CHAPTER 1

INTRODUCTION

This chapter gives an overview of the testing, challenges and motivation for the present research and thesis. It describes the problem at hand, the importance of the problem and why it needs to be addressed. It also provides the abstract overview of the contents and layout of the thesis.

1.1 OBJECT-ORIENTED TECHNOLOGY

Object oriented technology is the flavour of the day for software professionals. It is becoming more popular in several different contexts. The object-oriented techniques are applied in the areas of programming languages, databases, user interfaces, specification and design methodologies. As a result, perception towards object oriented software quality is changing very rapidly. Earlier the object oriented technology was considered very powerful without any additional efforts. Many design and analysis technologies [2. 3. 4] assumes that if the object oriented system is well designed then it will need minimum testing efforts. But object orientation is not a silver bullet in the area of software engineering. Inspite of its good characteristics like inheritance, polymorphism, encapsulation, information hiding, the software designed using this technology may also contain errors similar to traditional procedure oriented technology. Additionally the object oriented features introduces new kind of challenges for the software professionals. In the last one decade the main thrust area in the research has been the quality of the software. The researchers have emphasised on how to assess the quality of the software and how to achieve the quality actually. The research in the area of quality assessment have lead to object-oriented metrics [5. 6. 7. 8. 9. 10. 11. 12]. The metrics proposed by researchers indicates about the quality of the components of the system which are more likely to have errors. When the behaviour of certain component i.e. class, group of classes or the system as whole is not satisfactory, there should be certain method to improve it. To achieve a satisfactory level of quality in software generally two approaches are followed:
Introduction

1.1.1 Methodology Improvement

Various techniques and practices are used to improve the overall development process and specific attention is paid to analysis, design and development of the object oriented systems [3, 4, 13]. In these methodologies the main concentration is on the development process and the verification of the system is not given much importance.

1.1.2 Verification Improvement

After the system has been developed, it needs to be verified for the requirements of the user. The idea behind this is that the system developed by human beings may contain some errors, inspite of best efforts by the development team. Before final release of the system these faults should be uncovered and removed using static and dynamic techniques.

1.2 SOFTWARE TESTING TERMINOLOGIES

Software testing [23] is an activity aimed to evaluate an attribute or capability of a program or system and determining that it meets its required results. Due to limited understandability of principles of software, software testing still remains an art for programmers and testers. The difficulty in software testing stems from the complexity of software where a program cannot be completely tested with moderate complexity. Testing is more than just debugging. The purpose of testing can be quality assurance, verification and validation, or reliability estimation [23]. Testing can be used as a generic metric as well. Correctness testing and reliability testing are two major areas of testing. Software testing is a trade-off between budget, time and quality. Listed are some key terminologies that are used in testing.

1.2.1 Error

Whenever a development team member makes a mistake in any phase of SDLC [23], errors are produced. It might be a typographic error, a misleading of specification, a misunderstanding of what a subroutine does, and so on. Error is a very general term used for human mistakes. Thus an error causes a bug and a bug in turn causes failure.

1.2.2 Bug

A software bug may be defined as a problem in the logic of the program which causes its unexpected behaviour. Most of the bugs in software are caused due to human error
or negligence of the programmers. Bugs may not cause erroneous output always. They may go undetected for a long time unless circumstances are right for their detection.

1.2.3 Failure

When the software is tested [23], failure is the first term being used. It means the inability of a system or component to perform a required function according to its specification (in other words, the results or behaviour of system under test are different as compared to specified expectations). Failure is the term which is used to describe the problem in a system on the output side. Fault is a condition that in actual cause a system to produce failure. Fault is a synonymous with the words defect or bug. Therefore, fault is the reason embedded in any phase of SDLC and results in failure. It can be said that failures are the manifestation of bugs. Thus, when a bug is executed then failures are generated.

\[\text{ERROR} \rightarrow \text{BUG} \rightarrow \text{FAILURE}\]

1.3 SOFTWARE TESTING LIFE CYCLE

Software testing life cycle or STLC [24,25] consists of a detailed group of activities related with testing, which specify the details of every action along with the specification of the appropriate time to perform these actions. The testing process cannot be a standardized for all the organizations, however every organization dealing with software development, defines and follows some sort of testing life cycle.

STLC by and large comprises of following sequential phases:

1. Test Planning
2. Test Analysis
3. Test Design
4. Test Creation & Verification
5. Test Execution Cycle.
6. Performance Testing & Documentation
7. Actions after Implementation
1.3.1 Test Planning

Test Planning is entrusted to a senior person like the project manager and he plans & identifies all the areas where testing effort need to be applied, while operating within the boundaries of constraints like resources & budget. Unless judicious planning is done in the beginning, the result can be catastrophic with emergence of a poor quality product, dissatisfying the ultimate customer [24. 25].

Planning is not limited just to the initial phase; rather it is a continuous exercise extending till the end.

During the planning stage, the team of senior level person comes out with an outline of testing plan at high level. The high level test plan comprehensively describes the following [25]:

- Scope of Testing: Defining the areas to be tested, identification of features to be covered during testing.
- Identification of Approaches for Testing: Identification of approaches including types of testing.
- Defining Risk: Identification of different types of risks involved with the decided plan.
- Identification of Resources: Identification of resources like man, materials & machines which need to be deployed during Testing.
- Time Schedule: For performing the decided testing is aimed to deliver the end product as per the commitment made to the customer.

Involvement of software testers begins in the planning phase of the software development life cycles. During the design phase, testers work with developers in determining what aspects of a design are testable and with what parameters those test will work.

1.3.2 Test Analysis

The High Level Test Plan Document is analyzed in detail to further work out the following [24, 25]:

Introduction

- Identification of types of testing to be performed during various stages of Software Development Life Cycle.
- Identification of extent to which automation needs to be done.
- Identification of the time at which automation is to be carried out.
- Identification of documentation required for automated Testing.

The Software project cannot be successful unless there is a frequent interaction among various teams involved in Coding & Testing with the active involvement of the project managers, business analysts or even the customer [26, 27]. Any deficiencies in the decided test plans come to the surface, during such meetings of cross-functional teams. This provides an opportunity to have a rethinking & refining the strategies decided for testing.

Based upon the customer requirements a detailed matrix for functional validation is prepared to cover the following areas [27]:

- To ensure that each & every business requirement is getting covered through some test case or the other.
- To identify the test cases best suited to the automated testing.
- To identify the areas to be covered for performance testing & stress testing.
- To carry out detailed review of documentation covering areas like customer requirements, product features & specifications and functional design etc.

1.3.3 Test Design

This phase involves the following [26]:

- Further polishing of various test cases, test plans
- Revision & finalization of matrix for functional validation.
- Finalization of risk assessment methodologies.
- Identification of test cases suitable for automation.
- Preparation of test data.
- Establishing unit testing standards including defining acceptance criteria.
- Revision & finalization of testing environment.
1.3.4 **Test Creation & Verification**

This phase involves the following [26]:

- Finalization of test plans and test cases.
- Completion of script creation for test cases decided for automation.
- Completion of test plans for performance testing & stress testing.
- Bug logging in bug repository & preparation of detailed bug report.
- Performing integration testing followed by reporting of defects detected if any.

1.3.5 **Test Execution Cycle**

This phase involves the following [26]:

- Completion of test cycles by executing all the test cases till a predefined stage reaches or a stage of no detection of any more errors reach.
- This is an iterative process involving execution of test cases, detection of bugs, bugs reporting, modification of test cases if felt necessary, fixing of bugs by the developers & finally repeating the testing cycles.

1.3.6 **Performance Testing & Documentation**

This phase involves the following [27]:

- Execution of test cases pertaining to performance testing & stress testing.
- Revision & finalization of test documentation.
- Performing Acceptance testing, load testing followed by simulating conditions of actual usage.
- Verification of the software application by simulating conditions of actual usage.

1.3.7 **Actions After Implementation**

This phase involves the following [26]:

- Evaluation of the entire process of testing.
- Documentation of TGR (Things Gone Right) & TGW (Things Gone Wrong) reports. Identification of approaches to be followed in the event of occurrence of similar defects & problem in the future.
## Table 1.1 Software Testing Life Cycle (STLC)

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activities</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planning of Tests</td>
<td>Creation of a High Level Test Plan</td>
<td>Refined Test Plan and Specifications</td>
</tr>
<tr>
<td>Analysis of Tests</td>
<td>Creation of fully descriptive test plan</td>
<td>Refined Tests Plans, Test Cases and Matrix for Functional Validation</td>
</tr>
<tr>
<td></td>
<td>Creation of matrix for functional validation</td>
<td></td>
</tr>
<tr>
<td>Designing of Tests</td>
<td>Revision of Test Cases</td>
<td>Refined Tests Cases, Input Data Sets and Documents for Assessment</td>
</tr>
<tr>
<td></td>
<td>Selection of Test Cases fit for automation</td>
<td></td>
</tr>
<tr>
<td>Creation and</td>
<td>Creation of scripts suitable for Test Cases for</td>
<td>Detailed Procedures for Testing, Testing Scripts, Test Reports and Bug-Reports</td>
</tr>
<tr>
<td>Verification of</td>
<td>automation</td>
<td></td>
</tr>
<tr>
<td>Tests</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution of Testing Cycles</td>
<td>Completion of cycles of testing</td>
<td>Detailed test reports and Bug-Reports</td>
</tr>
<tr>
<td>Performance Testing, Documentation</td>
<td>Execution of test cases related to performance tests and stress testing</td>
<td>Test reports, documents on various metrics used during testing</td>
</tr>
<tr>
<td>Actions after</td>
<td>Evaluation of all Processes of Testing</td>
<td>Detailed plans for improving the process of testing</td>
</tr>
<tr>
<td>implementation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 1.4 SOFTWARE TESTING METRICS LIFE CYCLE

The process of setting up testing metrics involves:

- Recognizing the metric.
- Prioritizing metrics.
- Classifying metrics that may be project specific.
- Identifying the required data for the metric and if data is not available, identify/setup process to capture the data.
- Communicating the stakeholders.
- Capturing and verifying data.
- Analyzing and processing data.
1.5 ADVANTAGES OF SOFTWARE METRICS

Metrics help the software professionals in many ways like.

- In comparative study of various design methodology of software systems.
- For analysis, comparison and critical study of various programming language with respect to their characteristics.
- In the preparation of software quality specifications.
- In the verification of compliance of software systems requirements and specifications.
- In making inference about the effort to be put in the design and development of the software systems.
• In getting an idea about the complexity of the code.
• In taking decisions regarding further division of complex module is to be done or not.
• In providing guidance to resource manager for their proper utilization.
• In comparison and making design tradeoffs between software development and maintenance cost.
• In providing feedback to software managers about the progress and quality during various phases of software development life cycle.
• In allocation of testing resources for testing the code
• Test metrics data collection helps predict the long term direction and scope for an organization and enables a more holistic view of business and identifies high level goals.
• Provides a basis for estimation and facilitates planning for closure of the performance gap.
• Provide a means for control/status reporting.
• Identifies risk areas that require more testing.
• Quickly identifies and help resolve potential problems and identifies areas of improvement.
• Testing effectiveness and efficiency are provided by the test metrics.

1.6 LIMITATIONS OF SOFTWARE METRICS

There are certain shortcomings of the software metrics, like
• The application of software metrics is not always easy and in some cases it is difficult and costly.
• The verification and justification of software metrics is based on historical/empirical data whose validity is difficult to verify.
• These are useful for managing the software products but not for evaluating performance of the technical staff.
The definition and derivation of Software metrics is generally based on assuming which are not standardized and may depend upon tools available and working environment.

Most of the predictive models rely on estimates of certain variables which are often not known exactly.

1.7 NEED FOR SOFTWARE TESTING

The quality and reliability of software depends on its testing. If testing is done in proper way then there is a high probability for achieving better quality software. Here are some important defects that the testing would have found [23. 28. 29].

1. In October 1999[34], the $125 million NASA mars climate orbiter-an interplanetary weather satellite-was lost in space due to the data conversion error. Investigators discovered that the software on the craft performed certain calculation in English unit (yards) when it should have metric unit (meters).

2. In July 2001[35], a “serious flaw” was found in off- the shell that had long been used in system for tracking U.S. nuclear materials. The software had currently been donated to another country & scientist in that country discovered the problem and told U.S. official about it.

3. In February 2003 [35], the U.S. treasury department mailed 50,000 social security checks without a beneficiary name. A spoke person said that the missing names were due to a software program maintenance error.

1.8 TESTING LEVELS

Testing can be performed at different levels depending on the nitty-gritty of the code we are testing. Each level of testing presents specific problems that require specific techniques and strategies. It is possible to identify the following different levels:

1.8.1 Unit Testing

At this level of testing the main concentration is on testing of a single program unit, where the term unit can be any entity depending on the specific environment. A unit can be a single procedure or module. The main characteristic of unit testing is that, it
is usually carried on by the programmer that actually developed the code and thus has a complete visibility and maximum insight on the code.

Due to the maximum level of understandability of the software at this level, the selection of test data exercising particular elements of the code is usually much simpler in this case than advanced testing stages.

The major problem with unit testing is the requirement of the drivers, stubs, and oracles which allow actual execution of single units in isolation and that can be a very complex task. In particular, in the case of object-oriented systems the construction of drivers and stubs requires the simulation of classes which has not been implemented yet. It may be very difficult to provide complex functionalities, behaviour and interactions among simulated classes as if they are actual.

1.8.2 Integration Testing

Integration Testing is a systematic technique for constructing software architecture while at same time conducting tests to uncover errors associated within interfacing [38]. The unit tested components are integrated and a program structure is built that is as per the design document. With the testing of individual units the local faults may be removed. But unit testing does not exercise the interactions among different units. All the actual modules are pulled together to watch their interaction with each other during integration testing. The main activity is to involve different interacting units which have been developed by different programmers or teams. In this case the code is still visible, but with a higher granularity.

During integration testing many faults can be revealed like, interface problems, missing functionalities, and unexpected side-effects of method invocation.

Some problems are caused by a particular programming language only. Before choosing an integration testing strategy, it is thus very important to take into account the class of problems the test must address. For example, when using a strongly typed language is used for coding, many different interface errors as the ones related to the wrong type of parameters in a procedure call can statically be identified and removed.

In Integration testing most important issue is the choice of an integration order. i.e., the order in which the different units, or modules, are integrated.
Based on the order of integration, integration strategies are generally classified as under:

1.8.2.1 Top-Down Integration

It is an incremental approach, in which the integration begins with the main control module in the hierarchy defined by the use relation among modules, i.e., it starts with the module that is not used by any other module in the system. The other modules are then added to the system incrementally, following the use hierarchy. In this method, drivers are not required, but complex stubs are needed.

1.8.2.2 Bottom-up Integration

Bottom-up integration begins with construction and testing of individual modules with the modules that do not use any other module in the system, and continues by incrementally adding modules that are using already tested modules. In this method, complex drivers are needed but stubs are not required.

1.8.2.3 Big-Bang Integration

If the testers do not want to construct the drivers and stubs, then they can follow the big-bang integration order, where all modules are integrated at once. Although this approach is simpler as compared to other two approaches, it has some drawbacks also. Firstly, identification and removal of faults becomes very difficult when dealing with the entire system instead of subsystems. Secondly, the achieved degree of testing of the code is lower with respect to the two alternative approaches, where modules composing the incrementally growing subsystems are tested many times during each integration step.

1.8.3 System Testing

In system testing, the system is tested as a whole. It is performed on an invisible code, due to both accessibility and complexity reasons. In system testing all those issues are dealt which cannot be dealt with individual modules or components of the system. At this level the software behaviour is compared with the expected one according to the specifications. For example the system is load tested, in which the software under test is tested for its robustness to handle a load of work bigger than expected and its behaviour under that situation is studied. The system testing consists of following activities:
• Recovery Testing
• Security Testing
• Stress Testing
• Performance Testing

1.8.4 Acceptance Testing

Acceptance testing is also known as validation testing. The end-user performs all the activities on the system under test to verify whether the system is behaving as it was intended to. This level is characterized by the absence of visibility of specification and design documents. The source code is not provided to the tester.

1.8.5 Regression Testing

During integration of the modules, each time a new module is integrated with the already tested components, the behaviour of the system changes. It needs to be re-tested again. In regression testing some already executed tests are re-executed to ensure that the changes in the system have not introduced some new faults in the system. Regression testing is also performed during maintenance.

1.8.6 Smoke Testing

Smoke testing is another integration test strategy generally used during development stage. It is used as a pacing mechanism in time-critical systems. It allows the development team to assess their progress time to time. The modules which have been coded are integrated into a build. This build includes all the files, modules, libraries which are required to implement some function. Some tests are designed to find the errors due to which the build is not performing its function properly. The present build is then integrated with other builds and the whole product is tested on daily basis.

1.9 TESTING TECHNIQUES

There are three different sources for the tests derivation namely specifications, code, and fault models. The testing techniques are also categorized under three categories. These categories are not mutually exclusive but they are complementary to one another. A technique can be more effective in revealing a specific class of errors, i.e.,
there exist errors which can only be revealed by one technique and might remain unnoticed using the other ones. In addition, there may be cases such that one of the techniques is not applicable. In this section these two techniques of testing, Black Box and White Box techniques are discussed briefly.

1.9.1 Black Box Testing

Black box testing is also known as functional testing. These are based on software specifications, like requirements specification (for system testing), design specifications (for integration testing), and detailed module specifications (for unit testing). Functional specifications of the software are analyzed to derive test data and abstract elements to be covered during are identified. There some techniques, that can be used to map the specification entities to program entities directly. Following are the commonly used black box testing techniques:

1.9.1.1 Equivalence Partitioning

The testing strategy may be more effective if the tests are distributed evenly over the input domain rather than being concentrated on a particular area. this is the basic idea behind equivalence partitioning. Equivalent data defines a partition of the domain of the program. For the test data to be effective, it is better to chose inputs from each partition, than to choose inputs from only one partition.

1.9.1.2 Boundary Value Analysis

Many errors occur at the boundaries of the input domain rather than its centre. Boundary analysis is a special case of the equivalence partitioning technique. Values which lie at the edge of an equivalence partition are called boundary values. The basic idea of this approach is that it is very common that the module of the software under test may fail at the boundaries of the input domain. So the tests consisting of the boundary values are very likely to uncover certain errors.

1.9.2 White-Box Testing

White box testing is based on the internal structure of the program under test. This is also called structural testing or glass box testing. For the program under test, the priority of the tester is to identify the test data that will cover program’s entire structure. The test cases are designed in such a manner that all independent paths within a module must the exercised atleast once. all logical decisions are exercised
Introduction

for their true or false values, all loops are executed at their boundaries and internal data structures are exercised at least once to ensure their validity. Many structural metrics have been defined by different researchers to measure its structural complexity. The idea here is that the particular function or module may be called error free only if it has been executed on almost every class of inputs. The structural testing criteria may be classified under following two categories

1.9.2.1 Control-Flow Based

All the programs can be modeled as control-flow graphs in which the nodes represent the regions of the programs in which the entry and exit points are clearly defined and edges define the flow of control among these nodes. In control flow based testing, test cases are derived to exercise the basis set to execute every statement in the program at least once during testing.

1.9.2.2 Data-Flow Based

In data flow testing, the tester selects test paths of a program according to the locations of the definitions and uses of variables in the program. The specific paths to be tested are selected according to some particular permutations of the operations of operations on the data of the program. For each definition of variable the set of reachable uses is identified. Each definition-use pair identifies one or more paths containing both the definition and the related use. By selecting different subsets of all the possible combinations of definitions and uses. some criteria may be defined and thus different set of execution paths to be covered by the corresponding test data may also be identified.

There are certain problems as far which are left unsolved by structural testing. some of which are as under:

Scalability: Structural testing techniques are most suitable for unit and integration testing, but they can’t be used in system testing. In addition, some criteria may become impractical in the inter-procedural case. e.g., criteria that require the traversal of specific paths comprising procedure calls.

Instrumentality: Every time a structural testing technique is applied, it is applied from scratch. They are not applied in an incremental fashion. Due to this the small modifications in the large programs proves to be very time and resource consuming especially during regression testing.
Path-feasibility: There are certain paths that are present in the control-flow representation of the program, but are not actually executable, irrespective of the input data are used. It is very difficult to identify such paths in the program. So it becomes next to impossible to achieve desired level of coverage, which is dream of every tester. Due to the presence of these infeasible paths the evaluation of the actual coverage achieved by a given test set becomes very difficult and some approximate value has to be accepted.

1.10 TESTING TYPES

1.10.1 Dynamic Software Testing

Dynamic testing (or dynamic analysis) is a term used in software engineering to describe the testing of the dynamic behavior of code. That is, dynamic analysis refers to the examination of the physical response from the system to variables that are not constant and change with time. In dynamic testing the software must actually be compiled and run. Actually Dynamic Testing involves working with the software, giving input values and checking if the output is as expected. These are the Validation activities. Unit Tests, Integration Tests, System Tests and Acceptance Tests are few of the Dynamic Testing methodologies. Dynamic testing means testing based on specific test cases by execution of the test object or running programs.

When the piece of software under test is executed repeatedly to gain confidence that its runtime behavior is acceptable and it is error free and it can now be released as a final product. It is utmost important for any application to provide a high degree of applicability and user confidence and tested thoroughly to remove last bugs. A good testability measure should provide criteria that how much efforts should be put into testing an application and what portion of the overall budget for the project should be devoted for this purpose. Dynamic testing is used to test software through executing it, which can be done manually by a tester or automatically by the help of tools. This is in contrast to Static testing.

Dynamic Testing is generally categorized as under:

1. Specification Based.
2. Experience Based.
3. Structure Based
1.10.2 Static Software Testing

Under Static Testing code is not executed. Rather the code is checked manually, requirement documents, and design documents are examined to find errors. The main objective of this testing is to improve the quality of software products by finding errors in early stages of the development cycle. This testing is also called as non-execution technique or verification testing.

Static testing involves manual or automated reviews of the documents. This review is done during initial phase of testing to catch defect early in STLC. It examines work documents and provides review comments.

Work document can consist of the following:

- Requirement specifications
- Design document
- Source Code
- Test Plans
- Test Cases
- Test Scripts
- Help or User document

Various activities during static testing can be summarized as below:

- **Informal Reviews:** This is one of the types of review which doesn’t follow any process to find errors in the document. Under this technique, the document is just reviewed and the reviewer gives informal comments on it.

- **Technical Reviews:** A team consisting of peers review the technical specification of the software product and checks whether it is suitable for the project. They try to find any discrepancies in the specifications and standards followed. This review concentrates mainly on the technical document related to the software such as Test Strategy, Test Plan and requirement specification documents.

- **Walkthrough:** The author of the work product explains the product to his team. Participants can ask questions if any. Meeting is led by the author. Scribe makes note of review comments.
• **Inspection:** The main purpose is to find defects and meeting is led by trained moderator. This review is a formal type of review where it follows strict process to find the defects. Reviewers have checklist to review the work products. They record the defect and inform the participants to rectify those errors.

• **Static code Review:** This is systematic review of the software source code without executing the code. It checks the syntax of the code, coding standards, code optimization, etc. This is also termed as white box testing. This review can be done at any point during development.

![Testing Diagram](attachment:image.png)

**Figure 1.2 Testing Types**

### 1.11 TESTING PROCESS VS PRODUCT METRICS

Morgan et al. [22] showed that testing process measures do a better job of predicting fault detection effectiveness than product measures. Example of testing process measures includes test suite size & various coverage measures such as block, decision, and all-uses coverage. Example of product measures includes line of code and total count of blocks, decisions, and all-uses. Since testing process measures are more widely used in research as criteria than product measure, this work suggests that researchers are justified in doing this. Additionally, Morgan et al. also suggest the incorporating both testing process measures & product measures together can improve the chances of increasing fault detection effectiveness of suites. The work cautions.
however, that using either type of measure still does not allow researchers to predict fault detection effectiveness to a high degree.

1.12 METRICS COMPUTATION AND FAULT DETECTION FOR UML MODELS

Kawane [21] presents a brief case study comparing the fault detection effectiveness of various test adequacy criteria in the realm of UML design models. UML models are a formal specification-based language used by developers to model complex software systems. A GUI based tool called MagicDraw was used to compute the metric values of the software by applying backward engineering concept. Using this tool the UML model elements (class diagram & interaction diagrams) can be easily drawn for the software under consideration and the metric values for object oriented software can be easily computed. These values can be correlated with the actual experimental results. Other faults may be indicated by one of the following: a violation of constraint on the system operations, an inconsistent system configuration, or a deviation in system behavior from the specification in the use cases. The result of this work do not easily generalize due to the limited experimental setup and relatively small subject but the work is unique in that it applies the notion of metrics computation for object oriented software and can be utilized in the area of UML models.

1.13 WHEN TO STOP TESTING

This can be difficult to determine. Many modern software applications are so complex, and run in such as interdependent environment, that complete testing can never be done. “When to stop testing” is one of the most difficult questions to a test engineer. Common factors in deciding when to stop are:

- Deadlines (release deadlines, testing deadlines.)
- Test cases completed with certain percentages passed
- Test budget depleted
- Coverage of code/functionality/requirements reaches a specified point
- The rate at which Bugs can be found is too small
- Beta or Alpha Testing period ends
- The risk in the project is under acceptable limit.
Practically, we feel that the decision of stopping testing is based on the level of the risk acceptable to the management. As testing is a never ending process we can never assume that 100% testing has been done. We can only minimize the risk of shipping the product to client with X testing done. The risk can be measured by Risk analysis but for small duration / low budget / low resources project, risk can be deduced by simply:

- Measuring Test Coverage
- Number of test cycles
- Number of high priority bugs

**1.14 TESTABILITY**

**1.14.1 Definition**

Testability may be defined as the characteristics of a software artifact that decides with what ease the test goals can be achieved. We take the definition of testability as under:

“The degree to which a software artifact supports testing in a given test context”

Some more definitions of testability:

The relative ease and expense of revealing software faults [1].

IEEE standard glossary of software engineering [123] gives following definition of testability:

(a) The degree to which a system or component facilitates the establishment of test criteria and performance of tests to determine whether those criteria have been met, and

(b) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met [IEEE90].

ISO 9126-2 [122] gives the following definition of testability:

“A Set of efforts that bear on the effort needed for validating the modified software”.

---

*Introduction*
Bernd Kahlbrandt [36] has proposed that:

A software system is testable if 1) its components can be tested separately. 2) test cases can be identified in a systematic manner and repeated, and 3) the test results can be observed.

The below mentioned points are also relevant to our definition:

- It is not only the code or the executable portion of the software that effects testing but other features also effect testing i.e. SRS, design document etc.

- Whether a particular software artifact facilitates testing or not may be determined relative to the test context and it is affected by the given test goals, available resources, methodology used, technological support and tool support.

1.14.2 Motivation

Improved testability of the software makes it more reliable and cost effective and helps in continuous improvement of the software.

- Today each and every field of human life such as business, education, travelling and governance to name a few are dependent on software systems and are affected by the software in one form or another. But unfortunately the software development is an error prone process and testing is used to find and remove these errors.

- As software is developed by the human beings for the human beings and the human behaviour is unpredictable, so the requirements keep on changing. So the software developers should keep this thing in mind that requirements may change anytime and the same should be accommodated accordingly. The system being developed should be able to adapt according to new requirements and it should be upgradable. The new functionality added to the system should not hamper the existing functionality. Regression testing is one technique that ensures that existing functionality remains intact after incorporating changes.

- Although number of metrics are available in literature, but in absence of any criteria it is very difficult to choose the relevant set of metrics for a particular project. This gap gave us an idea to work upon ranking of metrics for object oriented software.
1.14.3 Testability of Software Components

The testability of software components (modules, classes) is determined by factors such as:

- **Controllability**: The degree to which it is possible to control the state of the component under test (CUT) as required for testing.
- **Observability**: The degree to which it is possible to observe (intermediate and final) test results.
- **Isolateability**: The degree to which the component under test (CUT) can be tested in isolation.
- **Separation of concerns**: The degree to which the component under test has a single, well defined responsibility.
- **Understandability**: The degree to which the component under test is documented or self-explaining.
- **Automatability**: The degree to which it is possible to automate testing of the component under test.
- **Heterogeneity**: The degree to which the use of diverse technologies requires to use diverse test methods and tools in parallel.

1.14.4 Factors Affecting Testability of Software

Software testability is dependent generally on six factors:-

1. Representational features of software under consideration
2. Implementational features
3. Built in test capability
4. Test suite structure
5. Support environment
6. The process in which testing is performed

The measure of testability of a particular system is decided by controlling its inputs and observing output. If inputs can not be controlled we can never be sure about the outputs to be produced by the software. If inputs are controllable and outputs are observable, we can be sure about the compliance.
Introduction

Although object oriented approach shares some common features with traditional procedure oriented approach, it introduces some new challenges for the software professionals as far as the issue of testing of these systems is concerned. Undecidability issues arises due to polymorphism and dynamic binding, inheritance causes the issue of incremental testing and encapsulation raises the visibility problems. The overall layout of an object oriented system is always different from traditional systems. In object oriented systems, the methods or member functions are usually small. These methods can be unit tested very easily. The main issue is of interaction among different objects and classes. Due to this the testing of object oriented systems is always more time and resource consuming. Several techniques have been proposed by the researchers for testing the object oriented systems. Most of work done in this area deals with the static behaviour of the system. This thesis proposes a new strategy to incorporate testability early in the software development life cycle. The main focus of our approach is to find new measures to calculate the overall testability of the system by considering the coupling (dependency) among classes in presence of object oriented features like polymorphism, inheritance and information hiding etc. The approach adopted is based upon the theory that there are some feature in the field of software engineering which are common to both traditional procedure oriented approach and certain features are specific to object oriented systems.

1.15 CONTENTS OF THE THESIS

The thesis is organized into seven chapters followed by list of references for the development and verification of the methodology discussed in the thesis. The measures and metrics developed in the chapters have been validated and applied on
Introduction

some projects to demonstrate their worth. Brief description of the contents of each chapter is as follows:

**Chapter-1 Introduction**

This is an introductory chapter. Basic concepts of software testing, terminology, need of software testing, software testability, software metrics, software metrics life cycle, advantages and disadvantages of metrics are discussed briefly. A basic overview of the software testing metrics along with their life cycle is presented.

**Chapter-2 Object Oriented Systems**

In this chapter the features of object oriented systems which differentiates them from other traditional software systems are discussed. The central theme of object oriented systems is the concept of object. An object can be regarded as abstract representation of a real-world entity or a conceptual element. The object is characterized by its properties called state, identity and services. Other object oriented features like classes, relationships, methods and attributes are also discussed. The other characteristics of object oriented systems which affect their testability and the testing process like information hiding, inheritance, and polymorphism are also explained briefly. Couplings or dependencies and cohesion among classes are also discussed.

**Chapter-3 Literature Review**

In this chapter the summary of the exhaustive literature survey done by us has been presented. The work done by the researchers has been studied thoroughly and the brief outline of the hypotheses, methods, metrics or approaches proposed by the researchers has been presented.

**Chapter-4 ATC: A Testability Metric for Object Oriented Software**

In this chapter, a new testability metric called “Average Temporal Complexity: ATC” has been proposed for object oriented systems. The value of ATC can be computed using the Sugeno Fuzzy Inference system. The proposed system is provided with three inputs and it produces one output depending on the current value of the three input parameters. The proposed metric ATC has been validated using the well established Weyuer’s criteria. ATC satisfies all the properties given by Weyuker for software metrics.
Chapter-5 RTM: Relation Based Testability Metric for Object Oriented Software

In this chapter, an algorithm has been proposed to compute the overall complexity of an object oriented system. An object oriented system is generally designed following a modular approach. At the module and class level we have to consider the object oriented features of the system like inheritance, cohesion, coupling etc. When we reach to the method level then the software complexity metrics like Cyclomatic Complexity used for procedural software can be used to compute the complexity of a method. In our approach we have explored the same idea and a new metric called “Relation Based Testability Metric: RTM” have been proposed and an algorithm to compute the value of RTM has also been proposed.

Chapter-6 Ranking of Software Metrics Based On Expert Opinion Aggregation

In this chapter, distance based approximation method has been used to rank the existing as well as newly proposed metrics for object oriented software. For the ranking purpose feedback from forty experts working in various firms dealing in the field of software and information technology and some educational intuitions located in National Capital Region Delhi have been taken. The Experts were given a questionnaire consisting of details of the twenty metrics used in object oriented software development and four criteria (seventeen sub- criteria). The aggregated ranking given by the experts has been used to find the overall ranking of the metrics using the DBA method.

Chapter-7 Conclusion and Future Scope

In this chapter concluding remarks and directions for future work has been suggested.