CHAPTER TWO
SOFTWARE TESTING

2.1. Introduction

2.2. Testing Strategies

2.3. Testing Methods

2.4. Automatic Test Data Generation

2.5. Software Quality and Reliability

2.6. Complexity Metrics

2.7. Software Testing Life Cycle

2.8. The Order of Testing (The ‘V’ Model)

2.9. Conclusion
Chapter 2: Software Testing

2.1. Introduction

From an economic perspective, 95% of total system development costs will be allocated to software, and 50% of software development cost is devoted to testing it. It becomes obvious that software testing merits considerable attention [19]. By testing a program we mean a set of executions of that program for different given input data and such that the execution outcome are compared correspondingly to the given set of expected outcomes to state that this is a correct implementation of the abstract specification [9].

Software testing is used with various strategies to test applications. In this chapter, software testing techniques and different test adequacy criteria is explained. An explanation of quality, reliability of software and some reliability metrics are discussed. It also explains how testing contributes to the quality of the software.
2.2. Testing Strategies

Testing is the process of executing a software program/segment of code with the intention of finding errors. This implies that testing is “destructive process”. It is done to catch bugs.

There are different test strategies for the software testing problem. A rational test strategy must be ‘step-wise’ [97] i.e. step by step. Different testing strategies are discussed below.

2.2.1 Unit Testing:

Unit testing is also referred as “software component testing” [41]. It represents the lowest level of testing. Where, one can test small pieces of code separately. It is conducted to ensure that reliable program units are produced that meet their requirements. Unit testing is typically conducted by the development team.

In unit testing the execution of a computer application is based on one of the modules. In addition to the module some inputs (test data) are provided, data which has been selected in such a way as to steer the execution through specific part of the module under test [9].
Unit testing is the means by which testers demonstrate that source code executes according to software design requirements defined in the program design language or flowcharts approved at the unit design walkthrough [34].

So the test strategy is to find suitable mix of test cases whose deployment, in testing, will provoke that specific part of the module under test to achieve testing objectives.

2.2.2. Integration Testing:
The integration testing is the process of combining and testing multiple components together i.e. testing that different piece of the code, including code written by different groups, works together. Once individual program components have been tested, and found to be working correctly to meet the objectives, they must be integrated to create a partial or complete system [106]. The objective of integration testing is to demonstrate that the software modules interact in a correct stable and coherent manner.

The main difficulty in integration testing is localizing errors that are discovered during the process. There are complex interactions between system components. When an anomalous output is discovered in such testing, it may be hard to find the source of the error.
To make it easier to locate error, testers should always use an incremental approach to system integration and testing. Initially a minimal system configuration is integrated and tested then adding components to this minimal configuration and test after each added increment. Integration testing is conducted by a dedicated testing team.

2.2.3. System Testing:

System testing is considered as a high level testing that involves testing the system as a whole. For the purposes of objectivity, it is most appropriately performed outside the development organization, usually by an independent test group or by other testing organization. System testing usually inspects the functionality of software.

The fundamental objective of system testing is to establish confidence that the application under test will be accepted by the users. During system testing, it is essential that truly independent observation of the testing process is performed. Other than system functionality and behavior, system testing may include testing configuration, throughput, security, resource utilization, and performance [60, 113].
2.2.4. Acceptance Testing:

The purpose of acceptance testing is to confirm that the system meets its business requirements, and to provide confidence that the system works correctly and is usable before it is formally “handed over” to the end user. Acceptance testing is performed by nominated user representatives under the guidance and supervision by the testing team.

2.2.5. Regression Testing:

As modifications have been added to an existing system, testing also has to be done to make sure that the modification has not had any undesired side effect of making some of the earlier functions or services faulty [86]. That is, besides insuring behavior of the new service, testing has to ensure that the new services are maintained. The basic regression testing approach is to incorporate selected test cases into a regression test bucket that is run periodically in an attempt to detect regression problems [53]. Regression testing is practically important in software maintenance. In many software organizations regression testing consists of rerunning all the functional tests every few months.
2.3. Testing Methods

There are two basic methods for constructing tests: “White–box” testing and “Black-box” testing. White box testing examines the internal design of a program and requires that tester has thorough knowledge of its structure.

Black-box test however are generally based on the functional requirements. White-box generally has the highest error yield of all test techniques [53]. Both methods are discussed in the following sections

2.3.1. Black-box Testing:

Black-box testing, also known as functional or specification-based testing [106], tests the functionalities of the software against its specifications. In black-box testing, the internal structure and behavior of the program under test is not considered. The objective is to find solely when the input-output behavior of the program does not agree with its specification [6, 55].

The strength of black-box testing is that tests can be derived early in development cycle. This can detect missing logic faults [110]. There are four types of black-box testing: equivalence partitioning, boundary-value analysis, cause-effect graphing and error guessing [83].
- *Equivalence partitioning*: partitions the input domain of a program into a finite number of equivalence classes such that one can reasonably assume (but not absolutely sure) that a test of a representative value of each class is equivalent to a test data of any other value within the corresponding class.

- *Boundary value analysis* is concerned with those situations directly on, above, and beneath the edges of input equivalent classes and output equivalent classes. Experience shows that a test case that explores boundary condition has a higher payoff than test cases that do not [83].

- *Cause-effect graphing* is a formal language into which a natural-language specification is translated, which also points out incompleteness and ambiguities in the specifications.

- *Error-guessing* is largely an intuitive and ad-hoc process, whose procedure is difficult to formalize. This technique needs an expertise that is able to smell out errors. The basic idea is to enumerate a list of possible errors or error-prone situations and then write test cases based on the list.
2.3.2. White-box Testing:

White box testing is more widely applied [83]. It is the verification of a single module, usually in an isolated environment (i.e. isolated from all other modules). In white box testing the internal structure and behavior of the program under test is considered. The test data are derived from the program’s logic. It is also called structural testing because it sees the structure of the program [82,119]. In a White-box test we consider for example various coverage criteria. White-box testing is far more likely to uncover some kinds of errors that black-box test may miss no matter how thoroughly done/conducted [53].

Structural testing tests those requirements are satisfied in terms of the coverage of particular set of elements in the structure of the program or the specifications [127]. The analysis of the code can be used to find how many test cases are needed to guarantee that all of the statements in the program or component are executed at least once during the testing process. By doing so one may discover unnecessary "dead" code (code that is of no use), or never get executed at all, which cannot be discovered by functional testing [88].

White-box testing, is sometimes called ‘glass-box’ testing or ‘clear-box’ testing to distinguish it from black-box testing [106].
2.3.3. White-box Test Adequacy Criteria:

An adequacy criterion is an essential part of any testing method. The crucial factor in test case design is the set of criteria used to select test case inputs. This set determines what feature of the program under test (PUT) will be covered, hence the extent of the test and the degree of confidence that can be attached to the PUT.

Miller [4] points out that the relationship between the achieved coverage and the chance of an error remaining in the software is an indirect one at best, even though there is a substantial evidence to indicate that the correlation is very high.

There are several white box (structural) testing criteria: statement, decision (a. k. a. branch), condition/decision, and path coverage, ordered from the weakest to the strongest [46, 66, 83, 110 and 127].

1) Statement coverage criterion requires every statement in the structure under test to be executed at least once, during testing. Unfortunately this is a weak criterion because while it exercises every statement at least once, it does not guarantee exercising the same statement in different flaws if any. For example in fig (2-1) one is not required to generate input test datum that exercises false
branch in order to satisfy statement coverage condition. In this case the test case checks for the correctness of the sequence S1-S2-S3, but not for the correctness of sequence S1-S3; which is possible to have a problem.

![Sample pseudo code]

Fig (2-1): Sample pseudo code

2) **Branch coverage** or decision coverage: is a stronger criterion than statement coverage. It requires that every possible outcome of all decisions to be exercised at least once [127]. Decision coverage can be shown, usually, to satisfy statement coverage. The problem with branch coverage is that it does not check for all the different sequences.

3) **decision/condition** coverage: this criterion requires sufficient-test cases that each condition in a decision takes all possible outcomes at least once, and each point of entry is invoked at least once. Moreover, **multiple condition** coverage requires sufficient test cases such that all possible combinations of condition outcomes in each decision, and all points of entry invoked at least once. This can be done by varying just one condition at a time while holding all other possible conditions fixed [13].
4) **Path coverage** criterion is the utmost coverage criteria, since it covers all the previous mentioned testing coverage criteria [109,127]. In path testing every possible path in the software under test is executed. This increases the possibility of error detection. Path testing is concerned with the execution of (selected) paths in the program since programs with loops can have infinite number of paths.

Test adequacy criteria plays two fundamental roles [127]. First, an adequacy criterion specifies a particular software testing requirement, and hence determines test cases to satisfy the requirements. Second, is to determine the observation that should be made during the testing process for example statement coverage requires that the tester, or the testing team, observe whether each statement is executed during the process of software testing. If path coverage is used, then the observation of whether statements have been executed is sufficient or not; execution of path should be observed and recorded. Almost all test adequacy criteria proposed in the literature explicitly specify particular requirements on software testing. For example, path coverage is a test requires that all branches of the program should be exercised. The objective of testing is to satisfy this requirement. The degree to which this objective is achieved can be measured quantitively by the percentage of paths exercised.
Testing is a process where a program is tested by feeding more and more test cases to it. Here, a test adequacy criterion can be used as stopping rule to decide when this process can stop. Once the test is adequate, it indicates that the test objectives have been achieved, and no further test case is needed.

Since White–box testing deals directly with the code, it is better suited to handle lower level testing, such as unit testing. Black-box testing is more appropriate for checking the functionality of the system, hence better suited for system testing and acceptance testing. However it can be applied to unit testing as well.

Finding error during unit or integration testing helps to point out where the errors are, whereas higher level testing only gives a rough idea where one should investigate further. It is cheaper to find an error earlier and fix it immediately.

2.3.4: Path Testing

Path testing is a structural testing strategy whose objective is to exercise every independent execution path through a component or program. If every independent path is executed then all statements in the component must have been executed at least once. Furthermore, all conditional statements are tested
for both true and false cases [106]. Path testing techniques are therefore mostly used at the module testing stages of the testing process.

Generally, exhaustive testing is completely impractical, even for the most trivial units [97]. Path testing searches the program input domain for suitable test cases that covers every possible path in the software under test (SUT). However it is impossible to achieve this goal for several reasons: (1) a program may contain an infinite number of paths when the program has loops. (2) The number of paths in a program is exponential to the number of branches in it. (3) the number of test cases is too large, since each path can be covered by several test cases, making the coverage, of all possible paths computationally impractical [23,69 and 103].

It is only practical to select a specific subset path to perform path testing. Since it is impossible to cover all paths in software, the problem of path testing selects a subset of paths to execute and find test data to cover it.

The starting point of path testing is a program control flow graph CFG. This is the skeletal model of all paths through the program. A flow graph consists of nodes representing decisions and edges showing flow of control. The flow graph is constructed by replacing program control statements by equivalent diagrams. If there is no “goto” statement in a program, it is simple
process to derive its flow graph. Sequential statements (assignment, procedure
calls and statements) can be ignored in the flow graph construction. Each
branch in a conditional statement (if-then-else or case) is shown by a separate
path and loops are indicated by an arrow looping back to the loop condition
node. An example of flow graph construction for binary search routine adopted
from [59] is shown in fig (2-2).

By tracing the flow graph, we see that there are four independent paths
as flows:

1, 2, 3, 8, 9
1, 2, 3, 4, 5, 7, 2
1, 2, 3, 4, 6, 7, 2
1, 2, 3, 4, 6, 7, 2, 8, 9

If all these paths are executed then:
1. Every statement in the program has been executed at least once, and
2. Every branch has been executed for true and false conditions.

An independent path is one which traverses at least one new edge in the
flow graph. The number of independent paths in a program can be discovered
by computing the McCabe’s cyclomatic complexity of the flow graph.
Test case generation for path testing consists of four basic steps:

1) *Control Flow Graph construction*: In this step, the source program is transferred to a graph that represents the control flow of the program. A cycle in CFG may imply that there is a loop in the code. Each branch of the graph is denoted by a label and different branches correspond with different labels.

2) *Target path selection*: In path testing, paths are extracted from the CFG, and some paths might be very meaningful and need to be selected as target path for testing (e.g. the path for Equilateral triangles in the experiment of chapter 4 is very critical).
3) *Test case generation and execution*: In this step the algorithm automatically creates new test cases to execute new path and leads the control flow to the target path. Finally, a suitable test case that executes the target paths could be generated.

4) *Test result evaluation*: This step is to execute the selected path and to determine if the test criteria are satisfied.

### 2.4. Automatic Test Data Generators:

An important factor for developing automatically generated test data cases is the issue of cognitive bias.

When called upon construct test cases, software developer usually suffers from biases that results in the under-exploration of many testing regions. Knowledge of the programs internals, mental models and expectations all influence the design of test suit.

Automated tools can augment the testing process by providing test cases that are not subject to the same biases as human developers. Combining both human and machine generated test cases from multiple sources may be the most robust strategy for software testing.
Automatic test data generator is a tool that assists the tester in creating test data. It is, more specifically, a system (program) that generates the input data for a target program, such that these input data satisfy a particular testing objective (i.e. adequacy criterion). Automatic testing tools consist of an instrumentator, test harness and a test generator [66]. A number of automatic test data generation techniques have been developed [50]. Automatic test data generators have been classified into random test data generators, structural test data generator and goal-oriented test data generator [74]. Others categorized test data generators into three classes: (1) Random test data generators. (2) Structural oriented test data generators (3) and data specification generators [25, 62].

Random test data generators select random test data from the domain of input variables. Structural oriented test data generators are based on covering structural elements in the program {statement, branch, path, def-use}. Data specification generators select test data from program specifications, in order to exercise features of the specifications [31].

Test data are generated to verify correctness of a program, but bad data also can be used. The tester can use the bad data (referred to as ‘worst case’ [82]) to test the error handling procedures in his program.
Generally, software testing techniques are classified into two techniques: static analysis (performed when the program is not actually executing) and dynamic analysis (done when the program is running) [12, 92].

2.4.1. Static Analysis:

Static analysis tools assess the properties of a software product without executing it. Typically, static analysis tools analyze some structural representations of a program to arrive at certain analytical conclusion. Static analyzer carries out an analysis of the source code of the program- its control flow, declared identifiers, etc. and reports on any defects detected, such as type incompatibilities that main complier would let through, use of un-initialized variables, instances of blocks of unreachable code, etc. it may also report on violation of coding or language standards [97, 106]. Code Walk-troughs and Code Inspection [128] can also be considered as static analysis tool; however the term static program analysis is generally denoting automatic static program analysis tools.

Symbolic execution is not the execution of a program in its true sense, but rather the process of assigning expressions to program variables as a path is followed through the code structure. Instead of executing a program on a set of sample inputs, a program is symbolically executed for a set of classes of inputs.
leading to symbolic values which describe the relation between input and output. These results can be checked against the program’s expectations for correctness [73, 92].

Symbolic execution is a popular static analysis tool [30]. They parse the program text and thus recognize the different types of statement in the program. They can detect whether or not statements are well formed, make inferences about the control flow of the program. They complement the error detection facilities provided by the language compiler.

Symbolic execution can be sued to derive test data; however, major practical limitation of the static analysis tools lies in handling dynamic evaluation of memory references at run-time. In higher-level programming languages, array subscripts and pointer variables provide dynamic memory references based on run time computation performed by the program. Static analyzer cannot evaluate pointer values and array subscripts [68].

2.4.2. Dynamic Analysis

As discussed in symbolic execution, the relationship between input data and internal variables for structural test data generation is difficult to analyze statically in the presence of loops and computed storage location such as arrays.
and pointers. Dynamic methods execute the program under test with some input. Dynamic program analysis techniques require the program to be executed and its actual behavior recorded. It solves some problems of symbolic analysis since values of variables are available and array subscripts and pointer values are known at run-time [73]. A dynamic analyzer usually instruments the code of the software to be tested in order to record the behavior of the software for different test cases.

Automated tools enable the test team to capture the state events during execution of a program, by watching and reporting the programs’ behavior. Then it can list the number of times a line of code is executed or a component is called. These statistics tell testers about the statement or path coverage of the test cases [97].

Normally the dynamic analysis result describes the structural coverage achieved for different modules of the program. The output of the dynamic analysis tool can provide evidence that thorough testing has been done [20, 24 and 85]. The dynamic analysis results also help us to comprehend the extent of testing performed in white-box mode. If the testing coverage is not satisfactory, more test cases can be designed and added to the test suit [68].
2.4. 3. Random Testing:

Random testing simply executes the program with random inputs and then observes the program structures executed. Random testing selects arbitrarily test data from the program’s input domain of the problem and then these test data are applied to the program under test [110].

Random number is just a number having a value that cannot be predicted in advance of its existence. Human mind is not very good at generating a complete unrelated set of numbers. Computers using algorithms can generate a sequence of numbers that appear random (pseudo-random numbers). For practical applications pseudo-random numbers suffice.

Humphry [53] states 3 major problems with random test methods: The first of these three main problems is that randomly generated tests rarely have high bug yield. Second, while such tools can usually produce large numbers of random input conditions, they are rarely able to generate the anticipated outputs. Analysis of the test results is thus time-consuming. Finally, the programmer who develops the random generator will also likely fail to consider some key conditions and may thus overlook an entire functional area.
McMinn [73] illustrates that random testing technique works well for simple programs. However, structures that are only executed with a low probability are often not covered. Considering the triangle classifier program (explained in Chapter 4), the equilateral triangle true branch is unlikely to be executed by chance. Even if the domain of integer values for each variable were limited to values between 1 and 100, the probability for three sides of a triangle inputs $a, b, c$, to be all equal and selected with the same value is 1 in 10,000.

Random testing is still used in the unit level, because it is competitive with systematic methods in failure finding [47]. Random testing has an important theoretical and experimental role; it serves as the standard of comparison by which other methods should be evaluated.

Random testing is a form of functional testing that is useful when the time needed to write and run directed tests is too long or the complexity of the problem makes it impossible to test every combination of the input. The conventional view considers random testing as a second class alternative to systematic testing. Systematic testing is preferred because it is directed usually towards exposing failures. However, there are situations where random testing must be chosen [46]. Random testing is useful even if it doesn’t find as many defects per time interval, since it can be performed without manual
intervention. An hour of computer time can be much less expensive than an hour of human time [47].

The comparison of systematic methods with random testing is a useful experience in its own right, since it has the potential for evaluating the systematic method to the random testing standard. It should be the default method by which other systems should be judged [55].

2.5. Software Quality and Reliability

The central goal of software engineering is to develop software products that possess ‘quality’. The notion of software quality may at first seem somewhat difficult to crystallize. However, one approach is to start with the observation that, ultimately, software engineering must generate products that are useful [97]. The implication is that software quality can be articulated in terms of product’s ‘usefulness’.

Some researchers [41, 64, 95] think of quality as achieving for high levels of user satisfaction, portability, maintainability, robustness and fitness for use. Others define quality as high levels of user satisfaction and low defect levels. Software reliability is the probability that a software system will
function without failure under a given environment and during a specified period of time [26].

Garvin (1984) (cited in [97]) has written about how different people perceive quality. He described quality from five different perspectives:

- The transcendental view, where quality is something we can recognize but not define.
- The user view, where quality is fitness for purpose.
- The manufacturing view, where quality is conformance to specification.
- The product view, where quality is connected to inherit product characteristics.
- The value-based view, where quality depends on the amount the customer is willing to pay for it.

Since many of the largest systems that exist (e.g. military applications) could cause catastrophe if they malfunctioned in some unpredictable way. This situation should be contrasted with other large systems (e.g. operating systems) where poor reliability is extremely irksome and annoying but not potentially disastrous. Today, quality and reliability is critical for survival and success. Customers demand it. Business organizations will not succeed unless they produce quality products (software/services).
General issues in software reliability have attracted attention to many researchers [33, 36 and 125].

Product software should fare better in respect of reliability, since it can undergo extensive testing and trial usage before being advertised and sold for general usage [97].

Rallis [96] proposed that sequential testing and correction improves the quality of a given software, and show confidence that no faults remain after several sequential tests.

Writing software is an art as well as a science [95]. During testing the actual results of each input are recorded as weights to provide estimate of reliability.

By comparison of expected and observed actual performance of software, measures of reliability and confidence (e.g. means time to failure MTT and errors per 1,000 lines of code) can be derived [95].
2.6. Complexity Metrics:

The objective of complexity metric is to identify the factors that cause defects to appear. The more complex the code the more difficult it is to understand therefore more difficult to debug and maintain, which in turn implies more defects and lower productivity.

We can use these metrics to identify code that should be used for simplification and, if not for simplification, additional testing [86]. We are going to present two different complexity aspects. Other software measures can be found in [39, 42, 67 and 82].

- **Lines of Code (LOC):**
  
  LOC is a traditional measure for software size. It is a simple measure. However it is not adequate measure for programs complexity. LOC measures the physical length of the software itself, it is a reliable measurement. However, it is always lacking representing the size of functionality of a system. Consequently there is need for some other size measurement as well. [90] Indicates that as a system increases in size, the relationship between defects and size is relatively linear.
Cyclomatic Complexity

The difficulty in software testing stems from the complexity of software; we cannot completely test a program with moderate complexity [88]. The most famous and popular complexity metric, is McCabe’s cyclomatic complexity (CC) [71] which is a measure of the number of control flows within a module. The underlying theory is that the greater the number of paths through a module, the higher the complexity.

McCabe’s metric was originally designed to indicate a module’s testability and understandability. It is based on the classical graph theory, in which you calculate the Cyclomatic number of a graph, denoted by \( V(g) \), by counting the number of linearly independent paths within a program, which allows you also to determine the minimum number of unique tests that must be run to execute every executable statement. Such metrics typically index the number of branches or paths created by the conditional expression within a program [22].

The cyclomatic number can be calculated in two ways, either by counting the nodes and edges of the graph or by counting the number of binary decision points. That is:

\[
V(g) = e - n + 2 \quad \text{................................................................. (2.1)}
\]
Where \( g \) is the control graph of the module, \( e \) is number of edges and \( n \) is the number of nodes. Or:

\[
V(g) = bd + 1 
\]

(2.2)

Where \( bd \) is the number of binary decision in the control graph.

Fig (2-3) Examples of different cyclomatic complexity

Complexity is often considered synonymous with understandability or maintainability [22]. Fig (2-3) gives some examples of different CC. It is expected that modules with higher Cyclomatic Complexity to be more difficult to test and maintain due to their higher complexity and modules with lower
Cyclomatic Complexity to be easier. Cyclomatic Complexity is useful to estimate maintenance effort, and estimate testing effort.

2.7. Software Testing Life Cycle:

Software testing life cycle (STLC), is the sequence of the different phases to be considered for testing software [4,132]. STLC consists of six phases. Table (2-1) shows STLC sequence starting with the test planning phase.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activities</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Planning</strong></td>
<td>Create high level test plan</td>
<td>Test plan, Refined Specification</td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td>Create detailed test plan, Functional Validation Matrix, test cases</td>
<td>Revised Test Plan, Functional validation matrix, test cases</td>
</tr>
<tr>
<td><strong>Design</strong></td>
<td>Test cases are revised; select which test cases to automate</td>
<td>Revised test cases, test data sets, sets, risk assessment sheet</td>
</tr>
<tr>
<td><strong>Construction</strong></td>
<td>Scripting of test cases to automate</td>
<td>Test procedures/Scripts, Drivers, test results, Bug reports.</td>
</tr>
<tr>
<td><strong>Testing cycles</strong></td>
<td>Complete testing cycles</td>
<td>Test results, Bug Reports</td>
</tr>
<tr>
<td><strong>Final testing</strong></td>
<td>Execute remaining stress and performance tests, complete documentation</td>
<td>Test results and different metrics on test efforts</td>
</tr>
<tr>
<td><strong>Post implementation</strong></td>
<td>Evaluate testing processes</td>
<td>Plan for improvement of testing process</td>
</tr>
</tbody>
</table>
Chapter 2: Software Testing

2.7.1. Test Planning

This is the phase where Project Manager has to decide what things need to be tested, do I have the appropriate budget, etc. Naturally proper planning at this stage would greatly reduce the risk of low quality software. This planning will be an ongoing process with no ending point.

Activities at this stage would include preparation of high level test plan-according to IEEE test plan template; the Software Test Plan (STP) is designed to prescribe the scope, approach, resources, and schedule of all testing activities. The plan must identify the items to be tested, the features to be tested, the types of testing to be performed, the personnel responsible for testing, the resources and schedule required to complete testing, and the risks associated with the plan. Almost all of the activities done during this stage are included in this software test plan and revolve around a test plan.

2.7.2. Test Analysis

Once test plan is made and decided upon, next step is to dive little more into the project and decide what types of testing should be carried out at different stages of Software Development Life Cycle (SDLC), do we need or plan to automate, if yes then when the appropriate time to automate is, what type of specific documentation is needed for testing.
Proper and regular meetings should be held between Testing Teams, Project Managers, Development Teams, and Business Analysts to check the progress of things which will give a fair idea of the movement of the project and ensure the completeness of the test plan created in the planning phase, which will further help in enhancing the right testing strategy created earlier. Testers will start creating test case formats and test cases itself. In this stage they also need to develop Functional validation matrix based on Business Requirements to ensure that all system requirements are covered by one or more test cases, identify which test cases to automate, begin reviewing of documentation, i.e. Functional Design, Business Requirements, Product Specifications, Product Externals, etc. they also have to define areas for Stress and Performance testing.

2.7.3. Test Design

In this phase, Test plans and cases which were developed in the analysis phase are revised. Functional validation matrix is also revised and finalized. Also in this stage risk assessment criteria is developed. If testers have thought of automation then they have to select which test cases to automate and begin writing scripts for them. Test data is prepared. Standards for unit testing and pass / fail criteria are defined here. Schedule for testing is revised (if necessary) & finalized and test environment is prepared.
2.7.4. Construction and Verification

In this phase testers have to complete all the test plans, test cases, complete the scripting of the automated test cases, Stress and Performance testing plans needs to be completed. They have to support the development team in their unit testing phase. And obviously bug reporting would be done as when the bugs are found. Integration tests are performed and errors (if any) are reported.

2.7.5. Testing Cycles

Every test should be treated like an experiment to be carefully designed, controlled, and recorded so that it can be reproduced. This requires defining and recording the test environment, the procedures and the test cases. In this phase testers have to complete testing cycles until test cases are executed without errors or a predefined condition is reached.

2.7.6. Final Testing and Implementation

At this stage testers have to execute remaining stress and performance test cases, documentation for testing is completed / updated, provide and complete different matrices for testing. Acceptance, load and recovery testing will also be conducted and the application needs to be verified under production conditions.
2.7.7. Post Implementation

In this phase, the testing process is evaluated and lessons learnt from that testing process are documented. Line of attack to prevent similar problems in future projects is identified. Create plans to improve the processes. The recording of new errors and enhancements are an ongoing process. Cleaning up of test environment is done and test machines are restored to base lines at this stage.

Reliability is clearly related to ‘correctness’ and correctness is the main concern of verification procedures that are carried out during product’s life cycle.

Reliability is a quality measure that relates to the operational correctness of software. Reliability, or lack of it, results from errors giving rise to failures. Failures are caused by errors in operational software that results from defects in requirements, specification, design or coding. However, this research work shall be concerned only with the coding errors.

As a summary, testing is an instrument that can contribute to the quality enhancement of the software products [44]. Testing should be embedded in the overall quality management of an organization. It is a Quality Control (QC) team’s responsibility.
2.8. The Order of Testing (The ‘V’ Model):

The traditional testing activities follow a general specified norms or procedures, as to when each activity must start. The so-called ‘V’ diagram (see fig (2-3) is one of the widely used models in many software projects [119]. It is used quite often in the literature [9]. It explains what the method is about, how to apply the model in testing activity, what needs to be done at each activity.

Every phase of the software testing life cycle (STLC) in this model corresponds to some activity in the software development life cycle (SDLC). As shown in fig (2-3) the domain description analysis will correspondingly have an installation acceptance testing activity at the end. The architecture design has integration testing and so on.

Fig (2-4) The V-Diagram
Conventionally one must first test on units (module) before one can perform test on components (aggregation of modules). And so, one can only do integration test once two or more components have been tested, and only perform system tests once all components have been tested. Once the product is in the field, it is being installation tested, through the daily use of the ultimate customers.

A Comprehensive discussion of software reliability models can be found in [27, 116].

2.9. Conclusion

This chapter gives a brief introduction to software testing. The test strategies: unit, integration, system, acceptance and regression testing have been explained. The chapter presents a discussion of testing techniques, different test adequacy criteria. The description of automatic test data generation methods and random test data generation is also given. Definitions of quality and reliability, software complexity metrics and measures have been presented. The chapter also gives the details of software testing life cycle. In the next chapter, the basics of genetic algorithms GA are presented. Structure, representation of elements and parameters is explained. And how to apply GA as an automatic test data generator is also given.