

A Method for Systematically Generating Tests from Object-Oriented Class Interfaces

Mahesh Babu Mungara

Thesis submitted to the faculty of
Virginia Polytechnic Institute & State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science & Applications

Dr. Stephen H Edwards, Chair
Dr. Richard E Nance, Member
Dr. Mary Beth Rosson, Member

September 15, 2003
Blacksburg, Virginia

Keywords: *Specification-based Testing, class Interfaces, all-DU-pairs, all-nodes*

Copyright 2003, *Mahesh B Mungara*

A Method for Systematically Generating Tests From Object-Oriented Class Interfaces

Mahesh Babu Mungara

(Abstract)

This thesis describes the development and evaluation of a manual black-box testing method inspired by Zweben's test adequacy criteria, which apply white-box analogues of all-DU-pairs and all-nodes to a flow graph generated from the black-box specification. The approach described herein generates tests from a matrix representation of a class interface based on the flow graph concept. In this process, separate matrices for all-DU-pairs and all-nodes guide the generation of the required tests. The primary goal of the research is not to optimize the number of tests generated but to describe the process in a user-friendly manner so that practitioners can utilize it directly, quickly, and efficiently for real-world testing purposes.

The approach has been evaluated to assess its effectiveness at detecting bugs. Both strategies—all-DU-pairs and all-nodes—were compared against three other testing methods: the commercial white-box testing tool Jtest, Orthogonal Array Testing Strategy (OATS), and test cases generated at random. The five approaches were applied across a sample of eleven java classes selected from `java.util.*`. Experimental results indicate that the two versions resulting from this research performed on par with or better than their respective equivalent approaches. The all-DU-pairs method performed better than all other approaches except for the random approach, with which it compared equally. Experimental evaluation results thus indicate that an automated approach based on the manual method is worth exploring.

Acknowledgments

This thesis is here today primarily as a result of the support given to me by my advisor, Dr. Stephen Edwards, whose technical guidance and insightful remarks has been indispensable. I would like to thank him for all of his help and guidance. Apart from his immense knowledge, his dedication and sincerity toward work has continuously inspired me and, as I have striven to emulate him, made me a better person.

Secondly, I would like to thank my committee members, Dr. Richard Nance and Dr. Mary Beth Rosson, for their helpful remarks and guidance. I also remain thankful to my mother and sister for their emotional help and support in tough times, not only during the years I have pursued my M.S. but in all times. Last, but definitely not least, I appreciate and thank my friend Kiran Indukuri for his company and motivation during those late night work schedules at McBryde 565.

Dedicated to my beloved father Seshagiri Rao Mungara, who is not here to see all this.

Table of Contents

Chapter 1: Introduction	1
1.1 Overview	1
1.2 Problem Statement	1
1.3 Research Contributions	3
1.4 Organization	4
Chapter 2: Related Research	6
2.1 Test Case Selection Criteria	6
2.1.1 Algebraic Approaches	7
2.1.2 Finite State Machine Approaches	8
2.1.3 Model-based Approaches	9
2.2 Test Data Adequacy Criteria	9
2.3 The Flow Graph Approach	13
Chapter 3: A Manual Method for Flow Graph Based Test Case Generation	15
3.1 The Matrix Method	15
3.1.1 A Flow Graph for a Class Interface	15
3.1.2 The Matrix Approach	17
3.2 All-nodes Approach	18
3.2.1 Overview	18
3.2.2 Step-by-Step Procedure	18
3.3 All-DU-pairs Approach	21
3.3.1 Overview	21
3.3.2 Definitions and Uses	22
3.3.3 Step-by-Step Procedure	22
3.4 Generating the Tests in JUnit	27
3.5 Infeasible Sequences	28
3.6 A Comparison between All-DU-pairs & All-Nodes	29
Chapter 4: Tutorial	31
4.1 Introduction	31
4.1.1 LinkedList Example	31
4.2 All-Nodes Approach	35
4.3 All-DU-pairs Approach	38
Chapter 5: Experimental Evaluation	47
5.1 The Five Approaches	47
5.1.1 Random Test Case Generation	48
5.1.2 JTest –Parasoft White-box Testing Tool for Java	49
5.1.3 OATS – Orthogonal Array Testing Strategy	49
5.2 Selecting Classes from java.util.*	50
5.2.1 Test Suites	52
5.3 Running the Tests	53
5.4 Mutation Testing	54
5.4.1 Jester – a Mutation Testing Tool for Java	54

5.4.2 Running the Mutants.....	56
5.5 Collection and Analysis of the Experimental Data.....	57
5.5.1 Results Gathered.....	57
5.5.2 Statistical Analysis of the Data Gathered.....	58
5.5.2.1 RCBD –Randomized Complete Block Design.....	59
5.5.2.2 Correlation Tests their Significance and Issues Involved.....	61
5.5.2.3 Correlation Tests.....	62
5.6 Summary of Results.....	67
Chapter 6: Conclusions and Future Work.....	71
6.1 Contributions of the Research.....	71
6.2 Future Work.....	72
References.....	74

List of Figures & Tables:

Figure Name	Page Number
Figure 3-1 Hypothetical stack class interface	16
Figure 3-2. Flow graph for the Stack class	17
Figure 4-1. A Simplified interface capturing the main features of java.util.LinkedList	32
Figure. 4-2. Flow graph for Linked List class specification	34
Figure 5-1. Experimental results- eleven classes, five approaches	58
Figure 5-2. Ranking of the five approaches	60
Figure 5-3. Number of Tests versus percentages- significant correlation ($r=0.28$)	63
Figure 5-4. Number of Methods in a class versus percentages-No correlation ($r=0.05$)	63
Figure 5-5. Average LOC versus percentages- No correlation ($r=-0.16$)	64
Figure 5-6. Correlation between Number of Tests and percentages ($r=0.02$)	65
Figure 5-7. Correlation between Number of Tests and percentages ($r=0.04$)	66

Table Name	Page Number
Table 3-1. Matrix for all-nodes	19
Table 3-2. Matrix for Stack self-data type	25
Table 3-3. Matrix for Stack Object data type	25
Table 4-1. Definitions and Uses separated for Linked List class	40
Table 4-2. Matrix for 'int' references for LinkedList class	41
Table 4-3. Matrix for LinkedList data type references	42
Table 4-4. Matrix for Object data type for Linked List class	42
Table 4-5. Matrix for boolean data type for Linked List class	43
Table 4-6. Matrix for Object[] data type for Linked List class	43
Table 4-7. Matrix for listIterator data type for LinkedList class	43
Table 4-8. Matrix for 'exceptions' as definitions for LinkedList class	43
Table 4-9. All-DU-pairs Matrix in progress	44
Table 4-10. Distribution of tests over data types for Linked List class	46
Table 5-1. Classes from java.util package ordered on various criteria	51
Table 5-2. Number of Tests for each of the 5 approaches over the eleven classes	53
Table 5-3. Number of mutants generated for eleven classes	56
Table. 5-4. Percentage of mutants caught for each test suite run	57
Table 5-5. Statistical parameters for the four approaches	59
Table 5-6. Ranking & Grouping of the five approaches	59
Table 5-7. First approach versus rest four	60
Table 5-8. Correlation table for data from all the five approaches	62
Table 5-9. Correlation table over all-DU-pairs	65
Table 5-10. Correlation table over Random data	65
Table 5-11. Correlation between Cyclomatic Complexity and percentage	67

Chapter 1: Introduction

1.1 Overview

Software testing strategies can be broadly classified into two categories according to how they generate tests: white-box and black box.

- White-box strategies involve developing tests from the structure of the code.
- Black-box strategies involve developing test cases from method signatures and specifications rather than the code or its structure.

In general, white-box methods are more widely used although they typically require more effort. When compared to their black-box counterparts, most white-box testing methods appear superior because they directly interact with the code and have a greater potential for exercising all its intricacies.

The main problem with black-box testing methods is that most of them are either difficult to use or require formal specifications. The current research, aims to develop a manual black-box testing method that is easy to use and that does not require any formal specifications. An evaluation was carried to assess the effectiveness of the approach by comparing it to other major testing approaches. In order to thoroughly describe the research undertaken by the project, this thesis covers two areas:

- Part one introduces the method and describes it in an easy-to-use manner, and
- Part two concentrates on the experimental evaluation of the method, the results obtained and their significance.

1.2 Problem Statement

For generating test cases, black-box testing methods are often considered less effective than their white-box counterparts because they don't directly deal with code but rather with its specification. Nonetheless, black-box methods do possess some key advantages,

which at times makes them equally—or even more—effective and efficient. One major advantage of black-box methods, for example, is that they require only a minimum understanding of the code for which test cases must be generated; hence, black-box test cases are often simpler to generate and require less effort [3].

Additionally, because black-box methods are not code-dependent, they are more generally applicable. Finally, in cases where the code becomes inaccessible and specification alone is available, such methods become indispensable. For example, during early stages of the software life cycle when code is unavailable, as in the requirements engineering and design phases, black-box methods prove crucial. White-box methods come into play only in later stages of the process when coding is complete.

The current research aims to develop a user-friendly manual black-box testing method that requires no formal specifications, and to describe it in such a manner that it can serve as a “cook book” for practitioners. The experimental evaluation aims to measure its effectiveness against other testing methods. Although many manual black-box testing methods exist, this research has been motivated by two entwined goals: establishing a process that poses fewer problems and is user-friendlier. To generate test cases, many existing manual black-box testing methods require formal code specifications. The method proposed here, requires none however; moreover, it is simple and logical in its formulation. Additionally, most extant manual methods—such as cause-effect graphs, error-guessing, and boundary value analysis—demand considerable skill and effort from users. Conversely, the method proposed by this research is relatively simple and can be applied like a “cook book” method.

Zweben *et. al.* [35] first proposed the idea of applying white-box analogues of all-definition-use-pairs (all-DU-pairs) and all-nodes to a flow graph generated from a component specification. The current research extends this concept in an object-oriented setting, to use a matrix-based approach to represent the definitions & uses as rows and columns of an $N \times N$ matrix. To check the cells and generate test cases, the method signatures of a class are taken, the parameters are divided into definitions and uses, the matrix is generated from the method signatures, and Zweben’s criteria are applied to the

matrix. The test cases are generated so that the criteria (here, all definition-use pairs and all-nodes) are covered. For the evaluation phase, eleven classes were selected based on various factors, such as the number of methods, average method size, and number of inherited and direct methods. For all eleven classes we compare the two test-suites generated by the all-DU and all-nodes criteria with randomly generated test suites, OATS test suites [28], and test suites generated by the white-box testing tool JTest [21].

1.3 Research Contributions

This study makes two significant contributions to research. First, since a survey of the pertinent literature reveals that most existing manual black-box testing methods are cumbersome and require formal specifications [3], it develops and explains a method that is user-friendlier and more easily adapted for practical application. To that end, this thesis not only describes the formal procedure of the method but also provides a tutorial-style “recipe” for applying the method to develop test cases for Java classes. Thus, because the method itself contributes both to research and to practical knowledge, care has been taken to describe it so that software testers can put it to immediate use.

By assessing the effectiveness of the proposed testing method against standard ones, both white-box and black-box, the evaluation phase of this project also makes a significant contribution to our understanding. In this phase, the all-DU-pairs method is compared to the basic all-nodes version to determine whether gains are proportional to required effort. A comparison against the random test suite—which has same number of test cases and average number of methods per test case—determines how effective this structured method is against a randomly generated (read “no method”) baseline method. By comparing the method to Java’s white-box testing tool Jtest, we can observe how it performs in contrast to a standard white-box method. Finally, comparing the proposed method against OATS will measure the effectiveness against an optimal method combination (OATS tests).

1.4 Organization

While the first chapter of this thesis introduces the problem and provides some basic understanding of purpose, the second chapter surveys literature relevant to the research and considers the problem in the context of existing work. Chapter 2 identifies patterns in the reference list and discusses existing testing methods, both black-box and white-box. Additionally, this chapter discusses the advantages of the proposed method, explains how it differs from other methods, and examines the basic idea of applying white-box analogues of all DU-pairs and all-nodes to black-box testing methods.

Chapter 3 describes the manual method proposed here, first discussing how it is adopted from Zweben's idea of applying all-DU-pairs and all-nodes criteria to black-box test case selection. This chapter goes on to describe how the all-nodes and all-DU-pairs approaches are used to generate test suites for Java classes. Finally, it discusses the differences between the two approaches and argues that despite the additional effort required, the all-DU-pairs method possesses key advantages.

Chapter 4 presents a tutorial for generating test cases following these two criteria. With the assistance of a running example, the method is illustrated; hence, the chapter also provides a "recipe" for users to follow the method. Finally, it identifies and discusses exceptions and critical issues that need to be recognized when using the method.

Chapter 5 discusses the experimental setup: it presents the various steps involved in conducting the experiment, from generating the mutants to collecting the data. The chapter discusses various issues involved in the experiment: the criteria behind the selection of the eleven classes for the experiment, how the Jester mutation tool was used to generate the basic set of mutants, and the approach followed in the generation of the randomly created, Jtest, and OATS test suites. The chapter explains the use of Junit in executing the test cases and the use of ANT in running the test suites over all the mutants. Chapter 5 also lists the experiment's dependent and independent variables and its various dimensions, along with an explanation of why the particular setup has been chosen. Results obtained from the experiment are given here, as are the statistical analysis of the data.

Chapter 6 presents conclusions drawn from the experimental analysis. The random and the all-DU-pairs test results are compared, since both contain the same number of test cases. Due to the fact that all combinations are exercised in the all-DU-pairs method—unlike that of the random method, where some combinations might repeat—the expected result was that all-DU-pairs would fare better. The actual result obtained was quite different: the all-DU-pairs and random methods measured equally. Similarly, we compared the proposed method to the all-nodes, JTest, and OATS methods, receiving from the examination a variety of results. Chapter 6 summarizes the research, explains all pertinent results, identifies significant implications, and projects future work.

Chapter 2: Related Research

In the software life cycle, testing is a pivotal phase. A successful testing strategy should concentrate on finding and eliminating any “bugs” in the system. Software testing literature can be primarily classified into methods that generate tests (test case selection) and methods that assess those testing strategies (test adequacy).

2.1 Test Case Selection Criteria

Test generating methods are broadly classified into white-box and black-box methods. White-box methods typically generate test cases by drawing a flow graph from the source code and applying control-flow and data-flow coverage criteria. Conversely, black-box methods involve generating the tests from the specification of the component. White-box testing methods have been more prevalent and have been widely automated, whereas most black-box approaches, for which no popular tools exist, generate tests manually. In practice, software projects use both methods: black-box, earlier in the cycle and during system level testing; white-box, during later stages, especially in unit testing. [4]

Currently, white-box methods are considered more effective than their black-box counterparts because most bugs arise from the source code. However, as programming environments and tools become more sophisticated, fewer bugs will arise due to mistakes in source code syntax or semantics. Rather, they will arise in the form of missing or inadequately understood requirements. In such cases black-box tests, which are developed from the specification of the component, will prove more effective.

As previously noted, black-box or specification-based testing methods involve generating test cases from component specifications. Their key advantage is that they are available very early in the life cycle. Unlike white-box methods, for which test cases are available only after the source code is developed, black-box methods can prove invaluable to testing throughout the cycle. Moreover, unlike white-box methods that concentrate only on the source code, black-box methods test both implementation and specification, thus ensuring specification consistency [3].

Specification-based test case generation approaches are broadly classified into three major types: (1) methods that use algebraic-based specifications, (2) methods that use model-based specifications, and (3) methods that use state-based specifications. Since it uses model-based specifications, the proposed approach falls into the second category [3].

2.1.1 Algebraic Approaches

Test case generation through path selection differs significantly from program level testing, where test cases are generated in the form of inputs. Among the methods that use algebraic-based specifications, two research projects come close to our approach: ASTOOT [8] and DAISTS [14].

In the ASTOOT [8] approach, each test case consists of a tuple of message sequences, along with tags indicating whether these sequences should put objects of the class under test into equivalent states and/or return objects that are in equivalent states. To execute tests, sequences are sent to objects of the class under test and then a user-supplied equivalence-checking mechanism is invoked. With this approach, if an algebraic specification of the class under test is available, testing can be automated simply. In the absence of a formal specification, tests are generated manually. In our approach, although some research issues remain to be solved, a UML specification for the Java class to be tested is required for automating the process.

The second research project in automated generation of specification-based test sets is DAISTS [14], a Data-Abstraction Implementation, Specification, and Testing System (DAISTS) specifically aimed at components implementing an ADT. Basically, the approach involves augmenting the program, implementing the ADT with an algebraic specification, and then generating cases that test the consistency of both the implementation and the specification. The test cases are derived from the algebraic specification axioms, and the entire process is automated. Since only inputs need to be supplied, the axioms serve as test oracles. Additionally, testing the axioms and the implementation individually is more effective.

In addition to these two algebraic-based specification test case generation approaches, Bouge et al. [5] suggested a logic programming approach to generating tests from algebraic specifications. Tsai, Volovik, and Keefe [30] used a similar approach, but started with relational algebra queries.

A primary difference exists between the two methods here discussed and the current work: while the former are based on algebraic specifications, the latter uses model-based specifications. Algebraic specifications often encourage the use of function-only operations and might suppress any explicit view of the content stored in an object. On the other hand, model-based specifications focus on abstract modeling of content. Unlike their algebraic counterparts, model-based specifications provide direct support for state-modifying methods and for operations with relational behavior.

2.1.2 Finite State Machine Approaches

State-based specification-based testing or test generation approaches focus on finite state machine (FSM) models of classes or programs [3, 25, 27]. An FSM model typically contains a subset of the states, as well as transitions supported by the actual component under consideration; it is developed by identifying equivalence classes of states that behave similarly. Test coverage is measured against the states and transitions of the model.

The current work follows a different approach: it draws a matrix from the component's specification and checks the cells to ensure that the generated tests satisfy the coverage techniques. In the proposed approach, there is no need to draw a state machine and slowly trace through it to generate test cases. Additionally, the work described here does not involve collapsing the state space of the component under test—or even directly modeling it. As a result, it gracefully degrades when only semi-or informal specifications are available.

To formalize the description of system level techniques, Offutt and Abdurazik [25] describe an approach, which until now has been described informally and confined to industry research. To derive test sets from the UML state charts, their approach adopts

state-based specification test data generation criteria. In the approach proposed by this research generates the test cases manually using Java API specification for the classes. To automate the current testing method we plan to follow the approach described by Offutt and Abdurazik and use Java UML specifications [25]. A second approach described by Offutt [27] bears similarity to the current work in that it also proposes a method to generate test cases manually, but it aims at a long-term goal of automation.

2.1.3 Model-based Approaches

To generate tests, model-based approaches often rely on a mathematical model of the specification. Since they derive from a mathematical basis, these methods provide solid ground upon which to analyze the specification and develop effective tests. In addition, their mathematical base makes it easy to measure their effectiveness and their coverage.

Since it also uses model-based specifications, the work of Hoffman, Strooper, and their colleagues is closely related [17]. Model-based specification languages, such as Z and VDM, attempt to derive formal software specifications based on mathematical models. Hierons [16] presents algorithms that rewrite Z specifications into a form that can be used to partition the input domain. Spence and Meudec [31] and Dick and Faivre [7] suggest using specifications to produce predicates and predicate satisfaction techniques to generate tests. Stocks and Carrington [32] and Ammann and Offutt [2] propose using a form of domain partitioning. Hayes [15] has suggested a dynamic scheme that uses run-time verification of the program.

Finally, Chang and Richardson [6] present techniques to derive test conditions from ADL specifications, a predicate logic-based language used to describe the relationships between inputs and outputs of program units.

2.2 Test Data Adequacy Criteria

Software testing strategies are as important as the methods chosen for measuring their effectiveness. In software testing, developing test adequacy criteria and measuring the effectiveness of the testing strategies have long preoccupied researchers. Once developed, a test data adequacy criterion can be used in two different ways.

First, an adequacy criterion is considered a 'stopping rule' that determines or measures whether enough testing has been done to permit stopping the procedure. Take, for example, the all-statements criterion. Each test case contains an input that exercises a particular part of the code. For this adequacy criterion, we can generate cases until they execute all code statements to be tested. Since software testing involves the program under test, the set of test cases, and the specification of the software, an adequacy criterion can be represented as a function C that takes a program p , specification s , and a test set t and then give a truth-value true or false. A test data adequacy criterion C is a function $C: P \times S \times T \rightarrow \{\text{true}, \text{false}\}$. $C(p, s, t) \rightarrow \text{True}$ means that t is adequate for testing program p against specification s according to the criterion C ; otherwise t is inadequate [34].

There is a second manner in which the test data adequacy criteria can be used. The function yields a real number, which indicates the current degree of testing that has been performed. This number can be used to determine testing adequacy: the greater the real number, the more adequate the testing.

Test data adequacy criteria can be helpful in many ways. An adequacy criterion specifies the requirements that must be met for testing to be considered adequate. In the case of software testing, especially in the unit testing phases where test cases are generated and fed to the code one by one, the adequacy criteria help by determining when enough testing has been done. Such criteria act both as a stopping rule to decide when to stop and, if required, as a means for generating additional tests. Additionally, these adequacy criteria are used at a higher level of testing where the code goes through the cycle of testing, debugging, and modifying until the required reliability level is achieved.

As previously noted, the test adequacy criteria can be classified into two basic types: white-box and black-box criteria. White-box criteria are based on the source code, while black-box criteria are based on externally observable features of the program or unit. The following definitions will assist in an understanding of these divisions.

- *Specification-based Criteria:* These criteria specify the required testing in terms of identified specification features or software requirements, so that tests are generated until all the identified requirements have been fully exercised.
- *Program-based Criteria:* These criteria are specified basing on the program under test. Based on the source code, the adequacy criteria specify what aspects of the code must be exercised to satisfy the adequacy criterion. Examples include criteria like all statements, all definitions, and all uses.
- *Interface-based Criteria:* These criteria are based on neither specification nor program source code, but on the range and type of input data.
- *Combined Specification-and Program-based Criteria:* These criteria are specified based on both the specification (requirements) and program (source code).

White-box criteria include both program-based and combined specification and program-based criteria, while black-box criteria include specification-based and interface-based criteria. Such adequacy criteria can be divided further into one of three types, depending upon the underlying test approach:

- Structural testing
- Fault-based testing
- Error-based testing

The latter two types are of less interest to this research. The structural testing approach, however, is important and can be divided into two types previously discussed: program-based and specification-based.

Program-based Structural Testing Criteria: There are two main groups of program-based structural test adequacy criteria: control-flow criteria and data-flow criteria. To give dependence-covering criteria, these two types are combined and extended. Most criteria from these two groups are based on the flow graph model of program structure. However, rather than using an abstract model of software structure, a few control flow

criteria define test requirements in terms of program text. The adequacy criteria defined in this model fall into three different types: control-flow, data flow, and dependency coverage.

The **control flow** coverage criteria are based on a control flow graph model of the program. The various criteria defined in this category include statement-coverage criterion, branch-coverage criterion, path-coverage criterion, cyclomatic number criterion, and multiple-condition coverage criterions.

The **data flow** coverage criteria are based on a data flow graph model of the program. The various criteria that are defined in this category are all definitions, all uses, and all definition-uses path criterion, among others.

Dependency-Coverage Criteria combine data and control flow criteria. In fact, these criteria serve as extended versions of control and data flow criteria.

Specification-based Structural Testing Criteria: These criteria are based on program specifications or requirements. The specifications can be used both to generate tests and to check for test data adequacy. The adequacy criteria can be developed from the specification syntax, as well as semantics. Zhu [34] describes the criteria basing on the former. These criteria are of basically two types: model-based formal functional specifications and algebraic formal functional specifications.

The **model-based formal functional specifications** specify the criteria basing on the program's model-based functional specifications, such as all-combination criterion, each-choice-used criterion, and base-choice-coverage criterion

The **algebraic formal functional specifications** derive criteria basing on the specifications to specify a set of properties that should be possessed by a program. In particular, an algebraic specification consists of a set of equations that must be satisfied by software operations. Checking if a program satisfies the specification means checking to determine whether the program satisfies all equations.

The work proposed by this study differs markedly from that undertaken by previous research: rather than describing new criteria, the current work will develop and assess practical test set generation strategies based on existing criteria. A large body of work exists on experimentally assessing the fault-detecting ability of various testing strategies or adequacy criteria [11, 12, 13]. For example, Frankl's work on statistically characterizing the effectiveness of adequacy criteria is notable. However, while her work focuses on statistically assessing the effectiveness of criteria by looking at large numbers of test sets, the work proposed here aims at assessing test sets using a specific strategy.

2.3 The Flow Graph Approach

The basic idea for this research was inspired by the work of Zweben, Heym, and Kimmich [35], who proposed test data adequacy criteria. These criteria were used by Edwards [9] in developing a method for generating black-box tests for software components. Unlike typical black-box testing methods, which involve drawing a state machine and tracing the states to generate test sets, Edwards' approach advocates generating a flow graph and applying the white-box analogues of all-DU-pairs and all-nodes.

According to this method, each of the modules in the specification forms a node or vertex in the flow graph. The constructor (if present) forms the initialization node of the graph, while the destructor forms the terminating node. An edge between two nodes in the graph indicates that the operations represented by the two nodes can come in a valid sequence in the component. Directed edges indicate operations that are feasible only in one direction, while undirected edges represent operations that can come in either combination. A path between two nodes indicates valid sequences of operations through which we can reach the second operation from the first.

According to Zweben *et al.* [35], for any given node in the flow graph, the parameters in the operation can be classified into definition and use basing on the following criteria. If the operation potentially alters the value of a parameter, then it is considered a definition of the parameter. All the parameters taken as input by the operation are considered as uses of the operation. In object-oriented languages, since it

both used and altered by the operation, the “self” or “this” parameter is considered to be both a definition and a use.

Given such a graph, any valid test on the component contains a sequence of operations that start from the constructor, follow some path in the graph, and end up in the destructor node. Zweben *et al.* [35] also propose other criteria like node, branch, definition, use, DU-path, and k -length path [35].

The current approach is similar to the work by Edwards [9] in that it also takes the Zweben’s criteria and applies them to the class specification. The only difference here is that while the work by Edwards uses flow graph representation of the specification, the current work generates tests from the matrix representation of the specification. The experimental evaluation performed here is on a larger scale and evaluates the approach against large number of other approaches.

Chapter 3: A Manual Method for Flow Graph Based Test Case Generation

3.1 The Matrix Method

Just as there are white-box and black-box methods, there are also manual and automated methods. The aim of the current research is to develop a manual black-box method. As previously noted, the basic idea for the research comes from Zweben *ET. Al.*'s [35] work on generating test cases from a flow graph developed from the component's specification. It adapts Zweben's idea of generating tests from the specification by applying flow graph coverage criteria.

The matrix method has been explored for two different test adequacy criteria: the all-nodes approach and the all-DU-pairs approach.

3.1.1 A Flow Graph for a Class Interface

The proposed approach involves developing a matrix from the specification and checking the cells in the matrix to generate a test case for each. To explain such an approach visually, this section develops a flow graph for an example class interface. This section explains the flow graph method proposed by Edwards [9] and section II talks about the matrix approach and explains it in relation to the flow graph approach.

To explain the process, the example we have chosen is a basic stack class [*Figure. 3-1*]. This example is a highly simplified version of one from the Java API. We chose this example for two reasons: it is easy to explain and, because there are a small number of methods in the class, the flow graph is easy to draw. Chapter 4 introduces an example that is larger and more realistic, as well as explains the process of test generation in more detail.

The labels 'Always' and 'Sometimes' on each arrow indicate that the two methods represented by that arrow can be called 'always in sequence' and 'sometimes in sequence,' respectively. Essentially, then, these labels should be interpreted as follows:

‘Always’ indicates that one method can always be called in sequence after another, and ‘Sometimes’ indicates that one method only sometimes leaves the object in a state where the second method can be called legitimately.

```
class Stack {
    Initialize();
    Object push(Object o);
    Object pop();
    int length();
    Finalize();
}
```

Figure 3-1 Hypothetical stack class interface

According to the flow graph approach, each of the methods in the class interface [Figure 3-1] form a node or vertex in the flow graph. The constructor (here, `initialize()`) forms the initial node of the graph, while the destructor (here, `Finalize()`) forms the terminating node. An edge between two nodes in the graph indicates that the operations represented by the two nodes can occur one after the other in a valid sequence in the component. A path between two nodes indicates valid sequences of operations through which we can reach the second operation from the first. In the stack class example the ‘A’ on the edge between `length()` and `push()` indicates that the `push()` method can be called ‘always’ after `length()`. Similarly the ‘S’ on the edge from `length()` to `pop()` indicates that `pop()` can be called only ‘sometimes’ after `length()` (when the stack is not empty).

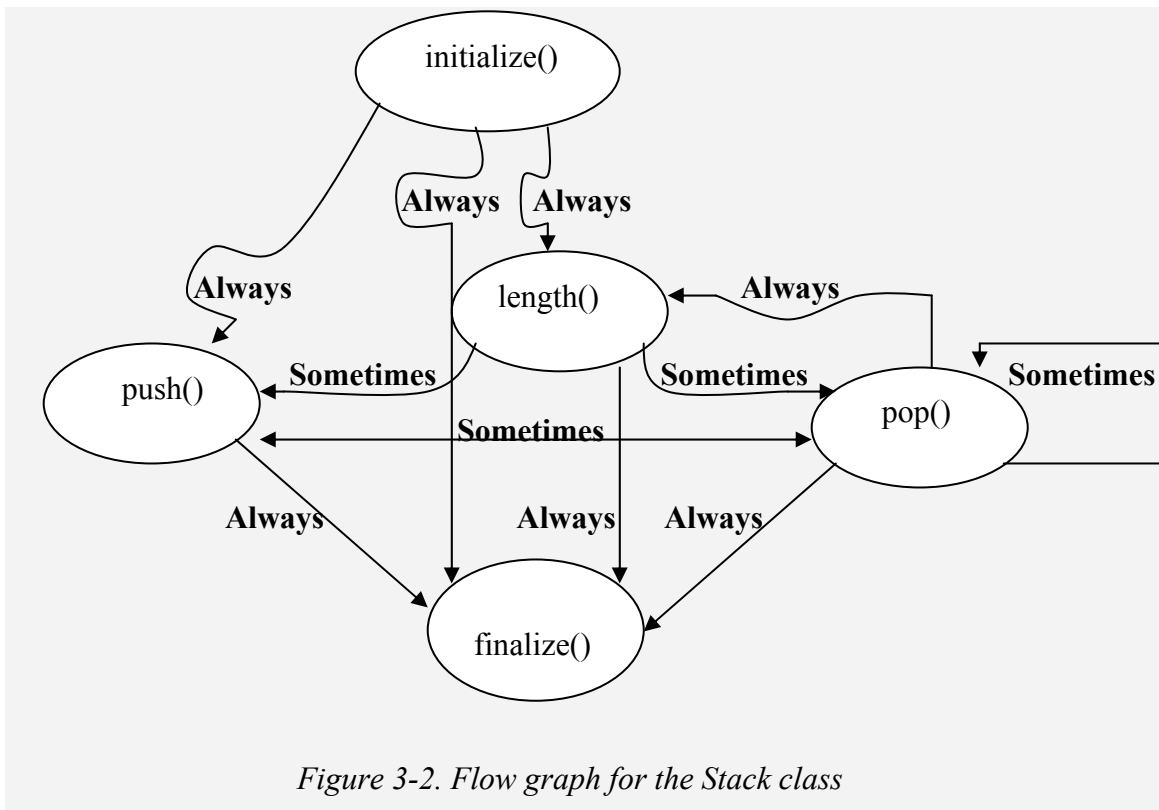


Figure 3-2. Flow graph for the Stack class

3.1.2 The Matrix Approach

The approach by Edwards[9] uses a flow graph representation of the specification, which has many disadvantages. The flow graph is generally drawn on a paper for generating the test cases. This works well when the number of the methods in the class is low but becomes very cumbersome to draw where the number of methods in the class under test increases. Hence we have chosen a matrix representation of the flow graph, which not only saves space by being more compact but also is more easy to use.

For all-DU-pairs, the model suggests drawing separate matrices for each data type definition [see Table 3-1]. For all-nodes, a matrix is not really required; instead, a list of methods is created, with N rows each representing one of the N methods.

For a flow graph, generating one test case involves selecting a path from the constructor node to the destructor node. For the matrix, a check in a cell indicates one test case, which includes the methods represented by the row and column of that cell in sequence. Coverage for the flow graph approach is phrased in terms of nodes in the

graph, while for the matrix approach; it is in terms of checking the cells. We select a test-generation criterion (like all-nodes or all-DU-pairs) and check the corresponding cells to generate tests satisfying the corresponding criterion. For example, to satisfy the all-DU-pairs criterion and generate test cases, we match against each other all the parameters of the same type over all the methods. A check placed in cell (x, y) indicates that the definition of the value in row “x” is being used by the method in row “y.” To generate a test case, the method represented by parameter in row “x” is called first, followed by the method represented by row “y.”

3.2 All-nodes Approach

3.2.1 Overview

Basically, the all-nodes approach described here adapts the white-box analogue of all-nodes criteria to the matrix generated from black-box specification or interface. The all-nodes approach suggests that every node in a flow graph, constructed from the specification of the class to be tested must be exercised at least once. Each test case corresponds to the exercising of one method, a node in the graph. Exercising of the node might often involve prerequisites, such as calling the constructor, and some initializing functions.

The current approach is defined for the Java language, with test cases generated per class. Java interface or class definitions contain method signatures, which are used to generate tests by hand. A method’s signature contains the name of the method, the types and names of its formal parameters, and the type of the return value.

3.2.2 Step-by-Step Procedure

All-nodes:

1. Create a list of methods exported by the class to be tested, including all constructors and finalizers.

2. Select a method from the list and create a new test case with this method call in the body.
3. If the selected method is not a constructor, choose a constructor to initialize an object and place the object initialization at the start of the test case body.
4. If the selected method cannot be applied to a newly created object, determine a legal state in which the method is callable, and insert other method calls following the constructor to transform the object to that state.
5. Write assertions to check that the method performs its required behavior and add them at the end of the test case.
6. Repeat the steps 2-5 for all the methods in the list, with each method leading to a test case.

Step 1: Create a list of methods exported by the class to be tested, including all constructors and finalizers.

The example stack class is shown in Figure 3-1. The method signatures can be extracted from the list and placed into spreadsheet.

Methods	checks
initialize()	X
push(Object)	X
pop()	X
finalize()	X
length()	X

Table 3-1. Matrix for all-nodes

Step 2: Select a method from the list and create a new test case with this method call in the body.

According to this step, a method must be selected from the list created in Step 1. For the all-nodes approach, each method or constructor in the class forms the basis for a test case. For our hypothetical stack class, we can select any of the methods from `push`, `pop`,

length and initialize. If we select the push method, we can construct the skeleton for the following test case:

```
public void testPush {
    Stack stk;
    stk.push(new Integer(4));
}
```

Step 3: If the selected method is not a constructor, choose a constructor to initialize an object and place the object initialization at the start of the test case body.

To form a test case, we add the constructor to the selected method (provided, of course, that the method is not already a constructor). The above test case takes the following form after this step:

```
public void testPush {
    Stack stk = new Stack();
    stk.push(new Integer(4));
}
```

Step 4: If the selected method cannot be applied to a newly created object, determine a legal state in which the method is callable, and insert other method calls following the constructor to transform the object to that state.

In certain cases, due to certain restrictions imposed on the methods, calling the method after the constructor is not feasible. To work correctly, setups for these cases must be created individually. For example, in our stack class, a `pop()` method cannot be called on an empty list. Hence, at no time can `pop()` be called immediately after the constructor. In the current step, we add an appropriate setup to make that sequence feasible. For example, we make sure that every `pop()` method is preceded by a `push()` method

Step 5: Write assertions to check that the method performs its required behavior and add them at the end of the test case.

To check for the correctness of the tests generated we must add proper assertions, which determine whether the actual result is indeed the expected one. Our example turns into the following test case:

```
public void testPush {
    Stack stk = new Stack();
    stk.push(new Integer(4));
    assertTrue(stk.pop().equals(new
    Integer(4)));
}
```

Step 6: Repeat steps 2-5 for all the methods in the list, with each method leading to a test case.

As indicated above, following the all-nodes criterion and applying it on one method will generate one test case. To satisfy the all-nodes test adequacy criterion, we must generate tests that correspond to all methods in the class interface. This involves generating a test case for each method or constructor in the list created in step 1.

3.3 All-DU-pairs Approach

3.3.1 Overview

The all-DU-pairs approach applies the white-box analogue of the all-DU-pairs test adequacy criterion to the matrix generated from a class interface. The term “all-DU-pairs” stands for “all definition use pairs,” which suggests that each test exercises one definition-use pair of a data type and that tests should be generated to cover all definition-use pairs over all data types.

The current approach is defined for the Java language, under which test cases are generated per class. Java interface of class definitions contain method signatures, which are used to generate tests by hand. A method’s signature contains the name of the method, the types and names of its formal parameters, and the type of the return value.

3.3.2 Definitions and Uses

Definition: If an object parameter is modified inside a method's body, the method is said to provide a definition of the new value. Some programming languages allow a programmer to specify the direction of information flow for each formal parameter as (in, out, in/out), based upon whether the corresponding actual value is used by the method (in), modified (out), or both used and modified (in/out). Java provides no such parameter mode declarations however. As a result the programmer must make a decision about which values accessible to a method are redefined, including the object receiving the method call, all of the method parameters and the method's return value. Normally the return value if any is considered as a definition. Treat all other values referenced as definitions when in doubt.

Use: An object or parameter is used by a method if its value is being accessed internally within the method's body. Again the programmer must make a decision about which values accessible to a method are actually *uses* including the object receiving the method call, as well as all the formal parameters. We can use the name *self* or *this* to refer to the object receiving the method call, which is normally always considered a *use*. Similarly because all parameters in Java are passed by value, it is reasonable to consider all parameters to be used when in doubt

3.3.3 Step-by-Step Procedure

All-DU pairs:

1. Create a list of methods exported by the class to be tested, including all constructors and destructors.
2. For each method in the list, identify the objects being referenced, including `self` (the object receiving the method call), all parameter values to the method, and the return value, if any.
3. For each object being referenced, identify its type and then classify it as a definition (its value is modified), a use (its value is read), or both.

4. Create separate matrices for each data type definition identified in step 3 by adding the definitions as rows and uses as columns. A spreadsheet is an excellent tool for this].
5. Select a matrix for generating test cases.
6. Select one cell in the matrix and create a new test case for it. Place the method call giving rise to the definition (row) followed by the method call giving rise to the use (column) in the body of the test case.
7. If the method associated with the selected definition is not a constructor, choose a constructor to initialize an object and place the object initialization at the start of the test case body.
8. If the method associated with the selected definition cannot be applied to a newly created object, determine a legal state in which the method is callable, and insert other method calls following the constructor to transform the object to that state.
9. Write assertions to check that the sequence of methods performs its required behavior and add them at the end of the test case.
10. Repeat steps 6-9 for all remaining cells in the selected matrix.
11. Repeat steps 5-10 for all remaining matrices.

Step 1: Create a list of methods exported by the class to be tested, including all constructors and finalizers.

For the classes currently selected, we have derived the class specification from the Java API specification available online [19]. The method signatures are extracted from the list and copied out into an Excel spreadsheet.

Step 2: For each method in the list, identify the objects being referenced, including `self` (the object receiving the method call), all parameter values to the method, and the return value, if any.

For each method in the class interface we have the return type if any, all the parameters, and one 'self' type which is none other than the object of the class itself. In this step we identify all such objects for all the methods from the list in step 1.

Step 3: For each object being referenced, identify its type and then classify it as a definition (its value is modified), a use (its value is read), or both.

The all-DU-pairs method works on the idea of data type definitions and uses. Hence, before we generate the tests we must first identify and categorize the data types identified in step 2 into definitions and uses.

The return type of every method is treated a definition. All parameters taken in by a method are treated as *used* by it. Additionally certain parameters are treated as definitions. `self` is considered both a definition and a use. In our stack class example, for example, every method is considered to a definition and use of the stack object receiving the method call.

Step 4: Create separate matrices for each data type definition by adding the definitions as rows and uses as columns. A spreadsheet is an excellent way to do this.

This step starts with figuring out the number of matrices to use. For each major data type having at least one definition **and** one use we create a matrix. In our stack class example, we create separate matrices for the stack and object data types. No matrix is created for the int data type since it has no uses.

Add each object reference identified in Step 3 to the matrix corresponding to its data type. If the reference is classified as a definition, add it as a new row in the matrix for its data type. If it is classified as a use, add it as a new column in the matrix for its data type. If it is classified as both, add both a new row and a new column.

Tables 3-2 and 3-3 depict the two matrices thus generated.

		USES			
		pop()	push()	length()	Finalize()
D E	Initialize()		X	X	X
	push(Object)	X	X	X	X
F	pop()		X		X
S	length()	X	X	X	X

Table 3-2. Matrix for Stack self-data type

		USES
		push(Object)
Definitions	push(Object)	X
	pop(Object)	X

Table 3-3. Matrix for Stack Object data type

Step 5: Select a matrix for generating test cases.

We can select either the stack or the object matrix for this purpose. Let us select the stack matrix.

Step 6: Select one cell in the matrix and create a new test case for it.

Place the method call giving rise to the definition (row) followed by the method call giving rise to the use (column) in the body of the test case.

Checking a cell from the selected matrix leads to one test case. Below for example the cell [1, 2] corresponds to the `initialize()` method from the row and `push()` method from the column. Calling the method represented by the row (`initialize()`) followed by the method represented by the column (`push()`) gives the following test case:

```
public void testStack1 {
    Stack stk = new Stack();
    stk.push(new Integer(4));
}
```

Step 7: If the method associated with the selected definition is not a constructor,

choose a constructor to initialize an object and place the object initialization at the start of the test case body.

In the current case since the definition method is already a constructor, no change is required and the test case remains as it is.

Step 8: If the method associated with the selected definition cannot be applied to a newly created object, determine a legal state in which the method is callable, and insert other method calls following the constructor to transform the object to that state.

In certain cases, due to certain restrictions imposed on the methods, calling again method immediately following the constructor is infeasible. To work correctly, setups for these cases must be created individually. For example, in our stack class, a `pop` method cannot be called on an empty list. Hence, at no time can `pop` be called immediately after the constructor. In the current step we add an appropriate setup to make that sequence feasible. For example, we make sure that every `pop` method is preceded by a `push` method. For the current test case no changes are required since the definition method is a constructor.

Step 9: Write assertions to check that the method performs its required behavior and add them at the end of the test case.

Since the tests generated by Step 4 are raw, we must make sure that the tests indeed are working correctly. For this purpose we must add proper assertions, which determine whether the actual result is indeed the expected one. Our example turns into the following test case:

```
public void testStack1 {
    Stack stk = new Stack();
    stk.push(new Integer(4));
    assertTrue(stk.pop().equals(new
    Integer(4)));
}
```

Step 10: Repeat steps 6-9 for all remaining cells in the selected matrix.

As indicated above, following steps 6-9 for one matrix cell will generate one test case. To satisfy the all-definition-uses criterion, we must generate test cases for all the definition use pairs for the selected matrix. This involves repeating the steps 6-9 over all the cells of the matrix.

Step 11: Repeat steps 5-10 for all remaining matrices.

Repeating steps 5-10 of checking cells over all the data type definition matrices will satisfy the all-DU-pairs criterion. For our stack example we repeat the process over the `stack` and `Object` matrices.

Since it is so crucial to both methods, the process of generating tests in JUnit—make the tests JUnit-compatible by adding assertions, checking for and handling exceptions, and wrapping them up in “test case” methods—will be discussed in detail in the next section.

3.4 Generating the Tests in JUnit

JUnit [18], a unit testing framework for writing repeatable tests in Java, is an instance of the XUnit architecture for unit testing frameworks. It is designed to support test driven development (TDD) as popularized by extreme programming. In TDD, tests are written before code and specify what behavior is desired. Since testing at the end of development, after coding is complete, will not help measure the progress of the development process, the TDD approach—with its “measure as you go” method—provides distinct advantages [18].

We used the JUnit framework for writing both all-DU-pairs and all-nodes tests. In addition to being user-friendly and highly functional, JUnit can be integrated easily with ANT [1] and other tools to facilitate the experimentation process.

To generate tests using the JUnit framework:

- All objects of the tested class are created, along with the required data types.
- The “setup” part of the code must contain all operations needed to initialize class variables and objects. The setup section often also contains the method calls that initialize the objects.
- Each of the tests is written as a separate test case embedded in a “test” function. For all-DU pairs, the test contains the function call of the “definition” method, followed by the function call of the “use” method, which uses the return value of the “definition” as one of its parameters. For all-nodes, a single method call is contained.
- For each test case, the expected outcome of the test case is calculated.
- Similarly, the “actual outcome” is calculated.
- An assertion is added at the end of the two function calls to test whether the expected value is equal to the actual value.

3.5 Infeasible Sequences

For the all-nodes approach, every function or node in the graph must be exercised. Often, exercising a node involves performing prerequisite activities, such as calling the constructor, and some initializing functions to reach the selected node in the graph. In turn, this often involves calling a constructor and then the node. Figure 3-2 illustrates the flow graph for the stack class example. To generate the test that exercises the `push()` method, we must call the constructor followed by the `push()` method.

On the other hand, if we want to generate a test case for `pop()` method, we cannot call it directly following the constructor since no feasible path exists between the two: the stack is empty. In order to call `pop()` method, we must first call the `push()` method. Hence, the test case for `pop()` would contain the constructor, followed by a call to `push()` followed by the call to actual `pop()`.

Similarly, for the all-DU-pairs approach, we must make sure we follow the all-nodes approach of calling the constructor and initialization functions before calling the actual “sequence” of functions. In addition to these problems, in the all-DU approach, calling the two functions in sequence might not always be feasible. For example, the stack class illustrated in Figure 3-2 explains this in more detail. The stack class contains the constructor, *pop*, *push*, and optional destructor methods. To call the *push* method, we must first call the constructor. However, to call the *pop* method we need to call both the constructor and *push* methods: we simply cannot call *pop* on an empty stack. To assure this condition, we must ensure that every time we call *push* before *pop*. Functions that cannot be called in sequence are called “infeasible.”

When generating tests, we must make such “infeasible” sequences feasible by adding appropriate initialization functions. For example, in the stack class, the *pop* function cannot be called immediately after *pop*. To handle this case, we must make sure that the stack contains at least two elements before this sequence is called: we insert additional *push* functions before the “infeasible pop sequence.” Care must be taken to insert such initialization functions only at the beginning of the “sequence,” because any added in the middle would spoil the concept of “testing definition use pairs.”

3.6 A Comparison between All-DU-pairs & All-Nodes

Both the all-nodes and all-DU-pairs approaches are specification-based and the respective analogues of white-box methods. In the all-nodes approach, each test case consists of one method corresponding to one node in the flow graph, and the tests for all methods are generated to satisfy the all-nodes criterion. Hence, each test case for all-nodes basically contains one method call. On the other hand, for the all-DU-pairs approach, we generate “pairs of function calls,” with each test case containing one pair. To satisfy the all-DU-pairs criterion, we generate all combinations of the functions in pairs, which corresponds to the number of methods squared.

For any given class, if the number of methods are “N,” the all-nodes approach generates $O(N)$ test cases, while the all-DU pairs approach generates $O(N \times N)$ test cases. Additionally the average number of lines in tests generated by the all-nodes approach is

one, while the average number in those generated by the all-DU pairs approach is *two*. We propose that both the average number of lines per test and the total number of tests identify the all-DU-pairs approach to be superior. As indicated later, experiments conducted in the evaluation phase reveal the same results.

Chapter 4: Tutorial

4.1 Introduction

This research aims to devise a user-friendly method of generating test cases that can be put to immediate use by practitioners. To that end, while Chapter 3 described the basic method and its underlying theories, this one takes a practical approach by devising a simple tutorial by which users can quickly and efficiently learn it. Quite simply, we take as an example the Linked List class and use it to explain the method. We use the same class for describing both the all-nodes and all-DU-pairs approaches. The second section of this chapter concentrates on the all-nodes approach and the third section on the all-DU-pairs approach.

4.1.1 LinkedList Example

To explain the all-nodes approach, consider the `java.util.LinkedList` class. Two criteria helped us choose this class: Its interface involves several other classes and it generates a large number of tests. The `LinkedList` class specification is available online [19].

Figure4-1 shows a simplified interface view of the class, the `LinkedList`.

```
public void interface LinkedList {
    LinkedList();
    LinkedList(Collection c);
    void add(int index, Object element);
    boolean add(Object o);
    boolean addAll(Collection c);
    boolean addAll(int index, Collection c);
    void addFirst(Object o);
    void addLast(Object o);
    void clear();
    Object clone();
    boolean contains(Object o);
    Object get(int index);
    Object getFirst();
    Object getLast();
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator listIterator(int index);
    Object remove(int index);
    boolean remove(Object o);
    Object removeFirst();
    Object removeLast();
    Object set(int index, Object element);
    int size();
    Object[] toArray();
    Object[] toArray(Object[] a);
}
```

Figure 4-1. A Simplified interface capturing the main features of java.util.LinkedList

The LinkedList class generates the flow graph shown in Figure. 4-2. Since the specification consists of 23 methods, it is rather large, which makes the graph difficult to draw. As a result, we divide the methods in the class into the following categories and draw the graph between them. The restrictions apply to the entire category of the methods rather than to individual methods. Hence, this optimization is safe.

Constructors:

```
LinkedList()
LinkedList(Collection c)
```

Methods that add to the list: There are six :


```
void add(int index, Object element)
boolean add(Object o)
boolean addAll(Collection c)
boolean addAll(int index, Collection c)
void addFirst(Object o)
void addLast(Object o)
```

Clear Method: This falls into a category of its own

```
void clear()
```

Methods that cannot be called on empty List: There are seven:

```
Object get(int index)
Object getFirst()
Object getLast()
Object remove(int index)
boolean remove(Object o)
Object removeFirst()
Object removeLast()
Object set(int index, Object element)
```

Methods that do not explicitly modify the list: There are eight:

```
Object clone()
boolean contains(Object o)
int indexOf(Object o)
int lastIndexOf(Object o)
ListIterator listIterator(int index)
int size()
Object[] toArray()
Object[] toArray(Object[] a)
```

The flow graph generated for this class in Figure. 4-2 is **not** part of the process, but is shown here to facilitate a deeper understanding of the method.

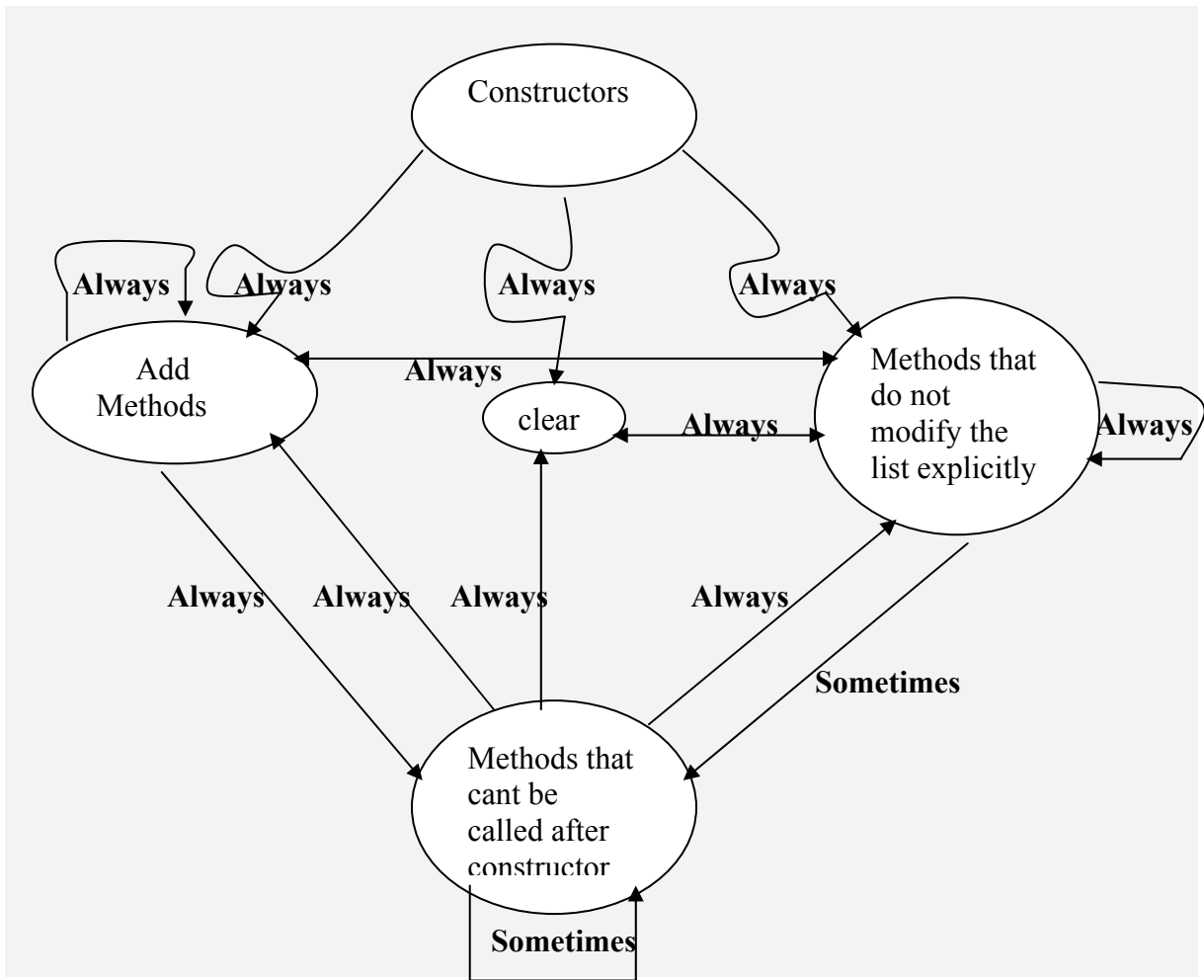


Figure. 4-2. Flow graph for Linked List class specification

The labels 'Always' and 'Sometimes' on each arrow indicate that the two methods represented by that arrow can be called 'always in sequence' and 'sometimes in sequence,' respectively. This kind of notion is derived from the languages, which have preconditions for calling the methods. Since the Java classes we have taken don't have any preconditions, any method can be called any time except when certain sequences generate exceptions.

Essentially, then, these letters should be interpreted as follows: 'A' indicates that methods can be called in sequence, and 'S' indicates that calling them in sequence generates exceptions which need to be handled.

4.2 All-Nodes Approach

The all-nodes approach described here is defined for java language, which generates test cases per class. Java class specifications contain the class method signatures, and the specification is used to generate tests by hand. The signature for a method contains the name of the method, the types and names of the variables it uses, and the type of the return value.

Here we have explained the method with the help of the example of the LinkedList class, which has a number of methods. The step-by-step procedure that is described in chapter 3 has been taken and the steps are explained in detail with the help of the LinkedList class example.

Step 1: Create a list of methods exported by the class to be tested, including all constructors and destructors.

Specifications for the selected classes are derived from online Java API specifications. Information for the Linked List class also is available online [19]. The method signatures can be extracted from the list and copied out into an Excel spreadsheet. The specification for our Linked List class is available from Figure. 4-1.

Step 2: Select a method from the list and create a new test case with this method call in the body.

Let us select the first method in the list. The add () method selected can be formed as a test case by creating an object of the class and calling the method on the object.

Step 3: If the selected method is not a constructor, choose a constructor to initialize an object and place the object initialization at the start of the test case body.

In the current case we have two constructors, one that takes no parameters and the other that takes as its parameter a “collection”:

```
LinkedList();  
LinkedList(Collection c);
```

To generate the simplest test case, we select the first constructor and call it before the selected `add()` method, which gives the following test case:

```
public void testAllDUpairs {
    LinkedList lobj;
    int i=0;
    lobj= new LinkedList();
    lobj.add (i, new Integer(4));
}
```

Step 4: If the selected method cannot be applied to a newly created object, determine a legal state in which the method is callable, and insert other method calls following the constructor to transform the object to that state.

In certain cases calling the selected method immediately after the constructor is infeasible. These cases need to be handled separately and setup procedures devised for each. The following three examples illustrate three cases:

where the selected method is a constructor,

```
public void testAllnodes1 {
    LinkedList lobj;
    lobj= new LinkedList();
}
```

where the selected method can be called on an empty list,

```
public void testAllnodes2 {
    LinkedList lobj;
    int i=0;
    lobj= new LinkedList();
    lobj.add (i, new Integer(4));
}
```

For the third test case, consider the method `void get(int index)`, which cannot be called directly after the constructor. If called, this method generates an `IndexOutOfBoundsException`. After adding the required setup to handle the exception, the third test case becomes

```
public void testAllnodes3() {
    LinkedList lobj;
    int i=0;
    int excep=1;
    LL1= new LinkedList();
    try {
        lobj.get(0);
    } catch(IndexOutOfBoundsException e) {
        e.printStackTrace();
        fail();
    }
}
```

Step 5: Write assertions to check that the method performs its required behavior and add them at the end of the test case.

Since the tests generated by Step 3 are raw, we must make sure they work correctly by adding proper assertions, which check whether the actual result is indeed the expected one.

To generate the assertions, the following example calculates and compares expected and obtained results:

Example:

```
Object element;
lobj= new LinkedList();
lobj.add(element);
```

Expected Result:

This test case adds an object to an empty list. Hence, after the operation, the expected state of the Linked List object is not an empty list but a list with a single element in it. The expected result is the Linked List Object 'lobj' with a single element 'o' inside it.

Assertion:

We must test the case for its expected result—that is, a linked list with one object inside it.

We can test this in two ways: determine if the list is no emptier or check it for the presence of the Object O. This leads to one of the following two assertions:

```
assertTrue(!lobj.isEmpty());  
assertTrue(lobj.contains(element));
```

This leads to the complete test case:

```
lobj= new LinkedList();  
lobj.add(element);  
assertTrue(lobj.contains(element));
```

Step 6: Repeat the above steps over all the method calls, with each method leading to a test case.

As indicated above, applying the all-nodes criterion to one method will generate one test case. To satisfy the exact all-nodes criterion, we must generate tests that correspond to all methods in the class specification, which involves matching each of the methods with the constructor.

4.3 All-DU-pairs Approach

The all-DU-pairs approach is defined for java language, with test cases generated per class. The Java class specification contains the method signatures of the class and is used to generate the tests manually. The signature for a method contains its name, the types and names of the variables it uses, and the type of the return value.

Here we have explained the method with the help of the example of the LinkedList class, which has a number of methods. The step-by-step procedure that is described in chapter 3 has been taken and the steps are explained in detail with the help of the LinkedList class example.

Step 1: Create a list of methods exported by the class to be tested, including all constructors and destructors.

This is similar to step 1 of all-nodes and gives the same results

Step 2: For each method in the list, identify the objects being referenced, including `self` (the object receiving the method call), all parameter values to the method, and the return value, if any.

For example consider:

```
boolean addAll(int index, Collection c)
```

The objects being referenced in this method call are return type (boolean), parameter (int), collection (parameter) and the LinkedList class object (self).

Step 3: For each object being referenced, identify its type and then classify it as a definition (its value is modified), a use (its value is read), or both.

All the return-types of a method are categorized as Definitions, while all of its parameters are categorized as “Uses” and/or definitions. Let us take the same method:

```
boolean addAll(int index, Collection c)
```

As inputs, the above function takes an integer parameter and a collection parameter along with `self`, it returns as output a boolean value. Hence, the definitions and uses for the above method are

Definitions: boolean, this

Uses: int, Collection, self

This can be represented visually in tabular form, with definitions on the left and uses on the right.

Definitions	Ret-type	Method call	Uses
Self, boolean	boolean	addAll(int index, Collection c)	int, Collection, Self

The LinkedList class contains: two constructors and 23 methods. Identifying all definitions and uses for all the methods in the class and representing all of them in the above form produced the information in Table 4-1. The left column of this table contains method definitions for the method in that row, while the right column contains uses for that method.

Definitions	Return type	Method call	Uses
Self		LinkedList()	
Self		LinkedList(Collection c)	Collection
Self	void	add(int index, Object element)	int, Object, Self
Self, boolean	boolean	add(Object o)	Object, Self
Self, boolean	boolean	addAll(Collection c)	Collection, Self
Self, boolean	boolean	addAll(int index, Collection c)	int, Collection, Self
Self	void	addFirst(Object o)	Object, Self
Self	void	addLast(Object o)	Object, Self
Self	void	clear()	Self
Self, Object	Object	clone()	Self
Self, boolean	boolean	contains(Object o)	Object, Self
Self, Object	Object	get(int index)	int, Self
Self, Object	Object	getFirst()	Self
Self, Object	Object	getLast()	Self
Self, int	int	indexOf(Object o)	Object, Self
Self, int	int	lastIndexOf(Object o)	Object, Self
Self, listIterator	listIterator	listIterator(int index)	int, Self
Self, Object	Object	remove(int index)	int, Self
Self, boolean	boolean	remove(Object o)	Object, Self
Self, Object	Object	removeFirst()	Self
Self, Object	Object	removeLast()	Self
Self, Object	Object	set(int index, Object element)	int, Object, Self
Self, int	int	size()	Self
Self, Object[]	Object[]	toArray()	Self
Self, Object[]	Object[]	toArray(Object[] a)	Object[], Self

Table 4-1. Definitions and Uses separated for Linked List class

Step 4: Create separate matrices for each data type definition identified in step 3 by adding the definitions as rows and uses as columns.

From the left column of Table 4-1 we can see that for the LinkedList class, we have around 44 definitions (**25 self, 5 boolean, 8 Object, 3 integer and 2 Object []**, **1 listIterator**). The total number of methods defined as “uses” numbers around 23. If we represent all these in a single matrix (K- Number of definitions, N1- Number of uses) we get a 40x23 matrix, which becomes very difficult to manage and manipulate. Hence, we opt for separate matrices for each data type involved.

For example, there are total of three int definitions and six int uses for each. Table 4-2 depicts this visually.

		Uses					
		add(int index, Object element)	addAll(int index, Collection c)	get()	listIterator()	remove()	set()
D e f s	indexOf(Object)						
	lastIndexOf(Object)						
	size()						

Table 4-2. Matrix for ‘int’ references for LinkedList class

The matrix for the ”self” definition of the constructor LinkedList() would have a 25*23 dimension, as shown in Table 4-3.

D e f i n i t i o n s	Uses					
	add(int, Object)	add(Object)	Addall(collection)	Addall(int, collection) ...	ToArray2	
LinkedList()						
LinkedList(Collection)						
add(int, Object)						
add(Object)						
addAll(int, collection)						
addFirst()						
addLast()						
clear()						
clone()						
contains()						
get()						
getFirst()						
getLast()						
indexOf()						
lastIndexOf()						
listIterator()						
remove(int)						
remove(object)						
removeFirst()						
removeLast()						
set()						
size()						
toArray()						
toArray(Object[])						

Table 4-3. Matrix for LinkedList data type references

D e f i n i t i o n s	Uses					
	add(int, Object)	add(Object)	addFirst(Object)	addLast(Object) ...	set()	
clone()						
get()						
getFirst(Object)						
getLast(Object)						
remove(Object)						
removeFirst(Object)						
removeLast(Object)						
set()						

Table 4-4. Matrix for Object data type for Linked List class

The definitions for Object [], ListIterator and boolean have no use to exercise upon. Hence, to generate matrices for them, we manufacture uses by adding print statements to exercise each of their definitions. As a result, the matrices shown in Tables 4-5, 4-6, and 4-7 are formed.

		Uses
D e f s		println()
		add(Object)
		addAll(Collection)
		addAll(int, Collection)
		contains(Object)
		remove(Object)

Table 4-5. Matrix for boolean data type for LinkedList class

		Uses
D e f s		println()
		toArray()
		toArray(Object[])

Table 4-6. Matrix for Object[] data type for LinkedList class

		Uses
D e f s		println()
		listIterator(listIterator)

Table 4-7. Matrix for listIterator data type for LinkedList class

Although we did not do it for our tests, treating each of the documented exceptions as a definition and generating tests that use those definitions helps in the process of detecting bugs. For the LinkedList class, Table 4-8 represents such a matrix.

		Exception	Method	Uses
D e f s				println()
		NoSuchElementException	removeLast	
		IndexOutOfBoundsException	addAll(I, C)	
		UnsupportedOperationException	clear	
		ArrayStoreException	toArray()	

Table 4-8. Matrix for 'exceptions' as definitions for LinkedList class

Step 5: Select a matrix for generating test cases.

For this step let us select the matrix for LinkedList references, which primarily contain definitions of and uses of self.

Step 6: Select one cell in the matrix and create a new test case for it.

Place the method call giving rise to the definition (row) followed by the method call giving rise to the use (column) in the body of the test case.

In the LinkedList data type matrix, consider cell [3, 3]. To generate the test case we must first call the method that corresponds to the definition: `void add (int index, Object element)` followed by that for the use: `boolean add(Object o)`

```
public void testallDU1 {
    LinkedList lobj;
    lobj.add(0, new String("abc"));
    lobj.add(new String("cab"));
}
```

At this point the matrix can be represented by Table 4-9.

	add(int, Object)	add(Object)	addAll(Collection)...	toArray(Object[])
LinkedList()						
LinkedList(Collection)						
add(int, Object)		X				
add(Object)						
addAll(Collection)						
.						
.						
.						
toArray(Object[])						

Table 4-9. All-DU-pairs Matrix in progress

Step 7: If the method associated with the selected definition is not a constructor, choose a constructor to initialize an object and place the object initialization at the start of the test case body.

Since in the present case, the defining method is not a constructor, we select a constructor to add to the test case:

```
public void testallDU1 {
    LinkedList lobj= new LinkedList();
    lobj.add(0, "abc");
    lobj.add("cab");
}
```

Step 8: If the method associated with the selected definition cannot be applied to a newly created object, determine a legal state in which the method is callable, and insert other method calls following the constructor to transform the object to that state.

In the present case, the method providing the definition can be called after the constructor and no additional set up is required.

Step 9: Write assertions to check that the sequence of methods performs its required behavior and add them at the end of the test case.

On adding the assertions, the test case takes the following form:

```
public void testallDU1 {
    LinkedList lobj= new LinkedList();
    lobj.add(0, "abc");
    lobj.add("cab");
    assertTrue(lobj.lastIndexOf("cab")== 1);
}
```

Step 10: Repeat steps 6-9 for all remaining cells in the selected matrix.

Repeating Steps 6-9 over all the 25 definitions (rows) and 23 uses (columns) for the LinkedList data type matrix, we get around 575 tests.

Step 11: Repeat the steps 5-10 for all the remaining matrices

Table 4-10 summarizes the number of test cases for the LinkedList class from each data type matrix

Data type	#Tests
S=25*23	575
O=8*9	72
I=3*6	18
Total	665

Table 4-10. Distribution of tests over data types for Linked List class

For the definitions and uses of self, every method (including constructors) is a self-definition, for a total of 25 methods. For each of these definitions, 23 methods (excluding constructors) serve as uses. Hence, the total number of self def-use tests generated is $25*23 = 575$ tests. Similarly, the object data type matrix generates 72 tests and the integer data type matrix generates 18. The tests generated are presented in appendix A.

Chapter 5: Experimental Evaluation

The current study contributes to research in two ways: by developing a method and by comparing it to other standard methods in order to evaluate its effectiveness. This chapter concentrates on the second contribution.

The experimental evaluation section of this research compares the proposed approach to a selection of four other testing methods in order to determine how effectively it catches “bugs.” In order to determine this bug detection capability, we constructed experiments using mutation-testing approaches. A sample population of eleven classes was selected to be representative across various factors (average and total Lines of code, number of direct, inherited and total methods and average method size). For each of these eleven classes, mutation [20] was used to seed defects uniformly throughout its implementation. The effectiveness of each approach is measured by the percentage of mutants it kills. Since we are comparing the five approaches across eleven samples, this leads to a 5 X11 experimental design.

5.1 The Five Approaches

Our approach includes two versions: all-nodes and all-DU pairs. In the all-nodes approach, each test case consists of one method corresponding to one node in the flow graph, and tests for all the methods are generated to satisfy the all-nodes criterion. Thus each test case for all-nodes contains one method call. The all-DU-pairs approach, however, generates “pairs of method calls.” Each test case contains one pair of the method calls. To satisfy the all-DU-pairs criterion, we generate all combinations of the definitions and uses in pairs, which generate tests in the order of the total number of methods squared.

For any given class, with “N” methods the all-nodes approach generates N test cases, while the all-DU-pairs approach generates O (N X N) test cases. Additionally, the all-nodes approach generates tests that contain an average of two lines, while the all-DU pairs approach generates tests with approximately three lines. Our research proposes that the all-DU-pairs approach performs more effectively than all-nodes because of the

pairing of definitions and uses. Hence, the experimental setup considers the all-nodes approach as one of the five methods. The following sections explain each of the three testing approaches against which all-DU-pairs and all-nodes are compared. Also noted are the ways each approach generates test cases and the significant differences between each and the proposed approach.

5.1.1 Random Test Case Generation

As its name suggests, random test case generation involves generating test cases in a random fashion rather than following a systematic scheme. In order to ensure the random generation process functions correctly, test cases are picked not by hand but with the help of a C program. Given a class, the random approach takes as input its method signatures. Selecting two methods in sequence in a random fashion forms a complete test case. In order to make the random tests Junit-compatible, assertions are added manually, along with a setup section to initialize the objects.

To provide an adequate basis for comparison, the random test suites were generated so that they had the same number of test cases as test suites from all-DU-pairs approach. Additionally, to ensure that the experimental results are valid, we made sure that for both approaches, the average number of lines per test case was equal. Since the all-DU-pairs approach generates pairs of method calls in a systematic fashion by pairing up all definitions and uses it generates all possible combinations definition-use sequences. However, since the random approach generates the same number of test cases but in no particular order, it will not necessarily generate all combinations of definition-use sequences.

Hence, the possibility exists that the random approach generates a smaller number of unique tests, and more redundant ones, than would the all-DU pairs approach. One of the reasons comparing the random approach to our method is to see whether the difference in the number of original tests effectively detects more bugs. The main reason for including the random approach is that it acts as a baseline for the other approaches to compare against. A random approach is literally no approach at all. Hence any structured approach should be able to perform better against the random approach if it is to be

considered effective. Hence the aim here is to compare the two versions of our method to the random approach to see if these structured approaches perform better than a baseline approach like random.

5.1.2 JTest –Parasoft White-box Testing Tool for Java

JTest [21] is Parasoft’s white-box testing tool for generating tests automatically from Java source code. It works by generating tests that exercise each method in the class. In practice, though, it generates more than one test for each method. Our approach differs from JTest in that it is a black-box method. We included JTest as one of the methods in the experimental setup to test the effectiveness of our approach against a traditional white-box testing method.

Although JTest is a white-box tool, it focuses on each method in isolation and generates few tests per method as a result. Thus while the all-DU-pairs method produces an average of two method calls per test case, JTest generates tests that generally contain one method call. In this regard the JTest approach more directly compares to the all-nodes version of the matrix approach, which also contains an average of one method call per test case.

5.1.3 OATS – Orthogonal Array Testing Strategy

The Orthogonal Array Testing Strategy (OATS) [28] is a systematic, statistical way to test pair-wise interactions. It provides representative, uniformly distributed coverage of all variable pair combinations, which makes it particularly useful for integration testing of software components (especially in OO systems where multiple subclasses can be substituted as the server). The most widely used application for OATS involves software components, which require testing multiple combinations of configurable options.

Orthogonal arrays are two-dimensional arrays of numbers, which possess an interesting quality: by choosing any two columns in the array you receive an even distribution of all the pair-wise combinations of values in the array. The research community has developed a number of such arrays for public use [28].

Each orthogonal array specifies a set of runs, which correspond to test cases in our experimental design. Each run consists of a combination of pre-specified factors—in the case of our design, the method calls—that might be included or excluded from it. By selecting an intelligent and optimum number of such “factors,” OATS aims to reduce the number of their possible combinations while still providing adequate coverage. While in the all-DU-pairs approach, we combine all the definition-use pairs, in OATS we use the orthogonal arrays to assist in selecting the method combinations. Although this method generates very few test cases, each test case on the average contains more method calls than do those created by either the all-DU-pairs or the all-nodes methods.

5.2 Selecting Classes from java.util.*

The experimental setup includes two dimensions: approaches and classes. This section discusses the second dimension of the experimental design, the classes chosen as test subjects from the java.util package provided as part of Sun’s Java 2 SDK V 1.4 [19]. The java.util package contains a total of 36 classes. Table 5-1 summarizes the features of all 36 java.util classes using the following statistics:

- Number of implemented methods
- Number of inherited methods
- Total number of methods
- Number of lines of code
- Average number of lines of code per method

Name	Direct	Inherited	Total Methods	Total LOC	Average LOC
AbstractCollection	15	12	27	130	8
AbstractList	17	29	46	355	20
AbstractMap	16	8	24	215	13
AbstractSequentialList	8	38	46	52	6
AbstractSet	4	33	37	43	10
ArrayList	22	28	50	212	9
Arrays	55	11	66	1059	19
BitSet	15	7	22	224	15
Calendar	46	7	53	432	9
Collections	31	11	42	1076	35
Date	16	7	23	413	26
Dictionary	8	11	19	13	2
EventObject	3	10	13	16	5
GregorianCalendar	23	38	61	852	37
HashMap	17	12	29	522	30
HashSet	12	24	36	71	6
Hashtable	34	7	41	560	16
LinkedList	25	27	52	342	13
ListResourceBundle	4	19	23	64	16
Locale	24	7	31	542	22
Observable	10	11	21	46	4
Properties	10	27	37	266	26
PropertyPermission	6	11	17	245	40
PropertyResourceBundle	3	19	22	50	16
Random	12	11	23	88	7
ResourceBundle	11	11	22	414	37
SimpleTimeZone	22	18	40	606	27
Stack	6	55	61	39	6
StringTokenizer	9	11	20	112	12
Timer	9	11	20	196	21
TimerTask	4	11	15	27	6
TimeZone	19	10	29	769	40
TreeMap	22	14	36	926	42
TreeSet	19	22	41	110	5
Vector	46	13	59	336	7
WeakHashMap	12	21	33	200	16

Table 5-1. Classes from java.util package ordered on various criteria

We selected eleven classes that captured large, average and small values across Table 5-1. That is, on each of the five criteria, the selected package contains classes that have *high*, *medium*, and *low* values. By selecting classes over these criteria, we can control the effect of each factor on the outcome. We plan to find the correlation between significant factors like *total number of lines of code* and *total number of methods over number of bugs detected*. The eleven classes selected were:

- BitSet
- Hashtable
- LinkedList
- Stack
- TreeMap
- TreeSet
- Vector
- Observable
- StringTokenizer
- Date
- GregorianCalendar

5.2.1 Test Suites

Applying the five testing methods over the eleven classes selected creates 55 test suites. Of these, both the random and the all-DU-pairs methods have approximately the same number of test cases over all the classes. Although the JTest test suites come next highest in terms of the number of tests, the OATS [28] suites are the next highest in terms of the total LOC. This occurs because OATS generates tests that on average contain about seven method calls. The all-nodes approach produced the fewest test cases on average as Table 5-2 specifies the number of tests generated in each case.

CLASS	All-DU-pairs#	Random#	JTest#	All-Nodes#	OATS #
BitSet	283	283	39	18	24
Hashtable	465	465	59	19	32
LinkedList	665	665	55	23	40
Stack	420	420	7	17	32
TreeMap	449	449	40	18	32
TreeSet	302	302	33	15	24
Vector	215	215	26	12	24
Observable	56	56	8	7	8
HashSet	68	68	32	6	8
Date	108	108	25	12	16
GregorianCalendar	388	388	30	14	24
Total	3419	3419	354	161	264

Table 5-2. Number of Tests for each of the 5 approaches over the eleven classes

5.3 Running the Tests

As our build tool, we chose Java-based Apache ANT [1], which is similar to `make`. Apache Ant writes all necessary operations in a “`build.xml`” file, which must be in the “current working directory.” Just a single command—“`ant`”—runs all operations. In the current experimental setup, each class’s source files are gathered from the `java.util` package into a directory. The Java source file is then compiled, a class version is created, and the class path is modified to include the directory that contains the class file. The test file for the corresponding class is then compiled with the class path thus set. Every reference to the class name in the test file now takes that of the original `java.util` package but the one to which the class path currently points. When the test file is executed it works on the version we created. It should detect any bugs that might be present (or purposefully injected) in our version of the source file. The basic idea involves creating several mutant versions of the source file and allowing our test file to run on each. We then collect results into a directory. [1]

Since completing these procedures manually entails a lot of work, plus there are good chances of introducing errors into the process, at this point the ANT build tool [1] proves vital. The basic operations can be written in the following steps:

- Compile our version of source file.
- Modify the class path so it points to the directory that contains the source.
- Compile the test driver.
- Run the test driver on the source.
- Repeat the above four steps for all five approaches.
- Repeat the above five steps for all eleven classes.

5.4 Mutation Testing

Mutation testing involves systematically seeding defects one bug at a time into the original source file in order to create of it mutant versions. The test suites are then run on every one of these versions with the expectation that they will catch all bugs. When the test suite successfully finds a bug within the mutant version, it is “killed”. The higher the number of mutants it kills, the more effective the test suite is.

5.4.1 Jester – a Mutation Testing Tool for Java

To create mutant versions for the source files we used the open source Java mutation testing tool Jester [20]. According to Moore [20], “Jester is a test tester for JUnit tests: it modifies the source in a variety of ways, and checks whether the tests fail for each modification. Jester indicates code changes that can be made that do not cause the tests to fail. If code can be modified without the tests failing, it either indicates that there is a test missing or that the code is redundant. Jester can be used to gain confidence that the existing tests are adequate, or give clues about the tests that are missing”.

When given a source file and test suite as input, internally Jester creates mutant versions of the source, runs the test suite on them, and gathers the results. However, the output data contains limited information: it does not indicate which tests failed. Moreover, the tool cannot by itself be integrated into an ANT build file so that the results can be gathered and used. Hence, we altered Jester’s source code (available openly) so that as mutants are created internally, the mutant versions are output as separate Java files.

One major problem with mutant testing involves the performance overhead that occurs when such a large number must be run simultaneously. To deal with this problem,

Offutt [26] proposes an optimum set of mutation operators that if run will function adequately to detect bugs and reduce the performance overhead. After inspecting how Jester generates mutants, we discovered its set of mutation operations are limited in comparison to the set proposed by Offutt.

Jester implements the following mutating operations:

1. Increment numbers (Mutate 0 to 1, 5 to 6, 9 to 0)
2. Flip boolean values (true to false and vice versa)
3. Mutate if (condition) to if (true | | condition)
4. Mutate if (condition) to if (false && condition)
5. Mutate ++ to -- and vice versa
6. Mutate != to == and vice versa

Offutt proposed an optimum set of the following five basic operators:

1. ABS – This forces each arithmetic expression to take a zero, positive and negative value
2. AOR - This replaces every arithmetic operator with all the legally possible operators
3. LCR – This replaces every Logical connector (AND, OR) with all legally possible connectors
4. ROR – This replaces every relational operator (<, >, = etc...) with all legally possible relational operators
5. UOI – This inserts unary operators in front of expressions

Although it implements certain other operations that are not present in the optimum set, Jester implements only **parts** of ROR and AOR and omits the remaining operations. To address this limitation, we modified Jester's source code to include the basic operations proposed by Offutt's optimum set. The basic operations then generate mutants as Java

source files. Table 5-3 indicates the numbers of mutants generated for each of the eleven classes

Java Class	Number of Mutants
BitSet	328
Hashtable	454
LinkedList	263
Stack	36
TreeMap	573
TreeSet	23
Vector	270
Observable	17
StringTokenizer	135
Date	559
GregorianCalendar	1708

Table 5-3. Number of mutants generated for eleven classes

5.4.2 Running the Mutants

Once created, the mutants are placed in a separate directory containing the test suite of the corresponding class. For each class and for each approach, the following steps are performed as a loop executed over a number of mutants:

```

Loop over all Mutants
{
Copy the mutant to the source directory
Compile Mutant under test
Compile Test Driver
Execute Test Driver on Mutant
Save the report file created with a different name
Remove the mutant from the source directory
}

```

The overall process is:

- Compile the source mutant.
- Modify the class path so that it points to the directory containing the mutant source file.
- Compile the test driver.
- Run the test driver on the mutant.

- Save the reports file created into a separate directory.
- Repeat the above for all the mutant versions.
- Repeat the above for all the five approaches for a given class.
- Repeat the above for all the eleven classes.
- Collect the data into corresponding directories for each approach and class.

5.5 Collection and Analysis of the Experimental Data

The output files generated for all the mutants run are collected into a separate directory. Then we determine if any of these mutants are “functionally equivalent”. Such mutants are detected by analyzing the respective output files. We then remove the functionally equivalent mutants. The summary of the results once all the functionally equivalent mutants are removed is shown in table 5-4.

5.5.1 Results Gathered

For each of the five approaches, Table 5-4 indicates the total number of mutants produced and the number of mutants “killed.” Figures in brackets show the percentage of mutants detected.

Class	Total #	DU-Pairs	Random	JTest	All-Nodes	OATS
BitSet	328	207 (63.1%)	195 (59.4%)	128 (39.0%)	124 (37.8%)	181 (55.2%)
Hashtable	454	166 (36.6%)	163 (35.9%)	91 (20.0%)	98 (21.6%)	117 (25.8%)
LinkedList	263	129 (49.0%)	131 (49.8%)	74 (28.2%)	72 (27.4%)	110 (41.8%)
Stack	36	34 (94.4%)	35 (97.2%)	13 (36.1%)	13 (36.1%)	35 (97.2%)
TreeMap	573	264 (46.1%)	269 (46.9%)	64 (11.2%)	129 (22.5%)	196 (34.2%)
TreeSet	23	2 (8.7%)	7 (30.4%)	2 (8.7%)	2 (8.7%)	2 (8.7%)
Vector	270	151 (55.9%)	153 (56.7%)	40 (14.8%)	41 (15.2%)	130 (48.1%)
Observable	17	8 (47.0%)	8 (47.0%)	1 (5.9%)	1 (5.9%)	1 (5.9%)
StringTokenizer	135	117 (86.7%)	114 (84.4%)	68 (50.4%)	95 (70.4%)	99 (73.3%)
Date	559	14 (2.5%)	14 (2.5%)	12 (2.1%)	13 (2.3%)	13 (2.3%)
GregorianCalendar	1708	499 (29.2%)	483 (28.3%)	233 (13.6%)	257 (15.0%)	275 (16.1%)

Table. 5-4. Percentage of mutants caught for each test suite run

A quick look at the table indicates that the random and all-DU-pairs methods performed equally, followed by OATS, all-nodes, and JTest. The above table is graphically summarized as follows

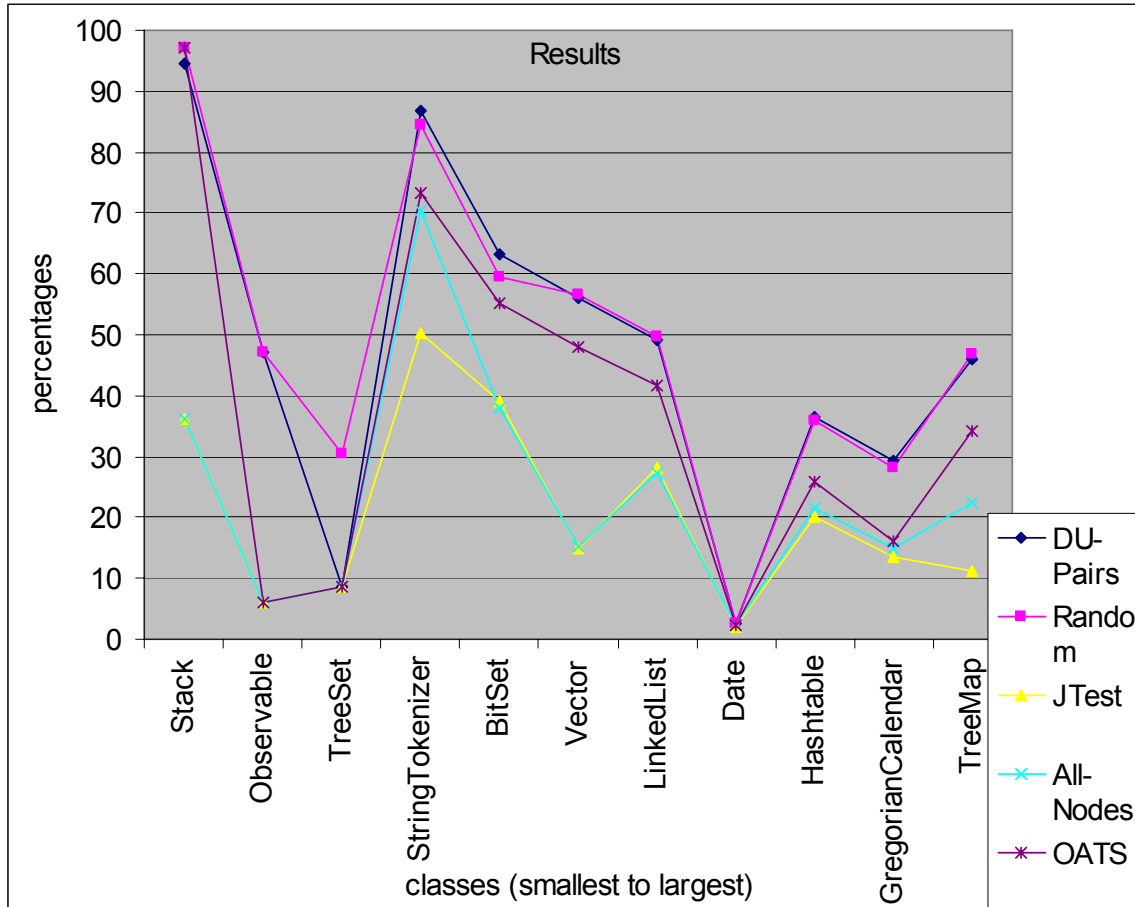


Figure 5-1. Experimental results- eleven classes, five approaches

5.5.2 Statistical Analysis of the Data Gathered

Results are gathered and taken to the statistics lab, where they are analyzed in detail. In the following two statistical cases, the dependent variable is the “percentage of mutants caught,” while the independent variables are the approaches and classes. Each of the following tests was conducted by creating SAS scripts and running them over the gathered data.

5.5.2.1 RCBD –Randomized Complete Block Design

The RCBD test determines whether the approaches differ with each other in a statistically significant manner. The test indicated that an overall significant difference ($p < 0.0001$) exists between all five approaches. As the result says only that the approaches differ overall, this difference can exist anywhere between any two approaches.

Source	DF	Type III SS	Mean Square	F Value	Pr > F
Method	4	7355.64756	1838.91189	16.22	<. 0001

The second part of the RCBD test involves ranking the five approaches based on their effectiveness -the higher the percentage, the more effective the approach.

Additionally, the approaches were organized using Duncan's method, which groups together means that are not significantly different. Approaches that fall in the same Duncan grouping and are assigned the same letter value are not found to differ statistically. The test was performed at an alpha value of 0.05, which corresponds to 95% confidence level. The number of degrees of freedom is 40, and the error mean square is found to be 113.4031 (see Table 5-6).

Number of Means	2	3	4	5
Critical Range	9.18	9.65	9.96	10.18

Table 5-5. Statistical parameters for the four approaches

Means with the same letter are not significantly different.

Duncan Grouping	Mean	N	Method
A	48.9%	11	2(Random)
A	47.2%	11	1(all-DU-pairs)
B	37.2%	11	5(OATS)
C	23.9%	11	4(all-nodes)
C	20.9%	11	3(JTest)

Table 5-6. Ranking & Grouping of the five approaches

As noted previously, the main goal of this experiment was to compare the method proposed by this research, that of all-DU-pairs, with the other four methods. The third and final part of the RCBD test involves performing this comparison, with the dependent

variable being the percentage of mutants caught. The results contrast the all-DU-pairs method with each of the rest four methods to determine any statistically significant difference.

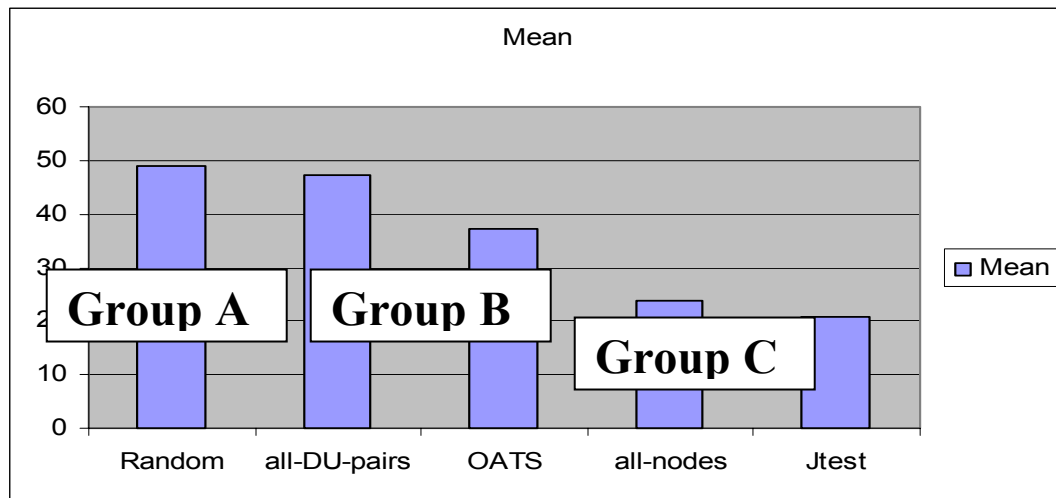


Figure 5-2. Ranking of the five approaches

Going by the numbers, Table 5-7 shows all-DU-pairs as the approach 1, while random, JTest, all-nodes, and OATS form 2 through 5 approaches respectively.

Contrast	Mean Square	F Value	Pr > F
1 VS 2	17.13	0.15	0.6996
1 VS 3	3803.52	33.54	<. 0001
1 VS 4	2988.47	26.35	<. 0001
1 VS 5	556.17	4.9	0.0326

Table 5-7. First approach versus rest four

The above comparisons indicate no statistically significant difference between the random and the all-DU pairs approaches ($p > 0.05$). In addition, the all-DU-pairs approach performed significantly better than did the JTest and all-nodes approaches (both $p \ll 0.05$). While the OATS approach identified more mutants than all-nodes and Jtest, the all-DU-pairs approach performed significantly better than it ($p < 0.05$).

5.5.2.2 Correlation Tests their Significance and Issues Involved

In our case, both all-DU pairs and random tests performed equally, which is contrary to the experimental findings of Edwards [9]. Additionally Statistical analysis suggests this overall order: both random and all-DU-pairs methods performed equally to or better than OATS, which performed better than both all-nodes and Jtest, both of which performed equally. Hence, the order is (all-DU, Random) > OATS > (all-nodes, JTest). On observation we can see that the performance of all five approaches ranked in the order of number of tests. Approaches that generated a higher number of tests performed better than those with lower. *These initial statistical tests suggested that the number of tests might be a more significant factor than the actual approach.* To check the validity of this suggestion, we performed additional **correlation tests** to measure the effect of tests on the percentages. We conducted the correlation tests to determine the effect of not only number of tests but also other influential factors like the LOC, and methods over percentages. For the current experimental setup, we performed four correlation tests.

The first test runs over all the test data to determine correlation between the number of tests, number of LOC, and number of methods in a class and the percentage of mutants caught. The **significance** of this test was to find out the effect of all these various factors over percentages.

As described later once we found that none of these factors except number of tests had a significant correlation with percentages of mutants killed. *This eliminated the effect of other factors on percentages. This indicated that number of tests could have a potential effect on percentages.* We then conducted additional correlation tests to see if this correlation between number of tests and percentages still holds when the testing methodology was kept constant. This resulted in correlation tests on more specific data sets (on all-DU-pairs and random separately). *These tests showed that the number of tests had no effect on the percentages of mutants detected and hence eliminate the effect of all the factors.* The classes that we dealt in our experiments were larger in size compared to the ones explored by Edwards [9]. *This led us to the thinking that the increased Cyclomatic complexity of the classes might be the reason for the low percentages*

observed. The Cyclomatic complexity test conducted proved that this was not the case. *This eliminated the effect of all the factors on percentages.*

The section below explains all the tests in detail and discusses the above-mentioned issues.

5.5.2.3 Correlation Tests

Correlation tests measured significant correlation between the various factors involved in the experimental setup. For the current experimental setup, we performed four correlation tests.

Test #1: The first test runs over all the test data (5*11) to determine correlation between the number of tests, number of LOC, and number of methods in a class and the percentage of mutants caught.

	Tests	Methods	LOC	Percentages
Tests	1.00 (X)	0.20 (0.20)	0.17 (0.27)	0.28 (0.06)
Methods		1.00 (X)	0.42 (0.01)	0.05 (0.72)
LOC			1.00 (X)	-0.16 (0.31)
Percentages				1.00 (X)

Table 5-8. Correlation table for data from all the five approaches

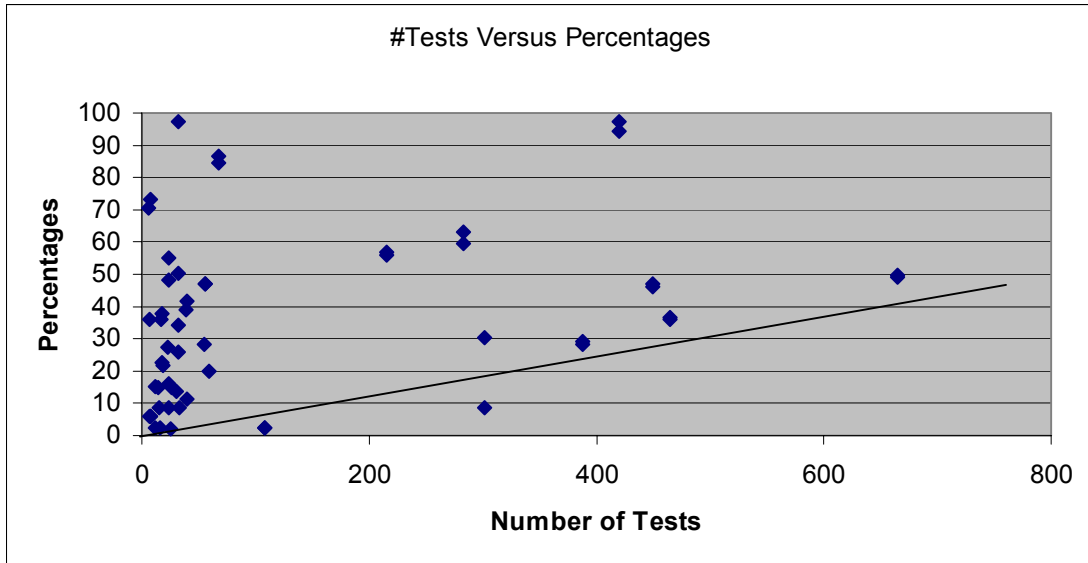


Figure 5-3. Number of Tests versus percentages- significant correlation ($r=0.28$)

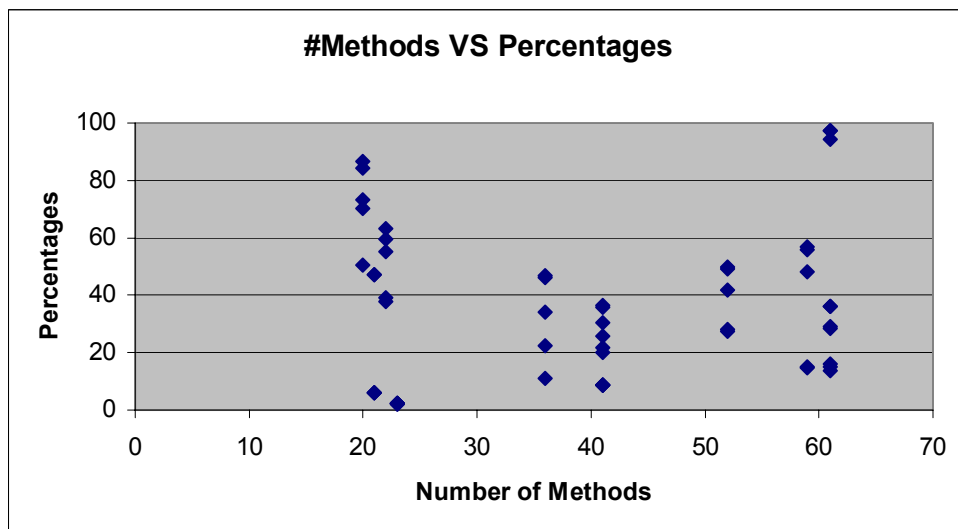


Figure 5-4. Number of Methods in a class versus percentages-No correlation ($r=0.05$)

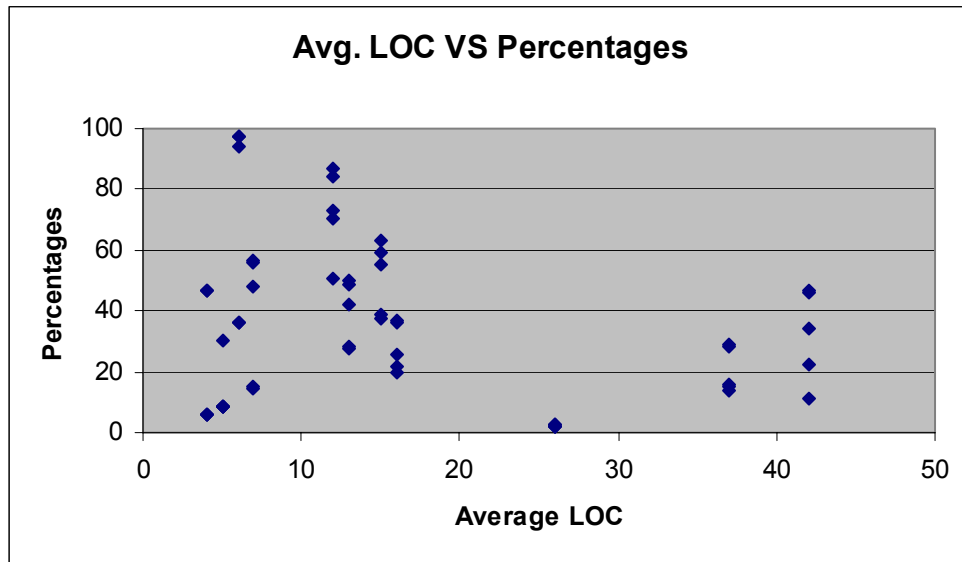


Figure 5-5. Average LOC versus percentages- No correlation ($r=-0.16$)

Results indicate no significant correlation found between LOC/percentage in Figure 5-4, methods/percentage in Figure 5-3. Statistically significant correlation occurs between the number of methods and LOC, which is an obvious relation. The comparison between tests and percentage in Figure 5-2 reveals a comparatively better value of 0.06 (slightly greater than 0.05), which can be considered a significant correlation—but not statistically significant. It is to be remembered here that we performed this correlation test on the data from all the five approaches. Hence there is another variable in the form of “approach selected” that can potentially affect the correlation between number of tests and percentages. Hence we performed additional correlation tests on the data from all-DU-pairs and random approaches separately to see if this correlation still holds.

Test #2: For a more specific test data set, the second test measures correlation between the factors and percentages. It performs the same correlation test as Test #1, but only on all-DU pairs and random data.

All DU pairs:

	Tests	Methods	LOC	Percentages
Tests	1.00 (X)	0.62 (0.04)	0.51 (0.11)	0.02 (0.95)
Methods		1.00 (X)	0.42 (0.20)	0.09 (0.78)
LOC			1.00 (X)	-0.19 (0.57)
Percentages				1.00 (X)

Table 5-9. Correlation table over all-DU-pairs

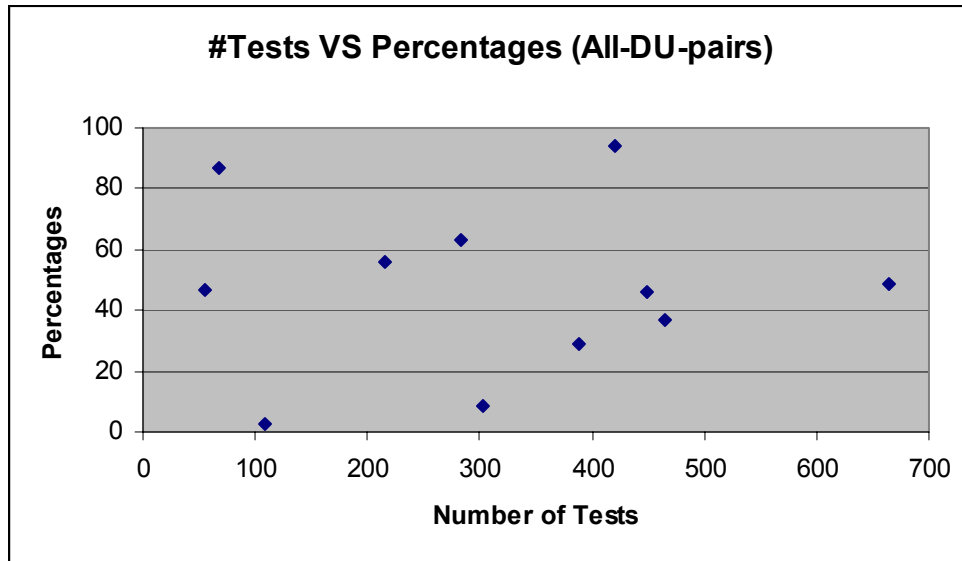


Figure 5-6. Correlation between Number of Tests and percentages ($r=0.02$)

Random:

	Tests	Methods	LOC	Percentages
Tests	1.00 (X)	0.62 (0.04)	0.51 (0.11)	0.04 (0.90)
Methods		1.00 (X)	0.42 (0.20)	0.15 (0.67)
LOC			1.00 (X)	-0.28 (0.41)
Percentages				1.00 (X)

Table 5-10. Correlation table over Random data

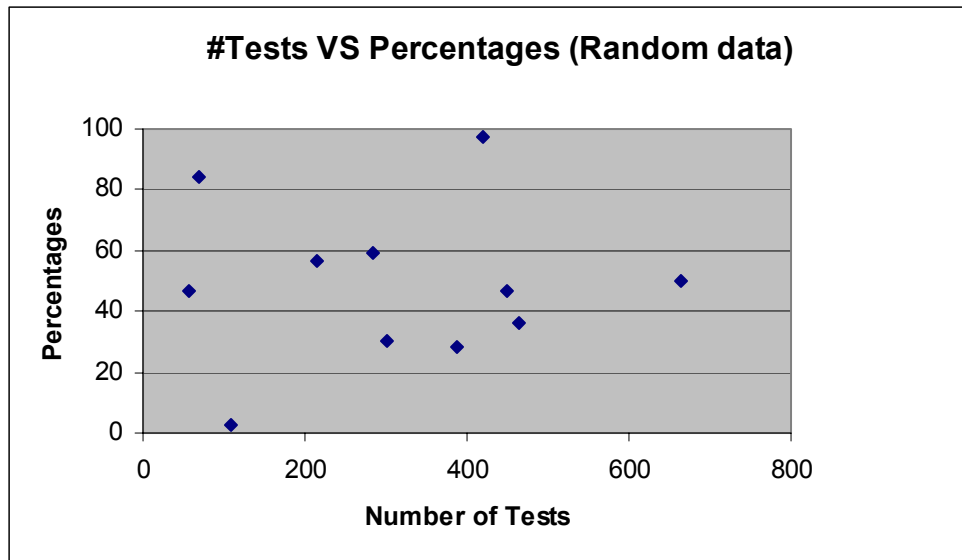


Figure 5-7. Correlation between Number of Tests and percentages ($r=0.04$)

We can see from the above tables that there is no significant correlation between tests and percentages when the approach is fixed (Figure 5-5, 5-6). This indicates that the number of tests is not a significant factor.

This indicates that the number of tests is not really the most important factor. These tests showed that the number of tests had no effect on the percentages of mutants detected and hence eliminate the effect of all the factors. The classes that we dealt in our experiments were larger in size compared to the ones explored by Edwards [9]. This led us to the thinking that the increased Cyclomatic complexity of the classes might be the reason for the low percentages observed. This led us to perform the Test # 3

Test # 3: The third test determines correlation between the cyclomatic complexity of the components and the percentages. The percentages of mutants detected in the current case are very low, which is contrary to the findings of Edwards [9]. This difference could arise as a result of an increase in the cyclomatic complexity of the components, which has made the bugs difficult to find. Hence, we performed this correlation test to see if complexity influences the percentage of mutants detected. The cyclomatic complexity of a class is measured by calculating its value for each of the methods and then summing up

all the values. Table 5-11 indicates no significant correlation between cyclomatic complexity and percentage. This indicates no significant correlation between cyclomatic complexity of classes and the percentages of mutants detected.

	CC	Percentage
CC	1.00 (X)	-0.14 (0.67)
Percentage		1.00 (X)

Table 5-11. Correlation between Cyclomatic Complexity and percentage

5.6 Summary of Results

Our major hypothesis was that the all-DU-pairs approach would perform better than the other approaches examined. Experimental evaluation determined that this is indeed the case, except in the case of the random approach. Random tests performed on par with the all-DU-pairs tests.

As we discussed earlier, we selected the random approach so that it acts as a **baseline** for other approaches to compare against. The results indicate the following ranking of the five approaches: (Random, All-DU-pairs), OATS, (JTest, all-nodes).

The all-DU-pairs approach generates in such a structured manner that all the definition-use combinations are exerted. Since the random approach generates the same number of tests as all-DU-pairs but in a random fashion, there is a possibility that not all definition-use combinations are exercised. Hence, the random approach might generate fewer unique tests than does the all-DU-pairs approach. The important research question here is whether the difference in the number of unique tests between random and all-DU-pairs are significant enough to bring out a difference in performance between the two approaches.

To determine this, we proposed that random approach would perform either on par with (if the difference is not significant) or less than (if significant) all-DU-pairs approach. The actual results proved that the difference is so insignificant that we cannot tout one method as being superior to the other. Even our statistical analysis proved that

both approaches fall in the same Duncan grouping, which indicates between them no statistically significant difference.

Statistical analysis suggests this overall order: both random and all-DU-pairs methods performed equally to or better than OATS, which performed better than both all-nodes and Jtest, both of which performed equally. Hence, the order is (all-DU, Random) > OATS > (all-nodes, JTest). On observation we can see that the performance of all five approaches ranked in the order of number of tests. Approaches that generated a higher number of tests performed better than those with lower. *These initial statistical tests suggested that the number of tests might be a more significant factor than the actual approach.* To check the validity of this suggestion, we performed **correlation tests** to measure the effect of three important factors on the percentages: tests, methods, and LOC.

The second test, which was performed on a large scale over the five approaches, indicated no statistically significant correlation between any of the three factors and percentages. The number of tests, though, showed significant correlation but as we discussed before these correlation tests were performed on the data from the five approaches. But this correlation did not stand when we performed the same tests by keeping the approach constant (on the data from all-DU-pairs and all-nodes). **This indicates that in the first correlation test, the approach was the important factor rather than the number of tests.** Coincidentally the five approaches were ranked in performance (see Figure 5-1.) basing on the number of tests they generate. That is the reason why number of tests showed a significant correlation with percentages when the correlation tests were performed over all the five approaches. But the latter tests indicated that the approach was the more important factor and number of tests had no effect on the percentages.

Compared to the classes we selected, those chosen by Edwards [9] are small. At the same time we found that the percentages of mutants detected in our experiments were less compared to those found in the experiments by Edwards. As a result, it is possible that the increased cyclomatic complexity of the classes has increased the difficulty of

finding bugs. This possibility was eliminated when the third test showed no correlation between the cyclomatic complexity of the class and the percentages. **This indicates, therefore, that the approach is factor with the largest effect.**

This result makes the equal performance of the all-DU-pairs and random approaches bit mysterious. Since we have no other factors affecting the percentages other than the approach, the equal performance of all-DU-pairs and random approaches could be because of inadequate testing. Even if we observe the percentages of mutants detected, we can see that they number fewer than those observed by Edwards [9]. This indicates that the amount of testing performed was not effective enough to create a difference between all-DU-pairs and random approaches.

We propose this difference to be because of the exceptions in Java language. For every Java class, the documentation contains a set of exceptions that can be generated by the methods of the class. These exceptions are called pre-defined exceptions since they are documented in the class specification. In our approach, we treated as definitions both the return values and the values a method modifies. We did not consider as definitions the set of exceptions documented as being generated by the methods of the class. These exceptions can be treated as definitions with uses in the form of `print ()` statements, which would generate additional tests and would reveal additional bugs. Hence, we hypothesize that this equal performance of random approach and all-DU-pairs approach could be due to these exception conditions, which if treated as definitions could have resulted in differing performance levels.

The surprising part of the result here is that for certain Java classes, the random approach performed slightly better than did the all-DU-pairs approach. This result is unexpected because the random approach generates fewer “original” tests does the all-DU-pairs approach. When we took out a sample of classes wherein the random approach performed better, we observed that the result is justified because of the manual addition of assertions. On average, each of these test cases contains two method calls. The assertion is added as the third method call, and since it is one-third of the entire test case, it plays a crucial role in detecting bugs. Selecting the appropriate method call, as

assertion is as important as selecting the test case itself. Since we have performed this function manually, the assertions will be very different for both all-DU-pairs and a random approach, which is why in some cases the latter outperformed the former.

The equal performance of random approach and all-DU-pairs led to the suspicion of the number of tests as an important factor (both all-DU-pairs and random contained equal number of tests), which was proved wrong by the later correlation tests. Since the approach is the only important factor, we feel careful generation of assertions, treating exceptions, as tests would have improved the performance of all-DU-pairs tests and made it look better against the random tests. On the overall we feel that the method proposed by this research has good potential for detecting bugs and needs to be explored further on a wide range of data sets.

Chapter 6: Conclusions and Future Work

The current research possesses two major components: method development and method evaluation. By explaining the method in a user-friendly fashion, Chapter 4, provides a recipe by which it can readily be put to use by the industry and then we evaluate its effectiveness against standard methods. The current chapter outlines the major results gathered from the experimental evaluation, identifies the major contributions of the research, and suggests avenues for future research.

6.1 Contributions of the Research

As previously stated, the work covered by this thesis contributes to research by proposing and evaluating a new testing method. Moreover, it aims to explain the method so clearly that it can be used quickly and efficiently by anyone who chooses to adopt it. Eschewing a theoretical purpose alone, the method is designed for real-world, practical application. In fact, such user-friendliness was the prime motivator behind the method's development. Unlike many manual methods, the one proposed here does not any formal specifications for test generation, nor is it difficult to use.

The experimental evaluation has taught us more about the key components of the method and has permitted us to realize how it fares against other testing methods. With its two versions—all-nodes and all-DU-pairs—the basic method measures effectively against random, Jtest, and OATS methods.

Although in the present case the structured all-DU-pairs approach appears to have performed with effectiveness equal to that of the random approach, in general the all-DU-pairs approach outperforms the random approach for the following reasons:

- **Preconditions:** For the Java classes with which we worked, there are no preconditions for calling the class methods. Any method can be called at any point of time with little setup required. In practice, however, most software possesses methods with preconditions, which makes the random approach—with its generation of numerous invalid test cases—ineffective.

- **Exceptions:** Many of the test cases generated under the random approach lead to exceptions; therefore, they become invalid if they are not dealt with quickly and efficiently. The proposed experimental setup avoids such problems: the Java classes used document all potential exceptions and do not generate new ones. Since in practice methods can generate new exceptions, many of the tests generated by the random approach can be invalid.

The fact that the all-DU-pairs and random approaches performed equally suggests that when automating the method, more randomness can be allowed into the process. Rather than generating the tests following a structured approach, a more random approach can be followed to generate numerous test cases, with invalid ones weeded out later in the process. Thus, while the all-DU-pairs approach took more work than did its complementary version, the all-nodes approach; the promising nature of the results far exceeded the additional time and effort. Although we cannot generalize and apply the less-than-stellar performance of JTest to all white-box testing methods, the JTest versus all-nodes comparison has proved to some extent that the black-box testing method is in all cases equal in quality to that of the white-box..

Since the OATS performance falls somewhere in the middle between the all-DU pairs and all-nodes approaches, the results rank the five approaches in the order of the number of tests and average number of method calls per test. The approaches that generated a higher number of tests performed better than did those that generated fewer ones. Hence, in the end, the number of tests and method calls—rather than the approach—might be more important. Even the correlation tests performed on the whole data (which had a lot of variance) suggest that the correlation between the number of tests and percentages of mutants caught was nearly significant ($p=0.06$).

6.2 Future Work

When the number of tests and average methods per test were the same, the experimental results indicate no significant difference between the random approach and a structured approach like all-DU-pairs. Additional work should be performed to see if this result stands for larger data sets, especially in cases where methods have preconditions and

cannot be called at any time. The experimental results thus give some insight into automating the approach.

The second major finding of the experiment is that the number of tests and average number of methods per test can be significant factors. Future work toward automation can use both of these results, allowing randomness and conducting more tests.

Since one major goal of this research was to develop a method that is easy to use, an important result can be evaluating this fact: how successful were we in accomplishing this goal? Our evaluation considers the approach only with respect to its effectiveness in finding bugs. We did not evaluate the method's "usability" or "ease of use," which we consider to be as significant as its effectiveness. Hence, evaluating the usability of this method constitutes important future work.

References

- [1] Open source tools web page, last accessed 12th March 2003 ANT build tool
<<http://www.apache.org>>
- [2] P. Ammann and A. J. Offutt. Using formal methods to derive test frames in category-partition testing. In Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS 94), pages 69{80, Gaithersburg MD, June 1994. IEEE Computer Society Press.
- [3] Beizer B. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley: New York, 1995.
- [4] Beizer B. *Software Testing Techniques* Van Nostrand Rheinhold, 1990.
- [5] L. Bouge, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test sets generation from algebraic specifications using logic programming. *The Journal of Systems and Software*, 6(4):343{360, November 1986
- [6] J. Chang and D. Richardson. Structural specification-based testing with ADL. In Proceedings of the 1996 International Symposium on Software Testing, and Analysis, pages 62{70, San Diego, CA, January 1996. ACM Press.
- [7] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In Proceedings of FME '93: Industrial-Strength Formal Methods, pages 268{284, Odense, Denmark, 1993. Springer-Verlag Lecture Notes in Computer Science Volume 670.
- [8] Doong R-K, Frankl PG. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Software Engineering Methodology*, 1994, 3(2): 101-130.
- [9] Stephen H. Edwards. Black-box testing using flow graphs: An experimental assessment of effectiveness and automation potential. *Software Testing, Verification and Reliability*, 10(4), December 2000, pp. 249-262.
- [10] Edwards SH. A framework for practical, automated black-box testing of component-based software. *Proc. 1st Int'l Workshop on Automated Program Analysis, Testing and Verification*, June 2000, pp. 106-114.

- [11] Edwards S, Shakir G, Sitaraman M, Weide BW, Hollingsworth J. A framework for detecting interface violations in component-based software. *Proc. 5th Int'l Conf. Software Reuse*, IEEE CS Press: Los Alamitos, CA, 1998, pp. 46-55.
- [12] Frankl PG, Weiss SN. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Software Engineering*, Aug. 1993, **19**(8): 774-787.
- [13] Frankl PG, Weiss SN, Hu C. All-uses versus mutation testing: An experimental comparison of effectiveness. *J. Systems and Software*, Sept. 1997, **38**(3): 235-253.
- [14] Gannon JD, McMullin PR, Hamlet R. Data-abstraction implementation, specification, and testing. *ACM Trans. Programming Languages and Systems*, July 1981, **3**(3): 211-223.
- [15] I. J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, SE- 12(1):124{133, January 1986.
- [16] Robert M. Hierons. Testing from a Z specification. *The Journal of Software Testing, Verification, and Reliability*, 7:19{33, 1997.
- [17] Hoffman D, Strooper P. The test-graphs methodology: Automated testing of classes. *J. Object-Oriented Programming*, Nov./Dec. 1995, **8**(7): 35-41.
- [18] Junit Web page, Last accessed 10th March 2003, Java's Unit Testing tool
<<http://www.junit.org>>
- [19] SUN java web page, Last accessed 20th April 2003, Java API documentation
<<http://java.sun.com/api/index.htm>>
- [20] Ivan Moore, Last accessed 15th March 2003 Jester, the Junit test tester
<<http://jester.sourceforge.net>>
- [21] Parasoft's JTest, Last accessed 15th April 2003, Java Unit testing tool
<<http://www.parasoft.com/jsp/products/home.jsp?product=Jtest>>
- [22] King KN, Offutt J. A Fortran language system for mutation-based software testing. *Software Practice and Experience*, July 1991, **21**(7): 686-718.
- [23] Mathur AP, Wong WE. Comparing the fault detection effectiveness of mutation and data flow testing: An empirical study. Tech. Report SERC-TR-146-P, Software Engineering Research Center, Dec. 1993.

- [24] Meyer B. *Object-Oriented Software Construction, 2nd Edition*. Prentice Hall PTR: Upper Saddle River, New Jersey, 1997.
- [25] Offutt J, Abdurazik A. Generating tests from UML specifications. *2nd Int'l Conf. Unified Modeling Language (UML99)*, Fort Collins, CO, Oct. 1999.
- [26] Offutt J, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Trans. Software Engineering Methodology*, April 1996, **5**(2): 99-118.
- [27] Offutt J, Xiong Y, Liu S. Criteria for generating specification-based tests. *5th IEEE Int'l Conf. Engineering of Complex Computer Systems (ICECCS '99)*, Las Vegas, NV, Oct. 1999.
- [28] Orthogonal Arrays Directory: <http://www.research.att.com/~njas/oadir/>
- [29] Roger S. Pressman, *Software Engineering : A Practitioner's approach*, Fourth Edition : McGraw Hill Companies
- [30] W. T. Tsai, D. Volovik, and T. F. Keefe. Automated test case generation for programs specified by relational algebra queries. *IEEE Transactions on Software Engineering*, 16(3), March 1990.
- [31] I. Spence and C. Meudec. Generation of software tests from specifications.
- [32] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777{793, November 1996.
- [33] Wing JM. A specifier's introduction to formal methods. *IEEE Computer*, Sept. 1990
- [34] Zhu H, Hall PAV, May JHR. Software unit test coverage and adequacy. *ACM Computing Surveys*, Dec. 1997, **29**(4): 366-427.
- [35] Zweben S, Heym W, Kimmich J. Systematic testing of data abstractions based on software specifications. *J. Software Testing, Verification and Reliability*, 1992, **1**(4): 39-55.