CHAPTER 1

INTRODUCTION AND LITERATURE REVIEW

1.1 THE NEED FOR TESTING A PROGRAM

1.1.1 Concepts of Software Engineering

Software Engineering is a systematic approach to development, operation, maintenance and retirement of the software according to IEEE definition. Software Engineering is a strategy for producing quality software. In general, Software Engineering as a discipline provides tools and techniques to develop software in an orderly fashion. Software Engineering as an engineering approach is used to build the software for end users using a well defined, well managed, consistent and cost effective process.

The Software Development Life Cycle (SDLC) is the sequence of different activities that take place during development. Any software development consists of five phases namely Requirement Analysis, Design, Implementation, Testing and Maintenances (Pressman 2000).

The SDLC begins with the identification of a requirement for software and ends with the formal verification of the developed software against that requirement. The software life cycle is the period of time that starts when a software product is conceived and ends when the product is no longer available for use. Traditionally, the models used for the software development lifecycle have been sequential, with the development progressing through a number of well defined phases. The sequential phases
are usually represented by a V or waterfall diagram. These models are
respectively called a **V-lifecycle model** and a **waterfall lifecycle model**
shown in Figure 1.1. The developer must complete each phase before the next
phase begins.

![V-Life Cycle Model](image)

**Figure 1.1 V-Life Cycle Model**

Requirement Analysis determines the functionality requirements
and obtains the requirements from customer or user. The goal of this phase is
to understand the exact requirements of the customer and to document them
properly. Requirement engineering activity is the most important in the
SDLC because the errors introduced at this stage are the most expensive
errors requiring lot of rework if detected later in the software life cycle and
output of this activity is used in successive stages of life cycle. The final
product of requirements engineering is normally a document referred to as the
software requirement specification. The objectives of this analysis are that
introducing the concepts of user and system requirements, describing
functional and non-functional requirements, explaining techniques for describing system requirements and explaining how software requirements may be organized in a requirements document. The fundamental principles of requirements are that all requirements are testable for presence in the real world of their implication, all requirements reflect someone’s subjective values and priorities and all requirements are in natural conflict with all others for common resources.

Design phase determines the functionality to be provided by the software. This phase determines how a software system should be produced in order to make it functional, reliable and reasonably easy to understand, modify and maintain. There are two ways of designing, namely, architectural and interface design. Architectural design is the one which designs the structure of the system. The architectural design phase is the one which implements the requirement of design and specification, identification of the components within the software and the relationships between the components. Interface design is to specify the interfaces between the parts of the system. Detailed design is the one which designs the algorithms for individual parts. The design provides a complete picture of the software, addressing the data, functional and behavioral domain from an implementation perspective.

Implementation is the coding phase which builds the software. The coding phase is the one in which each component of the software is coded and each unit is tested to verify whether it faithfully implements the detailed design or not. Testing is execution of the software with data to ensure that the software works correctly. Good code is the one which works bug-free, readable and maintainable. Some organizations have coding standards that all developers are supposed to follow. There are some important qualities of good code such as cleanliness in sense clean code that is easy to read,
consistency that makes it easy for people to understand how a program works, and extensibility that is easier to reuse and modify the code. The above three basic principles are statements that should describe the structure of the actions taken. The text should describe the structure of the program. The language should allow the underlying assignment oriented central processing unit to be used efficiently.

The Software Integration phase is the one in which progressively larger groups of tested software components are integrated and tested until the software works as a whole. The System Integration phase is the one in which the software is integrated into overall product. The target of integration testing is the interface in which it is checked whether parameters match on sides, types, permissible ranges, meaning and utilization. There are three integration strategies such as top down integration, bottom up integration and sandwich integration. Top down integration proceeds down the invocation hierarchy, adding one module at a time until an entire tree level is integrated. The bottom up strategy works from the bottom. A sandwich strategy runs from top and bottom concurrently, meeting somewhere in the middle. Each time a new module is added as part of integration testing as the software changes. New data flow paths are established, new input and output may occur and new control logic is invoked.

The Acceptance testing phase is the one in which tests are conducted and witnessed to validate that the software faithfully implements the specified requirement conducted by the end user rather than software engineers. A series of tests are conducted to enable the customer to validate all requirements. Acceptance testing may be conducted for few weeks or months. The discovered errors will be fixed and better quality software will be delivered to the customer.
Software maintenance is an activity that includes error correction, enhancement of capabilities, deletion of obsolete capabilities and optimization. Because change is inevitable, mechanisms must be developed for evaluating, controlling and making modifications. So any work done to change the software during its operation is considered to be maintenance work. The purpose of this phase is to preserve the value of the software over time. The value can be enhanced by expanding the customer base, meeting additional requirements, becoming easier to use, more efficient and employing newer technology. There are four major categories of software maintenance such as corrective maintenance, adaptive maintenance, perfective maintenance and preventive maintenance.

The advantages of using software engineering in software development are improved quality, improved requirement specification, improved cost and schedule estimates, better use of automated tools and techniques, less defects in final product, better maintenance of delivered software, well defined processes, improved productivity and improved reliability.

1.1.2 The Purpose of Testing

The role of computers in the technological advances is immense and unquestionable. It plays a major role in the growth and development of science, business and industries. The computers have the capability to perform at very high speed and accuracy thereby helping the man to achieve his goal at a faster rate. The software is the brain behind the computer without which computer cannot function. The software comes under various categories such as systems software, development software and application software. The software developers and computer professionals mostly use systems and development software whereas common man uses application software. In general all these software are designed to assist in performing
tasks quickly and accurately. With the technological advancement, every field depends on software for their requirements and lots of new software is developed and marketed everyday. All these software requires testing throughout their development stages and prior to shipping to customers. The main reason for software testing is to remove the flaws in the software and make them user friendly before marketing (Schach 1996) and (Edward Berard 1994).

One of the aims of software development is to create quality software artifacts. The quality of a software product has many aspects, one of which is correctness, which refers to the absence of defects. Software testing checks whether the product works in certain situations or it does not. Reliability estimation is a statistical sampling method to gain failure of data. Software without testing may lead to cost potentially much higher than that of testing, especially in the systems where human safety is involved. Cost factor plays a major role in many software developments. In the software life cycle, the earlier the errors are discovered and removed; the lower is the cost of their removal. The most damaging errors are those, which are not discovered during the testing process and therefore remain when the system goes live. The purposes of testing are save time and money by identifying defects early, avoid or reduce development time, provide better customer service by building a better applications, whether satisfied user requirements or not, build a desired modifications and enhancements for later versions, identify and catalog reusable modules and components, identify areas where programmers and developers need training design, or development phases. Because of software bugs like miscommunication, software complexity, programming errors, customer changing requirements, time pressures, poorly documented code testing is important. So testing is aimed at evaluating an attribute or capability of a program and determining that it meets its required result.
1.2 THE CONCEPTS OF TESTING

1.2.1 Types of Testing

Testing is the process of executing a program with the intent of finding errors. It provides the benefit of determining whether all software functions appear to be working according to the specification and verifies whether performance requirements have been fully met or not. Data collected during software testing are used as good measure for software reliability and software quality. The purpose of testing is to show the presence of defects (Whittaker 2000).

Traditionally there are two methods of testing software (1) Black Box Testing (Functional Testing) and (2) White Box Testing (Structural Testing).

White Box testing (1) guarantees that all independent paths within a module have been exercised at least once; (2) examines all logical decisions on their true and false sides; (3) executes all loops and tests their operations at their limits; and (4) exercises internal data structures to assure their validity.

White Box testing techniques are Condition testing, Basis path testing, Flow Graphs, Loop testing and Data flow testing. Condition testing is carried out to check all logical conditions in a program. Basis path testing defines basic set of execution paths in order to guarantee that every statement in the program is executed at least once during testing. This method enables the designer to derive a logical complexity measure of a procedural design and it acts as a guide for defining basis set of execution paths. An independent path of a program that introduces at least one new set of processing statements is called basis set. Flow graphs can be used to represent control flow in a program and can help in the derivation of the basis set. Each flow
graph node represents one or more procedural statements. The edges between nodes represent flow of control. An edge must terminate at a node, even if the node doesn’t represent any useful procedural statements. A region in a flow graph is an area bounded by edges and nodes. Each node that contains a condition statement is called predicate node. Cyclomatic complexity is a metric that provides quantitative measure of the logical complexity of a program. It defines the number of independent paths in the basis set and thus provides an upper bound for the number of tests that must be performed. Loop test focuses exclusively on the validity of loop constructors. Four different classes of loops can be defined: simple loops, nested loops, concatenated loops and unstructured loops. Data flow testing selects test paths of a program based on definition and use of variables in the program. Statements in a program are related to each other according to definitions and uses of variables. Challenges in White Box testing are, it requires a sound knowledge of the program code and the programming language, human tendency of a developer being unable to find defects in his or her code and fully tested code may not correspond to realistic scenarios.

Types of testing under White Box testing strategy are unit testing, static and dynamic analysis, statement coverage, branch coverage, security testing and mutation testing. The first test in the development process is the unit test. The source code is normally divided into modules called unit. Unit tests ensure that each unique path of the program performs accurately to the documented specifications and contains defined inputs and expected results. In statement coverage testing technique each statement of the code is executed at least once.

Static analysis involves going through the code in order to find out any possible defect in the code. Dynamic analysis involves executing the code and analyzing the output (Ball 1999). Static analysis covers the area of
performing all kinds of tests on the source code of software programs. Dynamic analysis covers the area of performing all kinds of tests on the object code and executables. The difference between these tests is determined by the state of post or pre compilation of source code. Before compilation the testing is targeted to improve and ensure the quality of the source code. After compilation the testing is targeted to improve and ensure the quality of the object code and the executable. Both areas of testing are targeted to improve and ensure the quality of the software program.

Branch coverage testing helps in validating of all the branches in the code and making sure that no branching leads to abnormal behavior of the application. Security testing is carried out in order to find out how well the program can protect itself from hacking, unauthorized access and any code damage. The application is tested for the code that was modified after fixing a particular bug. It helps in finding out which code and strategy of coding can help in developing the functionality effectively. Advantages of white box testing are as the knowledge of internal coding structure is prerequisite, it becomes easy to find out which input can help in testing the application efficiently, it helps in optimizing code and helps in removing the extra lines of code, which can bring in hidden defects. Disadvantages of White Box testing are a skill tester is needed to carry out the testing and it is difficult to check every bit of code to find out hidden errors, which may create problems, resulting in failure of the application.

Black Box testing is used to test those software functions are operational, the input is properly accepted and the output is correctly produced. It takes an external perspective of the test object to derive test cases. These tests can be functional or non functional, though usually functional. There is no knowledge of the test object’s internal structure.
Black Box testing should make use of randomly generated inputs. Data outside of the specified input range should be tested to check the robustness of the program. Boundary cases should be tested (top and bottom of specified range) to make sure the highest and lowest allowable inputs produce proper output. The number zero should be tested when numerical data is to be input. Stress testing should be performed (try to overload the program with inputs to see whether it reaches its maximum capacity), especially with real time systems. Crash testing is the one which is used to determine the fitness of a newly built system or release of software under test and it is used to check whether software under test is reliable and robust or it crashes during routine use.

Partitioning and Boundary Value Analysis (BVA), Graph Based testing establishes relationship between logical input combinations and finds corresponding effect. Equivalence class represents a set of invalid or valid states for input conditions. BVA test the boundaries of input domain (Perry 2006). Advantages of Black Box Testing are more effective on larger units of code than glass box testing, tester needs no knowledge of implementation, including specific programming languages, tester and programmer can be independent of each other and tests can be performed from a user's point of view. Disadvantages of Black Box Testing are only a small number of possible inputs can actually be tested, to test every possible input stream would take nearly forever, without clear and concise specifications, test cases are hard to design, there may be unnecessary repetition of test inputs if the tester is not informed of test cases the programmer has already tried, may leave many program paths untested.

Some of the testing terminologies used are, mistake is an action performed by a person leading to incorrect result; fault is the outcome of the mistake. It can be a wrong step, definition in the program; failure is the
outcome of fault; error is a deviation from the correct result. The bug is defined as a defect, or a function that does not work as defined in the requirements. Table 1.1 explains the difference between White Box Testing and Black Box Testing.

**Table 1.1 Difference between White Box Testing and Black Box Testing**

<table>
<thead>
<tr>
<th>WHITE BOX TESTING</th>
<th>BLACK BOX TESTING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Related to structural knowledge</td>
<td>Related to functional knowledge</td>
</tr>
<tr>
<td>Knowledge of the code is required</td>
<td>No knowledge of code is required</td>
</tr>
<tr>
<td>White Box testing approach one has to wait for the designing to complete</td>
<td>Black Box testing will begin early in the software development i.e. in requirement gathering phase itself.</td>
</tr>
<tr>
<td>White Box testing will be effective only for small lines of codes or piece of codes.</td>
<td>Black testing strategy will be effective to almost any size small or large.</td>
</tr>
<tr>
<td>In White Box Testing, performance of the application is not tested.</td>
<td>Performance of the application is tested.</td>
</tr>
<tr>
<td>In general one should write large quantity of test cases for white box</td>
<td>In Black Box, it’s a selection of sample test cases.</td>
</tr>
<tr>
<td>White Box testing is done by the Developers.</td>
<td>Black Box testing is done by the professional testing team</td>
</tr>
<tr>
<td>Synonyms for White-Box include: structural, glass-box.</td>
<td>Synonyms for Black-Box include: behavioral, functional, opaque-box, and closed-box</td>
</tr>
</tbody>
</table>

1.2.2 Data Flow Testing

There are many different techniques for testing software, one of which is data flow analysis. Data flow analysis was originally used as a technique for code optimization in compilers. It has also been shown to be a useful technique in other areas, such as performance tuning, testing and debugging. Data flow information is used in a number of different ways
within the testing domain. Data flow testing refers to the use of data flow information to guide the selection of test cases.

Data flow testing is based on use of data structures and flow of data in the program. Concepts of this test are:

Definition or def: A statement in the program where an initial value is assigned to a variable.

C-use: It is called computation use. It occurs when a variable is used for computation. A path can be identified starting from the definition and ending at a statement where it is used for computation. The value of variable also gets changed.

P-use: It is called predicate use. It occurs when a variable appears in the conditional statement. Path can be identified starting from definition of the variable and ending at the statement where the variable appears in the predicate.

All-use: Paths can be identified starting from definition of a variable to its every possible way of usage.

Du-use: Path can be identified starting from definition of a variable and ending at a point where it is used but its value does not get changed.

By using all the used paths, the errors such as “variable defined but not used in the program” can be detected. Data flow testing can be used for determining the semantic validity of a program, understanding the behavior of a program for debugging, maintenance and verification. Data flow testing methodologies can be suitable for both unit testing and integration testing. The Data flow testing approach is suitable for bugs detection. Data flow testing can be performed statically or dynamically: the static approach
performs the testing without executing the program, whereas dynamic is performed by executing the program. Static analysis is capable of detecting errors within the code. Dynamic analysis is capable of analyzing these data types, but is only performed on the parts of the program that are accessed during execution. With the help of data flow testing, unused variables, memory wastages, and specific programming language property bugs are identified. This thesis focuses on data flow testing of Object Oriented Programs with static analysis.

1.3 THE STATE OF THE ART

1.3.1 Procedure Oriented Testing Vs Object Oriented Testing

Traditional procedure oriented program testing and Object oriented program testing are different from each other (Firesmith 1993 and Ryder et al 2005). Procedure oriented languages were algorithm-centric in which they viewed the program as being driven by an algorithm that traced its execution from start to finish. Data was an external entity that was operated upon by the algorithm. Fundamentally, this class of programming languages was characterized by data being considered as separate from the operations or program and algorithm being the driver, with data being subsidiary to the algorithm. From a testing perspective, testing a conventional procedure oriented system therefore entailed testing the algorithm, treating data as secondary to testing the algorithm flow. The procedure oriented program breaks up a larger job into a number of tasks and programs are broken into tasks independently as functions or subroutines. These subroutines and functions are then combined to form a program. The general idea is to simplify debugging a program and to reuse the procedures in other programs. In due course, it was realized that this was not sufficient to enable re-use of programs. Subroutines and functions are too rigid in requiring a specific data type to be used and data to be passed to them in a rigid order. As the cost was
involved, it was realized that building programs using a reusable components was imperative. This led to the emergence of Object Oriented Program. Khan Al-A’ali and Girgis (1995), has described Object Oriented Programming language as data or object centric. There is no separation between data and the methods that operate on the data. The data and the methods that operate on the data go together as one individual unit. In OOP the concept of subroutines or functions is extended to that of “Class”. An object-oriented system is made up of objects and classes. An object is composed of a set of features, which define its states, and a set of operations, which define its behavior. A feature can either be a value or an object. Testing the object oriented properties is different from Procedure Oriented Program testing.

In the procedure oriented language, basic unit of testing is a subroutine or function whereas in an OOP testing it is a class. A class provides some of the properties such as polymorphism, Inheritance, Encapsulation of its instances to conceal the data structure and the details of implementation (Johnson 2000). Testing these properties is necessary in OOP. Inheritance is an important property in which the objects may acquire characteristics from other objects. So testing inheritance hierarchies is important. In object oriented life cycle inheritance related bugs may occur in two phases (i) Design Phase and (ii) Implementation Phase. By using data flow information one can test Implementation Phase Inheritance Related bugs like Class not defined error, Name Conflicts, Naked access, Inadvertent Bindings, Missing Override, Naughty Children and Spaghetti Inheritance. Design Phase Inheritance Related Bugs are Square Peg in a Round Hole and Worm Holes. This thesis focuses on Implementation Phase Inheritance Related Bugs.
1.3.2 Literature Review

Dorman (1993) presented a unit testing in C++ which tested call order, parameters, call original function, return value and throw exception. Unit testing is the lowest level of testing performed during software development. This paper has defined a unit level testing strategy for object based C++ developments based on the C++ class as the fundamental unit of test. A well designed class will provide an abstraction, with implementation details hidden within the class. Objects of such classes can be difficult to thoroughly perform unit test. This paper has discussed the problems involved in unit testing of C++ classes, presented strategies which solve these problems.

Object Oriented Programming concepts have implications on structured programming. This described a new approach to the unit testing of object-oriented programs, based on a set of tools. This approach allowed automation of many aspects of testing, including test case generation, test driver generation, test execution, and test checking. Experimental prototypes of tools for test generation and test execution were described. The test generation tool required the availability of an algebraic specification of the abstract data type being tested, but the test execution tool can be used when no formal specification is available. Using the test execution tools, with various parameters and combinations of operations were performed by Doong and Frankl (1994).

Testing of Object Oriented Programs is different from structured programming analysis. Testing Object Oriented software depends on many factors. They are application-under-test, the development approach, the organization of the development, quality assurance, the criticality of the application, the development environment, the implementation language, language features, project timing and resource constraints. The three
strategies namely intra-class, inter-class, system and acceptance testing are considered for classes and their features of their interactions (Binder 1994).

Object Oriented Metrics are dependent on programs and are estimated through empirical study. The empirical study focused on process improvement which increased necessity of using software measures and metrics. The need for such metrics was particularly notable when an organization is adopted a further enhancement in programs. An automated data collection tool was developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and suggested ways in which project managers may use these metrics for process improvement by the teams of Chidamber and Kemerer (1994) and Chin-ming et al (1997).

Harrold et al (1992) and the team of Smith and Robson (1992) created libraries of well-designed, thoroughly-tested classes that can be confidently reused for many applications. They have also developed few class testing techniques. In this paper, a class testing technique has been presented, explaining the hierarchical nature of the inheritance relation to test related groups of classes by reusing the testing information for a parent class to guide the testing of a subclass. Initially the base classes which have no parents are tested by designing a test suite that tests each member functions individually and also tests the interactions among member functions. To design a test suite for a subclass, it is mandatory to update all the modifications and changes in parent classes to be reflected upon the subclasses also.

David Hovemeyer and Frankl (1994) implemented automatic detector for a variety of bug patterns found in Java programs. This paper describes how bug pattern detector finds serious bugs in several widely used Java applications and libraries. By applying bug pattern detector in real programs four categories of bugs could be found out namely single threaded
correctness issue, synchronization correctness issue, performance issue, security and vulnerability to malicious untrusted code. Software quality concerns are correctness, robustness and readability.

Frank Tip et al (1996) have presented an algorithm for computing class hierarchy slices. This paper describes an algorithm for slicing class hierarchies in C++ programs. Program that uses the class hierarchy, eliminates from the hierarchy those data members, member functions, classes, and inheritance. Class slicing is especially useful when the program is generated from a larger program by a statement slicing algorithm. Such an algorithm eliminates statements that are irrelevant to a set of slicing.

Automated program analysis serves as the foundation for testing programs using inheritance feature supporting compiler concepts. This paper describes a concept named Unit Repeated Inheritance (URI) in Z notation to realize object-oriented testing of an inheritance hierarchy. Based on this unit, an inheritance level technique as a guide to test object-oriented software errors in the inheritance hierarchy is described. Two testing criteria, intra level and inter level, were formed based on the proposed mechanism. In order to make the test process automatic, lexical analyzer and a parser were used to demonstrate a declaration of C++ source code. Also constructed a windowing tool used in conjunction with a conventional C++ programming environment to assist a programmer to analyze and test C++ programs (Chun Chia Wang et al 1997). Chen and Low (1997), claim that program analysis plays a key role in many areas of software development, such as performance tuning, testing, debugging and maintenance. Program analysis can be carried out statically or dynamically. This paper focuses on runtime data analysis for object oriented systems in general and the Java 2 platform in particular. Especially it investigated various runtime approaches to monitor the access
and modification of variables in a Java program in order to keep track of their usage history and used different approaches.

Inheritance relationships were represented as a Flow Graph (Jiun-Liang and Feng-Jian Wang 1998). This paper presents an inheritance flow model. It represented the inheritance relationships among classes as a flow graph. A flow operation was associated with each attribute and method in a class to denote the defined or inherited member. An inherited member could be handled by a sequence of flow operations along a path in the flow graph. This model provided several analyses in a class hierarchy, such as implicit inherited member and polymorphic method invocation. These analyses may useful in various fields of software engineering, such as static analysis, maintenance, and complexity measurement.

Crnogorac et al (1997) discussed about inheritance mechanism in object oriented paradigm. This paper presents about abnormality of inheritance mechanism. Three kinds of anomalies such as state partitioning, state modification and history-only-sensitiveness related to history of an object. This leads to towards the design of better inheritance mechanism. Analyzing and coding programs is not easy because the programs in the Object Oriented Language hide much information, including the execution process of the programs.

Object-Oriented Programming increases software reusability, extensibility, interoperability and reliability. To realize these benefits development team should use software testing. Software testing aims to uncover as many programming errors as possible with minimum cost. How to reduce the cost and improve the quality of software testing remains a major challenge to the software engineering community. The requirements for testing Object-Oriented Programs differ from those for testing conventional programs. Object-oriented software testing problems are discussed and
focused on the difficulties and challenges in the Object Oriented Programming. It provided a general framework for class-level and system-level testing and examines object-oriented design criteria and metrics of high testability. It offers object-oriented testing techniques, ideas and methods for unit testing and object-oriented program integration-testing strategy. This may be useful to object-oriented program testers, program developers, software project managers and researchers working with object-oriented testing (Kung et al 1998).

A team of Souter and Pollock (1999) and an another team of Tai and Daniels (1999) discusses about static analysis of object-oriented software. They have focused on addressing the new features of classes, inheritance, polymorphism and dynamic binding. The use of Object-Oriented Program design principles has created a new set of problems for software tool developers and compiler writers. Besides having to represent and analyze programs that exploit the new features of classes, inheritance, polymorphism and dynamic binding, the style used to develop software applications and the units of analysis were changed. Object-Oriented Program design promotes the reuse of code not only through inheritance and polymorphism, but also through building server classes which could be used by many different client classes. This paper demonstrates how exploiting the nature of object-oriented design principles can enable development of scalable static analyses. The presented algorithm describes about computing definition-use information for a server class. This information may be useful for data flow testing and debugging. Static definition-use information was not only useful to optimizing and parallelizing compilers, but also for debuggers, software testing, editors, program integration and software maintenance.

Alexander (2001) performed dataflow testing in which polymorphic relationships were analyzed. It was based on inter procedural dataflow
analysis. All polymorphism definitions and their uses were analyzed between classes by using Java Parser generator. This design gives only about polymorphism dataflow information. Bush et al (2000) described about dynamic programming errors and behaviors. Some important classes of programming errors were hard to diagnose, both manually and automatically, because they involved a program’s dynamic behavior. This described a compile-time analyzer that detected dynamic errors in large, real-world programs. The analyzer traced execution paths through the source code, modeling memory and reported inconsistencies. This approach provided valuable contextual information to the programmer to understand and repair the defects. Automatically created models, abstracting the behavior of individual functions, allowed inter-procedural defects to be detected effectively and this product used effectively on several large commercial programs.

Antero Taivalsaari (1999) made a fundamental observation of underlying object-oriented inheritance mechanism which allows existing programs to be extended and refined without editing existing code. This article discussed about understanding of inheritance, examining its usage, surveying its varieties and presented a simple taxonomy of mechanism called late binding that minimizes the need for physical code duplication, by letting the same operations invoke different properties depending on the context from which these operations are invoked, thereby promoting sharing of code. Survey about testing of classes, generic classes, exceptions has been discussed (Binder 1999).

Min-Soo Jung et al (2000) observes that analyzing Java programs is not easy because these programs in object oriented language hide much information, including, the execution processes of the programs. Class file contains useful information that can trace the execution process. The author
has used a static Java class file analyzer that has shown the relationships among classes, caller/callee relationships, inheritance hierarchies and detailed information within classes. This analyzer displays decompiled source programs from the given class files. It can help users to understand Java programs better and help Java programs to be designed and developed in an efficient manner.

Labiche et al (2000) and a team of Harison and Counsell (2000) presented an approach for where to start test and how to define integration strategy for object oriented software. One of the characteristics of object-oriented software is the complex dependency that may exist between classes due to inheritance, association and aggregation relationships. These papers present an approach to define a test order by exploiting a model produced during design stages like Unified Modeling Language, namely the class diagram. The test takes account of dynamic dependencies, abstract classes that cannot be instantiated, making some testing levels infeasible. The test order is represented by a graph showing which testing levels must be done in sequence and which ones may be done independently. It also provided information about the classes involved in each level and how they are involved. The approach was implemented in a tool called TOONS (Testing level generator for Object-OrienNted Software).

This paper analyses and tests, the integration aspects of software components particularly the object-oriented software. The most of softwares were developed using object-based designs and object-oriented languages. The task of integrating the components, becomes more crucial to the success of the software. Data flow analysis has been applied for testing procedural and object-oriented programs especially in Java programs (Bowjarwah et al 2000).
Class libraries are generally designed with an emphasis on flexibility and extensibility. Applications used a library and exercised only part of the library's functionality. As a result, objects created by the application may contain unused members. The presented algorithm specialized in a class hierarchy with respect to its usage in a program. The algorithm analyzed the member access patterns for variables and created distinct classes for variables that access different members. This class hierarchy specialization algorithm reduced object size and execution time through reduced object creation/destruction time (Frank Tip and Peter Sweeney 2000).

A new method was presented for analyzing and reengineering class hierarchies. A class hierarchy was processed along with a set of applications with subtype relationships between objects, variables and class members. The method was primarily intended as a tool for finding imperfections in the design of class hierarchies and could be used as the basis for tools that largely automate the process of reengineering class hierarchies. The method could also be used for space-optimization and elimination of redundant fields from objects (Gregor Snelting and Frank Tip 2000).

Debugging tools for software specifications were rare. They discussed about an easier way to write requirements specification with the aid of Software Engineering tools. The tool suite was developed to speed up the process of understanding, debugging and re-testing software specifications. This tool suite provided a significant contribution to improve reliability and efficiency of software specification development, especially programming on the specification level in the telecommunications industry (Li and Horga 2000).

Object-Oriented measures predict classes having faults. Object-Oriented measures are used to find program complexity. A cognitive theory
proposes a threshold that effects many object-oriented measures. If Object-Oriented classes complexity is below a threshold, easy understandability increases. It explains how to design Object-Oriented programs. This paper empirically tested two C++ telecommunication systems (Benlarbi et al 2000).

By using data flow analysis, sequence of method invocation is identified in object oriented software. Programs developed with object technologies have unique features that often make traditional testing methods inadequate. The outcome of a method executed by an object often depends on the state of the object when the method is invoked. The techniques for testing of classes exercise on methods that are in different states in a class. The state of an object, at any given time, depends on the sequence of messages received by the object up to that time. Thus, methods for testing object-oriented software should identify sequences of method invocations that are likely to uncover potential defects in the code under test. Testing methods for traditional software do not provide this kind of information. In this paper, data flow analysis and symbolic execution were used to automate deduction to produce sequences of method invocations to test a class (Ugo Buy et al 2000).

Cara Stein et al (2002) have described about empirical evaluation of inheritance levels in five object oriented programs. Statistical correlation has been taken between four inheritance metrics like depth of inheritance, number of methods inherited by a class, number of methods overridden per class and a set of dependent variables and concluded whether inheritance make maintenance of OO systems easier or not. (Jonathan Gray 2000) has developed an automated tools which play an important role in the software engineering methods and processes. The development of these tools itself requires time and resources. Different kind of software engineering tools available differ in methods, activities and phases of SDLC.
Object-Oriented Programming consists of several different levels of abstraction, namely, algorithmic level, class level, cluster level and system level. The testing of Object-Oriented software at algorithmic and system level is similar to conventional program testing. Testing at the class and cluster levels was proposed and tools were developed by Chen et al (2001). Object Oriented Metrics like size, methods, nesting levels were collected analyzed and tested (Subramanian and Cortin 2001). Jeff Offutt et al (2001) have discussed about inheritance and polymorphism faults and investigated by using empirical study of Object Oriented Programming.

This paper (Banisiya and Davis 2002) describes an improved hierarchical model for the assessment of high-level design quality attributes in Object-Oriented designs. In this model, structural and behavioral design properties of classes, objects and their relationships have been evaluated using a suite of Object-Oriented design metrics. This model has related design properties such as encapsulation, modularity, coupling and cohesion to high-level quality attributes such as reusability, flexibility and complexity using empirical and unreliable information.

The construction of functional testing was presented with the algorithm that emphasizes the behavioral characteristics of objects. This algorithm provides a basis for automating the testing process for object oriented software system. This automation is useful because the typical development process for Object-Oriented software uses a large amount of regression testing, both during the original project and over the lifetime of the reusable components (Juliana Georgieva and Veskaganchere 2003). Many software defects result from the violation of programming rules. The rules describe how to use a programming language and its libraries and the dos and don’ts within a given application. Bugs finding engine was developed. It expressed many of these rules for programs written in Java. It found out
dozens of bugs such as string checking, warning if exceptions aren’t thrown, warning if Object.equals() is invoked on incompatible objects, null pointer, super finalizer warning if finalize() method doesn’t call super finalize() on all paths and flags redundant operations (Back and Engler 2003).

María Lucía Barrón-Estrada and Ryan Stansifer (2003), explained about Java adopted mechanism to support parameterized types on the next major release. All the type parameters defined in the parameterized type are replaced by type object in order to make them compatible with existing class libraries. When a client uses a parameterized type to define instances of it, in the translation process, the compiler will insert some bridge methods and coercions, which are guaranteed not to fail at execution time. They also described binary methods, multi-methods, bounded polymorphism, contravariance, constrained genericity, inheritance, parametric polymorphism and single dispatch.

The team of Alan Donovan and Ernest (2004) and Gilad Bracha (2003), have discussed about Java version 1.5 compiler which introduced generic types. This extension affects only the source language syntax by adding parameterized types and variables among object oriented features like primitive types and enumeration types. There are no changes in the Java Virtual Machine. Use of genericity, inheritance and binary methods were implemented but some problems arose as when to reuse code from a super class. The inclusion of generics in Java enhances its expressivity and produces safer code because more errors are caught at compile-time.

Exceptions due to dereferencing a null pointer are very common type of errors in Java programs. Atanas Rountaev et al (2004) have investigated static analysis for finding null pointer bugs in Java programs. They have detected memory access errors such a null pointer dereferences using data flow analysis. Some null pointer bugs arise because of null values
loaded from the heap or passed from a distant call site. The study of real Java applications and libraries has shown that many null pointer bugs are the result of simple mistakes, such as using the wrong boolean operator, infeasible paths. This described the basic analysis as well as the additional errors detected using inter-procedural analysis and found that static analysis techniques are able to pinpoint 50% to 80% of the defects leading to a null pointer exception at runtime in user applications.

The testing of polymorphism in object-oriented software may require coverage of all possible bindings of receiver classes and target methods at call sites (Atanas Rountev et al 2004). However, traditional whole-program class analysis cannot be used when testing of the programs was incomplete. To solve this problem, the author has presented a general approach for adapting whole-program class analysis to operate on program fragments. Furthermore, since analysis precision is critical for coverage tools, the approach provided precision measurements for several analysis by determining which of the computed coverage requirements were actually feasible for a set of subject components. This work enabled the use of whole-program class analysis for testing of polymorphism in partial programs and identified that potentially well for use in coverage tools.

Nghi Truong et al (2004) have reported that more than fifty percent of project budget of software is spent on activities related to improving software quality. Industry leaders claim that this is due to the inadequate attention paid to software quality in the development phase. This paper introduces a static analysis framework which can be used to give practice to the beginners of Java programs to write better quality programs. The framework is used, both software engineering metrics and relative comparison to judge the quality of student's programs and provide feedback about how result might be improved. It concludes poor programming practices like too
many loops, conditional statements, not enough methods, use of global variables rather than parameters to a method, too large methods, use of magic numbers, unused variables and performing unnecessary checking with Boolean expression. In addition it also shows poor performing practices like un-initialized variables, inappropriate access modifiers, omitted “break” statement in a case block, omitted “default” case in a switch statement, confusion between instance and local variables and omitted call to super class constructor. The static analysis framework consists of two analyses: software engineering metrics and structural similarity. This work first evaluated the quality and the second examined the similarity in structure of student programs compared with model solution. The analysis was performed on XML programs.

Rapid analysis testing object model recognized a new factor of testability in design stage that is referred to as reachability. It provides an alternative tester perspective for the developer and can lead the developer to estimate the polymorphic behavior in class inheritance hierarchy that may cause test obstacle at design phase (Lin et al 2005). Ophelia Chesley (2005) discusses about regression test after changes have been made in the program. A tool has been developed for constructing intermediate versions of a Java program named Crisp. If the program is edited, a programmer usually runs regression tests to make sure about functionality. If a test fails unexpectedly, Crisp uses the input for semantic change impact analysis allowing the programmer to select parts of the edit that affected the failing test and to add them to the original program, creating an intermediate version guaranteed to compile. Then the programmer can re-execute the test in order to locate the exact reasons for the failure by concentrating on those affecting changes that were applied. Using Crisp, a programmer can iteratively select, apply and undo, individuals affecting changes and thus effectively find a small set of failure-inducing changes.
Aggarwal et al (2006) investigated the correlation between Object-Oriented design metrics and the occurrence of Object Oriented faults. Study was conducted on three industrial real-time systems that contained a number of natural faults reported for the past three years. The faults found in these three systems have been classified into three types, namely Object-Oriented faults, object management faults and traditional faults. They have proposed a set of new metrics that can serve as an indicator of how strong an object-oriented program is. Different projects were analyzed for testing 22 metrics such as response for a class, number of attributes per class, number of methods per class, weighted methods per class, coupling between objects, data abstraction coupling, message passing, lack of cohesion, loose class information based cohesion, method hiding factor, information hiding, attribute hiding factor, information hiding, number of children, depth of inheritance, method inheritance factor, attribute inheritance factor, number of methods, overridden by a subclass and polymorphism factor.

Testing Object-Oriented systems is important compared with testing procedural software because it promotes reuse. The Object-Oriented software has various features like encapsulation, abstraction, polymorphism, inheritance and dynamic binding which made the testing of Object Oriented programs difficult and different from the conventional testing methods. This paper discussed about the different levels of testing Object-Oriented systems such as unit testing, integration testing and system testing, the Object Oriented testing problems, various testing techniques and the future directions for testing of Object Oriented systems (Singh et al 2006).

Inheritance is one of the main features of Object-Oriented Programming paradigm. There are several measures to characterize the inheritance tree complexity such as number of root classes, fan in, number of children and depth of inheritance tree. The depth of a class within the
inheritance hierarchy is the maximum length from the class node to the root of the tree, through the number of ancestor classes. The deeper a class within the hierarchy, the greater the number of methods it is likely to inherit. Depth of inheritance tree is considered as a factor influencing the cost of testing. Test is supposed to be more expensive if depth of inheritance is high. This paper related the analysis of depth of inheritance with respect to the number of methods to test in each class. This study is based on more than 1700 classes from 6 Java applications (Bruntink and Van Deursen 2006).

From the literature review, it is clearly understood that there is no model currently available for testing inheritance related bugs in Object Oriented Programs. Hence an innovative model has been proposed in this thesis.

### 1.4 THE SCOPE AND AIMS OF THE THESIS

Software must be designed and tested properly before it is delivered to customer. Data flow analysis is a testing technique that is used to identify the sequence of actions performed upon data elements of a program. Data flow testing is analyzed for Inter Procedural Data Flow Analysis, Flow Insensitive Data Flow Analysis, Context Sensitive Data Flow Analysis, Whole Program Analysis, Fragment Program Analysis and finding language property related bugs.

The following are the objectives of the proposed work:

i) Analyzing the source code of object oriented programs for finding number of lines, number of braces, number of comment lines, number of functions and number of constructors.
ii) Analyzing whole program and fragments of object oriented program.

iii) Analyzing classes, methods, attributes in a whole object oriented program.

iv) Finding classes with inheritance hierarchy using Class Hierarchy Analysis.

v) Identifying inheritance relationship between classes.

vi) Testing inheritance related bugs.

1.5 OUTLINE OF THE THESIS

The contents of the different chapters in the thesis are briefly described below.

In chapter 1, the need for testing a program, the concepts of testing and aims of the thesis are stated. Further, the organization of the thesis material in the subsequent chapters is briefly outlined.

In chapter 2, an innovative model, “Inheritance Related Bugs Testing” (IRBT) model is presented. IRBT model consists of four major steps, that is, Source Code Analysis of Object Oriented Programs, whole program analysis and fragment analysis testing, analysis of inheritance property of object oriented programs and testing Inheritance related bugs in object oriented programs. These are presented in the subsequent chapters.

In chapter 3, Object Oriented Program source codes are analyzed for general information presentation.

In chapter 4, the whole program analysis and fragment analysis testing are presented. The object oriented program is divided into small
fragments. Fragments are analyzed for classes and methods. This can be used for finding software metrics such as depth of inheritance, response for a class, number of attributes per class, number of methods per class, weighted methods per class, coupling between objects, data abstraction coupling, message passing, lack of cohesion and loose class information based cohesion.

In chapter 5, analysis of inheritance property of Object Oriented Program is designed to detect the set of classes and packages. Output hierarchies of the classes, methods, attributes detected by using data flow testing are presented. The testing is carried out with four major analyses such as function model, class model, variable model and interface model.

In chapter 6, some of the Object Oriented Program property bugs in the program are located. Testing for bugs in inheritance and Object Oriented Program test analyzes the structure of the software and gives the flow of property. This chapter presents the bugs with reference to the inheritance property.

In chapter 7, highlights of the thesis are briefly reviewed. Suggestions for future research are enlightened.