ENHANCING EVALUATION TECHNIQUES USING MUTATION OPERATOR PRODUCTION RULE SYSTEM AND ACCIDENTAL FAULT METHODOLOGY

by

PRANSHU GUPTA

B.S., Wilkes University, 2002
M.S., James Madison University, 2005

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2014

Abstract

Software testing is an essential component of software development life cycle, and certain software testing methodologies require enormous amounts of time and expense in order to detect and correct errors in a software system. The two primary goals of any testing methodology are error detection and increased reliability. Each methodology utilizes a unique technique to achieve these goals and detect faults in the software. In this paper, an evaluation approach is presented that can enhance evaluation techniques for software testing methodologies. Firstly, a new framework, Mutation Operator Production Rule System (MOPRS), is introduced that allows specifications of mutation operators that can be effective, precise, and focused on object-oriented faults. A new concept of effective mutation operator has been added to this system. An *effective mutation operator* is a precise set of rules that when applied to a program creates a set of mutants, which when killed by a test suite, will mean that further seeded or accidental faults characterized by the same fault type are highly likely to be killed by the same test suite. These effective mutation operators focus on fault types specific to object-oriented programming concepts. As a result, object-oriented faults are detected instead of finding traditional faults common to non-object-oriented and object-oriented programming. These mutation operators cover the gaps in the existing set of mutation operators. An evaluation method is described that can enhance the evaluation techniques, Accidental Fault Methodology (AFM), for software testing methodologies. When effective mutation operators are used along with this evaluation technique, it will demonstrate if the software testing methodology successfully detected induced faults and also any accidental faults specific to the object-oriented fault type.

ENHANCING EVALUATION TECHNIQUES USING MUTATION OPERATOR PRODUCTION RULE SYSTEM AND ACCIDENTAL FAULT METHODOLOGY

by

PRANSHU GUPTA

B.S., Wilkes University, 2002
M.S., James Madison University, 2005

A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2014

Approved by:

Major Professor
Dr. David A. Gustafson

Abstract

Software testing is an essential component of software development life cycle, and certain software testing methodologies require enormous amounts of time and expense in order to detect and correct errors in a software system. The two primary goals of any testing methodology are error detection and increased reliability. Each methodology utilizes a unique technique to achieve these goals and detect faults in the software. In this paper, an evaluation approach is presented that can enhance evaluation techniques for software testing methodologies. Firstly, a new framework, Mutation Operator Production Rule System (MOPRS), is introduced that allows specifications of mutation operators that can be effective, precise, and focused on object-oriented faults. A new concept of effective mutation operator has been added to this system. An *effective mutation operator* is a precise set of rules that when applied to a program creates a set of mutants, which when killed by a test suite, will mean that further seeded or accidental faults characterized by the same fault type are highly likely to be killed by the same test suite. These effective mutation operators focus on fault types specific to object-oriented programming concepts. As a result, object-oriented faults are detected instead of finding traditional faults common to non-object-oriented and object-oriented programming. These mutation operators cover the gaps in the existing set of mutation operators. An evaluation method is described that can enhance the evaluation techniques, Accidental Fault Methodology (AFM), for software testing methodologies. When effective mutation operators are used along with this evaluation technique, it will demonstrate if the software testing methodology successfully detected induced faults and also any accidental faults specific to the object-oriented fault type.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to acknowledge the inspirational instruction and guidance of Dr. David A. Gustafson. He has given me a deep appreciation and love for the beauty and detail of the subject.

I would also like to acknowledge the support and assistance given to me by my mentor, professor and friend Dr. Ramon A. Mata-Toledo.

Finally, I would like to express gratitude to my husband, Divye Kumar, for his support and encouragement. I could never have completed this effort without his assistance, tolerance and enthusiasm.

Dedication

I would like to dedicate this doctoral dissertation to my parents, Mrs. Asha Gupta and Mr. Devendra Veer Gupta. There is no doubt in my mind that without these two strong pillars it would not have been possible for me to achieve this goal. Their continued support and counsel have made this possible.

Chapter 1 - Introduction

Software Faults

An unintended mistake that causes failure of the system or a system component is known as a software fault [HA09]. For example, a simple error made by the programmer such as using "=" instead of "=="can result in a program fault. Errors can be caused by various factors such as a typo or a misunderstanding of the requirements. A fault created by an error can manifest into a failure when a system component or a system behaves in an unwanted manner. A fault that causes significant failure is analyzed and counted [HA96 & WK90].

Software faults in a system can be minimized if they can be induced, tested and removed. In order to create a fault-free software system, it is important to be able to induce software faults that represent commonly occurring errors in a program. If these faults can be induced in a program for the purpose of testing, they can be tested and subsequently removed from a program, thus creating a more fault-free system. Faults can be induced in a program by changing a single line of code or multiple lines of code.

Fault Types

A software fault type (ft) is a textual description of a specific kind of fault (f) that can occur in some program. An instance of a fault type can usually be induced in a program in diverse ways depending on where and what change is made in a program. In other words, a fault (f) is an instance of one or more fault types. For example, a "missing statement" is a software fault type that can exist in a constructor or a method of a program, creating two distinct faults in a program. Originally, the faults were categorized into two fault types: Bohrbugs and Mandelbugs [GR05]. Bohrbugs are easily isolated and Mandelbugs have complex activation and propagation. Fault types were introduced to relate software faults to other classes of faults resulting in a better understanding of faults. In this thesis, a categorization that is centered on software faults is used. The categorization is based on the object-oriented concepts of the object-oriented programming methodology. An example of such fault type is "An assignment statement that violates a state invariant" related to the object-oriented concept known as class invariants. A fault can fall under the umbrella of two different fault types based on changes made in a program. A fault type can generate a finite number of fault instances in a program.

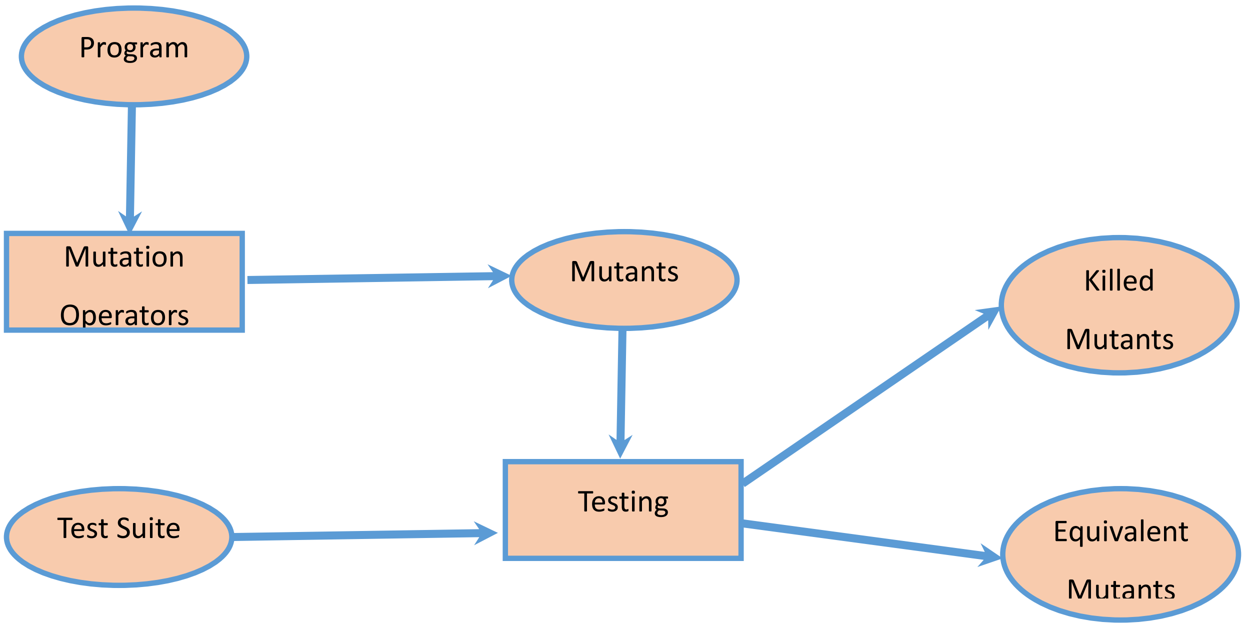
FaultTypes: *FT* is the set of all possible fault types in some program, $FT = \{ft_n \mid ft_n$ is a fault type$\}$.

FaultInstances: *FI* is the set of all possible fault instances in some program. FI_{ft} is a set of all possible fault instances of fault type *ft*. $FI_{ft} = \{f_{ft,\,n} \mid f_{ft,\,n}$ is a fault instance described by the fault type (ft)$\}$.

The Mutation Process

The mutation operators are applied to a program to create a set of mutants that are tested using a test suite. The testing generates two types of mutants – killed and equivalent. Killed mutants show that the test suite was able to find the fault in a program and the equivalent mutants show that the behavior of the program did not change and the fault was not found. The equivalent mutants are not helpful in finding faults in a program as they do not detect the erroneous behavior of a program.

Figure 1.1 The Mutation Process



Mutation Operators

Each software fault type can usually be represented by one or more mutation operators. Furthermore, a mutation operator can induce multiple instances of a fault type (ft) in a program. The definition and specification of a mutation operator are based on analysis of software fault types that can exist in non-object-oriented (non-OO) and object-oriented (OO) programs [AG89, CH01, DE06, KI00 & MA02]. A mutation operator definition describes how to insert a fault of a specific fault type in a program. An example of a traditional mutation operator definition is "changing an arithmetical operator in an expression." The specification of this mutation operator describes the changes to be made in a program, i.e. changing "+" sign operator to a "-" sign operator. In this thesis, the same strategy is used to define mutation operators and the focus is on OO mutation operators.

MutationOperators$_{NOO}$: MO_{NOO} is the set of non-object-oriented mutation operators collected from literature, $MO_{NOO} = \{\mathrm{mo}_{\mathrm{noo,\,n}} \mid \mathrm{mo}_{\mathrm{noo,\,n}}$ is a non-object-oriented mutation operator found in the literature$\}$.

MutationOperators$_{OO}$: MO_{OO} is the set of object-oriented mutation operators collected from literature, $MO_{OO} = \{\mathrm{mo}_{\mathrm{oo,\,n}} \mid \mathrm{mo}_{\mathrm{oo,\,n}}$ is an object-oriented mutation operator found in the literature$\}$.

A specific mutation operator (mo) creates multiple instances of a fault type (ft) in a program, each known as a mutant (mut). Multiple mutants are generated based on where and what changes are made in a program. A mutation operator may not necessarily generate all mutants i.e. it may not create all instances of a fault type. Each faulty version of a program created is called the "mutant" program.

MutationOperators$_{ft}$: MO_{ft} is the set of all mutation operators defined for a single fault type (ft), $MO_{ft} = \{\mathrm{mo}_{\mathrm{ft,\,n}} \mid \mathrm{mo}_{\mathrm{ft,\,n}}$ is a mutation operator defined for a fault type (ft) that can be applied to a program$\}$.

MutationOperators$_{new}$: MO_{new} is the set of newly defined mutation operators, $MO_{new} = \{\text{mo}_{\text{new, n}} \mid \text{mo}_{\text{new, n}}$ is a new object-oriented mutation operator added to the existing set of operators$\}$.

Mutants$_{mo, p}$: $Mut_{mo, p}$ is the set of all mutants generated when mutation operator (mo) is applied to program (p), $Mut_{mo, p} = \{\text{mut}_{\text{mo, p, n}} \mid \text{mut}_{\text{mo, p, n}}$ is a mutant created by applying a mutation operator (mo) to program (p)$\}$.

A number of sets for mutation operators, test cases, faults, fault types, and mutants have been defined in this paper. Ideally, these sets are finite but not complete. Sets defined in this paper are growing because various OO faults are being added to the list as they are analyzed and added to the literature. Literature review indicates that a number of OO fault types have not been represented by a mutation operator. A list of fault types that require a representation by a mutation operator is shown in Appendix A. The aim of this thesis is to define new mutation operators for some fault types, mentioned in Appendix A, which can occur in an OO program but are not currently defined using a mutation operator. The aim is to address gaps in the existing set of mutation operators for object-oriented programs.

Effective Mutation Operators

An *effective mutation operator* is a precise set of rules that when applied to a program creates a set of mutants (that represents a specific fault type) when killed by a test suite will mean that further seeded or accidental faults characterized by the fault type are highly likely to be killed by the same test suite. Effectiveness of a mutation operator can be calculated by taking into account the probability of killing accidental faults in a program. A higher probability of killing the accidental faults by the same test suite as the mutants will result in higher effectiveness of the mutation operator.

EffectiveMutationOperators$_{ft}$: EMO_{ft} is the set of mutation operators that when applied to a program must create a set of mutants, when killed by a test suite will mean that further seeded or accidental faults characterized by the fault type are highly likely to be killed by the same test suite, $EMO_{ft} = \{\text{emo}_{\text{ft, n}} \mid \text{emo}_{\text{ft, n}}$ is a mutation operator that generates a mutant (mut) that represent a fault ε ***DetectedFaults***$_{\boldsymbol{MOTS, ft, p}}$: $DF_{MOTS, p}$, where MOTS is a test suite that detects the mutant (mut) representing a fault type (ft) in a program (p) or an accidental fault in a program p} Formally, an effective mutation operator is when

$\forall$ mo $\in MO_{ft,p}$

$\forall$ mut $\in Mut_{mo, p}$

If all mut is killed by $MOTS_{Mutmo, p}$

Then this test suite ($MOTS_{Mutmo, p}$) is highly likely to kill any further instances of fault (f_{ft}) that are seeded by mutation operator (mo) or is an accidental fault belonging to the same fault type (ft).

Test Suites

A test suite is a collection of test cases used to find errors in a program. A test suite for a mutation operator is a collection of test cases used to kill mutants generated by the mutation operators belonging to a fault type. If all mutants induced using a mutation operator (mo) representing a specific fault type (ft) are killed, we can say that the test suite is sufficient for detecting that particular fault contained in a specific fault type (ft). A mutation operator test suite ($MOTS_Z$) is a set of test cases that is able to kill all non-equivalent mutants (Z) generated using the mutation operator. A minimal test suite (*MMOTS*) can be created from the *MOTS* that includes a minimum number of test cases that can kill all mutants for that mutation operator, i.e., *MMOTS* is a subset of *MOTS*. The *MOTS* may contain multiple minimal suites, $MMOTS_i$. The *MOTS* can also contain a sufficient test suite (*MOSTS* – mutation operator sufficient test suite) that contains test cases to kill all mutants such that *MMOTS* is a subset of *MOSTS* that is a subset of *MOTS*.

MutationOperatorTestSuite$_{\boldsymbol{Z}}$: $MOTS_Z$ is the set of all test cases able to kill at least one mutant, $MOTS_Z$ = {$mots_{z, n}$ | $mots_{z, n}$ is a test case that killed at least one mutant in the set of mutants (Z)}.

MinimalMutationOperatorTestSuite$_{\boldsymbol{Z}}$: $MMOTS_Z$ is a set of least number of test cases able to kill all mutants, $MMOTS_Z$ = {$mmots_{z, n}$ | $mmots_{z, n}$ is an essential test case to kill a specific mutant in the set of mutants (Z)}. A minimal test suite can be created such that for each $mmots_{z, n} \in$ $MOTS_Z$, there exists a mutant Z_i that is killed only by $mmots_{z, n}$.

MutationOperatorSufficientTestSuite$_Z$: MOSTS$_Z$ is a sufficient test suite that contains test cases to kill all mutants such that, $MOSTS_Z$ = {$\text{mosts}_{z,\,n}$ | $\text{mosts}_{z,\,n}$ is a test case where $\text{mosts}_{z,\,n} \in MOSTS_Z$ such that $MMOTS_Z \subset MOSTS_Z \subset MOTS_Z$}.

Detected Faults

The test suite detects fault instances of the fault type induced in a program. The set of detected faults is a set of fault instances (f_{ft}), seeded using the mutation operator (mo) which represents a fault type (ft) in a program (p), and discovered by a test suite (*MOTS*). It is not necessary that all fault instances are detected by a test suite. The ideology behind every testing methodology is to maximize the set of detected faults.

DetectedFaults$_{MOTS,\,ft,\,p}$: $DF_{MOTS,\,p}$ is the set of all faults of fault type (ft) in program (p) detected by the test suite (*MOTS*), $DF_{MOTS,\,ft,\,p}$ = {$\text{df}_{\text{MOTS, ft, p, n}}$ | $\text{df}_{\text{MOTS, ft, p, n}}$ is a fault of fault type (ft) that is detected by a test suite (*MOTS*) in a program (p)}.

The need for Additional Object-oriented Mutation Operators

Do new mutation operators need to be added for OO fault types or is the existing set a sufficient set of mutation operators? If the existing set of mutation operators is not generating instances of majority fault types, then it is not a sufficient set of mutation operators to generate OO fault instances. If the fault type is specific to one OO feature or OO property and is not represented by a mutation operator in the existing set, then new mutation operator(s) must be added to the existing set that represents that fault type. The addition of new mutation operators would allow generation of possible instances of more fault types in an OO program.

MutationOperators$_{ft}$: $\text{MutationOperators}_{\text{Existing}}$ + $\text{MutationOperators}_{\text{new}}$

MutationOperators$_{Existing}$: $MO_{Existing}$ is the set of existing mutation operators, $MO_{Existing}$ = {$\text{mo}_{\text{Existing, n}}$ | $\text{mo}_{\text{Existing, n}}$ is an existing object-oriented mutation operator listed in Appendix A}.

MutationOperators$_{new}$: MO_{new} is the set of newly defined mutation operators, MO_{new} = {$\text{mo}_{\text{new, n}}$ | $\text{mo}_{\text{new, n}}$ is a new object-oriented mutation operator added to the existing set of operators}.

OO mutation operators define and specify how fault types can be represented in a program. This study will collect and analyze these fault types and define mutation operators that can induce fault instances described by fault types so they can create a faulty version of a program for testing.

In order to create a standard set of mutation operators, a standard set of OO fault types must be available. A list of OO fault types was first mentioned in literature by Firesmith in 1993 [FI93]. A standard set of OO fault types collected from the literature and a list of OO fault types represented by an existing set of mutation operators are mentioned in Appendix A [BI96, CH01 & FI93]. The appendix also shows recent OO fault types added to the list. The list of mutation operators has grown based on the analysis of these fault types, and new fault types are added to the list with more research in this area of study.

Accidental Fault Methodology

In this section a new technique to evaluate the various software testing methodologies is defined. The existing evaluation techniques for software testing methodologies usually evaluate the method based on the number of seeded faults that are killed by a test suite in the software. The restriction of this approach is that it only details if a test suite is able to kill the fault generated by applying a mutation operator. In this thesis, the idea is extended to enhance the evaluation technique by proposing a new methodology, ***Accidental Fault Methodology (AFM)***. This methodology is based on two factors:

1) Probability of killing some or all mutants generated by a mutation operator.
2) Probability of killing the accidental faults of the fault type by the same test suite that killed the mutants generated by a mutation operator. The accidental faults are the fault instances that imitate the mutant generated using a mutation operator but cannot be generated when a mutation operator is applied to a program.

The AFM methodology calculates the effectiveness using the failure rate probability of the mutants and accidental faults. The probability of killing an accidental fault can be calculated using the Bayes' theorem that calculates the conditional probabilities of AF given M and M given AF. The probability for an accidental fault to be killed given that a mutant is killed by the same test suite can be calculated as,

$P(AF|M) = P(M|AF)\ P(AF)/P(M)$

where $P(M|AF)$ is the conditional probability of killing a mutant given an accidental fault is killed,

$P(AF)$ is the probability of killing an accidental fault, and

$P(M)$ is the probability of killing a mutant.

Now this calculation for probability is valid only if every test case that killed the mutants was run against the accidental faults. Let each accidental fault, *AF*, be associated with a failure rate θ_i such that

$P(AF|M) = P(M|AF)\ P(AF)/P(M)$

Finally, the probability of killing an accidental fault based on all the mutants generated by a mutation operator can be calculated as,

$$P_{AF} = 1 - ((1 - \theta_1) * (1 - \theta_2) * \ldots .* (1 - \theta_i)), \textit{ where i is the number of mutants} \quad (1)$$

If $P_{AF} = 1$, then we can definitively say that if a mutant is killed then the accidental fault will also be killed by the same test suite as the mutant. If $P_{AF} > 0.5$, then we can state that if a mutant is killed then the accidental fault is highly likely to be killed by the same test suite.

This new approach to enhance the evaluation technique for software testing methodology will depict the following:

1) If the software testing methodology was able to kill both mutants and the accidental faults belonging to the same fault type.
2) If the software testing methodology is able to kill a larger set of faults.
3) If the software testing methodology is successful in killing specific object-oriented fault type.

Based on the concepts discussed above, the hypothesis is stated in the next section.

Hypothesis

There are effective mutation operators that cover current gaps in the set of object-oriented mutation operators that can improve the accidental fault methodology for evaluation of software testing methodologies.

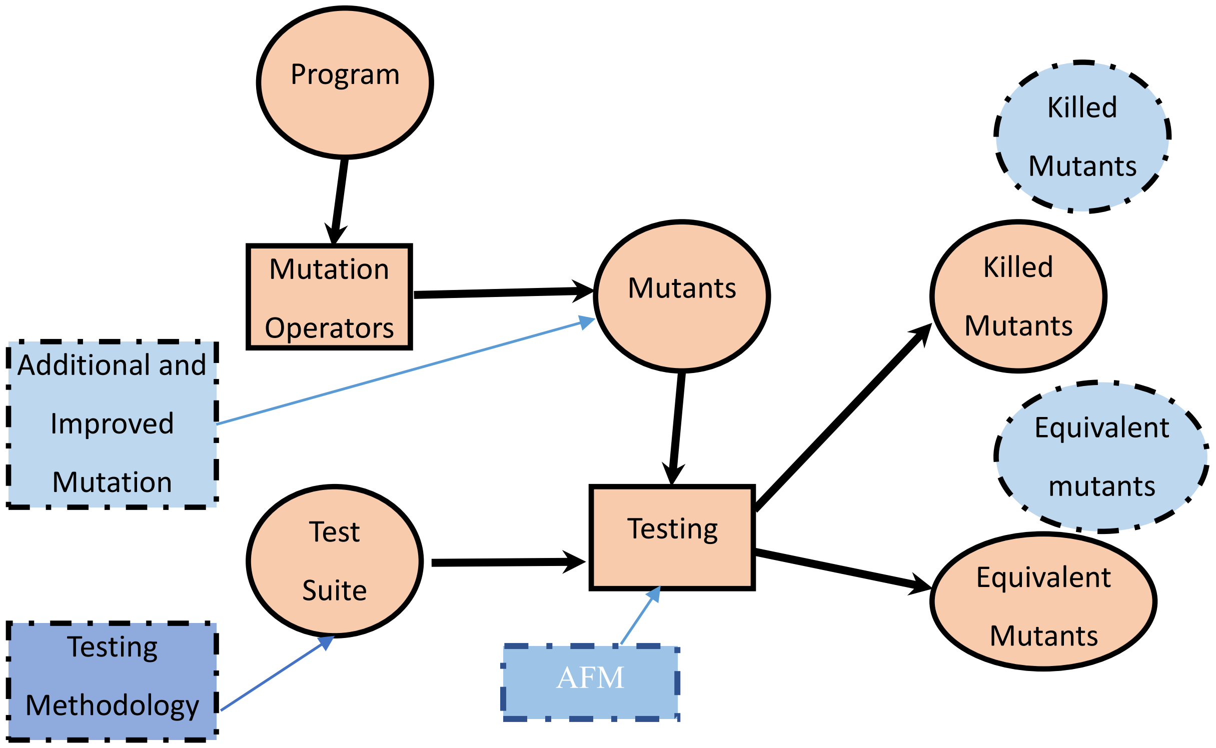
It is essential to improve the evaluation of software testing methodologies by requiring a test suite that detects the induced as well as any accidental faults in a program. In addition to detecting the mutants created using the current set of mutation operators, the test cases are also needed for any gaps in the current set of mutation operators. The existing set of mutation operators does not represent a full range fault types that can occur in a program and need new mutation operators to cover these gaps. In order to achieve this goal, effective mutation operators are needed. To be effective, a mutation operator must produce a suite of mutants such that any set of test cases that kill all of these mutants (characterized by a fault type (ft) that can exist in a program) should also be able to kill any further seeded or accidental fault of that fault type. This behavior of the effective mutation operator shows that the same test suite is able to detect a larger set of faults and not simply the mutants generated using a mutation operator. For example, a mutation operator specification can be defined as replacing the "positive (+) arithmetic operator" with a "subtraction (-) arithmetic operator." If the mutants generated by applying this specification to a program are killed by a test suite (T), then any accidental fault of the same fault type (e.g. changing the positive (+) arithmetic operator to multiplication (*) arithmetic operator) should also be killed by T for this mutation operator to be effective. Also, for a mutation operator to be effective it must focus on fault types specific to OO programming concepts (e.g. Class invariants, Inheritance, Polymorphism, Method overloading). The goal is to detect OO faults and not any traditional fault (e.g. changing relational operators) that is common to both non-OO and OO programming. As mentioned earlier, there is a gap in the current set of mutation operators. The existing set of mutation operators does not address all object-oriented faults mentioned in the literature. Appendix A lists the set of fault types that are not addressed by a mutation operator. Effective mutation operators need to be added to the existing set of mutation operators that represent object-oriented faults. These effective OO mutation operators when used for inducing an instance of a fault type in a program and a test case selection technique that requires a test suite that detects the induced as well as any accidental faults in a program, can enhance the evaluation technique for software testing methodologies. The mutants generated using the effective mutation operators when detected by this test suite will demonstrate if the testing methodology is successful in detecting any induced faults as well as any accidental faults specific to an OO fault type. This will enhance the ability of the evaluation technique to analyze the software testing methodology and demonstrate if the methodology was effective in killing a larger set of OO faults for the same fault type using the same test suite. The AFM technique described in the earlier section can compare the software testing methodologies and can be used as an evaluation technique.

The key contributions of this thesis are the following:

1) Categorization of faults – Fault Types: The faults are categorized based on the object-oriented property.
2) Traditional vs Object-oriented fault types: The focus of this thesis is on object-oriented fault types that emphasize the faults caused by object-oriented properties such as inheritance, method overloading, etc.
3) New and improved mutation operators: Mutation operators are defined for fault types not addressed by the existing set of mutation operators.
4) Formalization of mutation operators: A Mutation Operator Production Rule System is defined that uses a grammar to generate mutants in a program.
5) Accidental fault methodology: This methodology can enhance the evaluation technique for software testing methodologies.
6) The concept of effective mutation operators: The mutants generated by effective mutation operators can kill a larger set of faults i.e. both seeded and accidental faults.

The components are shown in the figure below.

Figure 1.2 Accidental Fault Methodology



The next chapter discusses related work and chapter 3 describes the utilized approach. Chapter 4 defines new and improved mutation operators and chapter 5 illustrates the results of applying the new mutation operators to the sample programs. Finally, chapter 6 presents conclusions and describes scope of future work.

Chapter 2 - Related Work

Traditional Mutation Operators

Originally, mutation operators were created for non-OO fault types but the idea of mutation operators was later applied to OO software fault types [DE78]. A few examples of the traditional fault types for non-OO programs are missing statement, incorrect arithmetic operations, incorrect Boolean expressions, and incorrect variable [AG89 & JI08]. Examples of traditional mutation operators that model these fault types are ABS (Absolute value insertion), AOR (Arithmetic operator replacement), LCR (Logical connector replacement), ROR (Relational operator replacement), and UOI (Unary operator insertion). Traditional mutation operators can be used to induce fault instances in both non-OO and OO programs [AG89, JI08, OF96 & WO90]. Furthermore, mutation operators have been defined for OO programs that focus on fault types specific to OO programming.

Object-oriented Fault Types

One challenge for this research was to collect a standard set of OO fault types. Few papers in the literature that list fault types that are specific to OO programs. The introduction of OO programming methodology created opportunities for new fault types known as OO fault types [HA94 & WA05]. Hayes analyzed OO fault types and conventional methods applied to these fault types in order to define fault taxonomy for OO systems [HA94]. His research used fault types from Purchase & Winder, Firesmith and Offutt et al. [FI93, OA01 & PU91]. Firesmith and Offutt et al. list OO fault types based on various OO program features such as class, attributes and methods as well as OO program properties such as inheritance, polymorphism, and method overloading [OA01 & FI93]. The research also suggests various types of testing methods for each fault type. Purchase & Winder presented nine types of OO fault types detectable by debugging tools [PU91]. Current research by Walkinshaw et al. mentions a number of OO fault types collected from various sources [AL02, BI99, HA94 & WA05]. Walkinshaw et al. states that little investigation has been done to find specifics of fault types likely to occur in OO projects; therefore, the development of testing techniques for OO software is difficult [WA05]. Offutt et al. focused on inheritance and polymorphism properties of OO programs [AL02]. This research stated that even though these properties are useful, they come at a cost. However, the analysis of fault types and testing methods associated with these fault types is crucial. Lin et al. analyzed fault types related to the polymorphic property of OO programming [LI05]. The goal of Lin et al. research was to provide a realization for OO fault types in polymorphism and also discuss specific categories of inheritance and polymorphism fault types. With the change of paradigms, the methods to develop and test the OO programs have changed [FI93]. In this thesis, the focus is on OO fault types that are collected from various studies [OA01 & FI93].

OO fault types already considered for creation of a mutation operator from Chevalley and Firesmith are incorrect overloading methods implementation, *super* keyword misuse, *this* keyword misuse and faults from programming experience [CH01 & FI93]. Other OO fault types considered by Offutt et al. are state visibility anomaly, state definition inconsistency, state definition anomaly, indirect inconsistent state definition, anomalous construction behavior; incomplete construction and inconsistent type use [OA01 & OF01]. Additional OO fault types used to create mutation operators are faults such as overloading methods misuse, access modifier misuse and *Static* modifier misuse [KI00]. These OO fault types have been represented by a mutation operator that can be used to induce possible faults in a program. Appendix A shows fault types collected from literature and mapped to existing mutation operators [BI96, CH01, DE06, FI93, KI00, MA02, OA01 & OF01].

Mutation Testing

The concept of mutation analysis was introduced to induce faults in a program, commonly known as mutation testing. The idea of mutation was introduced by Richard Lipton in 1971 in a class term paper entitled "Fault diagnosis of computer programs" [OA01 & OF01]. It was in the late 70s when the mutation model was researched and published [AC79, BU77, DE78 & LI78]. The mutation analysis includes steps to induce faults in a program using a series of mutation operators. Mutation operators model these faults and represent a skeleton of code for how these faults can be induced in a program. The mutated or incorrect version of a program is then subjected to a test suite to discern which mutation is killed. Implementation of the mutation analysis system was introduced in 1977 [AC79, BU77, BU78 & LI78]. This implementation created mutants for FORTRAN IV programs, accepted test cases from the user and then executed those test cases against mutant programs to detect the number of mutants killed by a test suite. Mutation testing has evolved since the 1970s [WO90]. Woodward et al. discussed concepts such as weak mutation, strong mutation and firm mutation for traditional or non-OO fault types [WO88]. Weak and strong mutation testing has been researched in detail [GI85, HO82 & WO88]. Offutt et al. discussed additional approaches to mutation testing such as "do-fewer" or "do-smarter" [OF01].

All mentioned research, applied mutation analysis to non-OO fault types. OO programming concept was introduced in 1950s, but the methodology was not dominant until the 1990s. Therefore, a majority of mutation operators and tools designed for mutation testing were focused on non-OO programming. Appelbe et al. applied mutation analysis method to programs written in Ada programming language with some OO mutations [AP88]. Other research applied mutation analysis to FORTRAN and COBOL [AC79]. In addition, it was applied to OO programs by Yoon et al. [YO88]. The paper discusses inter-class mutation testing in which mutations are defined to test the interaction of public methods calls between classes. The paper applies mutation analysis to the state diagram that represents class behavior based primarily on the inheritance property. Jia et al. analyzed a survey in detail on mutation testing [JI11]. The analysis took into account theories, optimization techniques, equivalent mutant detection, applications and mutation tools. Survey results showed an increase in the number of tools that can manage large and realistic programs. Results also revealed that further research is required to generate more realistic mutants for programs. Ma et al. evaluated the mutation testing technique for OO programs and proposed some mutation operators [MA06]. Mutation testing has been considered an effective testing method but this thesis does not concentrate on mutation testing but focus on a portion of mutation testing, i.e., mutation operators used for inducing faults in a program. As mentioned, additional research is required in order to create mutation operators specific to OO programs. Kim et al. and Ma et al. conducted initial research that created mutation operators specific to OO programs [KI00 & MA02].

Mutation Operators

Traditional faults or non-OO faults are represented by simple mutation operators. Mutation operators have been defined for the C programming language which is a non-OO programming language [AG89]. The paper classifies traditional mutation operators into four categories: statement mutations, operator mutations, variable mutations and constant mutations. Traditional mutation operators are valid for OO programs, but additional operators must be defined for the OO programs that represent the OO faults [CH01 & MA06].

Kim et al., Ma et al., and Derezińska defined mutation operators for OO programs [DE06, KI00, MA02 & MA06]. Kim et al. introduced *Class Mutation* and created mutation operators specific to OO programs [KI00]. Previous research did not address flaws or faults specific to OO programs. Kim et al. introduced a *Class Mutation* method that targeted fault types that could occur in OO programs [KI00]. This research categorized fault types based on polymorphic types, method overloading, method overriding, field variable hiding, information hiding, static/dynamic stated of objects, and exception handling. Ma et al. defined more mutation operators for OO programs and evaluated mutation testing for OO programs using empirical studies [MA02 & MA06]. The research provides an empirical study in order to compare the behavior of OO and non-OO mutation operators. Research by Ma et al. define mutation operators specifically for Java language [MA02 & MA06], but a majority of operators can be applied to other OO programming languages. Offutt et al. discussed the faults and mutation operators based on OO properties, such as polymorphism and Inheritance [OA01 & OF01].

Standard Set of Mutation Operators

In order to define a standard set of OO mutation operators, a standard set of possible OO fault types that can exist in a program must be defined. Faults specific to OO programs must be addressed and corresponding mutation operators must be defined. An OO mutation operator specifically designed to induce an OO fault instance is more likely to detect expected OO fault types in OO programs.

The faults collected from literature that are specific to OO programming should be represented by a mutation operator [BI96, FI93 & OA01]. Ideally, all OO faults should be represented by mutation operators that can be applied to a program in order to generate mutants, but, all OO fault types are not represented by an existing mutation operator [CH01, DE06, KI00, MA02 & OF01]. These OO fault types are shown in Appendix A.

Mutation Tools

Extensive research and development has created mutation tools for languages such as FORTRAN, Ada, COBOL, C and Java. Several mutation tools have been built, and research indicates that the number of tools able to handle large programs has increased in past years. Some initial mutation tools include PIMS and EXPER, built for FORTRAN programs [AC79, BU78 & BU80]. Another system that mutates FORTRAN programs was FMS.2 and the one built for mutating COBOL programs was CMS.1 [AC79, AC80, HA80 & TA81]. Mothra was developed to support multiple source languages, thus providing an integrated testing environment with the use of mutation analysis [DE88)]. This system has been enhanced since and is now available for Java programming language [MA06 & OF04]. *MILU*, a mutation tool for non-OO programs written in C language, is available for first order and higher order mutation testing [JI08]. *Judy*, system available for Java, allows estimation of the fault detection effectiveness of test suites for much larger systems [MA10]. *Proteum*, another mutation testing tool initially built for C programming language but has evolved and is now available for OO programs [DE96 & DE00]. *MuJava* is another mutation system built specifically for Java [OF04]. The system automatically generates mutants, runs them against a test suite and calculates the mutation score. The tools were built to save time when using mutation testing technique for testing programs. Mutation analysis is a fault-based testing technique and thus requiring knowledge of which fault types need to be tested. All tools use pre-defined mutation operators that represent these fault types to generate mutants in program, thus emphasis should be placed on mutation operators that are used to induce fault types in OO programs.

Software Testing Evaluation Techniques

The prime question in this area of study is how any two software testing techniques can be measured for effectiveness. The possible dimensions for evaluation are: if technique 1 find more faults than technique 2, or if technique 1 finds more critical faults than technique 2, or if technique 1 produces more reliable software then technique 2 [FA11]. In this thesis, the focus is on first dimension that can be used as an evaluation parameter. The research by Farooq and Quadri also state that focus of testing methodologies should not only be on fault finding ability but also factors such as fault severity, cost and efficiency should be considered. It mentions that two types of studies that has been carried out to evaluate the testing techniques: analytical and empirical. An analytical analysis would show relative effectiveness of software testing techniques. On the other hand, the empirical studies measure the observed experimental outcomes [FA11]. In this thesis, an empirical analysis of an experimental outcome is presented.

Another comparison is mentioned by Benli compares the performance of two testing methodologies based on the difficulty of creating test cases and detecting software errors [BE12]. It focuses on testing techniques for the Android operating system. With various platforms, applications being introduced, the effort of each testing methodology has become more complex and time consuming. A recent research weighs on several empirical studies conducted for both static and dynamic testing technique [TY14]. The results of this research contradicts some results in the research by Selby and Basili [SE12 & TY14]. Selby and Basili show results of an experiment, stating that code reading techniques found more faults as compared to other techniques whereas Tyagi and Malhotra state otherwise. The reason behind the contradiction is probably the diversity of the techniques that are used for the experiment. This shows the non-uniformity of the evaluation strategies being used to assess testing methodologies. Tyagi and Malhotra compare various results from authors that have used different evaluation techniques to compare testing methodologies [TY14]. All experiments rank testing methodologies with the exception of one research that states effectiveness of a technique is program dependent [FA13]. A recent research states there is a need in the industry to have guidelines on which testing technique should be based. This could be based on the testing objectives and the ranking can be measure on the scale of (effective, efficient, satisfactory) [VO12]. The absence of guidelines is very crucial to this area of study. It is difficult to achieve consensus until we have a common strategy to develop an evaluation framework for any program and then an evaluation guideline for ranking the technique.

Chapter 3 - Methodology

In this thesis, additional mutation operators that are effective, precise and geared towards object-oriented faults are created to suffice the hypothesis.

Addition to the existing set of mutation operators is required as there are OO fault types described in the literature that are not yet covered by any of the currently defined mutation operators. This new set of operators (MO_{new}) will be an addition to the existing set of mutation operators for OO fault types and *create a larger set of mutation operators* that can be applied to OO programs. List of known OO fault types is growing with ongoing research in the field and no research lists a complete set of OO fault types, therefore the proposed list of mutation operators in this thesis is *not complete*. A set of effective mutation operators is added to the existing set of mutation operators.

***Mutants**$_{moe,\,p}$*: $Mut_{moe,\,p}$ is the set of all mutants generated when existing mutation operator (mo) is applied to program (p), $Mut_{moe,\,p}$ = {$\mathrm{mut}_{\mathrm{moe,\,p,\,n}}$ | $\mathrm{mut}_{\mathrm{moe,\,p,\,n}}$ is a mutant created by applying an existing mutation operator (mo) to program (p)}. The existing mutation operators are listed in Appendix A.

***Mutants**$_{mon,\,p}$*: $Mut_{mon,\,p}$ is the set of all mutants generated when a new mutation operator (mo) is applied to program (p), $Mut_{mo,\,p}$ = {$\mathrm{mut}_{\mathrm{mo,\,p,\,n}}$ | $\mathrm{mut}_{\mathrm{mo,\,p,\,n}}$ is a mutant created by applying a mutation operator (mo) to program (p)}.

The three sets of faults that can exist in a program are mutants from the existing mutation operator, mutants from the new mutation operators and any accidental faults. A test suite that kills the $\mathrm{Mutants}_{\mathrm{moe,\,p}}$, $\mathrm{Mutants}_{\mathrm{mon,\,p}}$ and also the accidental faults in a program will show that a larger number of faults are killed by the same test suite. This methodology enhances the evaluation technique for software testing methodologies. These mutation operators when used for evaluating a software testing methodology will exhibit if the methodology is successful in finding any seeded faults as well as any accidental faults specific to an OO fault type.

Existing mutation operators have been detailed in a more verbose manner as compared to a precise and logical rule. Literature research shows that mutation operators have been defined using a verbose definition with a code example showing how the mutant can be generated in a program [MA2002, MA2006 and OF2001]. The existing mutation operators are not precisely defined and do not include information about the exact location of change to be made in a program. Example, Access Modifier Change (AMC) operator was defined to change the access modifier in a program. Now this operator can be applied to a class, method or variable declaration. The existing operator does not make a distinction between these different sections of a program to which the operator can be applied. This mutation operator is also missing the exact change of access level i.e. public->private or private->public. Depending on what change and where a mutation operator is induced, the behavior of the program may or may not change. The new and improved mutation operator specifications are a *precise and logical rule* that when applied to a program generates a set of mutants explicit to the fault type.

An important aspect of creating additional mutation operators is to focus on fault types specific to OO programs. A few researchers have made the distinction between mutation operators for traditional faults vs. object-oriented faults. The traditional mutation operators can still be applied to the OO programs but will not find faults specific to OO programming concepts such as inheritance, method overloading, polymorphism, etc. In this thesis, focus is on *object-oriented mutation operators* for object-oriented faults.

Collection of Fault Types and Mutation Operators

The first step of this research was to collect known OO fault types from the literature [OA01 & FI93]. As mentioned earlier, this list is growing with the ongoing research in the area of software faults. An effort has been made to collect a complete list of fault types from the published literature dated back to 1993. The author apologizes for missing any published fault types in the literature. The OO fault types (*FT*), and the already defined mutation operators (MO_{OO}), are collected from the literature in order to ascertain which fault types are not represented by a mutation operator [AG89, CH01, FI93, KI00, MA02, MA06, OA01, OF01 & PU91]. The list of collected fault types and mutation operators is available in Appendix A.

Specifying Mutation Operators

Appendix A lists a set of known fault types for which mutation operator is defined. The Appendix categorizes each fault type and classifies the related mutation operators under categories based on OO features and properties. Appendix A lists a set of fault types not addressed by existing mutation operators. The goal of this research was to carefully analyze fault types in detail and define new mutation operator(s) that provide a specification for how instances of fault types can be induced in a program. Depending on where the change is made in a program, the output or the behavior of the program may or may not be different and each change will represent a different instance of the fault type induced. Different mutation operators were defined for each fault type that are specific to where and what is changed in a program. Each mutation operator describes specific rules pertaining to how an instance of a fault type is seeded in a program. Chapter 4 details fault types and the specifications of mutation operators for how these fault types were generated in a program. The specifications were generated using the grammar outlined below. It provides a precise and logical set of rules that can be applied to a program. This thesis adds new mutation operators for some fault types mentioned in Appendix A.

Literature review shows that many research projects do not make a distinction between a fault and a fault type, and only describes an example of how a fault can be induced in a program using a mutation operator [AP89, CH01, DE06 and KI00]. The aim of this research was to analyze each fault type in detail and present an effective mutation operator that induces an instance of a fault type in a program. As mentioned above, there can be more than one mutation operator for each fault type depending on what changes are made in a program. It can be seen from Appendix A that usually for each fault type only one mutation operator has been defined in the literature. The focus is on the precision of rules. As each fault type can have more than one operator depending on the changes made in the program, there will be multiple rules that can be applied to a program to generate specific instances of fault types. In the next section, some design strategies are explained for how to precisely specify a mutation operator. These strategies were used to define new and improved mutation operators that are mentioned in chapter 4.

Designing New Mutation Operators

The design of an operator should answer the following questions for creating a precise specification of a mutation operator:

1. How (Insert, Delete, Change): Insert new LOC, Delete existing LOC, Change existing LOC
2. Where (Scope): Class (Constructor, Method, Variables), Method (Parameter, Body), Variable (Declaration/Initialization, Use)
3. When (After, Before, In): After certain LOC, Before certain LOC, In LOC
4. What (LOC) : LOC to be inserted, LOC to be deleted, LOC to be changed

Example: IVS10 Mutation Operator

1. How: Insert new LOC
2. Where: Class
3. When: After an assignment statement for "y"
4. What : Insert y = <expr> - n, where n is a randomly generated number

Using this approach, a production rule system is created that can be used to create a tool for mutant generation.

Mutation Operator Production Rule System

In this thesis, a system to precisely specify mutation operators that when applied to a program can generate mutants is proposed. A Mutation Operator Production Rule System (MOPRS) consists of a set of rules and behavior that defines how a mutation operator is induced in a program. The rules are known as productions that consist of two portions: the IF statement that is the syntactical precondition that can be true or false and the action that is based on this precondition is stated through the THEN statement. The production rule action is triggered when the precondition is true.

Depending on the fault type, there can be multiple mutation operators and thus multiple production rules for the same fault type. The production rules for each new mutation operator are shown in the chapter 4. These production rules are then converted into specifications that are applied to a program. A production rule is converted into a specification using two concepts: Keywords and Operators. Using a MOPRS grammar, the production rules are converted into a format that can be applied to a program to generate mutants. The IF precondition defines which

Keywords are used to search for specific statements and defines where changes are made in a program. The *Operators* define specific variables, expressions that can be changed or a reference for the location where changes are made, e.g. after or before a LOC.

For example, a production rule for a traditional mutation operator that replaces an arithmetic operator will be:

If there is an expression, x = y + z;

Then replace an instance of this expression with this statement, x = y – z;

Using the production rule shown above and the keywords and operators defined below, the specification using the grammar for this production rule will consist of:

```
IF MATCH: "‵ "(″ VAR "=″ VAR OP VAR′″ ")″
OP: "+″
THEN
REPLACE: VAR "=″ VAR OP1 VAR ";″
OP1: "-″
```

The specification examples for the new mutation operators are presented in chapter 4. Shown below are the numerous Keywords and Operators defined for MOPRS grammar.

Keywords

`IF MATCH/MATCH#`:

`MATCH`: Will find all instances of specified code and perform operations.

`MATCH#`: When a number (N) is specified it will skip the N-1 instances before it and perform operations on the Nth instance.

`IF MATCHINCOMMENT`: Will find the specified code within commented text.

`THEN`: The statements following this keyword are required as an action based on the `IF MATCH/MATCHCOMMENT.`

`FIND/FIND#`:

`FIND`: Will find all instances of this string.

`FIND#`: When a number (N) is specified, it will skip N-1 instances before it performs operations on the Nth instance.

`INSERTAFTER`: The string here will be placed after the Found item.

`INSERTBEFORE`: The string here will be placed before the Found item.

`REPLACE`: The string here will replace the Found item.

REPLACEVAR: The string here will replace the desired variable.

REPLACEPARAMETER: The string here will replace the desired parameter or the parameter list.

REPLACEMETHOD: The string here will replace the found method call with a compatible method.

NUMBERRANGE: The range of numbers in which the random NUM will be chosen.

` ': When between quotation marks denotes an area that white space might occur.

ASSOCIATEDMATCH: Specific class in consideration.

ASSOCIATEDFIND: Class where we find, replace or delete.

DELETE: Delete this line of code.

DELETE KEYWORD: Delete a specific keyword such as "super" or "this." Note: this keyword is specific to the programming language to which the mutation operators are applied.

CHILD: The child class involved in a mutation operator that is related to Inheritance property.

PARENT: The parent class involved in a mutation operator that is related to Inheritance property.

METHOD: A specific method in which the line of code needs to be changed or replaced.

METHODCALL: A specified method call that can be changed or replaced.

PARAMETER: Represents any parameter in the method declaration that can be changed or replaced.

PARAMETERLIST: Represents a parameter list in the method declaration that can be changed or replaced.

METHODBODY: Represents the method body that can be changed or replaced.

METHODNAME: Represents a method name that can be changed or replaced.

Operators

VAR: Represents any variable (VAR "=") that satisfies the MATCH;

OP: Represents any comparison operation (VAR OP) that satisfies the MATCH; can be specified within the specification with commas between multiple operators; example of common operators: "==", "<", "<=", ">", ">=", "!=".

OP1: Represents needed algebraic operation specified by designated OP within added text; example, when OP is ">" OP1 will be a "-". Validation will error if OP1 is used and OP is not included in the specification.

`EXPRESSION`: The expression `<expr>` that can be changed or replaced.
`NUM`: The random number in the `NUMBERRANGE`.
`EXPRESSION1`: The expression `<expr1>` that is not used. This is used a placeholder to denote code that will not be needed or saved.

Mutation Operator Production Rule System Grammar

In this section, a grammar is defined for the MOPRS. The production rules (IF-THEN) defined for each mutation operator need to be converted into some form of formal readable grammar. We define a grammar using the keywords and operators mentioned in the previous section. A grammar is formally defined as a set of grammar production rules that can be used to generate a specification which is further applied to a program to generate mutants. A grammar consists of following components:

a) A finite set of production rules – The production rules are defined in the MOPRS that are distinct from the grammar production rules. The MOPRS production rules define the mutation operator and the grammar production rules identify how to generate a specification for these mutation operators. The start symbol PR represents the production rule of the mutation operator.
b) A finite set of nonterminal symbols – The start symbol in this grammar is PR with nonterminal symbols A, B, C, E, F, G, H, K, L, S1, .
c) A finite set of terminal symbols – The keywords and the operators are the terminal symbols in this grammar. The language specific keywords are also included in this set of terminal symbols. D, I, J, OP, OP1 and Z are terminal symbols.

The grammar production rules for start symbol S are defined as follows:

```
S -> PR1 | PR2 | PR3 | PR4
PR1 -> IF MATCH A THEN B
PR2 -> IF MATCHINCOMMENT E THEN B
PR3 -> IF MATCH K THEN H
PR4 -> IF MATCH L THEN F
S1 -> “‘ “(” VAR OP EXPRESSION’” “)” D
A -> ‘assert S1’ D
A -> “‘ “(” VAR “I” VAR OP VAR’” “)” D
```

```
A -> ‘if S1’ D
B -> FIND VAR “I” EXPRESSION1 “Z” C
B -> PARENT CLASSNAME FIND VAR “I” EXPRESSION1 “Z” C
B -> ASSOCIATEDMATCH CLASSNAME ASSOCIATEDFIND CLASSNAME FIND VAR “I”
EXPRESSION1 C
C -> INSERTAFTER VAR “I” EXPRESSION OP1 NUM “Z” D
C -> INSERTAFTER VAR “I” EXPRESSION “Z” D
E -> ‘invariant S1’ “Z” D
C -> CHILD CHILDNAME INSERTAFTER VAR I EXPRESSION OP1 NUM D
C -> ASSOCIATEDFIND CLASSNAME INSERTAFTER VAR I EXPRESSION OP1 NUM D
K -> "‘EXPRESSION "J" VAR’" D
K -> EXPRESSION "J" METHODCALL "("")" D
L -> “‘METHODNAME (PARAMETER) “{” METHODBODY ’”“}” D
L -> “‘METHODNAME (“PARAMETERLIST”) “{” METHODBODY ’”“}” D
F -> FIND “{” METHODBODY “}” G
G -> REPLACE METHODNAME"("")" D
G -> REPLACE METHODNAME PARAMETERLIST[N] D
H -> REPLACE EXPRESSION “J” VAR1 D
H -> REPLACE EXPRESSION “J” METHODCALL"("")" “Z” D
I -> =
J -> .
OP -> >
OP -> <
OP -> <=
OP -> >=
OP -> ==
OP1 -> -
OP1 -> +
OP1 -> *
OP1 -> /
Z -> ;
D -> ε
```

This grammar generates the specifications that are applied to the sample program mentioned below. The next 2 sections detail the sample programs to which the new and improved mutation operators are applied.

Sample Programs

Sample Program 1 (GEO)

The first sample program used to apply the new mutation operator specifications was GEO (Geometric Objects). Examples of original and mutated code in the sample program GEO are shown in Appendix B. The program consists of 9 classes and has approximately 1500 LOC. The program is small but takes into account the properties of object-oriented programming such as Inheritance, polymorphism and method overloading.

Sample Program 2 (BANK)

The second sample program used was the BANK program that consists of 18 classes and has approximately 2500 LOC. The program is larger than GEO and also takes into account the properties of object-oriented programming such as inheritance, polymorphism and method overloading. The new and improved mutation operator specifications are applied to this sample program.

Using the definition of new mutation operators, specifications are created that when applied to the sample programs generate mutants. The new and smaller test suite as defined in chapter 4 is run against these mutants. Results in chapter 5 illustrate that the new mutation operators are able to kill a larger set of faults thus showing they are an “effective” representation of the fault type.

Test Cases

The test cases run against the sample program GEO are shown in Appendix C. The goal of this test suite was to test the normal behavior of a program, i.e. calculating area and perimeter of different geometric objects. We notice that some of the test cases show an overlap of constructors and methods. This overlap is used because different combinations of constructors and methods show different behavior of the program (GU10). Even though a mutation operator may mutate the constructor or the method, the test case includes the combination of constructors and method calls attached to the constructor. Thus, the result of the test cases is based on the construction of an object as well as which method calls are made in combination with the constructor (GU10). Due to this reason, the overlap of test cases is used in this paper. Appendix C shows a complete set of test cases that were used to create a mutation operator test suite (MOTS).

Chapter 4 - New Mutation Operators: Fault Types Associated with Classes and Methods

In this chapter, a fault type associated with invariants are examined and new mutation operators for possible faults that can be seeded in a program are proposed. A mutation operator specification details the production rules for generating the corresponding mutant in a program.

Fault Type 1– An Assignment that Violates a State Invariant

The property that defines the state of the object is called an invariant. A class invariant is a condition that defines the valid states for the object to ensure that the working of the class is correctly implemented. The invariants must hold when the object is constructed and should be preserved by all operations of the class. In other words, the class invariant is a relationship that includes the attributes of a class or a property that should hold for all instances of a class. For example, a *MinorChild* class may contain fields such as *age* and *birthdate*. A class invariant in this scenario can be (0 <= *age* <= 18) or (*birthdate* <= *todaysdate* – 216). The invariants should be preserved by all methods and constructors of the class. These are established during construction and maintained between calls to public methods. An invariant can be present in form of a condition or assertion.

The invariant needs to be preserved along the inheritance tree i.e. an invariant in the parent class should be preserved by the child class. Inheritance allows child class to change data that can become invalid as per the invariant defined in the parent class. The invariant makes sure that inheritance property does not break the encapsulation property of an OO program. In general, an invariant has the following structure:

Invariant: <a> <relationalOperator> <Expression>

When this mutation operator is applied to a program by changing only the relational operator, it can be said that a non-OO mutation operator has been applied to a program. In this research, we focus on the OO aspects of program. We will be analyzing the changes and effects of inducing this fault type in an OO program. For example, if there is an invariant (a>b) in a class. In order to induce this fault type, a mutation operator is defined that inserts an assignment statement (a=b) after any assignment statement in the code that reference object “a”. This will show the OO behavior of the program, i.e. if the class invariant is violated when a mutation operator is applied to the program.

Mutation Operators

An invariant that contains at least one attribute and a relational operator can be mutated. A mutation operator for this fault type ensures that invariant holds for the class as well as for any descendant class. The operator for this fault type takes into account the OO concepts such as invariants, inheritance and encapsulation. The mutation operators defined for this fault type will not change any existing lines of code. The code that violates the invariant is inserted in the class, i.e. a statement is inserted in the code to violate the invariant.

There are different ways in which invariants are introduced in a program. In Java™ programming language, invariants are introduced using pre-conditions and post-conditions. For example, /*@invariant x <= <expr>; */
A mutation operator for this scenario will be inserting an assignment statement for object "x" that violates the invariant, e.g. x = <expr> + 1.

Another way to check invariants in the code is by using assertions. If the assertion check option is available then the code can be mutated by inserting lines of code that results in violating the assert statement. The restriction is that assertions are disabled by default when executing a program because they reduce performance and are unnecessary for the program's user, but if the assertions are enabled and assert statements are already present in the code then this fault type can be induced by using assignment statement. A mutation operator in this scenario will insert a statement that violates the assert statement.

An invariant can also be preserved in the methods by using conditions that check the invariants in the method body. An invariant can be present in the method body in form of a condition that checks the relationship between attributes of the class. This mutation can be induced by mutating the condition that represents the invariant. Changing the position of attributes or the relational operator in the condition can be achieved by applying a non-OO mutation operator. For example, /*@invariant balance>= minimumBalance*/, if a conditional statement checking the invariant is "If (balance >= minimumBalance)", then this statement can be mutated to violate the invariant. The new mutation operators defined below will show the applicability of the OO mutation operators in this scenario.

Specification of Mutation Operators[1]

This section introduces the new mutation operator with respective production rules. A sample specification for each mutation operator is shown below. There can be multiple specifications for an operator subject to the changes being made by the operator. For example, IVS1 will generate all instances with the one specification shown below. Similarly, IVS2, IVS3, IVS9, IVS10 and IVS11 also generates all instances with one specification as shown below. On the other hand, IVS4 will need multiple specifications because of the type of changes being made in a program. This operator is applied to each parent-child relationship in the program and therefore a specification is needed for each relationship.

Correspondingly, IVS5, IVS6, IVS7 and IVS8 will need more than one specification. There are no changes in the definition of production rules. The same production rule will generate different specifications for all classes based on parent-child or association relationships.

Operator IVS1:

If there is an "assert" statement checking the invariant, then violate the invariant by inserting an assignment statement in the same class that violates the expression in the assert statement.

Production Rule:

If an assert statement exists in the class, assert(y >= <expr>)
Then insert an assignment statement y = <expr> - n after any assignment statement that references "y' in the code, where n is a randomly generated number.

Specification:

```
IF MATCH: "assert' "(" VAR OP EXPRESSION '""")"
OP: ">="
OP1: "-"
THEN
```

[1] The text highlighted in green shows the line of code to be searched and matched in the sample program. These statements are mapped to the MATCH, MATCHINCOMMENT keywords in the specification for the operator. Based on this statement the text highlighted in color red is inserted after a certain statement. These statement to be inserted is mapped to the INSERAFTER or INSERTBEFORE keywords in the specification. The FIND keyword is used to find that certain statement after (or before) which the line of code can be inserted.

```
FIND: VAR "=" EXPRESSION1 ";"
INSERTAFTER: VAR "=" EXPRESSION OP1 NUM ";"
```

Operator IVS2:

If the invariant is ensured using an "if" condition, then violate the invariant by inserting an assignment statement after any assignment statement for the assigned variable in the condition statement.

Production Rule:

If there is an "if" condition checking the invariant, if(y > <expr>)

Then insert an assignment statement y = <expr> in the class itself, after any statement that references "y" in the code.

Specification:

```
IF MATCH: "if' "(" VAR OP EXPRESSION'" ")"
OP: ">"
THEN
MATCHINCOMMENT: NULL
PARENT: NULL
CHILD: NULL
FIND: VAR "=" EXPRESSION1 ";"
INSERTAFTER: VAR "=" EXPRESSION ";"
INSERTBEFORE: NULL
REPLACE: NULL
```

Operator IVS3:

If the invariant is ensured using a statement defining pre-condition or post-condition, then violate the invariant by inserting an assignment statement after any assignment statement for the assigned variable in the pre-condition statement.

Production Rule:

If there is a pre-condition checking the invariant, /*@invariant x < <expr>; */

Then insert an assignment statement x = <expr> in the class itself, after any statement that references "x" in the code.

Specification:

```
IF MATCHINCOMMENT: "'/*@invariant VAR OP EXPRESSION ";" */'"
OP: "<"
PARENT: NULL
CHILD: NULL
THEN
FIND: VAR "=" EXPRESSION1 ";"
INSERTAFTER: VAR "=" EXPRESSION ";"
INSERTBEFORE: NULL
REPLACE: NULL
```

Operator IVS4:

If the invariant exists in the parent class as an "assert" statement, then insert an assignment statement that violates the invariant in the child class to check that invariant is preserved by the child class.

Production Rule:

If there is an assert statement in the parent class, assert (a >= <expr>)
Then insert an assignment statement a = <expr> - n after any assignment statement that references "a" in the child class, where n is a randomly generated number.

Specification:

```
IF MATCH: "assert '"(" VAR OP EXPRESSION'" ")"
OP: ">", ">="
OP1: "-"
THEN
PARENT: CIRCLE
CHILD: CYLINDER
FIND: VAR "=" EXPRESSION1 ";"
INSERTAFTER: VAR "=" EXPRESSION OP1 NUM ";"
INSERTBEFORE: NULL
```

Operator IVS5:

If the invariant is ensured using an “if” condition, then violate the invariant by inserting an assignment statement in the child class after any assignment statement for the assigned variable in the condition statement.

Production Rule:

If there is an “if” condition checking the invariant, if(y > <expr>)

Then insert an assignment statement in the child class y = <expr>-n, after any assignment statement that references “y” in the code, where n is a randomly generated number.

Specification:

```
IF MATCH: "if‘(" VAR OP EXPRESSION’")"
OP: ">"
OP1: "-"
THEN
PARENT: RECTANGLE
CHILD: SQUARE
FIND: VAR "=" EXPRESSION1 ";"
INSERTAFTER: VAR "=" EXPRESSION OP1 NUM ";"
INSERTBEFORE: NULL
REPLACE: NULL
```

Operator IVS6:

If the invariant is ensured using a statement defining pre-condition or post-condition, then violate the invariant by inserting an assignment statement in the child class after any assignment statement for the assigned variable in the pre-condition statement.

Production Rule:

If there is a pre-condition checking the invariant in a class, /*@invariant x > <expr>; */

Then insert an assignment statement x = <expr> in the child class, after any assignment statement that references “x” in the code.

Specification:

```
IF MATCHINCOMMENT: ‘/*@invariant VAR OP EXPRESSION ";" */’
```

```
OP: ">"
THEN
PARENT: RECTANGLE
CHILD: SQUARE
FIND: VAR "=" EXPRESSION1 ";"
INSERTAFTER: VAR "=" EXPRESSION ";"
INSERTBEFORE: NULL
REPLACE: NULL
```

Operator IVS7:

If there is an “assert” statement checking the invariant in a class, then insert an assignment statement in the associated class to violate the expression in the assert statement.

Production Rule:

An assert statement in a class, assert (a <= <expr>)

Then insert an assignment statement in the associated class, a = <expr> + n, after any assignment statement that references “a” in the code, where n is a randomly generated number.

Specification:

```
IF MATCH: "assert‘(" VAR OP EXPRESSION’")"
OP: "<="
OP1: "+"
THEN
ASSOCIATEDMATCH: RECTANGLE
ASSOCIATEDFIND: LINE
FIND: VAR "=" EXPRESSION1 ";"
INSERTAFTER: VAR "=" EXPRESSION OP1 NUM ";"
INSERTBEFORE: NULL
REPLACE: NULL
```

Operator IVS8:

If the invariant is ensured using an “if” condition, then violate the invariant by inserting an assignment statement in the associated class after any assignment statement for the assigned variable in the condition statement.

Production Rule:

If there is an "if" condition checking the invariant, if(y > <expr>)

Then insert an assignment statement in the associated class y = <expr>-n, after any assignment statement that references "y" in the code, where n is a randomly generated number.

Specification:

```
IF MATCH: "if‘(" VAR OP EXPRESSION’")"
OP: ">"
OP1: "-"
THEN
ASSOCIATEDMATCH: RECTANGLE
ASSOCIATEDFIND: LINE
FIND: VAR "=" EXPRESSION1 ";"
INSERTAFTER: VAR "=" EXPRESSION OP1 NUM ";"
INSERTBEFORE: NULL
REPLACE: NULL
```

Operator IVS9:

If the invariant is ensured using a statement defining pre-condition or post-condition, then violate the invariant by inserting an assignment statement in the associated class after any assignment statement for the assigned variable in the pre-condition statement.

Production Rule:

If there is a pre-condition checking the invariant, /*@invariant x < <expr>; */

Then insert an assignment statement in the associated class x = <expr>, after any assignment statement that references "x" in the code.

Specification:

```
IF MATCHINCOMMENT: ‘/*@invariant VAR OP EXPRESSION ";" */’
OP: "<"
THEN
ASSOCIATEDMATCH: RECTANGLE
ASSOCIATEDFIND: LINE
FIND: VAR "=" EXPRESSION1 ";"
```

```
INSERTAFTER: VAR "=" EXPRESSION ";"
INSERTBEFORE: NULL
REPLACE: NULL
```

Operator IVS10:

If the invariant is ensured using an "if" condition, then violate the invariant by inserting an assignment statement after any assignment statement for the assigned variable in the condition statement.

Production Rule:

If there is an "if" condition checking the invariant, if(y = <expr>)
Then insert an assignment statement y = <expr> - n after any assignment statement that references "y' in the code, where n is a randomly generated number.

Specification:

```
IF MATCH: "if‘(" VAR OP EXPRESSION’")"
OP: "="
OP1: "-"
THEN
FIND: VAR "=" EXPRESSION1 ";"
INSERTAFTER: VAR "=" EXPRESSION OP1 NUM ";"
REPLACE: NULL
```

Operator IVS11:

If the invariant is ensured using a statement defining pre-condition or post-condition, then violate the invariant by inserting an assignment statement after any assignment statement for the assigned variable in the pre-condition statement.

Production Rule:

If there is a pre-condition checking the invariant, /*@invariant x > <expr>; */
Then insert an assignment statement in the class itself x = <expr>, after any statement that references "x" in the code.

Specification:

```
IF MATCHINCOMMENT: '/*@invariant VAR OP EXPRESSION ";" */'
OP: ">"
THEN
FIND: VAR "=" EXPRESSION1 ";"
INSERTAFTER: VAR "=" EXPRESSION ";"
INSERTBEFORE: NULL
```

Fault Type 2 (New and Improved Specification) – Method Incorrectly Performed (MIP)

Object oriented programming added the concepts such as overloading and overridden methods to the programming methodology. It becomes vital to ensure that the correct method is used as well as the method is performing the desired function correctly. Defining functionally correct methods as per the requirements is an important aspect of object oriented programming.

Method overloading and method overriding features of OO programming have introduced the possibility of new faults in the OO programs. The method overloading features helps in defining methods with same name but a different parameter list, in other words more than one method is defined with different signatures. The method overriding allows the ability to define such methods that allows a child class to inherit from the parent class with some modified behavior.

The method declaration should perform the instructions as needed by the program. Method overloading is a powerful feature and overloaded methods with different signatures should be defined carefully. A method should not have excessive overloading which makes it difficult for the developers to understand the different method declarations.

Mutation Operators

Some of the components of method declarations that can be mutated are: access modifiers, method name, parameter list, method body and return type. The components that are related to incorrect functionality are parameter list, method body and return type. If these three components are incorrectly defined, then the method will not perform the intended behavior.

There are existing mutation operators for this fault type. Shown below are mutation operators that have improved specifications for the existing mutation operators and also the new mutation operators defined for this fault type. The existing mutation operator specifications were not defined precisely. These operators were analyzed in detail to outline a precise and logical rule for a mutation operator. The new mutation operators defined covers some gap in the existing set of operators.

Specification of Mutation Operators[2]

This section introduces the new mutation operator for MIP fault type with respective production rules. A sample specification for each mutation operator is shown below. MIP3 will generate all instances with the one specification. Similarly, MIP4 also generates all instances with one specification as shown below. On the other hand, MIP1 will need multiple specifications because of the type of changes being made in a program. This operator is applied to each parent-child relationship in the program and therefore a different specification is needed for each relationship. Correspondingly, MIP2 will need more than one specification to generate mutants. There are no changes in the definition of production rules.

Operator MIP1 (Improved Specification for OMR [KI2000, MA2002 & MA2006]):
If there is an overloaded method, then change the contents of the original method to the overloaded method (with no parameters) call.

Production Rule:
If there is an overloaded method, x(y) and x ().
Then change the contents of x(y) to a method call, x ().

Specification:
```
IF MATCH: "‘findPerimeter (PARAMETER) "{" METHODBODY ’"}""
THEN
FIND: "{" METHODBODY "}"
INSERTAFTER: NULL
```

[2] The text highlighted in green shows the line of code to be searched and matched in the sample program. These statements are mapped to the MATCH, MATCHINCOMMENT keywords in the specification for the operator. Based on this statement the text highlighted in color red is inserted after a certain statement. These statement to be inserted is mapped to the INSERAFTER or INSERTBEFORE keywords in the specification. The FIND keyword is used to find that certain statement after (or before) which the line of code can be inserted.

```
INSERTBEFORE: NULL
REPLACE: findPerimeter ()";"
```

Operator MIP2 (Improved Specification for OMR [KI2000, MA2002 & MA2006]):

If there is an overloaded method with some parameters, then change the contents of the original method to the nth overloaded method call.

Production Rule:

If there are N number of overloaded methods with some parameters, x(y, z), x(y), x (z)
Then change the contents of x(y, z) to a method call for x(y) and x (z) one at a time.

Specification 1:

```
IF MATCH: "'METHODNAME ("PARAMETERLIST") "{" METHODBODY '"}""
METHODNAME: findPerimeter
THEN
FIND: "{" METHODBODY "}"
INSERTAFTER: NULL
INSERTBEFORE: NULL
REPLACEPARAMETER: PARAMETERLIST[0]
REPLACE: METHODNAME "(" PARAMETERLIST[0] ")" ";"
```

Specification 2:

```
IF MATCH: "'METHODNAME ("PARAMETERLIST") "{" METHODBODY '"}""
METHODNAME: findPerimeter
THEN
FIND: "{" METHODBODY "}"
INSERTAFTER: NULL
INSERTBEFORE: NULL
REPLACEPARAMETER: PARAMETERLIST[1]
REPLACE: METHODNAME "(" PARAMETERLIST[1] ")" ";"
```

Operator MIP3 (New Specification):

If there reference to an attribute used in the calculation, then change the reference to another compatible attribute.

Production Rule:

If there an expression using the attribute, such as x = 2*this.radius

Then change the reference to another compatible attribute, such as y = 2*this.height

Specification:

```
IF MATCH: "‘EXPRESSION "." VAR’"
THEN
INSERTAFTER: NULL
INSERTBEFORE: NULL
REPLACEVAR: VAR1
REPLACE: EXPRESSION "." VAR1 ";"
```

Operator MIP4 (New Specification):

If there is another method call inside the body of the method, then change the call to another compatible method.

Production Rule:

If there is a method call x() inside the body of method z.

Then change the method call x() to y() in the body of the method z, where x and y are compatible.

Specification:

```
IF MATCH: EXPRESSION "." METHODCALL "("")"
THEN
FIND: NULL
INSERTAFTER: NULL
INSERTBEFORE: NULL
REPLACEMETHOD: METHODCALL1
REPLACE: EXPRESSION "." METHODCALL1"("")" ";"
```

Chapter 5 - Results and Discussion

In this thesis, new and improved mutation operators are created for two faults types - an assignment statement that violates a state invariant (IVS) and method incorrectly performed (MIP). The former fault type has not been addressed an existing mutation operator. In this paper, new mutation operators for this fault type are defined. The latter fault type contains new mutation operators as well as improved specifications for existing mutation operators that originate under this fault type. MIP fault type is related to the method overloading property of OO programming methodology. The new and improved specifications of the mutation operators are applied to the two sample programs: GEO and BANK.

Four mutation operators are applied to the two sample programs: IVS10, IVS11, MIP1, and MIP3. The reason for selecting IVS mutants for experimentation is that none of the existing research have addressed mutation operators related to the invariant fault type. On the other hand, MIP has been addressed by some of the research papers [MA02 & MA06]. But there is lack of mutation operators for the MIP fault type, thus new and improved specification for mutation operators is defined. Example, MIP1 is an improved specification of an existing mutation operator OMR and MIP3 is a new mutation operator created for fault type MIP as defined in the previous chapter. This selection shows the diversity of defining new mutation operators as well as improving the specifications of existing mutation operators.

All the mutants created by applying the mutation operators to the sample programs are killed by some of the test cases from the complete set of test cases. This subset of test cases formulate a new test suite which is run against the accidental faults belonging to the same fault type as the mutants. The results for both sample programs show if the mutants were killed by the new test suite then accidental faults are also killed by the same test suite. When IVS11, MIP1 and MIP3 mutation operators were applied to the BANK sample program, more accidental faults were killed by the new test suite created from killing the mutants. On the other hand, when IVS10 mutation operator was applied to the GEO sample program, more accidental faults were killed by the new test suite created from killing the mutants. The subsequent sections discuss this performance for all mutation operators in detail.

Operator IVS10 Results[3]

If there is an "if" condition checking the invariant, if(y = <expr>)

Then insert an assignment statement y = <expr> - n after any assignment statement that references "y' in the code, where n is a randomly generated number.

The results of the test suite run against the mutant programs for mutation operator IVS10 are shown in table 5.1. The row title lists the mutants and the accidental faults that are identified in the sample program. The mutants and the accidental faults for sample program GEO mentioned in the table can be mapped to the highlighted code in Appendix B. The column titles in table 5.1 represent the test case numbers. The complete set of test cases used in this experiment is shown in Appendix C. The test suite listed in table 5.1 killed either a mutant or an accidental fault. This test suite was then run against the accidental faults. The new and smaller test suite, *MOTS*, is created from selecting test cases that have killed any of the 7 mutants. The "X" in the table cell denotes that the test case has killed the mutant and/or the accidental fault.

The subset of test cases is then run against the accidental faults to list the test cases that killed the mutants as well as the accidental faults. Some test cases only killed one mutant generated using the mutation operator IVS10, e.g. test case number 2 killed only mutant 2 and test case number 4 killed only mutant 1. Other test cases that killed more than 1 mutant are test case number 7 killing mutant 1, 2 and 5. Example of test cases killing both mutants and accidental faults is all test cases except test case number 2. Test case 2 only killed mutant 7 but test case 2 was not able to kill any of the accidental faults.

The probabilities to kill the accidental fault instances based on the test cases killing the mutants are shown in table 5.2. These probabilities are calculated based on results shown in table 5.1 and equation (1) explained in chapter 1. The overlapping of test case killing both the mutant and the accidental fault is used in this calculation. For example, the probability of killing accidental fault 1 given that mutant 1 is killed using the test suite listed in table 5.1 can be

[3] The text highlighted in green shows the line of code to be searched and matched. These statements are mapped to the MATCH, MATCHINCOMMENT keywords in the specification for the operator. Based on this statement the text highlighted in color red is inserted after a certain statement. These statements are mapped to the INSERAFTER or INSERTBEFORE keywords in the specification. The FIND keyword is used to find the certain statement after (or before) which the line of code can be inserted.

Table 5.1 Results of IVS10 mutation operator for GEO program

	Test Cases																							
	2	***4***	***7***	***8***	***10***	***26***	***38***	***39***	***40***	***41***	***43***	***44***	***47***	***49***	***50***	***51***	***52***	***57***	***58***	***59***	***60***	***65***	***69***	***70***
Mutant 1		X	X	X				X			X							X	X	X	X			
Mutant 2			X	X								X										X		
Mutant 3							X	X	X							X	X						X	
Mutant 4				X						X	X	X							X				X	X
Mutant 5			X	X			X	X	X	X		X				X	X						X	
Mutant 6					X	X								X	X									
Mutant 7	X												X						X					
Accidental Fault 1			X							X		X												
Accidental Fault 2					X	X							X											
Accidental Fault 3									X							X	X						X	
Accidental Fault 4			X					X			X								X	X				
Accidental Fault 5		X						X			X	X						X		X		X	X	X
Accidental Fault 6								X									X						X	
Accidental Fault 7					X	X								X	X									
Accidental Fault 8										X											X	X	X	X
Accidental Fault 9				X	X		X			X	X	X	X		X				X				X	X
Accidental Fault 10				X	X						X	X											X	X

Table 5.2 Probabilities of killing IVS10 Mutants and Accidental Faults for GEO program[4]

	Mutant 1	*Mutant 2*	*Mutant 3*	*Mutant 4*	*Mutant 5*	*Mutant 6*	*Mutant 7*	*Probability*
Accidental Fault 1	0.1111	0.5000	0.0000	0.3333	0.3000	0.0000	0.0000	0.7926
Accidental Fault 2	0.0000	0.0000	0.0000	0.0000	0.0000	0.5000	0.3333	0.6667
Accidental Fault 3	0.0000	0.0000	0.6667	0.0000	0.4000	0.0000	0.0000	0.8000
Accidental Fault 4	0.5556	0.2500	0.1667	0.7143	0.2000	0.0000	0.3333	0.9577
Accidental Fault 5	0.5556	0.5000	0.3333	0.5714	0.2000	0.0000	0.0000	0.9492
Accidental Fault 6	0.1111	0.0000	0.5000	0.1429	0.3000	0.0000	0.0000	0.7333
Accidental Fault 7	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	0.0000	1.0000
Accidental Fault 8	0.1111	0.2500	0.1667	0.4286	0.3333	0.0000	0.0000	0.7884
Accidental Fault 9	0.3333	0.5000	0.3333	1.0000	0.8333	0.5000	0.6667	1.0000
Accidental Fault 10	0.2222	0.5000	0.1667	0.7143	0.5000	0.2500	0.0000	0.9653

Table 5.3 Probabilities of killing Accidental Faults (using AFM) vs Random Testing for IVS10 (GEO program)

	AFM	*Random Testing*
Accidental Fault 1	0.7926	0.0428
Accidental Fault 2	0.6667	0.0429
Accidental Fault 3	0.8000	0.0572
Accidental Fault 4	0.9577	0.0699
Accidental Fault 5	0.9492	0.1122
Accidental Fault 6	0.7333	0.0428
Accidental Fault 7	1.0000	0.0565
Accidental Fault 8	0.7884	0.0584
Accidental Fault 9	1.0000	0.1422
Accidental Fault 10	0.9653	0.0695

[4] The probability column is highlighted using darker color or relatively less probability and lighter color for higher probability. The same is used for other tables in this chapter.

calculated as – mutant 1 is killed by test cases (4, 7, 8, 39, 43, 57, 58, 59, and 60) – accidental fault 1 is killed by test cases (7, 41 and 44) – the overlap is test case number 7 – 1 out 3 test case overlap with the test suite from mutant 1. Therefore, if mutant 1 is killed by the test suite then accidental fault 1 is likely to be killed by the same test suite. We title this technique as the, Accidental Fault Methodology, that is discussed in chapter 1. Another example using the same approach is mutant 7. If only mutant 7 was used and killed by the test suite, then accidental fault 1 will not be killed by the same test suite. There is no overlap of test cases killing mutant 7 and accidental fault 1 in table 5.1. If all 7 mutants are induced and mutants 1, 2, 4, 5 are killed with probability shown in table 5.2, then the probability of killing accidental fault 1 by the same test suite is 0.7926. Now if we use all 7 mutants for seeding faults in the sample program the probability of killing all accidental faults is shown in table 5.2.

The calculated probabilities for each accidental fault using the AFM approach is shown in the 'Probability' column in table 5.2. The column is formatted such that darker color represents lower probabilities. This column showing probability is formatted such that dark colored cells show lower probability and lighter or no color shows that the probability of killing that accidental fault is 1. A value of 1 for probability means that the test suite killing the accidental faults fully overlap with the test cases killing one or more mutants. It shows from table 5.2 that the probability of killing any accidental fault falls in the range of ***0.6667-1.0000***. We use the probabilities calculated using the AFM approach in table 5.2 to compare with the probability of killing any accidental fault if test cases are selected randomly from the same test suite. Table 5.3 shows the probability of killing accidental faults calculated using the AFM approach. It also shows the probability of killing any accidental fault if test cases are selected randomly from the test suite. This probability falls in the range of ***0.0428-0.1422***. The probability for random testing to kill the accidental faults is lower in comparison to the probabilities as per the AFM calculations.

Based on the results from table 5.2, 20% of the accidental faults (7 and 9) are definitely killed and 80% of the accidental faults (1, 2, 3, 4, 5, 6, 8, and 10) are highly likely be killed by the same test suite. 100% of the accidental faults are highly likely to be killed by the same test suite. There are multiple smaller test suites possible from the complete set of test cases that can kill all 7 mutants. Some examples of such minimal test suites are shown in table 5.4. Table 5.5 and 5.6 illustrate the results of IVS10 mutation operator applied to sample program BANK.

Table 5.4 Possible Minimal Test Suites for GEO program

	Minimal Test Suite 1							Minimal Test Suite 2							Minimal Test Suite 3						
	2	4	7	8	10	26	38	39	40	41	43	44	47	49	7	38	39	40	41	47	49
Mutant 1		X	X	X				X			X				X		X				
Mutant 2			X	X								X			X						
Mutant 3							X	X	X							X	X	X			
Mutant 4				X						X	X	X							X		
Mutant 5			X	X			X	X	X	X		X			X	X	X	X	X		
Mutant 6					X	X								X							X
Mutant 7	X												X							X	
Accidental Fault 1			X							X		X			X				X		
Accidental Fault 2					X	X							X							X	
Accidental Fault 3									X									X			
Accidental Fault 4			X					X			X				X		X				
Accidental Fault 5		X						X			X	X					X				
Accidental Fault 6								X									X				
Accidental Fault 7					X	X								X							X

Table 5.5 Results of IVS10 mutation operator for BANK program

	Test Cases																												
	6	7	8	9	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	4	4	5	5	5	5	6	6	6
Mutant 1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Mutant 2						X	X														X								
Mutant 3					X			X																					
Mutant 4														X	X	X													
Mutant 5	X	X	X	X																									
Accidental Fault 1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Accidental Fault 2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Accidental Fault 3	X												X	X	X	X	X	X	X	X			X	X	X	X	X		
Accidental Fault 4		X																				X		X				X	X
Accidental Fault 5									X				X	X			X	X											

Table 5.6 Probabilities of killing IVS10 Mutants and Accidental Faults for BANK program

	Mutant 1	*Mutant 2*	*Mutant 3*	*Mutant 4*	*Mutant 5*	*Probability*
Accidental Fault 1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Accidental Fault 2	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Accidental Fault 3	0.2331	0.0000	0.0000	0.4828	0.1207	0.6512
Accidental Fault 4	0.0297	0.0000	0.0000	0.0000	0.0431	0.0715
Accidental Fault 5	0.0297	0.0000	0.0000	0.0575	0.0000	0.0855

Table 5.7 Results of IVS11 mutation operator for GEO program

	Test Cases												
	12	*30*	*38*	*39*	*40*	*41*	*46*	*51*	*52*	*60*	*65*	*69*	*70*
Mutant 1	X					X	X			X		X	X
Mutant 2											X		
Mutant 3			X	X	X			X	X				X
Mutant 4						X						X	X
Mutant 5		X				X							
Mutant 6						X							
Mutant 7								X	X			X	
Mutant 8					X								
Accidental Fault 1						X							
Accidental Fault 2													
Accidental Fault 3					X			X	X			X	
Accidental Fault 4				X									
Accidental Fault 5				X							X	X	X
Accidental Fault 6				X					X			X	
Accidental Fault 7													
Accidental Fault 8											X		
Accidental Fault 9			X										
Accidental Fault 10												X	X

Operator IVS11 Results

If there is a pre-condition checking the invariant in a class, /*@invariant x > <expr>; */ Then insert an assignment statement in the class x = <expr>, after any assignment statement that references "x" in the code.

The results of the test suite run against the mutant programs for mutation operator IVS11 are shown in table 5.7. The row title lists the mutants and the accidental faults that are identified in the sample program. The mutants and the accidental faults for sample program GEO mentioned in the table can be mapped to the highlighted code in Appendix B. The column titles in table 5.7 represent the test case numbers. The complete set of test cases used in this experiment is shown in Appendix C. The test suite listed in table 5.7 is a subset of the complete test suite. This test suite was then run against the accidental faults. The new and smaller test suite, *MOTS*, is created from selecting test cases that have killed any of the 8 mutants. The "X" in the table cell denotes that the test case has killed the mutant and/or the accidental fault.

The subset of test cases is then run against the accidental faults to list the test cases that killed the mutants as well as the accidental faults. Some test cases only killed one mutant generated using the mutation operator IVS11, e.g. test cases number 38 killed only mutant 3 and test case number 39 also killed mutant 3. Other test cases that killed more than 1 mutant are test case number 40 killing mutant 3 and 8. Example of test cases killing both mutants and accidental faults is all test cases except test cases 12, 30, 46 and 60.

The probabilities to kill the accidental fault instances based on the test cases killing the mutants are shown in table 5.8. These probabilities are calculated based on results shown in table 5.8 and equation (1) explained in chapter 1. The overlapping of test case killing both the mutant and the accidental fault is used in this calculation. For example, the probability of killing accidental fault 1 given that mutant 1 is killed using the test suite listed in table 5.7 can be calculated as – mutant 1 is killed by test cases (12, 41, 46, 60, 69 and 70) – accidental fault 1 is killed by test cases (41) – the overlap is test case number 41 – 1 out 1 test case overlap with the test suite from mutant 1. Therefore, if mutant 1 is killed by the test suite then the probability of killing accidental fault 1 i.e. 100% using the AFM technique. If all 8 mutants are induced and mutants 1, 4, 5 and 6 are killed with probability shown in table 5.8, then the probability of killing accidental fault 1 by the same test suite is 1. Now if we use all 8 mutants for seeding faults in the

Table 5.8 Probabilities of killing IVS11 Mutants and Accidental Faults for GEO program

	Mutant 1	*Mutant 2*	*Mutant 3*	*Mutant 4*	*Mutant 5*	*Mutant 6*	*Mutant 7*	*Mutant 8*	*Probability*
Accidental Fault 1	0.1667	0.0000	0.1667	0.3333	0.5000	1.0000	0.0000	0.0000	1.0000
Accidental Fault 2	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Accidental Fault 3	0.1667	0.0000	0.5000	0.3333	0.0000	0.0000	1.0000	1.0000	1.0000
Accidental Fault 4	0.0000	0.0000	0.1667	0.0000	0.0000	0.0000	0.0000	0.0000	0.1667
Accidental Fault 5	0.3333	1.0000	0.3333	0.6667	0.0000	0.0000	0.3333	0.0000	1.0000
Accidental Fault 6	0.1667	0.0000	0.3333	0.3333	0.0000	0.0000	0.6667	0.0000	0.8765
Accidental Fault 7	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Accidental Fault 8	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000
Accidental Fault 9	0.0000	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000
Accidental Fault 10	0.3333	0.0000	0.1667	0.6667	0.0000	0.0000	0.3333	0.0000	0.8765

Table 5.9 Probabilities of killing Accidental Faults (using AFM) vs Random Testing for IVS11 (GEO program)

	AFM	*Random Testing*
Accidental Fault 1	1.0000	0.0142
Accidental Fault 2	0.0000	0.0147
Accidental Fault 3	1.0000	0.0557
Accidental Fault 4	0.1667	0.0138
Accidental Fault 5	1.0000	0.0441
Accidental Fault 6	0.8765	0.0419
Accidental Fault 7	0.0000	0.0143
Accidental Fault 8	1.0000	0.0139
Accidental Fault 9	1.0000	0.0151
Accidental Fault 10	0.8765	0.0140

sample program the probability of killing all accidental faults is shown in table 5.8.

The calculated probabilities for each accidental fault using the AFM approach is shown in the 'Probability' column in table 5.8. The column is formatted such that darker color represents lower probabilities. This column showing probability is formatted such that dark colored cells show lower probability and lighter or no color shows that the probability of killing that accidental fault is 1. A value of 1 for probability means that the test suite killing the accidental faults fully overlap with the test cases killing one or more mutants. It shows from table 5.8 that the probability of killing any accidental fault falls in the range of ***0.0000-1.0000***. We use the probabilities calculated using the AFM approach in table 5.8 to compare with the probability of killing any accidental fault if test cases are selected randomly from the same test suite. Table 5.9 shows the probability of killing accidental faults calculated using the AFM approach. It also shows the probability of killing any accidental fault if test cases are selected randomly from the test suite. This probability falls in the range of ***0.0138-0.0557***. The probability for random testing to kill the accidental faults is lower in comparison to the probabilities as per the AFM calculations.

Based on the results from table 5.8, 50% of the accidental faults (1, 3, 5, 8, and 9) are definitely killed and 20% of the accidental faults (6 and 10) are highly likely be killed by the same test suite. 70% of the accidental faults are highly likely to be killed by the same test suite. A probability of less than 0.5 for an accidental fault is considered as less likely killed by the same test suite. Table 5.10 through 5.11 illustrates the results of IVS11 mutation operator applied to sample program BANK.

Operator MIP1 Results

If there is an overloaded method, x(y) and x()
Then change the contents of x(y) to a method call for x().

The results of the test suite run against the mutant programs for mutation operator MIP1 are shown in table 5.12. The tables are formatted the same as for mutation operator IVS10 and IVS11. The mutants and the accidental faults can be mapped to the code listed in Appendix B. The complete set of test cases used is shown in Appendix C. The new and smaller test suite was

Table 5.10 Results of IVS11 mutation operator for BANK program

	Test Cases																
	6	*7*	*8*	*9*	*30*	*31*	*32*	*33*	*34*	*49*	*53*	*54*	*62*	*67*	*68*	*69*	*70*
Mutant 1					X	X	X	X	X	X		X					
Mutant 2		X										X		X		X	
Mutant 3	X	X	X	X						X	X	X	X	X	X	X	X
Mutant 4		X								X		X		X		X	
Mutant 5					X	X	X	X	X			X					
Accidental Fault 1		X								X		X		X		X	
Accidental Fault 2	X	X	X	X						X	X	X	X	X	X	X	X
Accidental Fault 3		X								X		X		X		X	
Accidental Fault 4					X	X	X	X	X			X					
Accidental Fault 5	X	X	X	X						X	X	X	X	X	X	X	X
Accidental Fault 6	X	X	X	X	X	X	X			X	X	X	X	X		X	
Accidental Fault 7	X	X	X	X	X	X	X			X	X	X	X	X		X	
Accidental Fault 8	X				X	X	X				X	X	X				
Accidental Fault 9		X								X		X		X		X	
Accidental Fault 10					X												

Table 5.11 Probabilities of killing IVS11 Mutants and Accidental Faults for BANK program

	Mutant 1	*Mutant 2*	*Mutant 3*	*Mutant 4*	*Mutant 5*	*Probability*
Accidental Fault 1	0.2857	1.0000	0.4167	1.0000	0.1667	1.0000
Accidental Fault 2	0.2857	1.0000	1.0000	1.0000	0.1667	1.0000
Accidental Fault 3	0.2857	1.0000	0.4167	1.0000	0.1667	1.0000
Accidental Fault 4	0.8571	0.2500	0.4167	0.2000	1.0000	1.0000
Accidental Fault 5	0.2857	1.0000	1.0000	1.0000	0.1667	1.0000
Accidental Fault 6	0.7143	1.0000	0.8333	1.0000	0.6667	1.0000
Accidental Fault 7	0.7143	1.0000	0.8333	1.0000	0.6667	1.0000
Accidental Fault 8	0.5714	0.2500	0.3333	0.2000	0.6667	0.9429
Accidental Fault 9	0.2857	1.0000	0.4167	1.0000	0.1667	1.0000
Accidental Fault 10	0.1429	0.0000	0.0000	0.0000	0.1667	0.2857

Table 5.12 Results of MIP1 mutation operator for GEO program

	Test Cases													
	6	7	17	18	19	20	30	40	41	44	51	52	61	69
Mutant 1	X			X									X	
Mutant 2	X			X		X								
Mutant 3	X		X											
Mutant 4								X			X	X		X
Mutant 5		X					X		X	X				
Accidental Fault 1	X		X		X								X	
Accidental Fault 2	X			X		X								
Accidental Fault 3								X				X		X
Accidental Fault 4									X					
Accidental Fault 5		X					X		X	X				

Table 5.13 Probabilities of killing MIP1 Mutants and Accidental Faults for GEO program

	Mutant 1	Mutant 2	Mutant 3	Mutant 4	Mutant 5	Probability
Accidental Fault 1	0.6667	0.3333	1.0000	0.0000	0.0000	1.0000
Accidental Fault 2	0.6667	1.0000	0.5000	0.0000	0.0000	1.0000
Accidental Fault 3	0.0000	0.0000	0.0000	0.7500	0.0000	0.7500
Accidental Fault 4	0.0000	0.0000	0.0000	0.0000	0.2500	0.2500
Accidental Fault 5	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000

Table 5.14 Probabilities of killing Accidental Faults (using AFM) vs Random Testing for MIP1 (GEO program)

	AFM	*Random Testing*
Accidental Fault 1	1.0000	0.0571
Accidental Fault 2	1.0000	0.0421
Accidental Fault 3	0.7500	0.0433
Accidental Fault 4	0.2500	0.0146
Accidental Fault 5	1.0000	0.0567

Table 5.15 Results of MIP1 mutation operator for BANK program

	Test Cases														
	4	*30*	*31*	*32*	*33*	*34*	*43*	*44*	*45*	*46*	*47*	*48*	*49*	*50*	*61*
Mutant 1								X	X	X	X				
Mutant 2							X		X			X		X	
Mutant 3												X	X		
Mutant 4	X														X
Mutant 5		X	X	X	X	X									
Accidental Fault 1							X		X			X		X	
Accidental Fault 2							X		X			X		X	
Accidental Fault 3												X	X		
Accidental Fault 4	X														X
Accidental Fault 5		X	X	X	X	X									

Table 5.16 Probabilities of killing MIP1 Mutants and Accidental Faults for BANK program

	Mutant 1	*Mutant 2*	*Mutant 3*	*Mutant 4*	*Mutant 5*	*Probability*
Accidental Fault 1	0.2500	1.0000	0.5000	0.0000	0.0000	1.0000
Accidental Fault 2	0.2500	1.0000	1.0000	0.0000	0.0000	1.0000
Accidental Fault 3	0.0000	0.2500	1.0000	0.0000	0.0000	1.0000
Accidental Fault 4	0.0000	0.0000	0.0000	1.0000	0.0000	1.0000
Accidental Fault 5	0.0000	0.0000	0.0000	0.0000	1.0000	1.0000

created from selecting test cases that killed any of the 5 mutants generated by this mutation operator.

The calculated probabilities for each accidental fault using the AFM approach is shown in the ‘Probability’ column in table 5.13. This column showing probability is formatted such that dark colored cells show lower probability and lighter or no color shows that the probability of killing that accidental fault is 1. A value of 1 for probability means that the test suite killing the accidental faults fully overlap with the test cases killing one or more mutants. We use the probabilities calculated using the AFM approach in table 5.13 to compare with the probability of killing any accidental fault if test cases are selected randomly from the same test suite. It shows from table 5.13 that the probability of killing any accidental fault falls in the range of ***0.2500-1.0000***. Table 5.14 shows the probability of killing accidental faults calculated using the AFM approach. It also shows the probability of killing any accidental fault if test cases are selected randomly from the same test suite. This probability falls in the range of ***0.0146-0.0571***. The probability for random testing to kill the accidental faults is lower in comparison to the probabilities as per the AFM calculations.

Based on the results, 60% of the accidental faults are definitely killed and 80% of the accidental faults (all except 4) are highly likely to be killed by the same test suite. Table 5.15 through 5.16 illustrates the results of MIP1 mutation operator applied to sample program BANK.

Operator MIP3 Results

If there an expression using the attribute, such as x = 2*this. Radius

Then change the reference to another compatible attribute, such as y = 2*this. Height

The results of the test suite run against the mutant programs for mutation operator MIP1 are shown in table 5.17. The tables are formatted the same as for mutation operator IVS10 and IVS11. The mutants and the accidental faults can be mapped to the code listed in Appendix B. The complete set of test cases used is shown in Appendix C. The new and smaller test suite was created from selecting test cases that killed any of the 5 mutants generated by this mutation operator.

The calculated probabilities for each accidental fault using the AFM approach is shown in the ‘Probability’ column in table 5.18. This column showing probability is formatted such that dark colored cells show lower probability and lighter or no color shows that the probability of

Table 5.17 Results of MIP3 mutation operator for GEO program

	Test Cases											
	6	*7*	*8*	*18*	*30*	*40*	*41*	*44*	*51*	*52*	*61*	*69*
Mutant 1	X			X							X	
Mutant 2					X	X			X	X		X
Mutant 3		X	X		X	X	X	X		X		X
Mutant 4						X						X
Mutant 5					X	X			X	X		X
Accidental Fault 1	X										X	
Accidental Fault 2	X			X								
Accidental Fault 3						X				X		X
Accidental Fault 4			X				X					
Accidental Fault 5		X			X		X	X				

Table 5.18 Probabilities of killing MIP3 Mutants and Accidental Faults for GEO program

	Mutant 1	*Mutant 2*	*Mutant 3*	*Mutant 4*	*Mutant 5*	*Probability*
Accidental Fault 1	0.6667	0.0000	0.0000	0.0000	0.0000	0.6667
Accidental Fault 2	0.6667	0.0000	0.0000	0.0000	0.0000	0.6667
Accidental Fault 3	0.0000	0.6000	0.3750	1.0000	0.6000	1.0000
Accidental Fault 4	0.0000	0.0000	0.2500	0.0000	0.0000	0.2500
Accidental Fault 5	0.0000	0.2000	0.5000	0.0000	0.2000	0.6800

Table 5.19 Probabilities of killing Accidental Faults (using AFM) vs Random Testing for MIP3 (GEO program)

	AFM	*Random Testing*
Accidental Fault 1	0.6667	0.0275
Accidental Fault 2	0.6667	0.0287
Accidental Fault 3	1.0000	0.0418
Accidental Fault 4	0.2500	0.0287
Accidental Fault 5	0.6800	0.0563

Table 5.20 Results of MIP3 mutation operator for BANK program

	Test Cases														
	4	7	28	31	33	37	39	49	52	54	55	56	61	67	69
Mutant 1				X			X								
Mutant 2			X		X	X									
Mutant 3		X						X		X				X	X
Mutant 4	X												X		
Mutant 5									X		X	X			
Accidental Fault 1			X		X	X									
Accidental Fault 2			X		X	X									
Accidental Fault 3		X						X		X				X	X
Accidental Fault 4	X												X		
Accidental Fault 5			X		X	X									

Table 5.21 Probabilities of killing MIP3 Mutants and Accidental Faults for BANK program

	Mutant 1	*Mutant 2*	*Mutant 3*	*Mutant 4*	*Mutant 5*	*Probability*
Accidental Fault 1	0.0000	1.0000	0.0000	0.0000	0.0000	1.0000
Accidental Fault 2	0.0000	1.0000	0.0000	0.0000	0.0000	1.0000
Accidental Fault 3	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000
Accidental Fault 4	0.0000	0.0000	0.0000	1.0000	0.0000	1.0000
Accidental Fault 5	0.0000	1.0000	0.0000	0.0000	0.0000	1.0000

killing that accidental fault is 1. It shows from table 5.19 that the probability of killing any accidental fault falls in the range of ***0.2500-1.0000***. Table 5.19 shows the probability of killing accidental faults calculated using the AFM approach. It also shows the probability of killing any accidental fault if test cases are selected randomly from the test suite. This probability falls in the range of ***0.0148-0.0563.*** The probability for random testing to kill the accidental faults is lower in comparison to the probabilities as per the AFM calculations.

Based on the results 20% of the accidental faults will definitely be killed and 80% of the accidental faults are highly likely to be killed by the same test suite. An observation from table 5.17 shows that test case 51 killed some mutants but did not kill any of the accidental faults. Table 5.20 through 5.21 illustrates the results of MIP3 mutation operator applied to sample program BANK.

The fault type MIP was easier to detect in the BANK program and the other fault type IVS behaved similarly for both sample programs i.e. more than 70% of the accidental fault instances are highly likely killed by the same test suite as the mutants. The improved specification for MIP proved to be efficient in killing the mutants and accidental faults of that fault type.

All the mutants for both fault types are killed by the test suite; more than 70% of accidental fault instances are killed by the same test suite. Thus, we can say that the mutation operators are effective in creating mutants that are killed by the test suite and are a good representation of faults that can be present in a program for that fault type. This analysis shows how MOPRS can be used to induce mutants and how AFM can be used to enhance the evaluation techniques for software testing methodologies. These results can show if a software testing methodology is effective in killing a larger set of faults.

Chapter 6 - Conclusion and Future Work

The results show that there is a need for new and improved mutation operators. The new operators create faults in the programs that cannot be generated using the existing mutation operators in the literature. Fault types addressed are focused on object-oriented programming concepts and concentrate on issues related to inheritance, polymorphism, method overloading, etc.

The new system, Mutation Operator Production Rule System (MOPRS), proposed in the thesis generates mutants in the sample program. This system uses a set of precise production rules to generate mutants in object-oriented programs. The precise production rules are very effective in creating the specifications to generate mutants in a program. Using this precision, it becomes easier to state the specific fault type being killed by a test suite. The production rules are converted to a specification using the newly defined grammar. The specification for a mutation operator generate one or more mutants in a program.

Furthermore an evaluation technique, Accidental Fault Methodology (AFM), is defined to evaluate a software testing methodology. This technique shows if a test suite generated using a software testing methodology can kill the mutants generated using mutation operators as well as the accidental faults in a program. Using this technique we can analyze the probability of a test suite killing the mutants and the accidental faults. Thus, for a given software testing methodology one can calculate the probability of killing the mutants and the accidental faults and conclude if the methodology in question can kill a larger set of faults in the program and compare different software testing methodologies.

The results for all four new and improved mutation operators defined in the thesis are consistent. The probability of killing faults generated using MOPRS can be calculated using the AFM technique. It shows from the results that new mutation operators are effective in imitating fault types. For all fault types, the probability of killing both mutants and accidental faults fall in the range of 60% to 100%. These results are compared to the random testing methodology. The probabilities of killing mutants and accidental faults using this methodology as lower than the expected values from the new technique.

The future work will involve creating new mutation operators for the object-oriented faults that are not yet addressed in this thesis. The list shown in Appendix A needs to be analyzed and more mutation operators need to be added to the existing set of operators. Upcoming work will also focus on existing mutation operators not addressed in this thesis. There is a need for analysis in detail so the mutation operator definition and specification can be improved and is precise. The precision of the definition will lead to precision in the production rules and initiate the process of killing specific fault types.

With increase in number of mutation operators, the grammar for the production rules also need to be expanded. As the mutation operators are added to the existing set, more grammar keywords and operators may be needed to create the specification to generate mutants.

Along with defining new mutation operators for fault types, future work will focus on fault types that show anomalous behavior. During the analysis of results two anomalies' were encountered. Firstly, the test suite killing some accidental faults but not killing any mutants. Secondly, a mutant is killed by the test suite but no accidental faults are killed by the same test suite. Such instances may require additional experimentation and new mutation operators.

There is a possibility that a mutation operator when defined may not be effective or cannot be defined precisely. In this scenario, the hypothesis would not suffice and new technique may need to be researched to be able to induce such type of faults in a program that cannot be induced using the new mutation operators. The production rule for effective mutation operators is the key notation in the process of mutant generation. If the mutation operator cannot be defined precisely, the production rule cannot be stated and furthermore the specification cannot be generated for the mutation operator.

Each fault type can have multiple mutation operators. When designing mutation operators for a fault type, there is possibility of running into multiple and infinite set of mutation operators. Each mutation operator has multiple instances in a program called mutants. Some work is required to define the finiteness or sufficiency of mutation operators and mutants that can be induced and generated in a program, respectively.

Bibliography

[AC79] Acree, A. T., Budd, T. A., DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1979). *Mutation Analysis.* Technical Report GIT-ICS-79/08, Georgia Institute of Technology, School of Information and Computer Science, Atlanta, GA.

[AC80] Acree, A. T. (1980). *On Mutation.* PhD Thesis, Georgia Institute of Technology.

[AG89] Agrawal, H., Spafford, E., DeMillo, R. A., Hathaway, B., Hsu, W., Krauser, E. W., Martin, R. J. (1989). *Design of Mutant Operators for the C Programming Language.* SERC-TR41-P, Purdue University, West Lafayette.

[AL02] Alexander, R., Offutt, J. A., & Bieman, J. (2002). Syntactic fault patterns in Object-oriented programs. *Proceedings of The Eighth IEEE International Conference on Engineering of Complex Computer Systems*, (pp. 193-202). Greenbelt, MD.

[AP88] Appelbe, W. F., DeMillo, R., Guindi, D. S., King, K. N., & McCracken, W. M. (1988). Using mutation analysis for testing Ada programs. *Proceedings of the Ada-Europe International Conference*, (p. 65). Munich.

[AP89] Appelbe, W. F., DeMillo, R. A., Guindi, D. S., King, K. N., & McCracken, W. M. (1989). *Using Mutation Analysis for testing ADA programs.* Technical Report SERC-TR-9-P, Purdue University, West Lafayette, IN.

[BE12] Benli, S., Habash, A., Herrmann, A., Loftis, T., Simmonds, D. (2012). A Comparative Evaluation of Unit Testing Techniques on a Mobile Platform, *Information Technology: New Generations (ITNG), 2012 Ninth International Conference*, pp.263-268.

[BI96] Binder, R. V. (1996). Testing Object-oriented Software: A Survey. *Journal of Software Testing, Verification and Reliability, 6*, 125-252.

[BI99] Binder, R. V. (1999). *Testing Object-oriented Systems.* Addison Wesley.

[BU77] Budd, T. A., & Sayward, F. G. (1977). *Users Guide to the Pilot Mutation System.* Technical Report 114, Yale University, Department of Computer Science.

[BU78] Budd, T. A., DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). The design of a prototype mutation system for program testing. *In Proceedings NCC, AFIPS Conference Record*, (pp. 623-627).

[BU80] Budd, T. A. (1980). *Mutation analysis of program test data.* Yale University. New Haven, CT: PhD Thesis.

[CH01] Chevalley, P. (2001). Applying Mutation Analysis for Object-oriented Programs Using a Reflective Approach. *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference* (pp. 267-270). Washington DC: IEEE Computer Society.

[DE78] DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978, April). Hints on test data selection: help for the practicing programmer. *Computer, 11*(4).

[DE88] DeMillo, R. A., Guindi, D. S., King, K. N., McCracken, W. M., & Offutt, A. J. (1988). An Extended Overview of the Mothra Software Testing Environment. *Proceedings of Second Workshop Software Testing, Verification, and Analysis*, (pp. 142-151).

[DE96] Delamaro, M. E., & Maldonado, J. C. (1996). Proteum—A Tool for the Assessment of Test Adequacy for C Programs. *Proceedings of Conference Performability in Computing Systems*, (pp. 79-95).

[DE00] Delamaro, M. E., Maldonado, J. C., & Vincenzi, A. (2000). Proteum/IM 2.0: An Integrated Mutation Testing Environment. *Proceedings of First Workshop Mutation Analysis.*

[DE06] Derezińska, A. (2006). Advanced mutation operators applicable in C# programs. *Software Engineering Techniques: Design for Quality*, 283-288.

[FA11] Farooq, S. U. & Quadri, S. M. K. (2011). Evaluating Effectiveness of Software Testing Techniques with Emphasis on Enhancing Software Reliability. *Journal of Emerging Trends in Computing and Information Sciences*, *2*(12), 740-745.

[FA13] Farooq, S. U. & Quadri, S. M. K. & Ahmad N. (2013). A controlled experiment to evaluate effectiveness and efficiency of three software testing methods. *IEEE Conference on Software Testing, Verification and Validation*.

[FI93] Firesmith, D. G. (1993). Testing Object-oriented Software. *Proceedings of the Eleventh International Conference on Technology of Object-oriented Languages and Systems* (pp. 407-426). Englewood Cliffs, NJ: Prentice Hall.

[GI85] Girgis, M. R., & Woodward, M. R. (1985). An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis. *Proceedings of Eighth International Conference of Software Engineering*, (pp. 313-319).

[GR05] Grottke, M., & Trivedi, K. S. (2005). A classification of software faults. *Journal of Reliability Engineering Association of Japan*, *27*(7), 425-438.

[GU10] Gupta, P., & Gustafson, D. A. (2010). Object-oriented Testing beyond Statement Coverage. *Proceedings of International Conference on Computer Applications in Industry and Engineering*, (pp. 150-155).

[HA80] Hanks, J. M. (1980). *Testing Cobol Programs by Mutation.* PhD Thesis, Georgia Institute of Technology.

[HA94] Hayes, H. J. (1994). Testing of Object-oriented Programming Systems (OOPS): A Fault-Based Approach. *Proceedings of Object-oriented Methodologies and Systems* (pp. 205-220). Berlin: Springer.

[HA96] Hatton, L. (1996). Software Faults: The Avoidable and the Unavoidable: Lessons from Real Systems. *Proceedings of the Product Assurance Workshop.* Noordwijk, The Netherlands.

[HA09] Hamill, M., & Goseva-Popstojanova, K. (2009, July). Common Trends in Software Fault and Failure Data. *IEEE Transactions in Software Engineering, 35*(4), 484-496.

[HO82] Howden, W. E. (1982, July). Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions of Software Engineering, 8*(4), 371-379.

[JI08] Jia, Y., & Harman, M. (2008). MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. *Proceedings of Third Testing: Academic and Industrial Conference Practice and Research Techniques*, (pp. 94-98).

[JI11] Jia, Y., & Harman, M. (2011). An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions of Software Engineering, 37*(5), 649-678.

[KI00] Kim, S., Clark, J., & McDermid, J. (2000). Class mutation: Mutation testing for Object-oriented programs. *Proceedings of Net Objectdays Conference on Object-oriented Software Systems.*

[LI78] Lipton, R. J., & Sayward, F. G. (1978). The status of research on program mutation. 355-373.

[LI05] Lin, J. C., Huang, Y. L., & Liu, C. H. (2005). Testability Analysis for Polymorphism. *Proceedings of the Second IASTED International Multi-Conference on Automation, Control, and Information Technology* (pp. 98-103). Novosibirsk, Russia: IASTED/ACTA Press.

[MA02] Ma, Y. S., Kwon, Y. R., & Offutt, J. A. (2002). Inter-class mutation operators for Java. *Proceedings of the 13th International Symposium on Software Reliability Engineering* (pp. 352-363). Annapolis, MD: IEEE Computer Society Press.

[MA06] Ma, Y. S., Harrold, M. J., & Kwon, Y. R. (2006). Evaluation of Mutation Testing for Object-oriented programs. *Proceedings of the 28th international conference on Software engineering*, (pp. 869-872). New York, NY.

[MA10] Madeyski, L., & Radyk, N. (2010). Judy - a mutation testing tool for Java. *IET Software, 4*, 32-42.

[OF96] Offutt, J. A., Lee, A., Rothermal, G., Untch, R. H., & Zapf, C. (1996, April). An experimental determination of sufficient mutant operators. *ACM Transactions of Software Engineering Methodology, 5*(2), 99-118.

[OF01] Offutt, J. A., & Untch, R. H. (2001). Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century* (pp. 34-44). Norwell, MA, Norwell, MA: Kluwer Academic Publishers. Retrieved from http://dl.acm.org/citation.cfm?id=571305.571314

[OF04] Offutt, A. J., Ma, Y. S., & Kwon, Y. R. (2004, September). An Experimental Mutation System for Java. *ACM SIGSOFT Software Engineering Notes, 29*(5), 1-4.

[OA01] Offutt, J. A., Alexander, R., Wu, Y., Xiao, Q., & Hutchinson, C. (2001). A fault model for subtype Inheritance and Polymorphism. *Proceedings of the 12th International Symposium on Software Reliability Engineering* (pp. 84-93). Hong Kong, China: IEEE Computer Society Press.

[PU91] Purchase, A. J., & Winder, R. L. (1991, June). Debugging Tools for Object-oriented Programming. *Journal of Object-oriented Programming, 4*(3), 10-27.

[SE12] Selby, R.W. and Basili, V.R. (1984). Evaluating Software Engineering Testing Strategies. *Proceedings of the 9th Annual Software Engineering Workshop*, pp.42—53. NASA/GSFC, Greenbelt, MD.

[TA81] Tanaka, A. (1981). *Equivalence Testing for Fortran Mutation System Using Data Flow Analysis.* MS Thesis, Georgia Institute of Technology.

[TY14] Tyagi, M. & Malhotra, S. (2014). A Review of Empirical Evaluation of Software Testing Techniques with Subjects. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, *3*(2), 519-523.

[VO12] Vos, T. E. J., Marin, B., Escalona, M. J., Marchetto, A. (2012). A Methodological Framework for Evaluating Software Testing Techniques and Tools. *Quality Software (QSIC), 2012 12th International Conference*, pp.230-239.

[WA05] Walkinshaw, N., Roper, M., & Wood, M., (2005). Collecting and Categorizing Faults in Object-oriented Code. *Sheffield: UKTest*.

[WO88] Woodward, M. R., & Halewood, K. (1988). From weak to strong, dead or alive? An analysis of some mutation testing issues. *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis IEEE*, (pp. 152-158).

[WO90] Woodward, M. R. (1990). Mutation Testing - an Evolving Technique. *Proceedings of IEE Colloquium on Software Testing for Critical Systems*, (pp. 3/1-3/6).

[WK90] Wohlin, C., & Körner, U. (1990). Software Faults: Spreading, Detection and Costs. *Software Engineering Journal, 5*(1), 33-42.

[YO88] Yoon, H., Choi, B., & Jeon, J. O. (1988). Mutation-Based Inter-Class Testing. *Proceedings of Fifth Asia Pacific Software Engineering Conference*, (pp. 174-181).

Appendix A - Fault Types vs. Mutation Operators

The faults collected from the literature corresponding to each mutation operator are shown below [FI93, MA2002 & OA01].

Classes

- Incorrect initialization or missing re-initialization: *JDC (Java-supported default constructor create)*

Attributes

- Incorrect values of attributes: *ISK (super keyword deletion)*
- Not initialized: *JID (Member variable initialization) deletion)*
- Incorrect visibility, access modifier misuse: *AMC (Access modifier change)*

Methods

- Incorrect method used: EAM (Accessor method change), EMM (Modifier method change)
- Parameter mismatch: OAO (Argument order change)

Inheritance

- Deletion of resource in subclass violates abstraction of super class: *IHD (Hiding variable deletion), IHI (Hiding variable insertion)*
- Incompatibility between subclass resources and existing super class resources: *IOP (Overridden method calling position change)*
- Incorrect initialization (super not used for initialization): *IPC (Explicit call of a parent's constructor deletion)*
- Inheritance allows super class features with the same names, providing the opportunity for misnaming. Changes to either super class will affect the subclass: *IOR (Overridden method rename)*
- Original resource in super class not overridden in subclass: *IOD (Overriding method deletion)*
- *super* keyword misuse: *ISK (super keyword deletion)*

Polymorphism

- Failure to check the compatibility of the actual parameters: *PPD (Parameter variable declaration with child class type)*
- Failures associated with instantiation: *PNC (new method call with child class type), PMD (new method call with child class type)*

Method Overloading

- Misusing a dynamically bound method: *OMD (Overloading method deletion)*
- Message not received by correct method or message received by incorrect method: *OMD (Overloading method deletion), OAO (Argument order change), OAN (Argument number change)*
- Method incorrectly performed: *OMR (Overloading method contents change)*

Inheritance manifested by Polymorphism [OA01]

- State visibility anomaly: *SVA*
- State definition inconsistency due to state variable hiding: *SDIH*
- State definition anomaly: *SDA (possible post-condition violation)*
- State defined incorrectly: *SDI*
- Inconsistent type use: *ITU (context swapping)*
- Incomplete construction: *IC*

- Indirect inconsistent state definition: *IISD*
- Anomalous construction behavior: *ACB*

Programming language specific features [MA2002]

- *this* keyword misuse: JTD
- *Static* modifier misuse: JSC

The faults that have not been addressed by a mutation operator are listed below [FI93].

Classes

- Invariants violated or missing

Attributes

- Invariant relationships between attributes violated
- Encapsulated in wrong object
- Failure of pre-conditions and post-conditions of attributes
- Incorrect value of attributes passed through parameters
- Incorrect unit of measure

Methods

- Proper value not returned or improper value returned by method
- Abstraction violated by the performance of the associated method
- Pre-condition, post-condition, or invariants violated
- Method incorrectly performed

Inheritance

- Unexpected changes in subclass due to changes in super class
- Classes improperly instantiated
- Wrong resource inherited, confusion resulting from multiple Inheritance, Polymorphism, or dynamic binding
- Incorrect parameters
- Incompatibility between subclass and super class resources

Polymorphism

- Polymorphic substitution of subclasses for super classes not supported

Method Overloading

- Values of attributes unnecessarily passed through a method

Appendix B - Code Examples of Seeding Mutants Using New Mutation Operators[5]

This Appendix includes code examples of generating mutants using mutation operator IVS10 and IVS11 for GEO including the accidental faults.

1) GeometricObjects.cpp

```
class GeometricObjects{ private float area;
        public float getArea() { return area;}
        public void setArea(float theArea) { area = theArea;}
        private float perimeter;
        public float getPerimeter() { return perimeter;}
        public void setPerimeter(float thePerimeter) { perimeter = thePerimeter;}
}
```

2) Point.cpp

```
class Point extends GeometricObjects { private float x;
        public float getX() { return x;}
        public void setX(float theX) { x = theX;}
        private float y;
        public float getY() { return y;}
        public void setY(float theY) { y = theY;}
        public Point()  { this.setX(0); this.setY(0);}
        public Point(float theX, float theY) { this.setX(theX); this.setY(theY);}
        public float findArea(float theX, float theY)  { return 0;}
        public float findArea() { return 0;}
        public float findPerimeter(float theX, float theY) { return 0;}
        public float findPerimeter() { return 0;}
```

3) Line.cpp

```
class Line extends GeometricObjects { public static float length;
        private float x1, y1, x2, y2;
        public float getX1()  { return x1;}
        public void setX1(float theX1) { x1 = theX1;}
        public float getY1()  { return y1;}
        public void setY1(float theY1) { y1 = theY1;}
        public float getX2() { return x2;}
        public void setX2(float theX2) { x2 = theX2;}
```

[5] The highlighted text in yellow shows the mutants and the accidental faults used in the sample programs

```
public float getY2() { return y2;}
public void setY2(float theY2) { y2 = theY2;}
public float getLength() { return length;}
public void setLength(float theLength) { length = theLength;}
public Line() { x1=0; y1=0; x2=1; y2=1; length = this.findLength(x1,y1,x2,y2);}
public Line(Point p1, Point p2)  { Line l1 = new Line();
        l1.setX1(p1.getX());
        l1.setY1(p1.getY());
        l1.setX2(p2.getX());
        l1.setY2(p2.getY());
        this.setLength(this.findLength(p1, p2));
        length = 0; //IVS11 seeded mutant 7, Invariant: length > 0 for line }
public Line(float theX1, float theY1, float theX2, float theY2) { x1 = theX1;
        y1 = theY1;
        x2 = theX2;
        y2 = theY2;
        this.setLength(this.findLength(theX1, theY1, theX2, theY2));
        length = 0; //IVS11 seeded mutant 8, Invariant: length > 0 for line }
public float findLength(Point p1, Point p2)  { x1 = p1.getX();
        y1 = p1.getY();
        x2 = p2.getX();
        y2 = p2.getY();
        float xdis = (x1 - x2) * (x1 - x2);
        float ydis = (y1 - y2) * (y1 - y2);
        float x= (float) (Math.sqrt(xdis + ydis));
        return x;}
public float findLength(Point p1) {       x1 = p1.getX();
        y1 = p1.getY();
        float xdis = (x1) * (x1);
        float ydis = (y1) * (y1);
        float x = (float) (Math.sqrt(xdis + ydis));
        return x;  }
public float findLength(float theX1, float theY1, float theX2, float theY2) { x1 = theX1;
        y1 = theY1;
        x2 = theX2;
        y2 = theY2;
        float xdis = (x1 - x2) * (x1 - x2);
        float ydis = (y1 - y2) * (y1 - y2);
        float x = ((float) (Math.sqrt(xdis + ydis)));
        return x;  }
public float findLength() { float xdis = (this.x1 - this.x2) * (this.x1 - this.x2);
        float ydis = (this.y1 - this.y2) * (this.y1 - this.y2);
```

```
                float x = (float) (Math.sqrt(xdis + ydis));
                return x;  }
        public float findArea(Point p1, Point p2) { return 0;}
        public float findPerimeter(Point p1, Point p2)  { float x = this.findLength(p1, p2); return x;}
        public float findPerimeter(Point p1)  { float x = this.findLength(p1); return x;}
        public float findPerimeter() {float x = this.findLength(); return x;}
}
```

4) Parallelogram.cpp

```
class Parallelogram extends GeometricObjects { protected float rWidth;
        public float getRWidth() { return rWidth;}
        public void setRWidth(float theRWidth) {  rWidth = theRWidth;
                rLength = rWidth; //IVS11 seeded mutant 1, Invariant: length > width for parallelogram }
        protected float rLength;
        public float getRLength() { return rLength;}
        public void setRLength(float theRLength) { rLength = theRLength;}
        public Integer sides = 4;
        public int getSides() { return sides;}
        public void setSides(int theSides) {      sides = theSides;
                sides = theSides -1; //IVS10 seeded mutant 5, Invariant: sides = 4 for parallelogram }
        public Parallelogram() { sides = 4;
                sides = 4 -1; //IVS10 seeded mutant 1, Invariant: sides = 4 for parallelogram
                rLength = 0; rWidth = 0;}
        public Parallelogram(float theRWidth, float theRLength) { sides = 4;
                sides = 4 -1; //IVS10 seeded mutant 2, Invariant: sides = 4 for parallelogram
                rLength = theRLength;
                rLength = rWidth; //IVS11 seeded mutant 2, Invariant: length > width for parallelogram
                rWidth = theRWidth; //Changed to next line
                rWidth = theRWidth + 100; //IVS accidental fault 8 }
        public Parallelogram(Line l1, Line l2) { sides = 4;
                sides = 4 -1; //IVS10 seeded mutant 3, Invariant: sides = 4 for parallelogram
                rWidth = l1.findLength(); //Changed to next line
                rWidth = l2.findLength()+100; //IVS accidental fault 9
                rLength = l2.findLength();
                rLength = rWidth; //IVS11 seeded mutant 3, Invariant: length > width for parallelogram }
        public Parallelogram(Point p1, Point p2, Point p3, Point p4)  { sides = 4;
                sides = 4 -1; //IVS10 seeded mutant 4, Invariant: sides = 4 for parallelogram
                Line L1 = new Line(p1, p2);
                Line L2 = new Line(p2, p3);
                rWidth =  L1.findLength(); //Changed to next line
                rWidth = (l2.findLength())+100; //IVS accidental fault 10
                rLength = L2.findLength();
```

```
          rLength = rWidth; //IVS11 seeded mutant 4, Invariant: length > width for parallelogram }
     public float findArea(Point p1, Point p2, Point p3, Point p4)  { Line temp1 = new Line(p1, p2);
          Line temp2 = new Line(p2, p3);
          float theRWidth = temp1.findLength();
          float theRLength = temp2.findLength();
          theRLength = theRWidth; //IVS11 seeded mutant 5, Invariant: length > width for parallelogram
          this.sides = 3; //IVS accidental fault 1
          if (this.sides == 4) { float x = theRWidth * theRLength; return x; } else { return 0; }}
     public float findArea(Line l1, Line l2) { float theRWidth = l1.getLength();
          float theRLength = l2.getLength();
          sides = 3; //IVS accidental fault 3
          if (this.sides == 4) { float x = theRWidth * theRLength; return x; } else { return 0; }}
     public float findArea(float theRWidth, float theRLength) { sides = 3; //IVS accidental fault 4
          if (this.sides == 4) { float x = theRWidth * theRLength; return x; } else { return 0; }}
     public float findArea() { sides = 10; //IVS accidental fault 5
          if (this.sides == 4) { float x = this.getRWidth() * this.getRLength(); return x;} else { return 0; }}
     public float findPerimeter(Line l1, Line l2) {
          if (this.sides == 4) { float x = 2 * (l1.getLength() + l2.getLength()); return x; } else { return 0; }}
     public float findPerimeter(Point p1, Point p2, Point p3, Point p4) { Line temp1 = new Line(p1, p2);
          Line temp2 = new Line(p2, p3); rWidth =  temp1.findLength(); rLength = temp2.findLength();
          rLength = rWidth; //IVS11 seeded mutant 6, Invariant: length > width for parallelogram
          if (this.sides == 4) { float x = 2 * (this.getRWidth() + this.getRLength()); return x;}
          else { return 0; }     }
     public float findPerimeter() { sides = 1; //IVS accidental fault 6
          if (this.sides == 4) { float x = 2 * (this.getRWidth() + this.getRLength()); return x; }
          else  { return 0; }     }
     public float findPerimeter(float theRWidth, float theRLength) {
          if (this.sides == 4) { float x = 2 * (theRWidth + theRLength); return x;} else { return 0; }}}
```

5) Circle.cpp

```
class Circle extends GeometricObjects { protected float radius;
     public float getRadius()  {      return radius;}
     public void setRadius(float theRadius) { radius = theRadius;}
     private Point center;
     public Point getCenter() { return center;}
     public void setCenter(Point theCenter) { center = theCenter;}
     public Circle()  {     this.setRadius(0); this.setCenter(new Point(0,0));}
     public Circle(float theRadius)  { radius = theRadius; this.setCenter(new Point(0,0)); }
     public float findArea(float theRadius) { this.setRadius(theRadius);
          float x = (float) (3.14 * this.getRadius() * this.getRadius()); return x; }
     public float findArea(int theRadius) {
          float x = (float) (3.14 * this.getRadius() * this.getRadius()); return x; }
```

```
        public float findPerimeter(float theRadius) throws Exception { this.setRadius(theRadius);
                float x = (float) (2 * 3.14 * this.getRadius()); return x; }
}
```

6) Rectangle.cpp

```
class Rectangle extends Parallelogram { public Rectangle() { super();}
        public Rectangle(float theRWidth, float theRLength) { super(theRWidth, theRLength);}
        public Rectangle(Line l1, Line l2) { super(l1, l2);}
        public Rectangle(Point p1, Point p2, Point p3, Point p4) { super(p1, p2, p3, p4);}
}
```

7) Cylinder.cpp

```
class Cylinder extends Circle { private float volume;
        public float getVolume() { return volume;}
        public void setVolume(float theVolume) { volume = theVolume;}
        private float height;
        public float getHeight() { return height;}
        public void setHeight(float theHeight) { height = theHeight;}
        public Cylinder()  { super(); height = 1;
                volume = (float) (3.14 * super.radius * super.radius * this.getHeight()); }
        public Cylinder(float theRadius, float theHeight)  { super(theRadius); this.height = theHeight;
                volume = (float) (3.14 * super.radius * super.radius * this.getHeight());}
        public Cylinder(float theRadius, float theHeight, float theVolume) { super(theRadius);
                this.height = theHeight; this.volume = theVolume;}
        public float findVolume(float theRadius, float theHeight) { this.setRadius(theRadius);
                this.setHeight(theHeight);
                float x = ((float) (2 * 3.14 * this.getRadius() * this.getRadius() * this.getHeight())); return x;}
        public float findVolume() {
                float x = (float) (2 * 3.14 * this.getRadius() * this.getRadius() * this.getHeight()); return x;}
        public float findPerimeter(float theRadius) { this.setRadius(theRadius);
                float x = (float) (4 * 3.14 * this.radius); return x;}
        public float findPerimeter() { float x = (float) (4 * 3.14 * this.getRadius()); return x;}
        public float findArea(float theRadius, float theHeight) { this.setRadius(theRadius);
                this.setHeight(theHeight);
                float x = (float) (2 * 3.14 * this.getRadius() * this.getRadius() * this.getHeight()); return x;}
        public float findArea() { float x = (float) (2 * 3.14 * this.getRadius() * this.getRadius() * this.getHeight());
                return x;}
}
```

8) Square.cpp

```
class Square extends Rectangle {
        public Square() {    super();}
        public Square(float theRLength) { super.setRLength(theRLength);}
        public Square(Point p1, Point p2, Point p3, Point p4) { super(p1, p2, p3, p4);}
        public float findPerimeter(Point p1, Point p2, Point p3, Point p4)  { Line temp1 = new Line(p1, p2);
                Line temp2 = new Line(p2, p3);
                float theRWidth = temp1.findLength() - 1; //IVS accidental fault 7
                float theRLength = temp2.findLength(p2, p3);
                theRLength = temp2.findLength() - 2; //IVS10 seeded mutant 6, Invariant: length = width for square
                if (theRWidth == theRLength) { float x = 4 * (theRWidth); return x;} else  { return 0;}}
        public float findPerimeter(float theRLength) { float x =  4 * theRLength; return x;}
        public float findPerimeter() { float x = this.findPerimeter(this.getRLength()); return x;}
        public float findArea(float theRLength) { this.setRLength(theRLength);
                float x = this.getRLength() * this.getRLength(); return x;}
        public float findArea(Point p1, Point p2, Point p3, Point p4) { Line temp = new Line();
                float theRWidth = temp.findLength(p1,p2) - 1; //IVS accidental fault 2
                float theRLength = temp.findLength(p2,p3);
                theRLength = temp.findLength(p2, p3) - 1; //IVS10 seeded mutant 7, Invariant: length = width for square
                if (theRWidth == theRLength) { float x = theRWidth * theRWidth; return x; } else { return 0; }}
        public float findArea() { float x = this.findArea(this.getRLength()); return x;}
}
```

---

9) Triangle.cpp

```
class Triangle extends Parallelogram { public int sides = 3; private float height;
        public float getHeight() { return height; }
        public void setHeight(float theHeight) { height = theHeight; }
        public Triangle() { this.setSides(sides); }
        public Triangle(Point p1, Point p2, Point p3) throws IOException { this.setSides(sides);
                Line temp1 = new Line(p1, p2); Line temp2 = new Line(p2, p3);
                this.setHeight(temp1.findLength(p2, p1)); this.setRWidth(temp2.findLength(p2, p3)); }
        public float findArea(Point p1, Point p2, Point p3) throws IOException { Line temp1 = new Line(p1, p2);
                Line temp2 = new Line(p2, p3);
                float l1 = temp1.findLength(p1, p2); float l2 = temp2.findLength(p2, p3);
                if (this.sides == 3) { return (l1 * l2) / 2; } else { return 0; }}
        public float findArea(Point p1, Point p2) throws IOException { Point p3 = new Point(0, 0);
                Line temp1 = new Line(p1, p2); Line temp2 = new Line(p2, p3);
                float l1 = temp1.findLength(p1, p2); float l2 = temp2.findLength(p2, p3);
                if (this.sides == 3) { return (l1 * l2) / 2; } else { return 0; }}
        public float findPerimeter(Point p1, Point p2, Point p3) throws IOException {
                Line temp1 = new Line(p1, p2); Line temp2 = new Line(p2, p3); Line temp3 = new Line(p3, p1);
                float l1 = temp1.findLength(p1, p2); float l2 = temp2.findLength(p2, p3);
                float l3 = temp3.findLength(p3, p1);
                if (this.sides == 3) { return (l1 + l2 + l3); } else { return 0; }}
        public float findPerimeter(Point p1, Point p2) throws IOException { Point p3 = new Point(0, 0);
                Line temp1 = new Line(p1, p2); Line temp2 = new Line(p2, p3); Line temp3 = new Line(p3, p1);
```

```
            float l1 = temp1.findLength(p1, p2); float l2 = temp2.findLength(p2, p3);
            float l3 = temp3.findLength(p3, p1);
            if (this.sides == 3) { return (l1 + l2 + l3); } else { return 0; }}
}
```

Appendix C - Sample Test Cases

Test Case 1

```
Point pc = new Point(); pc.findArea();
```

Test Case 2

```
Point p = new Point(5, 5); Point p1 = new Point(1, 1); Point p2 = new Point(2, 2);
Line l = new Line(p, p1);
l.findArea(p, p1); l.findPerimeter(); l.findPerimeter(p1);
```

Test Case 3

```
Circle c1 = new Circle(); c1.findArea(3);
```

Test Case 4

```
Parallelogram poly1 = new Parallelogram(); poly1.findArea(); poly1.findPerimeter();
```

Test Case 5

```
Point p2 = new Point(2, 0); Point p3 = new Point(0, 2);
Triangle t1 = new Triangle();
t1.findArea(p2, p3); t1.findPerimeter(p2, p3);
```

Test Case 6

```
Cylinder cy1 = new Cylinder(); Cylinder cy2 = new Cylinder(3,4);
cy1.findArea(); cy2.findArea(cy2.getRadius(), cy2.getHeight());
cy2.findPerimeter(); cy2.findPerimeter(cy2.getRadius());
cy2.findVolume(); cy1.findVolume(cy1.getRadius(), cy1.getHeight());
```

Test Case 7

```
Point p = new Point(5, 5); Point p1 = new Point(7, 5);
Point p2 = new Point(7, 7); Point p3 = new Point(5, 7);
Rectangle r = new Rectangle(); Rectangle r1 = new Rectangle(3, 4);
r.findArea(3, 4); r1.findArea(p, p1, p2, p3);
```

Test Case 8

```
Point p = new Point(5, 5); Point p1 = new Point(7, 5);
Point p2 = new Point(7, 7); Point p3 = new Point(5, 7);
Rectangle r = new Rectangle(); Rectangle r1 = new Rectangle(3, 4);
Rectangle r2 = new Rectangle(p, p1, p2, p3);
r.findPerimeter(3, 4); r1.findPerimeter(p, p1, p2, p3); r2.findPerimeter();
```

Test Case 9

```
Square s1 = new Square(); s1.findArea(); s1.findPerimeter();
```

Test Case 10

```
Point p = new Point(5, 5); Point p1 = new Point(7, 5);
Point p2 = new Point(7, 7); Point p3 = new Point(5, 7);
Square s1 = new Square(); Square s2 = new Square(3); Square s3 = new Square(p, p1, p2, p3);
```

s1.findPerimeter(); s2.findPerimeter(); s1.findPerimeter(p, p1, p2, p3); s3.findPerimeter();

s1.findArea(p, p1, p2, p3);

Test Case 11

Square s1 = new Square(); Square s2 = new Square(3); s1.findPerimeter(); s2.findPerimeter(0);

Test Case 12

Point p1 = new Point (2,1); Point p2 = new Point (3,1);

Point p3 = new Point (3,6); Point p4 = new Point (2,6);

Rectangle r1 = new Rectangle(p1,p2,p3,p4); Line l1 = new Line();

r1.findPerimeter (); r1.findArea(); l1.findLength();

Test Case 13

Point p1 = new Point(2, 0); Point p2 = new Point(4, 0); Point p3 = new Point(3, 2);

Triangle t1 = new Triangle(); Triangle t2 = new Triangle(p1, p2 , p3);

t1.findArea(p2, p3); t2.findArea();

Test Case 14

Point p1 = new Point(2, 0); Point p2 = new Point(4, 0); Point p3 = new Point(3, 2);

Triangle t1 = new Triangle(); Triangle t2 = new Triangle(p1, p2 , p3);

t1.findPerimeter(p2, p3); t2.findPerimeter(p1, p2, p3);

Test Case 15

Circle c1 = new Circle(); Circle c2 = new Circle(10); c1.findArea(10); c2.findArea();

Test Case 16

Circle c1 = new Circle(3); c1.findPerimeter(3);

Test Case 17

Cylinder cy1 = new Cylinder();

cy1.findArea(); cy1.findVolume(cy1.getRadius(), cy1.getHeight()); cy1.findPerimeter();

Test Case 18

Cylinder cy2 = new Cylinder(6,10);

cy2.findArea(cy2.getRadius(), cy2.getHeight()); cy2.findPerimeter(); cy2.findPerimeter(cy2.getRadius());

cy2.findVolume();

Test Case 19

Cylinder cy1 = new Cylinder(); cy1.findArea();

Test Case 20

Cylinder cy2 = new Cylinder(10,4);   cy2.findArea(cy2.getRadius(), cy2.getHeight());

Test Case 21

Cylinder cy1 = new Cylinder(); cy1.findVolume(cy1.getRadius(), cy1.getHeight());

Test Case 22

Cylinder cy2 = new Cylinder(3,4); cy2.findPerimeter();

Test Case 23

Cylinder cy2 = new Cylinder(10,10); cy2.findPerimeter(cy2.getRadius());

Test Case 24

Cylinder cy2 = new Cylinder(3,4); cy2.findVolume();

Test Case 25

Point p = new Point(5, 5); p.findArea();

Test Case 26

Point p = new Point(5, 5); p.findPerimeter(3, 4);

Test Case 27

Point p = new Point(5, 5); p.findPerimeter();

Test Case 28

Point p = new Point(5, 5); Point p1 = new Point(1, 1); Line l = new Line(p, p1); l.findPerimeter(p, p1);

Test Case 29

Point p = new Point(5, 5); Point p1 = new Point(1, 1); Line l = new Line(p, p1); l.findPerimeter(p1);

Test Case 30

Point p = new Point(5, 5); Point p1 = new Point(1, 1); Line l = new Line(p, p1); l.findPerimeter();

Test Case 31

Point p = new Point(5, 5); Point p1 = new Point(1, 1); Line l = new Line(p, p1); l.findLength();

Test Case 32

Point p = new Point(5, 5); Point p1 = new Point(1, 1); Line l = new Line(p, p1); l.findArea();

Test Case 33

Line l1 = new Line(1, 1, 2, 2); l1.findLength(1, 1, 2, 2);

Test Case 34

Line l1 = new Line(1, 1, 2, 2); l1.findLength();

Test Case 35

Point p2 = new Point(2, 2); Line l1 = new Line(1, 1, 2, 2); l1.findLength(p2);

Test Case 36

Circle c1 = new Circle(); c1.findPerimeter(3);

Test Case 37

Circle c2 = new Circle(3); c2.findArea(3);

Test Case 38

Point p = new Point(1, 1); Point p1 = new Point(3, 1);

Line l = new Line(p, p1); Line l1 = new Line(4, 2, 2, 2); Parallelogram poly3 = new Parallelogram(l, l1);

poly3.findPerimeter(); poly3.findPerimeter(3, 4);

Test Case 39

Point p = new Point(1, 1); Point p1 = new Point(3, 1);

Line l = new Line(p, p1); Line l1 = new Line(4, 2, 2, 2);

Parallelogram poly1 = new Parallelogram(); Parallelogram poly3 = new Parallelogram(l, l1);

poly1.findArea(); poly3.findArea(30, 43);

Test Case 40

Point p = new Point(1, 1); Point p1 = new Point(3, 1);

```
Line l = new Line(p, p1); Line l1 = new Line(4, 2, 2, 2); Parallelogram poly3 = new Parallelogram(l, l1);
poly3.findPerimeter(l, l1); poly3.findArea(l, l1);
```

Test Case 41

```
Point p = new Point(1, 1); Point p1 = new Point(3, 1);
Point p2 = new Point(4, 2); Point p3 = new Point(2, 2);
Parallelogram poly2 = new Parallelogram(p, p1, p2, p3);
poly2.findPerimeter(); poly2.findArea(p, p1, p2, p3);
```

Test Case 42

```
Point p1 = new Point(4, 5); Point p2 = new Point(8, 5); Point p3 = new Point(6, 10);
Triangle t2 = new Triangle(p1, p2 , p3);
t2.findArea(p1, p2, p3); t2.findPerimeter();
```

Test Case 43

```
Point p = new Point(5, 5); Point p1 = new Point(7, 5);
Point p2 = new Point(7, 7); Point p3 = new Point(5, 7);
Rectangle r = new Rectangle(); Rectangle r2 = new Rectangle(p, p1, p2, p3);
r.findArea(3, 4); r2.findArea();
```

Test Case 44

```
Point p = new Point(5, 5); Point p1 = new Point(7, 5);
Point p2 = new Point(7, 7); Point p3 = new Point(5, 7);
Rectangle r1 = new Rectangle(3, 4); Rectangle r2 = new Rectangle(p, p1, p2, p3);
r1.findArea(p, p1, p2, p3); r2.findArea();
```

Test Case 45

```
Square s1 = new Square(); s1.findArea();
```

Test Case 46

```
Square s2 = new Square(3); s2.findArea(3);
```

Test Case 47

```
Point p = new Point(5, 5); Point p1 = new Point(7, 5);
Point p2 = new Point(7, 7); Point p3 = new Point(5, 7); Square s3 = new Square(p, p1, p2, p3);
s3.findArea(); s3.findArea(3); s3.findArea(p, p1, p2, p3);
```

Test Case 48

```
Square s1 = new Square(); s1.findPerimeter();
```

Test Case 49

```
Square s2 = new Square(3); s2.findPerimeter(3);
```

Test Case 50

```
Point p = new Point(5, 5); Point p1 = new Point(7, 5);
Point p2 = new Point(7, 7); Point p3 = new Point(5, 7); Square s3 = new Square(p, p1, p2, p3);
s3.findPerimeter(); s3.findPerimeter(3); s3.findPerimeter(p, p1, p2, p3);
```

Test Case 51

```
Point p1 = new Point (1,1); Point p2 = new Point (3,1);
Point p3 = new Point (3,3); Point p4 = new Point (1,3);
```

```
        Line l1 = new Line (p1, p2); Line l2 = new Line (p3,p4);
        Rectangle r1 = new Rectangle (l1, l2); r1.findArea(l1, l2);
```

Test Case 52

```
        Point p1 = new Point (1,1); Point p2 = new Point (3,1);
        Point p3 = new Point (3,3); Point p4 = new Point (1,3);
        Line l1 = new Line (p1, p2); Line l2 = new Line (p3,p4);
        Rectangle r1 = new Rectangle (l1, l2); r1.findPerimeter (l1, l2); r1.findArea(l1,l2);
```

Test Case 53

```
        Point p1 = new Point (1,1); Point p2 = new Point (3,1); Point p3 = new Point (3,3);
        Triangle t1 = new Triangle(); Triangle t2 = new Triangle(p1,p2,p3);
        t1.getHeight(); t2.getHeight(); t2.findArea(p1, p2);
```

Test Case 54

```
        Point p1 = new Point (0,0); Point p2 = new Point (1,1); Point p3 = new Point (2,2);
        Line l1 = new Line(p1,p2); Line l2 = new Line (p1,p3);
        l1.getX1(); l1.getX2(); l1.getY1(); l1.getY2(); l2.getX1(); l2.getX2(); l2.getY1(); l2.getY2();
        l1.findLength(); l2.findLength();
```

Test Case 55

```
        Point p1 = new Point (0,0); Point p2 = new Point (4,5); Point p3 = new Point (0,0);
        Line l1 = new Line(p1,p2); Line l2 = new Line (p1,p3);
        l1.getX1(); l1.getX2(); l1.getY1(); l1.getY2(); l2.getX1(); l2.getX2(); l2.getY1(); l2.getY2();
        l1.findArea(p1, p2); l2.findPerimeter(p1);
```

Test Case 56

```
        Circle c1 = new Circle(); c1.getCenter(); c1.findArea(2); c1.findPerimeter(56);
```

Test Case 57

```
        Parallelogram p1 = new Parallelogram();Square s1 = new Square();
        p1.getSides(); p1.findArea(); s1.findArea(); s1.findPerimeter();
```

Test Case 58

```
        Point p = new Point(5, 5); Point p1 = new Point(7, 5);
        Point p2 = new Point(7, 7); Point p3 = new Point(5, 7);
        Rectangle r = new Rectangle(); Rectangle r2 = new Rectangle(p, p1, p2, p3);
        r.findArea(3, 4); r2.findArea(2,3); r2.findPerimeter(2,3);
```

Test Case 59

```
        Parallelogram poly1 = new Parallelogram();
        poly1.findArea(); poly1.findPerimeter(); poly1.findArea(7,8); poly1.findPerimeter(2,4);
```

Test Case 60

```
        Square s1 = new Square(); s1.findArea(); s1.findPerimeter(); s1.findArea(2); s1.findPerimeter(3,4);
```

Test Case 61

```
        Cylinder cy1 = new Cylinder();
        cy1.findArea(); cy1.findPerimeter(); cy1.findPerimeter(3); cy1.findArea(3);
```

Test Case 62

```
Point p = new Point(6, 5); Point p1 = new Point(1, 2); Line l = new Line(p, p1);
l.findPerimeter(p, p1); l.findLength();
```

Test Case 63

```
Point p1 = new Point(1,1); Point p2 = new Point(5,1);Point p3 = new Point(5,5);
Triangle t1 = new Triangle(); Triangle t2 = new Triangle(p1, p2 , p3);
t1.findArea(p2, p3); t2.findArea(p1, p2, p3); t1.findPerimeter(); t2.findPerimeter();
```

Test Case 64

```
Circle c1 = new Circle(); c1.findArea(3); c1.findPerimeter(9); c1.findArea((float) 7.4);
```

Test Case 65

```
Parallelogram p1 = new Parallelogram (2,3); p1.findArea(); p1.findPerimeter();
```

Test Case 66

```
Point p1 = new Point(2, 0); Point p2 = new Point(4, 0); Point p3 = new Point(3, 2);
Triangle t1 = new Triangle(); Triangle t2 = new Triangle(p1, p2 , p3);
t1.findArea(); t2.findArea(p1, p2); t2.findArea(); t2.findArea(p1,p2,p3);
```

Test Case 67

```
Point p1 = new Point(1,1); Circle c1 = new Circle(3);
c1.setCenter(p1); c1.getCenter(); c1.findPerimeter((float)9.3);
```

Test Case 68

```
Point p = new Point(5, 7); Point p1 = new Point(9, 0); Line l = new Line(p, p1);
l.findPerimeter(); l.findArea(p, p1); l.findLength();
```

Test Case 69

```
Point p1 = new Point (2,1); Point p2 = new Point (3,1);
Point p3 = new Point (3,6); Point p4 = new Point (2,6);
Line l1 = new Line (p1, p2); Line l2 = new Line (p3,p4);
Rectangle r1 = new Rectangle (l1, l2); Rectangle r2 = new Rectangle(p1,p2,p3,p4);
r1.findPerimeter (l1, l2); r1.findArea(l1,l2); r2.findArea(); r2.findPerimeter(); r2.findPerimeter(3,9);
```

Test Case 70

```
Point p1 = new Point (2,1); Point p2 = new Point (3,1);
Point p3 = new Point (3,6); Point p4 = new Point (2,6); Line l1 = new Line();
Rectangle r1 = new Rectangle(p1,p2,p3,p4); r1.findPerimeter (); r1.findArea(); l1.findLength();
```