**Module-1**

**Basics of Software Testing**

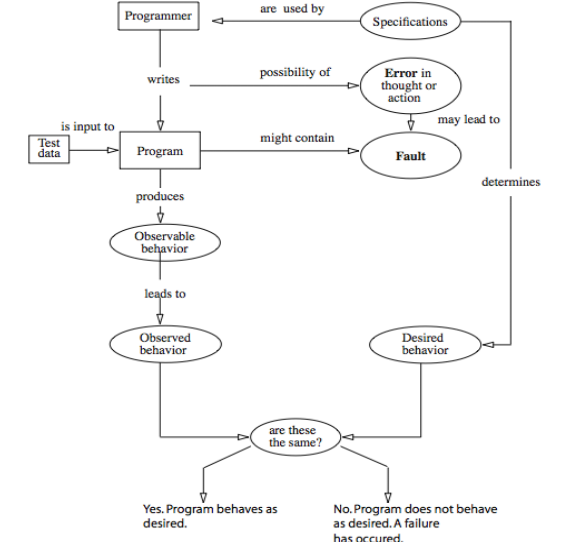
1. **Basic Definitions:**
   1. **What is Testing:**

* Testing is the process of executing a program with the intent of finding faults.
* In the process of testing, a tester must find answers to the questions that developers generally ignore such as:
  + 1. How is the software?
    2. How good it is?
    3. How do u know that it works?
    4. What are the critical areas?
    5. What are the weak areas and why?
    6. What is serious design issue?
    7. What do you feel about the complexity of the source code?
  1. **What are the objectives of a tester?**
* Good testing means uncovering the hidden faults in a software program, checking if the application is behaving as it must behave.
* A tester is expected to
  + Understand the application well both in terms of functional and structural
  + After gaining product knowledge, a tester is expected to make test plan, create test cases, execute test cases, track defects and provide production support.
  1. **Errors, Fault and Failures:**
* Humans make errors in their thoughts, in their actions, and in the products that might result from their actions.
* Humans can make errors in a field.

Ex: observation, in speech, in medical prescription, in surgery, in driving, in sports, in love and similarly even in software development.

**Example:**

* An instructor administers a test to determine how well the students have understood what the instructor wanted to convey
* A tennis coach administers a test to determine how well the understudy makes a serve.



The figure above describes the differences between fault, error and failure.

**Error:**

* + Informally error denotes the mistake in coding, and this can happen because of the syntactical faults.
  + Developer can’t compile or run a program due to a coding mistake in a program.

**Fault:**

* Fault is a manifestation of one or more errors.
* Fault generally happens due to omission or commission error.
* error of omission results in a fault in which something is missing that should be present in the representation.
* Error of commission results in a fault in which something extra is present in the representation.

**Failure:**

* Presence and execution of faulty code leads to failure and generally failures are identified by the customers.
* A failure occurs when the code corresponding to a fault executes.
* Two forms of failure can arise one that is related to only execution that may be caused due to source code or loaded object code. This form is generally viewed as failure due to fault of commission.

Other form of definitions you should be aware with respect to this topic is

**Incident:**

* When a failure occurs, it may or may not be readily apparent to the user (or customer or tester).
* An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure.

**Example:**

Consider the case of an accounting software that manages all the account related information of an organization. None of the users can notice a failure related to income tax calculation till someone reports on it. This reporting can be viewed as an incident that alerts the people using the accounting software about the income tax calculation failure.

|  |  |
| --- | --- |
| **Defect** | **Bug** |
| Not satisfying the requirements | Result of a coding error |
| Identified by the user outside premises after going into production | Identified as a part of unit testing within the premises before production |
| Eg: User finding grammatical mistake in product | Eg: In an application where both phone number and license number are considered but the linking has happened interchangeably. |

* 1. **Testing related terminologies:**

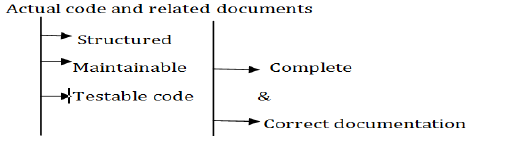
|  |  |
| --- | --- |
| **Verification** | **Validation** |
| Evaluating work products | Evaluating the product to check if it meets the business needs |
| Eg: Booking of hotel, verification means checking reviews, photos etc and then deciding. | Eg: Go to the hotel directly, check if it suits you or not is validating |
| Verification is generally carried out through reviews, walkthroughs and inspections. | Validation is carried out through the testing. |
| Verification is done thoroughly or not is evaluated by checking the requirement specification, design specification code and test cases | Validation is evaluated by checking the actual product/software. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test** | **Test Case** | **Test Suite** | **Test Procedure** | **Test Plan** | **Test Strategy** |
| A test is the act of exercising software with test cases. A test has two distinct goals: to find failures or to demonstrate correct execution. | Test case is a sequence of steps that help us perform a certain test on the application. | Test suite, on the other hand, is the list of all the test cases that must be executed as a part of a test cycle or a regression phase etc. There is no logical grouping based on functionality. The order in which the constituent test cases get executed may or may not be important. | Test procedure is a combination of test cases based on a certain logical reason, allowing us to identify the order of execution of test cases | Test plan is a document that lists all the activities in a QA project, schedules them, defines the scope of the project, roles & responsibilities, risks, entry & exit criteria, test objective and anything else that you can think of. | Test strategy outlines the testing approach and everything else that surrounds it. It is different from the test plan, in the sense that a Test strategy is only a subset of the test plan |

1. **Software Quality:**

* The product is said to possess good quality if it meets the specification as well as the product is measured for its quality.
* The quality attributes are the ones that is required to measure quality for a given product.
* Based on the type of functionality performed the quality attributes are classified as static and dynamic.
* **Static quality attributes** refer to the actual code and related documents as the name suggests it is static in nature and that can only include verification procedures.

Some of the static quality attributes are depicted below:



**Example:** A poorly documented piece of code will be harder to understand and hence difficult to modify. A poorly structured code might be harder to modify and difficult to test. (Imagine a code without any comment line…denotes a poorly documented code)

* Dynamic quality attributes can reflect the behaviour of a system during execution.
* Some of the dynamic attributes are

**Reliability:**

It refers to the probability of failure free operation.

**Correctness :**

correct operation of product or process and is always with reference to some artefact.

For a Tester, correctness is w.r.t to the requirements

For a user correctness is w.r.t the user manual

**Completeness :**

Refers to the availability of all the features listed in the requirements or in the user manual. An incomplete software is one that does not fully implement all features required.

**Consistency:**

Refers to adherence to a common set of conventions and assumptions.

Ex: All buttons in the user interface might follow a common-color coding convention.

**Usability :**

How easily the application can be used refers to the usability?

Usability is generally done by potential users and the users tests the application in terms of user centric view where the ease of use, functionality as expected, performance, safety and security are tested.

Psychology plays an important role in the design of techniques for usability testing.

**Performance :**

Refers to the time the application takes to perform a requested task. Performance is considered as a non-functional requirement.

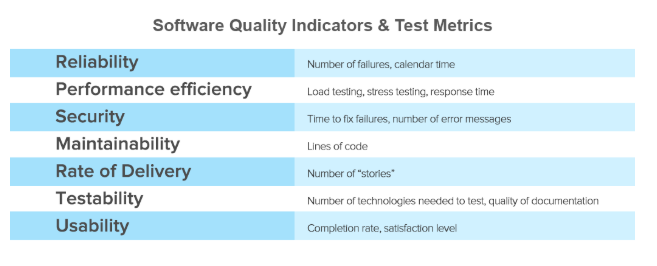
**Security :**

Assesses how well an application protects information against the risk of software breaches. The quantity and severity of vulnerabilities found in a software system are indicators of its security level. Poor coding and architectural weaknesses often lead to software vulnerabilities.

**Maintainability:**

Is the ease with which you can modify software, adapt it for other purposes, or transfer it from one development team to another.

Compliance with software architectural rules and use of consistent coding across the application combine to make software maintainable.



The above tabulation from altexsoft clearly indicates the way in which a product is measured for its quality.

1. **Requirements,Behaviour and Correctness:**

* These three terminologies are used in conjunction with each other and are directly proportional to each other as the correctness of the product is completely dependent on the way the tester understood the requirements.
* To explain this consider the two examples as below:

**Example:**

**Requirement 1: It is required to write a program that inputs and outputs the maximum of these.**

**Requirement 2: It is required to write a program that inputs a sequence of integers and outputs the sorted version of this sequence.**

* Suppose that the program max is developed to satisfy requirement 1 above. The expected output of max when the input integers are 13 and 19 can be easily determined to be 19.
* Suppose now that the tester wants to know if the two integers are to be input to the program on one line followed by a carriage return typed in after each number.
* The requirement as stated above fails to provide an answer to this question. This example illustrates the incompleteness requirements 1.
* The second requirement in (the above example is ambiguous. It is not clear from their requirement whether the input sequence is to be sorted in ascending or descending order. The behaviour of sort program, written to satisfy this requirement, will depend on the decision taken by the programmers while writing sort. Testers are often faced with incomplete/ambiguous requirements. In such situations a testers may resort to a variety of ways to determine what behaviour to expect from the program under test).
* Regardless of the nature of the requirements, testing requires the determination of the expected behaviour of the program under test. The observed behaviour of the program is compared with the expected behaviour to determine if the program functions as desired.

**3.1 Exhaustive Testing:**

* Suppose there is a program that takes an integer for calculation and presenting results must be evaluated for its correctness attribute. How can we evaluate it?
* The simplest answer would be to check if it meets the requirements or not. But this is not correct from tester’s perspective as the requirements should be checked for all the possible inputs. So, if you have one integer as input the set of all possible inputs are -32678 to 32678
* Testing a program on all possible inputs is known as “exhaustive testing”.
* If the requirements are complete and unambiguous, it should be possible to determine the set of all possible inputs.

**3.2 Definition: *Input Domain***

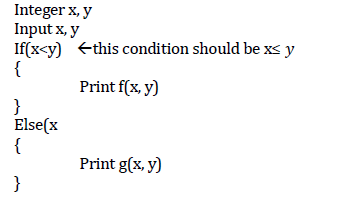
* The set of all possible inputs to program P is known as the input domain, or input space, of P.
* Modified requirement 2: It is required to write a program that inputs a sequence of integers and outputs the integers in this sequence sorted in either ascending or descending order. The order of the output sequence is determined by an input request character which should be “A” when an ascending sequence is desired, and “D” otherwise while providing input to the program, the request character is entered first followed by the sequence of integers to be sorted. The sequence is terminated with a period.
* A program is considered correct if it behaves as expected on each element of its input domain.

**3.3 Valid and Invalid Data:**

* The input domains are derived from the requirements. It is difficult to determine the input domain for incomplete requirements.
* Identifying the set of invalid inputs and testing the program against these inputs are important parts of the testing activity. Even when the requirements fail to specify the program behaviour on invalid inputs, the programmer does treat these in one way or another. Testing a program against invalid inputs might reveal errors in the program.
* Ex: sort program < E 7 19...> The sort program enters an infinite loop and neither asks the user for any input nor responds to anything typed by the user. This observed behaviour points to a possible error in sort.

1. **Correctness Vs Reliability:**

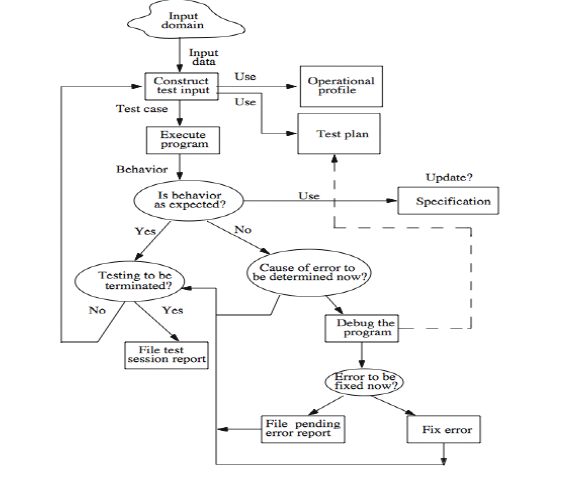
* Though correctness of a program is desirable, it is almost never the objective of testing.
* To establish correctness via testing would imply testing a program on all elements in the input domain, which is impossible to accomplish in most cases that are encountered in practice.
* Thus, correctness is established via mathematical proofs of programs.
* While correctness attempts to establish that the program is error-free, testing attempts to find if there are any errors in it.
* Thus, completeness of testing does not necessarily demonstrate that a program is error-free.
* Removal of errors from the program. Usually improves the chances, or the probability, of the program executing without any failure.
* Also testing, debugging and the error-removal process together increase confidence in the correct functioning of the program under test.
* **Example:**



* Suppose that function f produces incorrect result whenever it is invoked with x=y and that f(x, y)≠ g(x, y), x=y. In its present form the program fails when tested with equal input values because function g is invoked instead of function f. When the error is removed by changing the condition x<y to x≤𝑦, the program fails again when the input values are the same. The latter failure is due to the error in function f. In this program, when the error in f is also removed, the program will be correct assuming that all other code is correct.
* A comparison of program correctness and reliability reveals that while correctness is a binary metric, reliability is a continuous metric, over a scale from 0 to 1.
* A program can be either correct or incorrect, it is reliability can be anywhere between 0 and 1. Intuitively when an error is removed from a program, the reliability of the program so obtained is expected to be higher than that of the one that contains the error.

1. **Testing and Debugging:**

* Testing is a process to execute the program to find errors.
* During the testing process, error is detected, the steps taken to determine the cause of the error and removing it is Debugging.



**The Steps involved in the test/debug cycle are as follows:**

1. **Preparing a test plan :**

* Test cycle commences with the guidance from the test plan,
* Test plan details the
* methods used for testing,
* methods for evaluating the adequacy of test cases
* and a method to determine if the program has failed or not.

***Test plan for sorting program:***

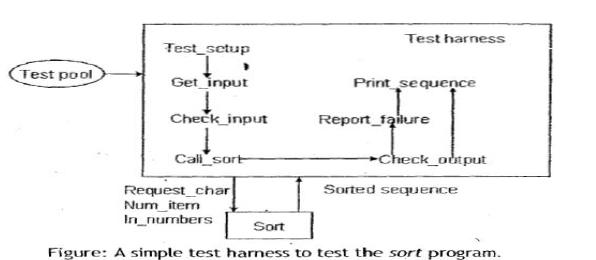
1. Execute the program on at least two input sequence one with “A” and the other with “D” as request characters.
2. Execute the program on an empty input sequence
3. Test the program for robustness against erroneous input such as “R” typed in as the request character.
4. All failures of the test program should be recorded in a suitable file using the company failure report form.
5. **Constructing test data :**

* Test plan aids one in constructing test data.
* Test data is nothing but the input to the program or application.
* For every input variable used in the program a value is associated which is termed as test data.
* Execution of the program on test data might begin after all or a few test cases have been constructed.
* Based on the results obtained, the testers decide whether to continue the construction of additional test cases or to enter the debugging phase.
* The following test cases are generated for the sort program using the test plan in the previous figure.



1. **Executing the Program:**

* With the available test plan and test data, the next significant step in testing is execution that includes construction of test harness that can assist in program execution.
* Test harness typically includes global variables, test case for carrying out the execution. The output of a harness is typically stored in a file.
* The below figure shows the harness for the sort program.



In preparing this test harness assume that:

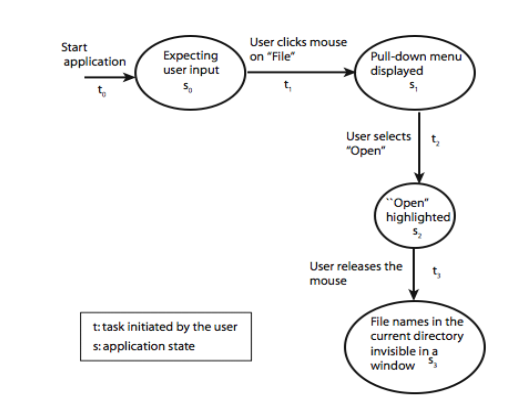
1. Sort is coded as a procedure
2. The get-input procedure reads the request character & the sequence to be sorted into variables request\_char, num\_items and in\_number, test\_setup procedure-invoked first to set up the test includes identifying and opening the file containing tests. Check\_output procedure serve as the oracle that checks if the program under test behaves correctly. Report\_failure: output from sort is incorrect. May be reported via a message(or)saved in a file. Print\_sequence: prints the sequence generated by the sort program. This also can be saved in file for subsequent examination.
3. **Specifying Program Behaviour:**

Representation of program behaviour can be specified in several ways as formal mathematical specification, state diagram, plain natural language etc.



Consider a menu driven application as below:





* The above figure shows the state diagram for the menu driven application.
* State sequence diagram can be used to specify the behavioural requirements.
* This same specification can then be used during the testing to ensure if the application confirms to the requirements.

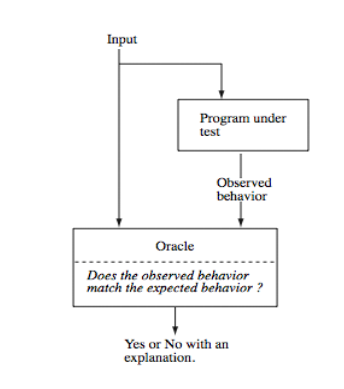
1. **Assessing the correctness of the program:**

Behaviour: It has two steps:

1. Observes the behaviour

2. Analyzes the observed behaviour.

* Above task, extremely complex for large distributed system.
* The entity that performs the task of checking the correctness of the observed behaviour is known as an oracle.



* But human oracle is the best available oracle.
* Oracle can also be programs designed to check the behaviour of other programs.

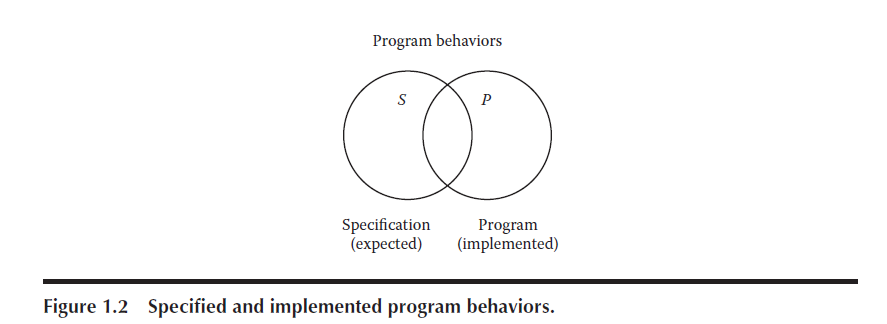
1. **Construction of oracles:**

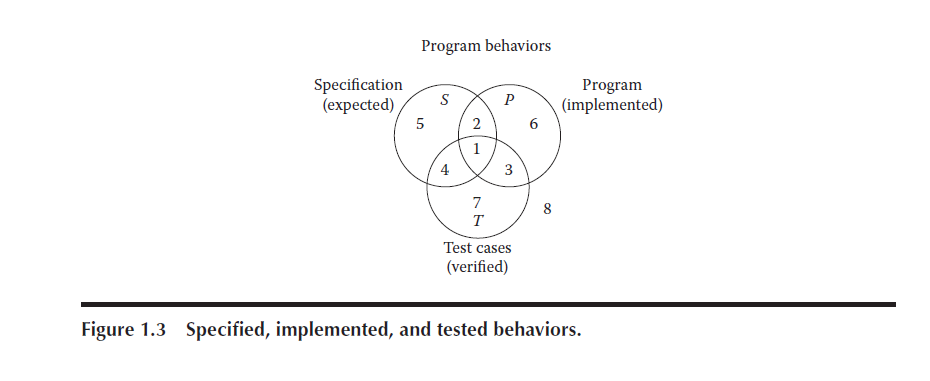
* Construction of automated oracles, such as the one to check a matrix multiplication program or a sort program, Requires determination of I/O relationship.
* When tests are generated from models such as finite-state machines (FSMs)or state charts, both inputs and the corresponding outputs are available.
* This makes it possible to construct an oracle while generating the tests.
* Example: Consider a program named Hvideo that allows one to keep track of home videos. In the data entry mode, it displays a screen in which the user types in information about a DVD. In search mode, the program displays a screen into which a user can type some attribute of the video being searched for and set up a search criterion.
* To test Hvideo we need to create an oracle that checks whether the program function correctly in data entry and search nodes. The input generator generates a data entry request. The input generaor now requests the oracle to test if Hvideo performed its task correctly on the input given for data entry.



* The oracle uses the input to check if the information to be entered into the database has been entered correctly or not. The oracle returns a pass or no pass to the input generator.

**6.Insights from a Venn Diagram:**

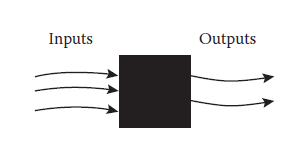
* Through a Venn diagram representation, several nagging questions about testing are clarified by making a clear distinction between behaviour and code-based view.
* Software developer’s perception is code based and this is orthogonal to behaviour.
* Major difficulty faced by testers lies in terms of the resources a tester gets while starting the testing process.
* Most of the base documents emphasis on code based lacking behavioural information making the job of testers difficult.
* The relationship between the programmed and specified behaviour is represented through the venn diagram below and this can clearly depict the problems that confront a tester.
* In the universal set of all program behaviors, S denotes the set of specified behaviour and P denotes the set of Programmed behaviour.
* This diagram clearly denotes a problem faced by tester that is the faults of omission that is What if certain specified behaviors have not been programmed? and fault of commission what if certain programmed (implemented) behaviors have not been specified?
* The intersection of *S* and *P* (the football-shaped region) is the “correct” portion, that is, behaviors that are both specified and implemented.
* A very good view of testing is that it is the determination of the extent of program behavior that is both specified and implemented.



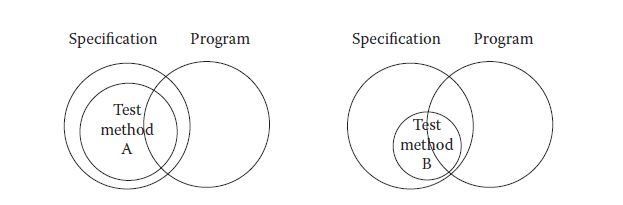
* Now, consider the relationships among sets *S*, *P*, and *T*.
* There may be specified behaviors that are not tested (regions 2 and 5), specified behaviors that are tested(regions 1 and 4), and test cases that correspond to unspecified behaviors (regions 3 and 7).Similarly, there may be programmed behaviors that are not tested (regions 2 and 6), programmed behaviors that are tested (regions 1 and 3), and test cases that correspond to behaviors that were not implemented (regions 4 and 7).
* Each of these regions is important.
* If specified behaviors exist for which no test cases are available, the testing is necessarily incomplete.
* If certain test cases correspond to unspecified behaviors, some possibilities arise: either such a test case is unwarranted, the specification is deficient, or the tester wishes to determine that specified non-behavior does not occur.
* what can a tester do to make the region where these sets all intersect (region 1) as large as possible? Another approach is to ask how the test cases in set *T* are identified. The short answer is that test cases are identified by a testing method.

1. **Identifying Test Cases:**

* The process of deriving test cases for a given problem or application is carried out by specification-based testing or code-based testing.
* Both approaches have several distinct test case identification methods; they are generally just called testing methods. They are methodical in the sense that two testers following the same “method” will devise very similar test cases.
  1. **Specification Based Testing:**
* More commonly termed as functional or Black box testing.
* To derive test cases adopting specification-based testing, knowledge about how the program/application is developed is not required.
* In functional testing, the program is a function that maps values from its input domain to values in the output domain.
* With the specification-based approach to test case identification, the only information used is the specification of the software.



* Figure below shows the results of test cases identified by two specification-based methods.
* Method A identifies a larger set of test cases than does method B. Notice that, for both methods, the set of test cases is completely contained within the set of specified behavior. Because specification-based methods are based on the specified behavior, it is hard to imagine these methods identifying behaviors that are not specified.



* specification-based testing, including boundary value analysis, robustness testing, worst-case analysis, special value testing, input (domain) equivalence classes, output (range) equivalence classes, and decision table-based testing.

**Advantages:**

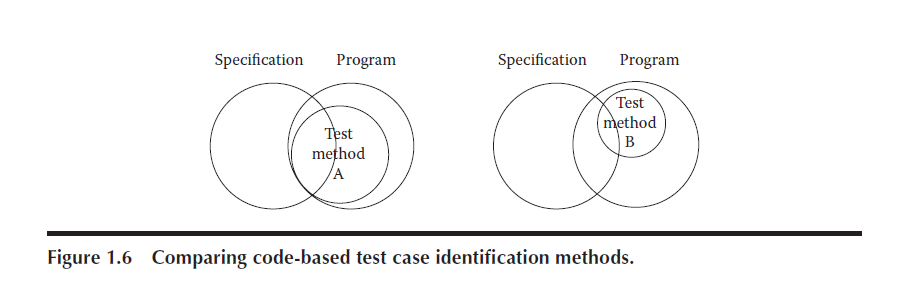
(1) They are independent of how the software is implemented, so if the implementation changes, the test cases are still useful.

(2) Test case development can occur in parallel with the implementation, thereby reducing the overall project development interval.

**Disadvantages:**

1. Redundancies among test cases.
2. Gaps of untested software.
   1. **Code Based Testing:**

* Contrast to black box, this is generally termed as white box or clear box testing.
* Here the implementation details are needed to generate test cases.
* The ability to “see inside” the black box allows the tester to identify test cases based on how the function is implemented.
* Code based testing requires strong theoretical basis thus the usage of test metrics is quite often in code-based testing.
* Figure below shows the results of test cases identified by two code-based methods. As before, method A identifies a larger set of test cases than does method B. Is a larger set of test cases necessarily better? This is an excellent question, and code-based testing provides important ways to develop an answer. Notice that, for both methods, the set of test cases is completely contained within the set of programmed behavior. Because code-based methods are based on the program,it is hard to imagine these methods identifying behaviors that are not programmed. It is easy to imagine, however, that a set of code-based test cases is relatively small with respect to the full set of programmed behaviors.

****

**Which is better?**

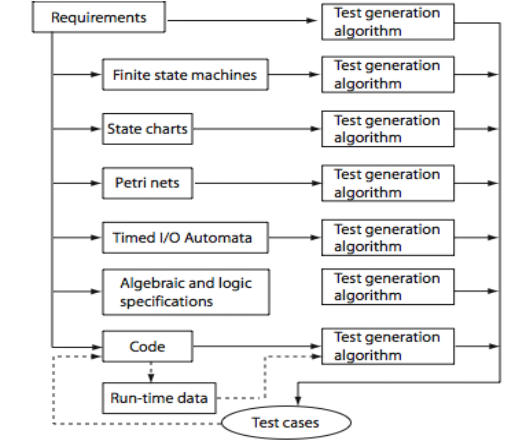
* There is no single rule which can say whether white or black box is better.
* QA has to concentrate on functional more than that of coding approach, however to be an efficient QA coding knowledge helps.
* Developer/ unit tester will concentrate more on white box testing strategies.
* Much of white-box testing is involved with coverage, making sure we have tested everything we need to based on the context of project needs.
* Using white-box testing on top of black-box testing allows us to measure the coverage we got and add more testing when needed to make sure we have tested all the important complexity we should.

1. **Test Generation Strategies:**

* Any form of test generation uses a source document.
* In the most informal of test methods, the source document resides in the mind of the tester who generates tests based on a knowledge of the requirements.
* In several commercial environments, the process is a bit more formal.
* The tests are generated using a mix of formal and informal methods either directly from the requirements document serving as the source. In more advanced test processes, requirements serve as a source for the development of formal models.
* The common classification for the test generation strategies are as follows:
* Model based: require that a subset of the requirements be modeled using formal notation (usually graphical).

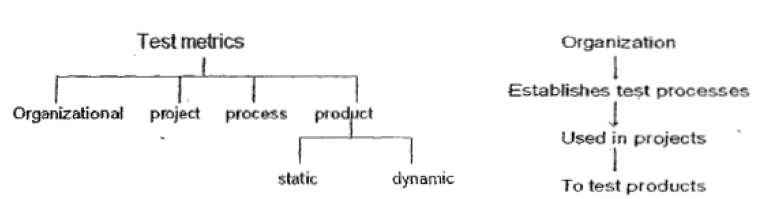
Models: Finite State Machines, Timed automata, Petri net, etc.

* Specification based: require that a subset of the requirements be modeled using a formal mathematical notation. Examples: B, Z, and Larch.
* Code based: generate tests directly from the code.



1. **Test Metrics:**

* To measure the quality of the software and to improve the effectiveness and efficiency of the testing process, test metrics are required.
* The term metric refers to a standard of measurement. In software testing, there exist a variety of metrics.
* The major classification of test metrics includes base and derived metrics whose definition varies according to the way in which the values are derived.



* ***Schedule related metrics:*** Measure actual completion times of various activities and compare these with estimated time to completion.
* ***Quality related metrics:*** Measure quality of a product or a process
* ***Resource related metrics:*** Measure items such as cost in dollars, manpower and test executed.
* ***Size-related metrics:*** Measure size of various objects such as the source code and number of tests in a test suite

***Organizational metrics:***

* Metrics at the level of an organization are useful in overall project planning and management. Ex: the number of defects reported after product release, averaged over a set of products developed and marketed by an organization, is a useful metric of product quality at the organizational level.
* Organizational metrics allow senior management to monitor the overall strength of the organization and points to areas of weakness. Thus, these metrics help senior management in setting new goals and plan for resources needed to realize these goals.

***Project metrics:***

* Project metrics relate to a specific project, for example the I/O device testing project or a compiler project. These are useful in the monitoring and control of a specific project.
  + Actual/planned system test effort is one project metrics. Test effort could be measured in terms of the tester\_man\_months.
  + Project metric= 𝑛𝑜.𝑜𝑓 𝑠𝑢𝑐𝑐𝑒𝑠𝑠𝑓𝑢𝑙 𝑡𝑒𝑠𝑡𝑠/𝑡𝑜𝑡𝑎𝑙 𝑛𝑢𝑚𝑏𝑒𝑟 𝑜𝑓 𝑡𝑒𝑠𝑡𝑠 𝑖𝑛 𝑡ℎ𝑒 𝑠𝑦𝑠𝑡𝑒𝑚 𝑝ℎ𝑎𝑠𝑒

***Process metrics:***

* Every project uses some test process. Big-bang approach well suited for small single person projects. The goal of a process metric is to assess the goodness of the process.
* Test process consists of several phases like unit test, integration test, system test, one can measure how many defects were found in each phase. It is well known that the later a defect is found, the consttier it is to fix.

**Product metrics: Generic**

* Cyclomatic complexity
* Halstead metrics

**Product metrics:**

* OO software **Metrics** are reliability, defect density, defect severity, test coverage, cyclomatic complexity, weighted methods/class, response set, number of children.
* Static and dynamic metrics: Static metrics are those computed without having to execute the product.
* Ex: no. of testable entities in an application.
* Dynamic metric requires code execution.
* Ex: no. of testable entities covered by a test suite is a dynamic quality.

**Testability:**

* According to IEEE, testability is the “degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met”.

**Two types:**

*  static testability metrics
* dynamic testability metrics

**Static testability metric:**

Software complexity is one static testability metric. The more complex an application, the lower the testability, that is higher the effort required to test it.

**Dynamic metrics** for testability includes various code-based coverage criteria.

Ex: when it is difficult to generate tests that satisfy the statement coverage criterion is considered to have low testability them one for which it is easier to construct such tests.

**Specifically testing related metric that are commonly used are:**

* 1. **Test case Execution Percentage:**

Percent of Test Case Execution = {(Number of Passed Test Cases + Number of Failed Test Cases + Number of Blocked Test Cases)/Total Number of Test Cases} \*100

* 1. **Defect Density:**

Defect Density = Total Number of defects/Total lines of code

* 1. **Defect Detection Percentage:**

Defect Density = Total Number of defects/Total lines of code

* 1. **Mean Time to Detect:**

Defect Density = Total Number of defects/Total lines of code

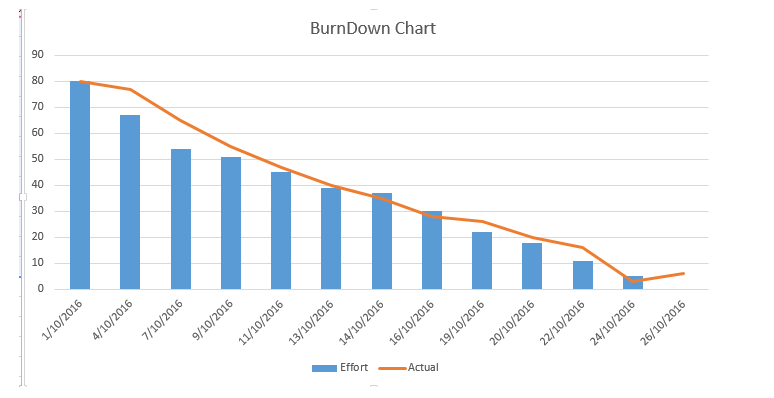
* 1. **Mean Time to Repair:**

Defect Density = Total Number of defects/Total lines of code

* 1. **Defect Leakage:**

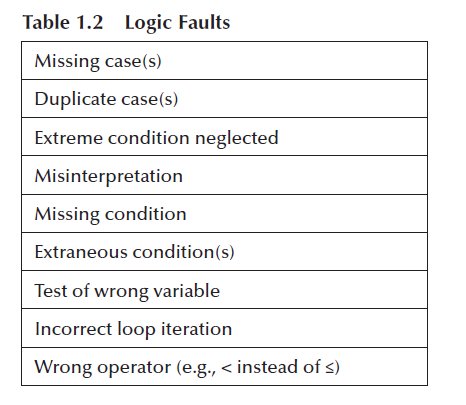
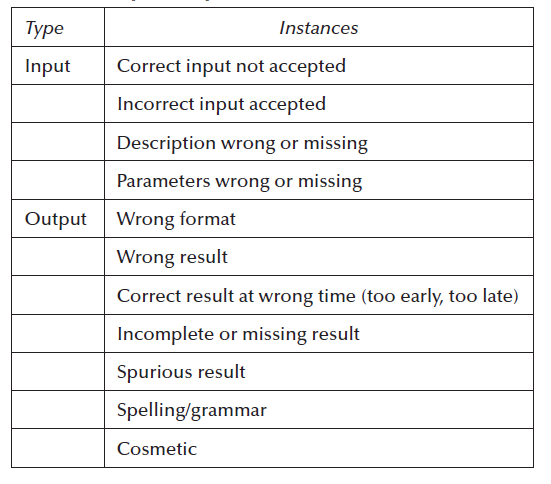
Defect Density = Total Number of defects/Total lines of code

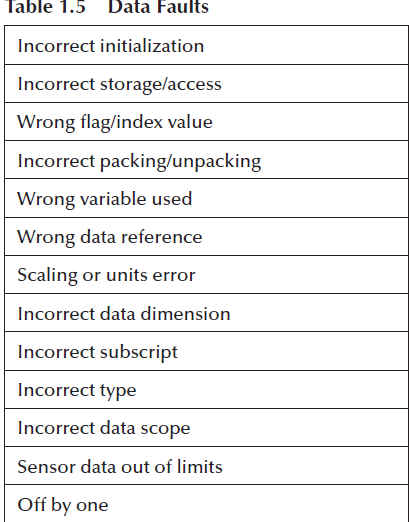
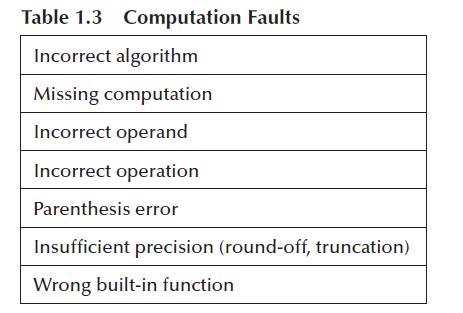
* 1. **Burn Down Chart:**

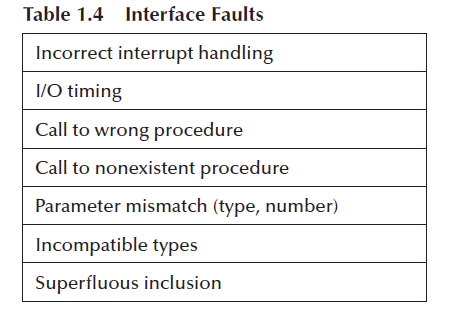
****

**10.Error and Fault Taxonomies:**

* Taxonomy as the name signifies represents the classification among the various types of fault and error.
* Distinction among the process and product can also be understood with this classification.
* Testing aims for a good quality product which says testing is product oriented and this goal of good quality product is possible by reducing errors endemic in the development process.
* Both process and product can be made better if a clear identification of faults is done.
* Faults can be classified in several ways: the development phase in which the corresponding error occurred, the consequences of corresponding failures, difficulty to resolve, risk of no resolution, and so on.
* For a comprehensive treatment of types of faults, see the IEEE Standard Classificationfor Software Anomalies (IEEE, 1993). (A software anomaly is defined in that document as“a departure from the expected,” )
* The IEEE standard defines a detailed anomaly resolution process built around four phases (another life cycle): recognition,investigation, action, and disposition.
* Some of the more useful anomalies are given in Tables below:

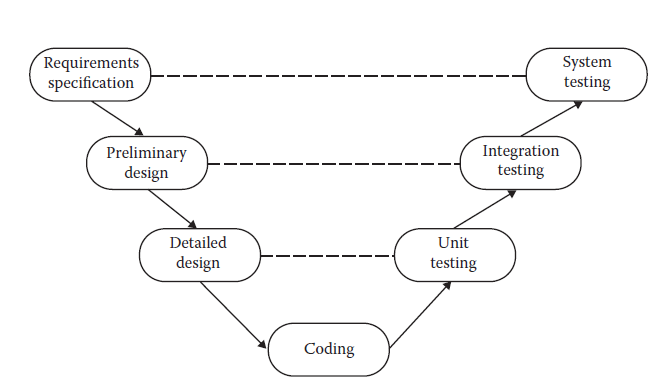






1. **Levels of Testing:**

* Levels of testing echo the levels of abstraction found in the waterfall model of the software development life cycle.
* Although this model has its drawbacks, it is useful for testing as a means of identifying distinct levels of testing and for clarifying the objectives that pertain to each level.
* A diagrammatic variation of the waterfall model, known as the V-Model in ISTQB parlance,is given below:



* To perform a unit level testing, code based testing is essential whereas to do a system testing, specification based testing knowledge helps.
* Unit testing denotes testing the individual components.
* Integration testing denotes the how the individual modules are working together.
* System testing denotes how the application behaves under environmental conditions.