Chapter 1

Introduction

The increasing importance of software applications in our daily lives has emphasized on the development of the software which is reliable. For the software to be reliable the first and foremost requirement is to test it adequately. Testing is the process of establishing confidence that a program or system does what it is supposed to do [2, 30, 64, and 105]. However, software testing is one of the most expensive phases of the software development as it is estimated that in between 40% - 70% of the effort of software development is devoted to the testing of the software [26, and 55]. Therefore, there is a need to make the process of testing easier and more effective. This can be achieved by two ways: firstly, by developing the testing techniques which ease the testing process and secondly, by improving the testability of the software.

Testing object oriented software is different from testing the procedural software due to the various object oriented features like abstraction, encapsulation, inheritance, polymorphism, exception handling etc. There are a number of object oriented testing techniques [1, 45, 51, 79, 82, 98, 100, 107, 111, 112, 116, and 146] some of which are an adaptation of the procedural testing techniques and some are pure object oriented testing techniques. The object oriented testing techniques have been developed at various levels of testing. Traditional levels of testing defined for procedural programs do not fit well in case of object oriented software. The main difference between the testing levels of object oriented software and the procedural
software comes from the fact that the smallest unit of testing in object oriented software is different from the one in the procedural software. A class is the smallest unit of testing in object oriented software whereas in procedural software, it is a procedure or a function. Various researchers have proposed different levels of testing for object oriented software. The most commonly used levels are: method level, class level, cluster level, and system level. A number of object oriented testing techniques are developed which are based on various concepts like data flow analysis, algebraic specification, evolutionary algorithms, data bindings, and the UML models etc. Still there is a need to develop the object oriented testing techniques which improve the software testability. Software contracts are known to improve the testability of the software. A software contract [21] is a specification to prove that a system is correct or not. It has three components: precondition of the methods, postcondition of the methods of a class; and a class invariant for the class. A precondition is a condition which should be true for the execution of the method of a class. A postcondition is a condition which should be true after the execution of the method of a class. A class invariant is a condition which should be true after the execution of the method if it was satisfied before the execution of the method of the class. The testing techniques developed in this thesis are based on the concept of software contracts which leads to the reduced testing effort and increased software testability.

Also, there is a need to make an assessment of the software testability so that it can be improved. Object oriented measures have been used in the literature to predict various software quality attributes like reliability [5], fault proneness [121, 122, and 150], maintainability [65, 95] etc. Software testability is also a software
quality attribute which is part of the software maintainability attribute in the software quality model. The work in this thesis uses various statistical techniques to assess the testability of the object oriented software using open source java software.

The work in this thesis emphasizes on: 1) developing the object oriented testing techniques which improve the testability of the software by reducing the testing effort, and 2) to assess the testability of object oriented software using various object oriented design measures through empirical studies using large open source software written in java. The assessment of testability through OO design metrics will help the software practitioners to plan the testing activities in advance, leading to a reduced cost of testing. Also, it will help in planning and executing the testing activities by paying more attention to the classes with low testability. These classes will need extra attention during the remainder of the development. Also, if required, developers can reconsider the design of these classes and take corrective actions. The remainder of this Chapter is devoted to a brief introduction of the field and disciplines related to this dissertation.

1.1 Software Testing

Testing is the process of executing a program with the intent of finding errors [2, and 106]. Testing is performed to make a judgment about the software quality [113]. The testing process comprises of the following two processes [26]:

a) Verification and

b) Validation
• Verification: The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [50].

• Validation: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements. [50]

Some terminologies used in software testing are [2]: Error, Mistake, Bug, Fault, and Failure. People make errors and a good synonym for an error is mistake. This may be a syntax error or misunderstanding of the specifications. Sometimes, there are logical errors. When developers make mistakes while coding, these mistakes are called “bugs”. A fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault. A failure occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised. A test case describes an input description and an expected output description. Test and Test case terms are used interchangeably. In practice, both are same and are treated as synonyms. The set of test cases is called a test suite. Hence, any combination of test cases may generate a test suite.

Testing is often divided into two types: black box testing and white box testing. Black box testing or functional testing is a strategy in which testing is based solely on the requirements and specifications. Unlike its complement, white box testing, black box testing requires no knowledge of the internal paths, structure, or implementation of the software under test. There are a number of functional testing
techniques like: Boundary Value Analysis, Equivalence Class Testing, Decision Table Based Testing, and Cause Effect Graphing Technique etc. White box testing is a strategy in which testing is based on the internal paths, structure, and implementation of the software under test. White box testing generally requires programming skills. There are a number of white box testing techniques like: path testing, data flow testing etc.

Levels of Testing: The various levels of testing software are as follows.

Unit Testing - A unit is the "smallest" piece of software that a developer creates. It is typically the work of one programmer and is stored in a single disk file. Different programming languages have different units: in C++ and Java the unit is the class; in C the unit is the function; in less structured languages like Basic and COBOL the unit may be the entire program.

Integration Testing - In integration we assemble units together into subsystems and finally into systems. It is possible for units to function perfectly in isolation but to fail when integrated.

System Testing - A system consists of all of the software (and possibly hardware, user manuals, training materials, etc.) that make up the product delivered to the customer. System testing focuses on the defects that arise at this highest level of integration. Typically, system testing includes many types of testing: functionality, usability, security, internationalization and localization, reliability and availability, capacity, performance, backup and recovery, portability, and many more.

Acceptance Testing - Acceptance testing is defined as that testing, which when completed successfully, will result in the customer accepting the software.
1.2 Object Oriented Paradigm

Object oriented paradigm was evolved to remove some of the problems encountered in the procedural software. The most important concepts in this paradigm are: objects, encapsulation, information hiding, inheritance, and polymorphism. A brief description of some of the basic features of this paradigm is provided below [10]:

**Object:** Objects are the building blocks of object oriented paradigm. An object has three characteristics: a name (identity of the object), a state (represented by the attribute value), and behaviour (methods). An object communicates with another object by passing messages.

**Classes:** A class is a template for building the objects. A class can be instantiated to get any number of objects.

**Data abstraction and encapsulation:** The concept of wrapping up of data and functions into a single unit called class is called encapsulation. Through this property of object oriented paradigm the data is not accessible to the outside world. Only the functions of the class can access the data of the class.

Data Abstraction is the representation of all the essential features of an object, without including the background details [10]. Creation of new data types using the encapsulated items which are suitable to an application to be programmed is known as data abstraction. Classes use the concept of the abstraction and they encapsulate all the essential properties of the objects. Attributes are called data members and functions which operate on these data members are called methods.

**Inheritance:** Inheritance is a process through which one object can acquire the properties of another. It allows the declaration and implementation of one class to be
based on an already existing class. The concept of inheritance helps in reusability which means that we can add additional properties to an already existing class without modifying it.

**Polymorphism:** Polymorphism means having many forms. An operation may show different behaviors in different instances depending upon the type of data used in the operation. Polymorphism enables objects having different internal structures to share the same external interface. The feature of polymorphism helps in implementing the inheritance.

**Dynamic Binding:** Dynamic binding (late binding) means that the code associated with a given procedure call is not known until its call at the run time. This feature is related to polymorphism and inheritance.

**Message passing:** The objects in object oriented software communicate with each other by passing messages to each other.

### 1.3 Object Oriented Testing

Object oriented software is more difficult to test than the procedural software [113]. Object oriented testing is different from the procedural testing, because of the various concepts of the object oriented paradigm. Traditional unit and integration levels defined for procedural programs do not fit well in case of object oriented systems.

The main difference between the testing levels of object oriented software and procedural software comes from the fact that the smallest unit of testing in object-oriented software is different from the one in the procedural software. The smallest
unit of testing in object oriented software is class whereas, in procedural software, it is a procedure or a function. Testing levels as defined by Orso [3] are as follows:

**Basic unit testing:** It is defined as the testing of a single method of the class and is called intra-method testing.

**Unit testing:** It is defined as the testing of a class and is called intra-class testing.

**Integration testing:** It is defined as the testing of the interaction among classes and is called inter-class testing.

**Regression testing:** It is defined as the testing of the system when some change has been made to one or more classes.

Object oriented software has different concepts like object, class, inheritance, encapsulation, polymorphism, dynamic binding etc. and they introduce different kinds of problems for testing these systems [3, 41, 99, 109, and 123]. For example one of the testing problems caused by inheritance is that, the inherited methods must be retested in the context of child class. A number of testing techniques have been given by various researchers to test the object oriented software. Some of the techniques are only an adaptation of the procedural testing techniques to the object oriented software and some are pure object oriented testing techniques. In addition to the two testing styles of procedural software: white box and black box testing, there is another testing style applicable to object oriented software and is called state based testing [101]. State based testing focuses on testing the state dependent behaviour of the objects. State based testing tests the state of an object throughout the life cycle of an object. A detailed description of the object oriented testing problems, object
oriented testing levels, and object oriented testing techniques is presented in Chapter 2.

1.4 Software Testability

Software testability is one of the important characteristics of the software quality. It shows the effect of the structure and the semantics of the software on the testing effectiveness, following a given criterion [153]. The quality of the software also depends upon the software testing. Software testability has been defined by various researchers with different points of view for procedural and object oriented software. Software testability is defined at various phases of the software development life cycle: requirement phase, design phase, and the implementation phase. A brief description of a few definitions of software testability is described in this section. A detailed description is given in Chapter 2. Software testability is defined as the probability that the software will fail on its next execution if it contains fault [81].

IEEE glossary of software engineering [50] defines software testability as 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. ISO [77] defines software testability as: 'attributes of software that bear on the effort needed to validate the software product'. According to Binder [25] software testability can be defined as the relative ease and expense of revealing the software faults. He defines software testability in terms of the two concepts which are controllability and observability. Controllability is the ability to manipulate the input to the software as well as to place the software into a particular state, whereas
observability refers to the ability to observe the output and state change that occurs in the software. Binder emphasizes that, a piece of software is said to be testable if it has these two properties.

Bache and Mullerburg [9] measure the testability in terms of the effort needed for testing the software. The testing effort is measured as the minimum number of test cases required to obtain the full coverage of a given coverage criterion. Baudry et al. [12] suggests that only class diagrams and state charts of UML are the main models that should be analysed for testability. Freedman [49] emphasizes upon the testability of the components as important to the software engineering process. He defines domain testability based on the concepts of controllability and observability of the software. McGregor et al. [102] focuses on estimating the effort needed to test a class as early as possible in the development of an object oriented system. They have introduced the concept of visibility component of a method and define testability of a method as a function of its visibility component (VC). The testability of the class is calculated from the testability of its methods.

In the literature, we find many other studies related to the software testability. Most of them focus on measuring the testability in terms of the testing effort. A number of empirical studies [7, and 32] have also been performed to assess the software testability from the design metrics but these studies are very few. There is a need to perform more empirical studies to assess the software testability. A detailed description of the testability assessment techniques for procedural and object oriented software is presented in Chapter 2.
1.5 Goals of the Thesis

A number of object oriented testing and testability assessment techniques are available in literature. There is an utmost need to reduce the testing cost by improving the testability of the software. Hence, we need to develop the testing techniques to improve the software testability. Also, the assessment of software testability should be done so that it can be improved.

Software contracts are known to improve the software testability [20]. Hence, in this work we develop the object oriented testing techniques based on the concept of software contracts. One of the techniques is a state based testing technique and another is an enhancement of data flow testing technique for object oriented software. It is demonstrated that software contracts improve the software testability.

There exist empirical studies that assess the software testability using design measures in procedural as well as object oriented software. But these studies are very few and more studies on different data sets are needed in order to generalize the conclusions. Thus, recognizing this need we conduct empirical studies to assess the testability of the object oriented software using various open source software. The empirical studies are performed at the class level and the package level. The statistical methods are used to predict the testability using design metrics. In this work, we also analyze and compare the performance of statistical and neural network models for predicting the software testability.

Thus, in this thesis, we focus on the following primary goals:

1. To develop new object oriented testing techniques based on the concept of software contracts.
2. To demonstrate the improvement in software testability through the use of the software contracts.

3. To assess the testability of object oriented software at the class level and the package level using the object oriented design measures.

4. To compare the assessment of testability using statistical techniques and neural networks.

1.6 Organization of the Thesis

The organization of the thesis is presented in this section. Chapter 1 presents the introduction. Chapter 2 focuses on the literature survey of object oriented testing and testability techniques. Chapter 3 concentrates on the concept of software contracts. In Chapter 4, an object oriented testing technique is proposed which is based on software contracts. In Chapter 5, a state based class testing technique is proposed. Chapter 6 presents the assessment of testability for a class. Chapter 7 presents the prediction of testability at package level. Chapter 8 presents the prediction of testability using the the regression analysis for object oriented software. Chapter 9 presents the application of artificial neural networks for assessing the testability of the object oriented software. Chapter 10 presents the conclusions of the work carried. Chapter-wise description of the work carried is as follows:

Chapter 1. This chapter is devoted to an introduction of the field and disciplines related to this dissertation.
Chapter 2. This chapter concentrates on the literature survey of the existing object oriented testing techniques and software testability techniques in conventional and object oriented software.

Chapter 3. This chapter focuses on the concept of software contracts on which the proposed testing techniques are based. Also, it is shown in this Chapter how software contracts can be used to reduce the number of test cases and improve the testability of an object oriented software. To accomplish this, software contracts are instrumented in a class and test cases are designed for this class using the path testing technique and a comparison is made with a class without instrumenting the software contracts. It is found that the instrumentation of software contracts reduces the number of test cases and hence, improves the testability.

Chapter 4. This chapter proposes an object oriented testing technique based on the software contracts. This testing technique has been developed at the class level. The proposed testing technique is data flow based which combines data flow testing and software contracts. In this technique a class flow graph is generated from the contract specification of the class and then conventional data flow testing is applied to find the test cases. This technique helps in finding the infeasible sequences and the implementation errors. A contract instrumented class is compared with a non contract instrumented class. It is found that the number of du pairs is reduced in an instrumented class. Software contracts are used to improve the software testability. Hence, this testing technique helps in improving the software testability.

Chapter 5. This chapter proposes a state-based class testing technique. State-based testing is one of the most important testing techniques of object oriented software. It
complements the traditional approaches of testing: black box and white box testing. State based testing focuses on testing the state of an object throughout its life cycle. The proposed state-based testing technique uses software contracts (method preconditions, method postconditions, and class invariants). A state test model is developed from the contract based specification of a class. The test cases are generated from the state test model using the state testing criteria.

Chapter 6. This chapter concentrates on the assessment of testability for a class using the four open source Java systems which total over 281K lines of code. The testability is an external attribute which makes it difficult to measure. But it can be assessed through the testing effort. Assessment of the testing effort is made by finding a correlation between fourteen design metrics and three JUnit based test metrics. The testing effort in turn provides an assessment of the software testability. The results show that the testability can be assessed from the designs metrics, as there is a significant correlation between most of the design metrics and the test metrics. The testability assessment can help the testers in planning the testing activities.

Chapter 7. This chapter focuses on the prediction of testability at package level using one of the biggest open source software (Eclipse). The approach used is same as used for the class level assessment of testability. An Eclipse plugin is used to extract the value of the design metrics and test metrics from the source code of Eclipse project. The design metrics and the test metrics of Eclipse project are calculated at the package level. A correlation is found between the design metrics and the test metrics. The results show that testability can be assessed from the design metrics at the
package level. This study can help software practitioners to have an understanding of package level testability.

Chapter 8. This chapter presents the prediction of testability from the design metrics of object oriented software using univariate and multivariate linear regression analysis. This study is based on the experimental analysis that uses fourteen design metrics as the independent variables and two JUnit based test metrics as the dependent variables. Three large open source java systems, which have their JUnit classes, are used in this study. The results of the study indicate a number of promising effects of design metrics on testability of a class in object oriented software. The use of results from the relatively early phases of the software development can significantly help software practitioners to improve the testing costs and thus improve the quality of the software.

Chapter 9. This chapter presents the application of artificial neural networks for predicting the software testability using the design metrics. The testability is generally measured in terms of the effort required for testing. The object oriented design metrics are used as the independent variables and two JUnit based test metrics are used as dependent variables in this study. This study compares the prediction performance of neural networks to the two types of statistical analysis methods: least squares regression and robust regression. This study is conducted on an agile based software, written in Java, having 40K lines of code. The results of the study indicate that the prediction model using neural networks is better than that of the regression models in terms of the statistical measures of the model evaluation.
Chapter 10. This chapter includes the conclusions of the research work and suggests the few directions for further research.