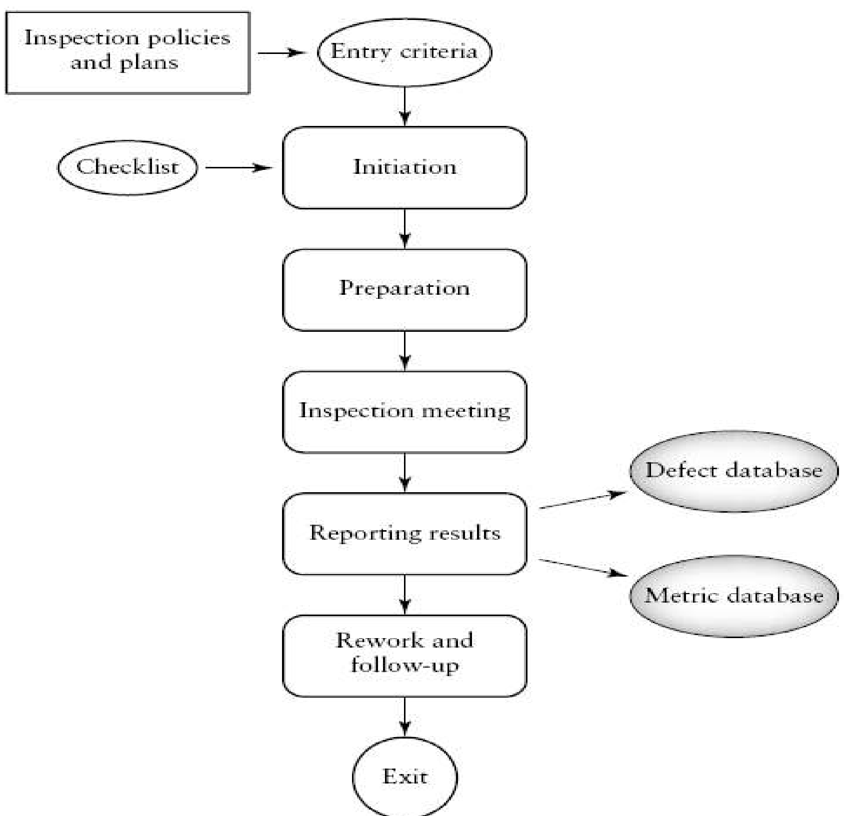


FIG. Steps in the inspection process

- Generally involve the author of a product.
- The inspector team may consist of different expertise, such as domain expertise, or design method expertise, or language expertise, etc. Inspections are usually conducted on a relatively small section of the product.
- Often the inspection team may have had a few hours to prepare, perhaps by applying an analytic technique to a small section of the product, or to the entire product with a focus only on one aspect, e.g., interfaces.
- A checklist, with questions germane to the issues of interest, is a common tool used in inspections.
- Inspection sessions can last a couple of hours or less, whereas reviews and audits are usually broader in scope and take longer.

Audits

- An audit is an **independent evaluation of conformance** of software products and processes to applicable regulations, standards, plans, and procedures
- An audit is a formally organized activity, with participants having specific roles, such as lead auditor, other auditors, a recorder, an initiator, and a representative of the audited organization
- Audits may examine plans like recovery, SQA, design documentation, etc.
- Audits can occur on almost any product at any stage of the development or maintenance process

14.4 REVIEWS AND TESTING

- A software system is more than the code; it is a set of related artifacts; these may contain defects or problem areas that should be reworked or removed; quality-related attributes of these artifacts should be evaluated
- Reviews allow us to detect and eliminate errors/defects *early* in the software life cycle (even before any code is available for testing), where they are less costly to repair
- Most problems have their origin in requirements and design; requirements and design artifacts can be reviewed but not executed and tested
  - Early prototyping is equally important to reveal problems in requirements and high-level architectural design
- A code review usually reveals directly the location of a bug, while testing requires a debugging step to locate the origin of a bug
- Adherence to coding standards cannot be checked by testing

14.5 TECHNICAL AND MANAGEMENT REVIEWS

- **Technical Reviews** - examine work products of the software project (code, requirement specifications, software design documents, test documentation, user documentation, installation procedures) for V&V and QA purposes
  - Multiple forms: Desk checking, Walkthroughs, Inspections, Peer Reviews, Audits
  - Covered here
- **Management Reviews** - determine adequacy of and monitor progress or inconsistencies against plans and schedules and requirements
  - Includes what Ian Somerville calls **Progress Reviews**
  - May be exercised **on plans and reports** of many types (risk management plans, project management plans, software configuration management plans, audit reports, progress reports, V&V reports, etc.)

14.6 COMPONENTS OF A REVIEW PLAN

- Review goals
- Items being reviewed
- Preconditions for the review
- Roles, team size, participants
- Training requirements
- Review steps and procedures
- Checklists and other related documents to be distributed to participants
- Time requirements
- Nature of the review log and summary report
- Rework and follow-up

Review Goals

(i) identification of problem components or components in the software artifact that need improvement,
(ii) identification of specific errors or defects in the software artifact,
(iii) ensuring that the artifact conforms to organizational standards, and
(iv) communication to the staff about the nature of the product being developed.

Additional goals might be to establish traceability with other project documents, and familiarization with the item being reviewed.

Preconditions and Items to Be Reviewed

In many organizations the items selected for review include:

- requirements documents;
- design documents;
- code;
- test plans (for the multiple levels);
- user manuals;
- training manuals;
- standards documents.

The preconditions need to be described in the review policy statement and specified in the review plan for an item. General preconditions for a review are:

(i) the review of an item(s) is a required activity in the project plan. (Unplanned reviews are also possible at the request of management, SQA or software engineers. Review policy statements should include the conditions for holding an unplanned review.)
(ii) a statement of objectives for the review has been developed;
(iii) the individuals responsible for developing the reviewed item indicate readiness for the review;
(iv) the review leader believes that the item to be reviewed is sufficiently complete for the review to be useful the design document for a procedure-oriented system may be reviewed in parts that encompass:

(i) the overall architectural design;
(ii) data items and module interface design;
(iii) component design.

Review Roles, Responsibilities and Attendance
Two major roles that need filling for a successful review are
(i) a leader or moderator, and
(ii) a recorder.
These are shown in the following Figure – Review Roles.

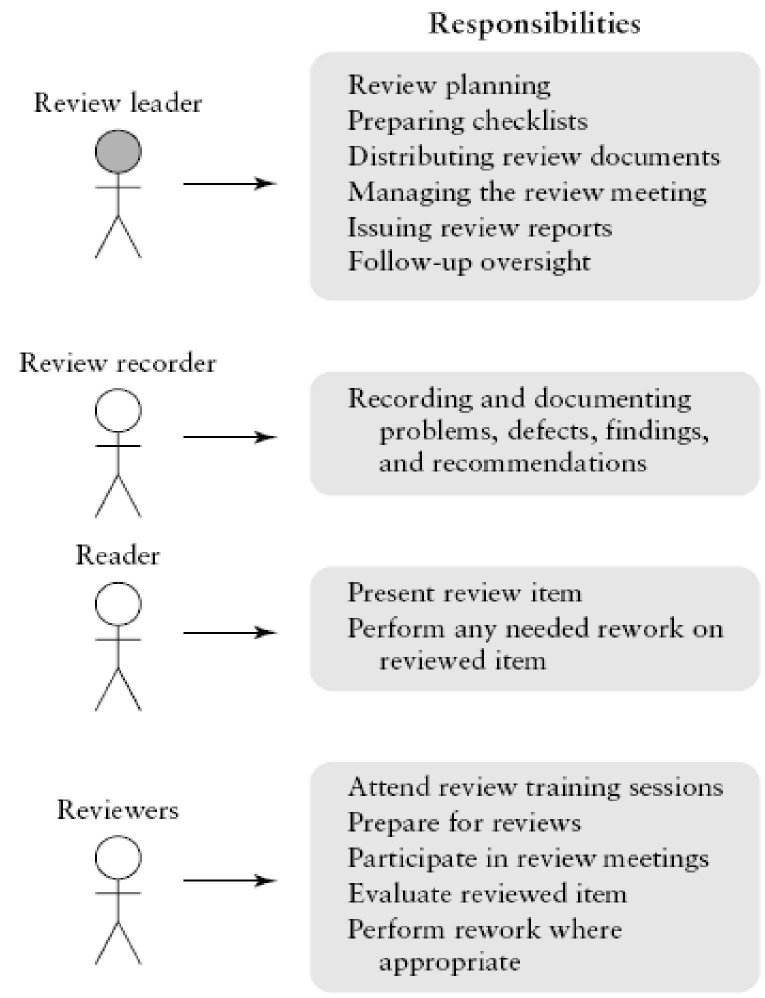


Figure: Review Roles

It is the author's option that testers take part in all major milestone reviews to ensure:

- effective test planning;
- traceability between tests, requirements, design and code elements;
- discussion, and support of testability issues;
- support for software product quality issues;
- the collection and storage of review defect data;

- support for adequate testing of “trouble-prone” areas.

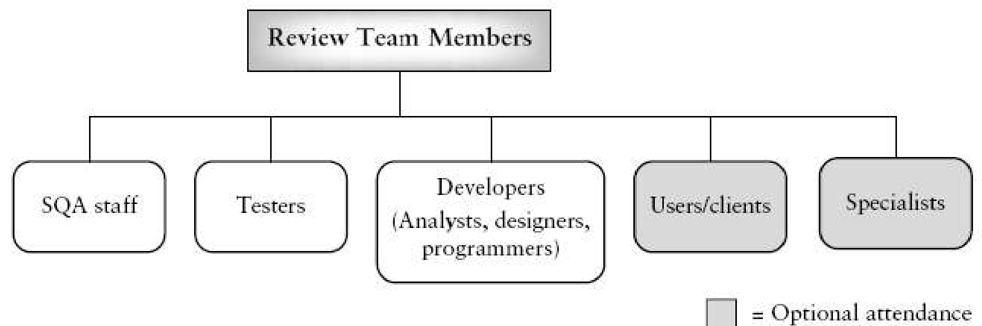


Figure: Review Team Members

Review Procedures

For each type of review that an organization wishes to implement, there should be a set of standardized steps that define the given review procedure. These are initiation, preparation, inspection meeting, reporting results, and rework and follow-up. For each step in the procedure the activities and tasks for all the reviewer participants should be defined. The review plan should refer to the standardized procedures where applicable.

Review Training

1. **Review of Process Concepts.**

Reviewers should understand basic process concepts, the value of process improvement, and the role of reviews as a product and process improvement tool.

2. **Review of Quality Issues.**

Reviewers should be made familiar with quality attributes such as correctness, testability, maintainability, usability, security, portability, and so on, and how can these be evaluated in a review.

3. **Review of Organizational Standards for Software A r t i f a c t s .**

Reviewers should be familiar with organizational standards for software artifacts.

4. **Understanding the Material to Be Reviewed.**

Concepts of understanding and how to build mental models during comprehension of code and software-related documents should be covered.

5. **Defect and Problem Types.**

Review trainees need to become aware of the most frequently occurring types of problems or errors that are likely to occur during development. They need to be aware what their causes are, how they are transformed into defects, and where they are likely to show up in the individual deliverables.

6. **Communication and Meeting Management S k i l l s .**

These topics are especially important for review leaders. It is their responsibility to communicate with the review team, the preparers of the reviewed document, management, and in some cases clients/user group members. Review leaders need to have strong oral and written communication skills and also learn how to conduct a review meeting.

7. **Review Documentation and Record Keeping.**

Review leaders need to learn how to prepare checklists, agendas, and logs for review meetings.

8. **Special Instructions.**

During review training there may be some topics that need to be covered with the review participants.

9. Practice Review Sessions.

Review trainees should participate in practice review sessions. There are very instructive and essential. One option is for instructors to use existing documents that have been reviewed in the past and have the trainees do a practice review of these documents.

Review Training Topics
Topic1. Basic concepts Topic 2. Review of quality issues Topic 3. Review of standards Topic 4. Understanding the material to be reviewed Topic 5. Defect and problem types Topic 6. Communication and meeting management skills Topic 7. Review documentation and record keeping Topic 8. Special instructions Topic 9. Practice review sessions

FIG. Topics for review training sessions.

14.7 CHECKLIST FOR REVIEWING

A Sample General Checklist for Reviewing Software Documents/Requirements Review

- Coverage and **completeness**
  - Are all essential items completed?
  - Have all irrelevant items been omitted?
  - Is the technical level of each topic addressed properly for this document?
  - Is there a clear statement of goals for this document?
  - (Don't forget: more documentation does not mean better documentation)
- **Correctness**
  - Are there incorrect items?
  - Are there any contradictions?
  - Are the any ambiguities?
- **Clarity** and **Consistency**
  - Are the material and statements in the document clear?
  - Are the examples clear, useful, relevant and correct?

- Are the diagrams, graphs and illustrations clear, correct, use the proper notation, effective, in the proper place?
- Is the terminology clear and correct?
- Is there a glossary of technical terms that is complete and correct?
- Is the writing style clear (nonambiguous)?

- References and Aids to Document Comprehension
  - Is there an abstract or introduction?
  - Is there a well placed table of contents?
  - Are the topics or items broken down in a manner that is easy to follow and is understandable?
  - Is there a bibliography that is clear, complete and correct?
  - Is there an index that is clear, complete and correct?
  - Is the page and figure numbering correct and consistent?

A sample supplementary checklist for design reviews
(for high-level architectural design and detailed design)

- Are the high-level and detailed designs consistent with requirements? Do they address all the functional and quality requirements? Is detailed design consistent with high-level design?
- Are design decisions properly highlighted and justified and traced back to requirements? Are design alternatives identified and evaluated?
- Are design notations (ex: UML), methods (ex: OOD, ATAM) and standards chosen and used adequately?
- Are naming conventions being followed appropriately?
- Is the system structuring (partitioning into sub-systems, modules, layers, etc.) well defined and explained? Are the responsibilities of each module and the relationships between modules well defined and explained? Do modules exhibit strong cohesion and weak coupling?
- Is there a clear and rigorous description of each module interface, both at the syntactic and semantic level? Are dependencies identified?
- Have user interface design issues, including standardization, been addressed properly?
- Is there a clear description of the interfaces between this system and other software and hardware systems?
- Have reuse issues been properly addressed, namely the possible reuse of COTS (commercial off the shelf) components (buy-or-build decision) and in-house reusable components?
- Is the system designed so that it can be tested at various levels (unit, integration and system)?

A sample general code review checklist

- Design Issues
  - Does each unit implement a single function?
  - Are there instances where the unit should he partitioned?
  - Is code consistent with detailed design?

  - Does the code cover detailed design?
- Data Items
  - Is there an input validity check?
  - Arrays-check array dimensions, boundaries, indices.
  - Variables - are they all defined, initiated? have correct types and scopes been checked?
  - Are all variables used?
- Computations
  - Are there computations using variables with inconsistent data types?
  - Are there mixed-mode computations?
  - Is the target value of an assignment smaller than the right-hand expression?
  - Is over- or underflow a possibility (division by zero)?
  - Are there invalid uses of integers or floating point arithmetic?
  - Are there comparisons between floating point numbers?
  - Are there assumptions about the evaluation order in Boolean expressions?
  - Are the comparison operators correct?
- Control Flow Issues
  - Will the program, module or, unit eventually terminate?
  - Is there a possibility of an infinite loop, a loop with a premature exit, a loop that never executes?
- Interface Issues
  - Do the number and attributes of the parameters used by a caller match those of the called routine? Is the order of parameters also correct and consistent in caller and callee?
  - Does a function or procedure alter a parameter that is only meant as an input parameter?
  - If there are global variables, do they have corresponding definitions and attributes in all the modules that use them?
- Input/output Issues
  - Have all files been opened for use?
  - Are all files properly closed at termination?
  - If files are declared are their attributes correct?
  - Are EOF or I/O errors conditions handed correctly?
  - Is I/O buffer size and record size compatible?
- Portability Issues
  - Is there an assumed character set, and integer or floating point representation?
  - Are their service calls that mar need to be modified?
- Error Messages
  - Have all warnings and informational messages been checked and used appropriately?
- Comments/Code Documentation
  - Has the code been properly documented? Are there global, procedures, and line comments where appropriate?

  - Is the documentation clear, and correct, and does it support understanding?
- Code Layout and White Space
  - Has white space and indentation been used to support understanding of code logic and code intent?
- Maintenance
  - Does each module have a single exit point?
  - Are the modules easy to change (low coupling and high cohesion)?

A sample code review checklist for C programs

- Data Items
  - Are all variables lowercase?
  - Are all variables initialized?
  - Are variable names consistent, and do they reflect usage?
  - Are all declarations documented (except for those that are very simple to understand)?
  - Is each name used for a singe function (except for loop variable names)?
  - Is the scope of the variable as intended?
- Constants
  - Are all constants in uppercase?
  - Are all constants defined with a "#define"?
  - Are all constants used in multiple files defined in an INCLUDE header file?
- Pointers
  - Are pointers declared properly as pointers?
  - Are the pointers initialized properly?
- Control
  - Are if/then, else, and switch statements used clearly and properly?
- Strings
  - Strings should have proper pointers.
  - Strings should end with a NULL.
- Brackets
  - All curly brackets should have appropriate indentations and be matched
- Logic Operators
  - Do all initializations use an " = " and not an " = ="?
  - Check to see that all logic operators are correct, for example, use of = / = =, and ||
- Computations
  - Are parentheses used in complex expressions and are they used properly for specifying precedences?
  - Are shifts used properly?

14.8 REPORTING REVIEW RESULTS

Contents of a formal review report

- Checklist will all items covered (with a check mark) and comments relating to each item
- List of defects found, with
  - description
  - type
  - frequency
  - defect class, e.g.
    - missing
    - incorrect
    - superfluous
  - location
    - cross-reference to the place or places in the reviewed document where the defect occurs
  - severity, e.g.
    - major
    - minor
- Summary report, with
  - list of attendees
  - review metrics, such as
    - number of participants
    - duration of the meeting
    - size of the item being reviewed (usually LOC or number of pages)
    - number of defects found
    - total preparation time for the review team
    - number of defects found per hour of review time
    - number of defects found per page or LOC
    - LOC or pages reviewed per hour
    - ...
  - status of the reviewed item (requirements document, etc.)
    - **accept** – the item is accepted in its present form or with minor rework required that does not need further verification
    - **conditional accept** – the item needs rework and will be accepted after the moderator has checked and verified the rework
    - **reinspect** – considerable rework must be done to the item. The inspection needs to be repeated when the rework is done.
  - estimate of rework effort and the estimated date for completion of the rework
  - signatures and date